AI for robotics

Master SETI

DO NASCIMENTO SANTOS Alaf

2023/2024

Contents

1 Introduction

As part of the AI for Robotics module, the main aim of this work is to study Laser telemetry and localisation using a differential-motion robot, Thymio model, simulated with the **webots** software version 2021b and **Python 3.9v**.

Below are the objectives of the practical work:

- Getting to know the laser telemeter embedded in the Thymio
- Perform reference point changes to re-project data
- Programming an ICP localisation

Three Python files have been created: the first for the `controller` (**tp3.py**), which uses the `kinematicsFunctions` class (**kinematics_func.py**), which calculates positions and kinematics, the `graphWalls` class (**graph_walls.py**), which is dedicated to plotting with matplotlib, and one more file containing some functions (**functions.py**) that where developed and/or adapted from the provided assignment.

> **Note**: To solve the problem related to the use of matplotlib on the work computer, a solution was found based on the use of the simulator through the Wine software used to generate portability between the Windows executable and the Linux system. In the end, the program is exactly the same as the Windows version, but with an extra layer of compatibility.

2 Embedded laser telemeter

In this section, we want to explore the onboard laser telemeter embedded in the robot. We start by programming the retrieval of data from the sensor. The initialisation of the equipment in Python is done by the following:

```
lidar = robot.getDevice('lidar')
lidar.enable(timestep)
lidar.enablePointCloud()
```

Next, we can use `point_cloud = lidar.getRangeImage()` to read the information measured at a given instant. The acquired data is in polar format, its first value corresponds to angle 0, the second 0+1*2Pi/lidar.getHorizontalResolution(), the third 2*2Pi/lidar.getHorizontalResolution(), and so on. The following code was used to transform it into rectangular coordinates:

```
angle = 0
for i in point_cloud:
    xy = [i*math.sin(angle), 0, i*math.cos(angle)]
    angle+= 2*math.pi / lidar.getHorizontalResolution()
```

At this point, we want to see a graphical representation of the information we have treated so far. For this purpose, we use the plotting function developed in the previous practical work, where we can also see the labyrinth walls. Since the data from the robot's laser telemeter is expressed in the mobile robot, in order to display it in the global map it will be necessary to change the reference from the Robot to the Global map. The following function was used to do that operation for each robot pose:

```
def multmatr(X,Y,T):
    res = []
    res.append( X[0] * Y[0] + X[3] * Y[1] + X[6] * Y[2] - T[0])
    res.append( X[1] * Y[0] + X[4] * Y[1] + X[7] * Y[2] + T[1])
    res.append( X[2] * Y[0] + X[5] * Y[1] + X[8] * Y[2] + T[2])
    return res
```

When using the mentioned function for changing the reference, we must remember to apply the polar to rectangular transformation explained before. So the snippet of code below is used to process the entire set of telemeter points acquired at the `getRangeImage()` function call. Firstly we acquire the current robot pose from the simulator; and then we go through the data using a `for` loop, reading each value from the sensor (360 different angles); after, inside the loop, we apply the rectangular transformation, to obtain the x-axis and y-axis values, so we can apply the reference changing operation and we finish the loop iteration by increment the angle so we can continue to the next point.

```
node = robot.getFromDef("Thymio")
angle = 0
rotation = node.getOrientation()
xyz = node.getPosition()
for i in point_cloud:
    xy = [i*math.sin(angle), 0, i*math.cos(angle)]
    pt = multmatr(rotation,xy,xyz)
    angle+= 2*math.pi / lidar.getHorizontalResolution()
```

Finally, the laser rangefinder points can be displayed on the global map. Using the code presented, in the same webots world presented in the previous work, we can generate Figure 1. When moving the robot, it is possible to see the point cloud sticking to the wall. To achieve this result, as seen in the figure, it was necessary to walk the maze with the robot and save the data from each point in a list (which can be plