Project 1 - Odometry and Feature Extraction

Gabriel Capiteli Bertocco*,Thales Mateus Rodrigues Oliveira†,Elisangela Silva dos Santos ‡

*Computer Science - PhD Program - E-mail: gabriel.bertocco@ic.unicamp.br

, †Computer Science - Masters Program - E-mail:ra148051@students@ic.unicamp.br

, ‡Computer Science - Special Student - E-mail:ra149781@ic.unicamp.br

***Abstract* – In this project, the group was given task of building an odometry and feature extraction system for an autonomous wheeled robot, P3-DX. The mentioned goals are the basis for robot performance in the environment, as they guarantee robust and reactive operation in real time. Reliable enough odometry leads to trustworthy features, hence more complex actions such as control, mapping and navigation. To fulfill the requirements, the algorithm implemented deals with the kinematic model of the robot for odometry calculation, its sensors for acquiring information from the environment and a RANSAC algorithm to extract features. The V-REP simulator was used to test the developed structure. The group was able to generate different odometry qualities depending on sensor usage, with good results in the best case. The point extraction performs well, considering that the current position of the robot is reliable. The feature extraction algorithm is able to calculate obstacles such as walls and supports of objects, not considering possible outliers.**

***Keywords* – autonomous systems, robot kinematics, robot sensing systems**

I. Introduction

A. Background and motivation

The study of mobile robotics is of great importance and current, since there is growing demand and investments related to the area in various sectors of technology[1]. Studies, research and development made a major leap in technologies involving the application of mobile robots. Today, there are domestic, industrial, urban, militar and security applications, exemplifying the great importance of mobile robots today.

The area of Autonomous Robotics, that is, the performance of robot actions through space without any external assistance from humans, has gained importance as they can be extremely advantageous in many applications, reducing many expenses and increasing reliability.

In order to implement a robust autonomous system, the robot has to be able to execute a simple pipeline execution, which consists of sense, plan and act. The system needs to have the capacity of acquiring information from the environment (which is done by readings of sensors, such as cameras, Ultrasonic sensors, LIDARs) and itself (like encoders, gyroscopes[2], temperature and so on), decide what to do based on these readings, and perform the decided action.

The ability of a mobile robot to move in an environment is one of robotics' most basic and important activities. Successful locomotion is linked to the use of reliable sensors to provide the correct robot position information. With the right position information, the robot can make safe decisions, such as moving to interest points, map an unkown environment, perform specific actions it was supposed to (*e.g.:* deliver of machinery for industrial robots, identifying victims and their spots in desaster zones, map structures in cities).

To be able to move with accuracy to its goals, the robot relies on a good odometry[3] calculation, which consists in being able to analyze, by direct kinematics, its position and orientation based on the volocity applied in its wheels, in the case of wheeled robots.

Sensing the environment is also an important issue for autonomous robots. Choosing the right sensor is of paramount importance for the application, in order to avoid obstacles, do reliable mapping, and so on. With the information extracted from those sensors, local features from the environment can be processed, such as walls, unleveled structures, objects of interest, and others.

B. Work Objectives

The general objective of this work is to build an odometry and feature extraction systems able to estimate the position of the robot over time, and also to process information obtained from the sensors, for the robot Pioneer P3-DX, shown in figure 1 to be tested in the V-REP [4] Robotics Simulator.

Figure 1: Pioneer P3-DX.

C. Organization

The following work is divided into 4 sections. Section II deals with points extraction from the scene, section III with the Odometry, section IV talks about the evaluation of Points Extraction and Odometry together, and section V discuss the Feature Extraction algorithm. Those sections explain the implemented algorithms, related tests and discussions. Section VI sums up the work with the conclusion.

II. Points Extraction

The robot that we are working here has two kind of distance sensors: laser sensor and ultrasound sensor. To perform the point extraction we used only the laser sensor since its distance range from the robot is greater than the one returned by the ultrassound sensor. Besides, laser sensor get 1500 points at once in each calling to the reading data function as long as we have only 16 points got by the ultrassound sensors.

The laser sensor has its own reference system which is not the same of the global reference system when the robot is in different positions. So, to localize the points extracted from the enviroment we must have a standard coordinate system in which we will project all the points in a same space in order to have a consistent representation of the enviorment independent of the robot orientation. In this case, our standard system is the global reference system of the scene, therefore, given the coordinates of a point detected by the sensor we need to project this point from the local coordinate system of the sensor to the global reference system of the scene. Basically, we need to apply two transformation in the point: rotation and translation.

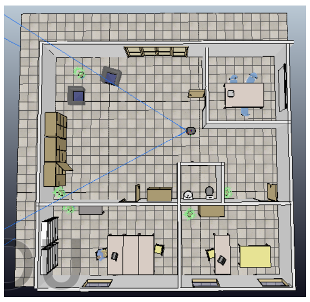
Once a point is detected by the laser sensor, we can access its coordinates on the local coordinates system. To get its values in the global one we need, firstly, to rotate the local system in order to have the same directions. We performed this applying the following equation to a point $P_0 = (x_0, y_0)$ detected by the laser sensor:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \quad (1)$$

Where θ is the rotation angle between the local e global reference frame and $P_1 = (x_1, y_1)$ is the same point detected by the laser but referenced in the rotated local system. It is importante to realize that the z coordinate is always the same value for all points in the same reference system since the robot does not change its altitude (robot does not climb up or go down), that is why we did not need to put it on the account. Besides, coordinate systems of the laser sensor has exactly the same directions of the coordinate system of the robot, that is, it is exaclty over the center of the robot. So we are always assuming that the system origin of both are the same. To get the coordinates of the point in the global coordinate system we need only to translate the local reference frame for coincide with the global reference frame, which can be mathematically expressed in the following equation:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_{laser} \\ y_{laser} \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \quad (2)$$

Where $P_{laser} = (x_{laser}, y_{laser})$ is the position of the laser related to global reference frame and $P_2 = (x_2, y_2)$ is the position of the point detected by the laser in the global frame. Now, the robot can move in any direction and all the points will be represented in the same system. So we proposed a trajectory to be followed by the robot in order to get into all parts (rooms) of the scene and get points from the whole enviorment. Once the robot finishes the trajectory we can see all the points representing the whole scene. This is shown in Figure 2.


(a) Upper view of the whole scene.

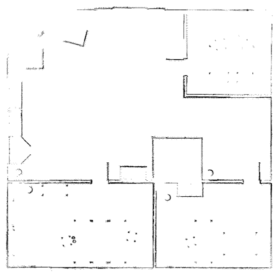

(b) Points extracted from the scene.

Figure 2: Result of Point Extracted compared to the real scene.

A. Discussion

Since in each iteration we are getting 1500 points, we filtered them out in order to have only 10 points equally having the same angle between two consecutives ones. We tested with another values of downsampling and we concluded that even with 10 points in each iteration, we are having a good representation of the points of the enviorment. The downsampling has been done because sometimes many beams scan over the same part of the enviorment or even the same beam might capture again the same point. This happens because of the high discretization of the laser sensor (1500 points at once), since we read the sensor in each iteration of the algoirthm, we could end up with 10 millions of points in a short distance that robot goes by. This leads to lot of redundancy, memory consuming and time consuming in the feature extraction stage. Therefore we chose to apply this downsampling strategy and we ende up with a lot of points but over the whole enviorment. Figure 2(a) shows us an upper view of the whole enviorment as long as Figure 2(b) show us the points extracted from it. We see that the majority has been captured which allow us to have a good representation and enough data for the feature extraction. Points that have not been scanned are mainly the ones in rooms where the door is closed, for instance the bathroom, others are behind the wardrobes and in the upper left nand right corners of the scene where the laser did not reach. Once we have the points extracted, we can perform the feature extraction.

III. Odometry

Odometry is the tracking of the robot trajectory performed based on its sensors information. In this work, we used two sensors to calculate the robot odometry: the encoders in the wheels and the gyroscope over the robot positioned exactly over the robot's center in order to have the axes of the gyroscope and of the robot in the same directions. So, the coordinates reported by the gyroscope frame will be the same reported in the robot frame. In fact, it could have a little difference in the altitude reported (z axe), however, since the robot does not change his altitude in the kinematic model, we only worry with the x and y coordinates that are the same in both frames. So, three experiments have been performed to calculate odometry: based only on the encoders wheels, based only on the gyroscope and based on the fusion of both. In order to refine the results, we performed a filtering in the angles calculated by the sensors in the sense that we multiplied the angles by a factor in order to increase or decrease the angle value. This has been done since we might have noise, errors, delays and some asynchronous behaviour between the python code and the V-REP simulator which could lead us to bad acquired data.

When we are tracking the robot trajectory in the global frame, we need to describe the robot position based on three values: x coordinate, y coordinate and the θ angle between the x (or y) axe of the robot (local frame) and the x (or y) axe of the global frame. We are going to reference them as (x_{robot}, y_{robot}) and (x_{global}, y_{global}) respectively. When the robot goes from the position $P^{(k)} = (x^{(k)}_{robot}, y^{(k)}_{robot}, \theta^{(k)}_{robot})$ to position $P^{(k+1)} = (x^{(k+1)}_{robot}, y^{(k+1)}_{robot}, \theta^{(k+1)}_{robot})$, we have a variation in each coordinate, mathematically we have: $\Delta x = x^{(k+1)}_{robot} - x^{(k)}_{robot}$, $\Delta y = y^{(k+1)}_{robot} - y^{(k)}_{robot}$, $\Delta\theta = \theta^{(k+1)}_{robot} - \theta^{(k)}_{robot}$. To calculate the variation in each coordinate, we have to consider the moviment of the wheels of the robot. Since we are working with P3dx robot, we need consider only two directed standard wheels in each side of the robot to get the values to the calculations. P3dx also has a castor wheel, but it is not taken in the account since it is only a suport wheel.

Let is suppose that the robot is doing a circular moviment to the left direction, so the velocity in the right wheel (V_r) is greater than the velocity in the left wheel (V_l) and we can aproximate the distance coursed by center of the robot to: $\Delta s = \frac{(V_r + V_l)\Delta t}{2}$. Since we know that $V_r = \omega_r R$, $V_r = \omega_l R$, $\omega_r = \Delta\phi_r \Delta t$ and $\omega_l = \Delta\phi_l \Delta t$, where Δt is the time taken by the robot for moving, $\Delta\phi_r$ and $\Delta\phi_l$ are the angular variations of the right and left wheels respectively, and R is the radius of the wheels, we conclude that $\Delta s = \frac{(\Delta\phi_l + \Delta\phi_r)R}{2}$. Then we can infer that $\Delta x = \Delta s . \cos(\theta^{(k)}_{robot} + \frac{\Delta\theta}{2})$ and $\Delta y = \Delta s . \sin(\theta^{(k)}_{robot} + \frac{\Delta\theta}{2})$. Now, we only need to calculate the $\Delta\theta$ which can be obtained by $\Delta\theta = \frac{(V_r - V_l)\Delta t}{2l}$. Applying the same idea that we did to calculate Δs, we have $\Delta\theta = \frac{(\Delta\phi_r - \Delta\phi_l)R}{2l}$, where l is the distance of each wheel to the center of the robot. Here we always assuming that the center of the robot is the midpoint between the two wheels in the robot chassi. Finally, we can put everything together to obtain the overall changing in the position point of robot:

$$\begin{bmatrix} x^{(k+1)}_{robot} \\ y^{(k+1)}_{robot} \\ \theta^{(k+1)}_{robot} \end{bmatrix} = \begin{bmatrix} x^{(k)}_{robot} \\ y^{(k)}_{robot} \\ \theta^{(k)}_{robot} \end{bmatrix} + \begin{bmatrix} \Delta s . \cos(\theta^{(k)}_{robot} + \frac{\Delta\theta}{2}) \\ \Delta s . \sin(\theta^{(k)}_{robot} + \frac{\Delta\theta}{2}) \\ \Delta\theta \end{bmatrix} \quad (3)$$

To obtain $\Delta\phi$ from the two wheels, we must use the encoders associated to them. To do this, we read the encoders in a instant of time t_0 and 0.1 seconds later we read them again and perform the subtraction of the angles obtained to have the angle variation in each wheel. This sleeping time has been chosen due to the simulation time. If we read the same encoders two times consecutively without any stopping, we have $\Delta\phi = 0$, since the execution of the code is so fast that does not allow to get a little significative change in the angles.

In V-REP, the angles returned by the encoders have a discontinuity, since they are read from π to $-\pi$. To get over this issue we used a conditional statment in order to have the right variation when the final and initial angles have different signals.

To calculate the angle variation in the wheels, we always need the encoders, however, to calculate $\Delta\theta$ we can also use them or we can use the gyroscope put over the robot. The gyroscope calculates the rotation angle in each of the three coordinates x, y and z, since the robot does not change its altitude along its trajectory, we only have rotation on the z coordinate when it turns to left or to the right. This rotation leads to a angle variation of the robot, so we can apply it directly to the $\Delta\theta$ in equation 3.

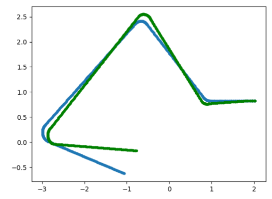
Based on these approaches, we performed three experiments to get the odometry: based only on the encoders (they are also used to obtain $\Delta\theta$) , based only on gyroscope ($\Delta\theta$ obtained from it), and based on the fusion of both ($\Delta\theta$ is get from a combination of gyroscope and encoders). Each approach and result are explained and shown in the following subsections.

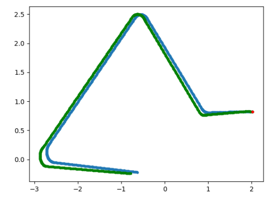
For each setup we applied an angle filtering multiplying by a factor c. We applied this strategy in order to get an improvment in the odometry since many times the angles returned by the fuction might have a lot of noise, erros or problems due to the assynchronous behaviour between the code of the process and the V-REP simulator. So looking for a smoothing, we seek to a factor value by a grid-searching in possible candidates.

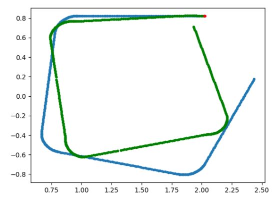
In all strategies (with and without filtering) we applied a threshold in the angle variations ($\Delta\phi$) of the wheels and in the rotation angle of the robot ($\Delta\theta$) in the sense that if a variation value is lower than a threshold tr, it is assigned directly to zero. We did this for avoinding accumulate low errors along the trajectory since the odometry is resulted from a accumulative calculation in each step of the trip. In all experiments we choose empirically $tr = 0.01$.

A. *Odometry only with encoders*

In fact, we do not have a native encoder associated to the wheels, but a sensor that able us to read the angle variation of them. In order to have a polished explanation we will always use the term ”encoders” to refer to angle reading from the wheels. So, in each wheel of the robot we have an encoder that get their angles and allow us to get the angle variation ($\Delta\phi$) of them as the robot moves on the scene. In this setup, these values are used to calculate the three variations: Δx, Δy, $\Delta\theta$. In order to check the performance of the encoders we selected three trajectories to test them, they can be seen in figure 3. The red point indicates where the trajectory starts, the green line is the ground truth (the real trajectory) and the blue line is the trajectory predicted by the odometry based only on encoders.


(a) Circuit 01 without filtering.


(b) Circuit 01 with filtering ($c = 1.05$).


(c) Circuit 02 without filtering.

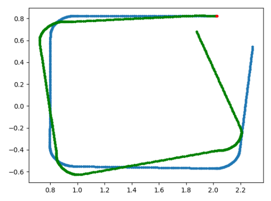

(d) Circuit 02 with filtering ($c = 1.05$).

Figure 3: Result of Odometry using only encoders with and without angle filtering.

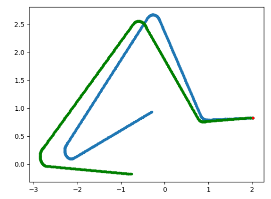
B. Odometry only with gyroscope

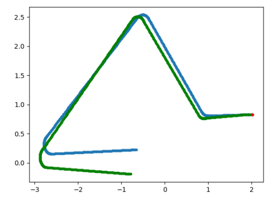
In this setup, we applied almost all the calculations of the previous one with only one difference: $\Delta\theta$ is get directly from the gyroscope without using any information from the encoders. The encoders are still being used to get the angle variation on the wheels but those results are only used to calculate Δx and Δy. The idea is to try to check the behaviour of the predicted trajectory using another sensor to help. In order to have a fair comparison, we kept all the same trajectories from the previous approach. The results are shown in the figure 4.

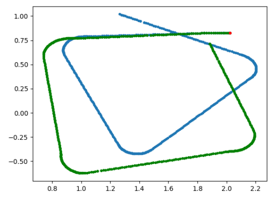
C. Odometry with encoders and gyroscope (Sensor Fusion)

After applying the two previous strategies, we can see that each one still have errors more associated to the curves of the trajectory. More specificaaly, setup with only encoders tends to predict angles lower than the real ones performed by the robot as long as the setup only with gyroscope tends to predict angles greater than the ones in ground truth. This can be easier checked when we compare the figures 3(c) and 4(c). Even after applying the filtering process in the angles, we got lower errors but the same conclusions remain for encoders and gyroscope strategies. Therefore we decied to put the both together in order to achieve a best result. Now the $\Delta\theta$ is obtained through the following equations to fuse both result sensors:

$$\Delta\theta_{fusion} = (\Delta\theta_{enco} + \Delta\theta_{gyros})/2 \qquad (4)$$


(a) Circuit 01 without filtering.


(b) Circuit 01 with filtering ($c = 0.85$).


(c) Circuit 02 without filtering.

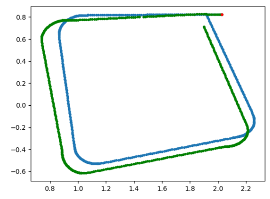

(d) Circuit 02 with filtering ($c = 0.85$).

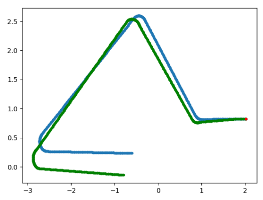
Figure 4: Result of Odometry using only gyroscope with and without angle filtering.

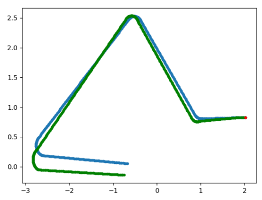
$$\Delta\theta_{fusion} = (c_{enco}.\Delta\theta_{enco} + c_{gyro}.\Delta\theta_{gyro})/2 \qquad (5)$$

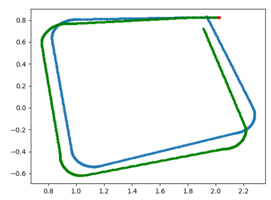
Where $\Delta\theta_{enco}$ is the angle variation obtained using the encoders, $\Delta\theta_{gyro}$ is the angle variation using gyroscope, c_{enco} and c_{gyro} are the factors that multiply each angle in the weighted sensor fusion. Since the filtering has been a good way to reduce error, as we will explain in the following subsection, we applied the same idea weighting each of the angle variations and taking the mean as shown in equation 5. The results of the fusion are in the Figure 5.

D. Discussion of the results

In the first setup based only on encoders, we can see that the error increases mainly after a curve that changes the trajectory in ninety or more degrees. This is because the way that the angle variations are calculated: the encoder of a wheel takes the initial angle, wait for a little time, and takes again the final angle and then the code perfoms the subtraction and get the $\Delta\phi$ variation in each wheel. During a curve, maybe it does not have enogh time to calculate, in each calling to the function, all the possible variations of the wheel that runs the further distance, since we are making the curve setting a wheel to zero and the another one to a velocity greater than zero (we always used 2.0). So the wheel with higher velocity will run the greater distance and then the encoders cannot take in the account all of the variations. It leads to mispredictions of trajectories that turns angles lower than the real ones as we can see mainly in figures 3(a) and 3(c). With filtering, the error is smoothed, by increasing the value of the angle multypling it by a fcator higher than 1.0, but we still have some great diferences mainly in the middle and in the end of


(a) Circuit 01 without filtering.


(b) Circuit 01 with filtering ($c_{enco} = 1.05$ and $c_{gyro} = 0.90$).


(c) Circuit 02 without filtering.

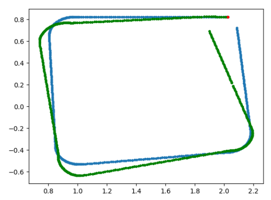

(d) Circuit 02 with filtering ($c_{enco} = 1.05$ and $c_{gyro} = 0.90$).

Figure 5: Result of Odometry using fusion of encoders and gyroscope.

trajectory since the error is propagated to the future steps of the odometry. Even if the prediction in a specific point is good, the summation of previous erros might overcome this right prediction. We can easier see this in figure 3(d) in the end of the trajectory. The first circuit with filtering is almost perfect due to the fact that we have a lower distance to be travelled which allow us to have a lower accumulated error and only two curves with angles higher than ninety degrees (the second and last ones). Figure 3(b) illustrates this explanation.

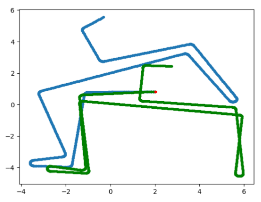
In the second setup using only the gyrsocope, we have the opposite behaviour of the preious one: higher predicted angles compared to the real ones. We can see this in Figures 4(a) and 4(c). We think this is related to the greater sensibility of the gyroscope in which it reports lower variation angles than the encoders but in a higher frequency and then we have erros acumullated faster in the curves. Therefore, in contrast to previous setup, we multiply the angles by a factor lower than 1.0 in order to get lower angles to be added to the odometry. Figures 4(b) and 4(d) can illustrate that the results are improved a lot but due to the sensibility, noise, and some assynchronous behavious we still have errors.

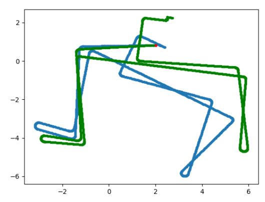
Compared to the first setup, the second one is a little worse, however, they are complementars, that is, the former predict angles lower than the real ones and the last predicts angles higher than the groud truth. So the combination of the them seems promising in a way that we can fuse them in a single result and get a better prediction. The results without any filtering and combining them using equation 4 is showed in figures 5(a) and 5(c). As we expected, the resulting odometries for each circuit is closert o the ground truth when we compared to the other non-filtered circuits. We have little improvement when comparing 3(a) and 4(a) to 5(a) due to the lower distance of the circuit 01 compared to the cirucit 02. In this last one we see a higher improvement with fusion not only in the endpoint but also along all the trajectory which the robot remained cosed to the ground truth. To be fair, we also applied the filtering strategy in the fusion using one factor to each angle (Equation 5). The factors are shown in the captions of the Figures 5(b) and 5(d). We see that for the first circuit we had a worse result, so encoders are better to be used in shorter distances with smoother angles. But ofr the circuit 02 which is longer only with ninety-degree curves, we see a better behaviour along the trajectory mainly in the curves which are better predicted and well shaped, even with a further endpoint compared to the gyroscope alone (Figure 4(d)).

Now that we exposed our conclusions to shorter distances, we will present the performance considering the trajectory used in Points Extraction of the scene, since it has angles in different degrees and covers almost all rooms and longer distances.

IV. Putting it togheter: Points Extraction with Odometry

Now that we have exposed all of the explanations and results of Points Extraction and Odometry, we will evaluate the odometry setups in a higher distance, with angles in different degrees, longer distances and covering almost all the enviorment. This will put in a higher test, since the erros tends to be higher as the distance and angles degrees increase. Since the filtering strategy was the better for the three setups, we only shows the results to them. Figure 6 shows the results.


(a) Odometry based only on encoders.


(b) Odometry based onlt on gyrosocope.

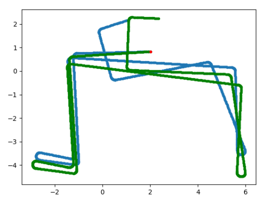

(c) Odometry with fusion.

Figure 6: Odometry result for each setup with filtering.

As previously concluded, even with filtering, setup with encoders tends to predict angles lower than the ground truth, and then in each curve (mainly in sharp curves) the error increases and eu to the longer distances, we have more accumulation of erros, and then teh final result is very different from the ground truth (Figure 6(a)). This enforces our previoius conclusion: encoders are better in shorter distances and lower angles. Following the same way of first setup, the second one based only on gyroscope still leads to high erros because of the same reason that we have already discussed: predicts higher angles possibly due to ythe sensibility (Figure 6(b)). However, as we were expecting, the fusion with filtering from both sensors gives us the best odometry as illustrated in Figure 6(c). Along almost all the trajectory the robot keeps closer to the ground truth accumulating lower erros with little deviation. During the turns the robot keeps closer to the expected trajectory without big discrenpancies as we saw two the previous ones. Therefore, the sensor fusion gave us the best performance and tends to get good results in further trajectories.

V. Feature Extraction

As the robot acquires information from the environment, the data must be processed in order to be useful in more complex actions such as decision making, mapping, navigation. In the case of autonomous robots, the distance points obtained from a variety of sensors can be used to deduce obstacles and landmarks in the environment, such as walls, doors, supports of objects, and so on. The recognition of those structures is fundamental to solve its main actions, like identifying victims in a desaster scenario, delivering resources in an unknown environment, robust mapping withoud human interference.

In our case, the 2D LIDAR sensor was used to obtain obstacle points related to the robot frame, and then converted to the global frame, as mentioned in the previous subsection. As the number of obtained points is huge (order of 10^6 samples for full exploration of our test scene), an algorithm for extracting line features from those points was implemented, based on the idea of Random Sample Consensus (RANSAC[5]). As a result, it is desired to obtain a map of line features representing obstacles in the enviroment, which could be later used in an Occupation Grid algorithm for mapping, and as landmarks in a A* algorithm for navigation, for example. The main implementation decisions are listed next.

A. Feature Extraction Trigger

To execute the Feature Extraction step for implementation, it was decided to do it offline, which is, after the exploration step is concluded. Even though this is a naive solution and possibly would not be recomended for real operations, it is a good choice for prototyping and validation of the algorithm. So, when discussing the execution pipeline, after the exploration and data acquirement step, the output points with respect to the global reference frame is saved and used as input to the Feature extraction algorithm, named **feature_extraction.py**. Then, it starts its execution.

B. Line Calculation with RANSAC

In order to find lines, the obstacle points are order with respect to their x and y coordinates,and then the RANSAC algorithm starts operating. As we are dealing with lines in the cartesian plane, the method struggles when trying to resolve vertical lines. To fix this issue, the algorithm executes two times, one for the regular xy plane, and another for the inverted yx plane, to be able to find the vertical ones. The names of the RANSAC execution functions inside the code are *ransac_regular_plane* and *ransac_inverted_plane*. The inputs for the functions are the obstacle points. As their functioning is almost the same (with the difference of with type of lines they calculate), a general explanation of the case for the regular plane is given.

For a predefined number of execution times, two points from the obstacle points list are taken randomly, and then a line crossing both points are considered. For every point in the set, if the point is close enough to the line (considering a threshold), the point is considered to belong to that respective line, and then saved in other structure. If the number of points (hit points) is high enough (considering a specific metric), we probably have a line. Analyzing those hit points, as they fit the calculated line, it is known that they are colinear enough, but not necessarily close enough to belong to the same structure (like a door, wall). Therefore, those points are passed through and the distance between neighbors is checked. If two consecutive neighbors are close enough (with respect to a metric), they probably belong to the same line. If not, we probably found a discontinuity. If the discontinuity is found, the number of consecutive points found until this step is counted. If they are greater than a metric, a line has been found. If not, those consecutive points are discarded, as they are probably outliers. If any lines have been found, they are stored by saving the x, y of the first point that fits the line and the x, y of the last point that fits that same line. So the algorithm returns a list of lines, in which every element contains a list with the following chracteristics: $\begin{bmatrix} x_{firstPoint} & y_{firstPoint} & x_{lastPoint} & y_{lastPoint} \end{bmatrix}$. Pseudocode is shown in algorithm 1

The threshold to analyze if points are close to the calculated line is done by taking the distance from all points in the set to the line, and then normalizing them. If the normalized distance to the point is below 10%, the point is considered belonging to the line. The number of minimum points necessary to form a line was chosen empirically, with the value of 10000. The number of minimum consecutive points to form a line was also defined empirically, with the value of 1000.

C. Line merging and Printing

After calculating the lines, one last attempt is done of trying to merge lines that are colinear and close enough. The obtained lines are ordered, and every line is checked with its right neighbor, to analyze if they are colinear and close enough. The metrics for comparison where obtained empirically. If they do, they are merged into a single one. If they do not, the line is painted on canvas for user output, but its parameters could be easily saved for usage in other modules. The process is done for lines in regular and inverted plane.

Algorithm 1 RANSAC

```
1: procedure RANSAC(POINTS)
2:     while $loop\_idx < MAX\_ITER$ do
3:         $guess[2] \leftarrow random(points)$
4:         $line\_incl \leftarrow (point[0].y - point[1].y)/(point[0].x - point[1].y)$
5:         $cross\_y \leftarrow point[1].y + point[1].x * ((point[0].y - point[1].y)/(point[1].x - point[0].x))$
6:         for $point\ in\ points$ do
7:             $d.append(CALCULATE\_DISTANCE(point,\ line\_incl,\ cross\_y))$
8:         end for
9:         $d \leftarrow NORMALIZE\_DISTANCE(d)$
10:        for $distance\ in\ d$ do
11:            if $distance < DIST\_THRESHOLD$ then
12:                $hit\_rate \leftarrow hit\_rate + 1$
13:                $hit\_idx.append(distance)$
14:            end if
15:        end for
16:        if $hit\_rate > MIN\_HITS$ then
17:            for $idx\ in\ hit\_idx$ do
18:                $d\_between\_points \leftarrow DISTANCE(point[idx],\ point[idx - 1])$
19:                if $d\_between\_points > MAX\_D$ then
20:                    if $n\_points > MIN\_CONS\_P$ then
21:                        $lines.append(LINE(start,\ points[idx]))$
22:                    end if
23:                else
24:                    $n\_points \leftarrow n\_points + 1$
25:                end if
26:                $start \leftarrow point[idx]$
27:            end for
28:        end if
29:        $loop\_idx \leftarrow loop\_idx + 1$
30:    end while
31: end procedure
```

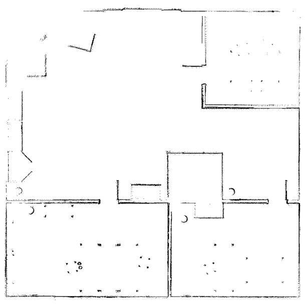


Figure 7: Canvas with extracted points painted on it

D. Experiments and Results

For features extracted from our test scene, the group uses the points selected from the exploration step, as mentioned before. For printing the lines, in other to make it easier for the reader to analyze the output, a canvas 1024x1024 pixels is created, and then the lines are written in blue color. To make it comparable with the points, figure 7 shows the canvas painted with the original points obtained by the sensors, in which every pixel represents one or more incidences. Figures 8 shows the execution of the algorithm for 10, 50 and 100 RANSAC iterations.

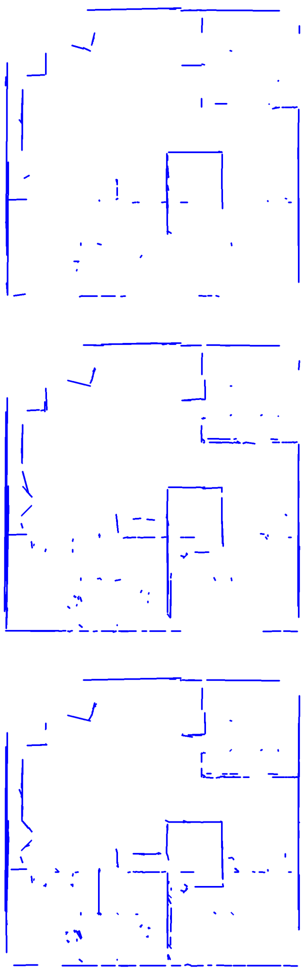


Figure 8: Canvas with extracted features. a) 10 RANSAC iterations. b) 50 RANSAC iterations. c) 100 RANSAC iterations

From the output images, it is possible to conclude that the RANSAC method can build features from the points with good consistency. The more iterations, the greater is the number of features built. As the method is sthocastic, the features change from execution to the other, so one interesting strategy is to guarantee a minimum number of execution times to extract all main features.

The line merging approach, although it is able to perform its objective sometimes, still needs to be improved, as it produces false positives in some cases (*e.g.:* false vertical line in bottom left room of canvas, connecting two support feet of table, for 100 iterations case).

A great number of features was not fully produced, as we can see some incomplete walls in Figure 8. The possible reasons for these cases are: lack of treatment for noisy data acquirement, poor strategy at choosing guess candidates (as we have 10^6 points magnitude), bad parameter tuning (as the majority of the comparison parameters where set empirically). The ”brute force” strategy, which consists of doing a greater number of RANSAC iterations (approx. 500 times), showed to be innefective as a trial to produce more features, which shows that fixing some of the problems could improve our results.

VI. Conclusion

The implementation of odometry and feature extraction modules to autonomous mobile robotics allowed us to understand some of the challenges in the area.

The module built to extract points from the action environment showed to be adequate, given the fact that the robot position is known. The odometry module also showed good results when sensor fusion was performed. It was able to see in practice the problems related to odometry solutions when relying on few sensors. The feature extraction module returned good results, and presents great opportunity for improvement in the line selection strategies.

Overall, the project was a great opportunity to get used to the tools explored in the area, and provided base material for future improvements, considering that more complex actions will be implemented for the robot during the semester.

+——————-+

References

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] W. is a Gyroscope?, “sparkfun,” 2019, [Online; accessed 28-September-2019]. [Online]. Available: https://learn.sparkfun.com/tutorials/gyroscope/all

[3] C. Esther, “Autonomous robotics class, lectures 5 to 8,” 2019, [Online; accessed 28-September-2019]. [Online]. Available: https://www.ic.unicamp.br/~esther/teaching/2019s2/mo651/index.html

[4] C. Robotics, “V-rep (virtual robot experimentation platform),” *Available: http://www.coppeliarobotics.com/.Last Access: September 28, 2019*, 2016.

[5] M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.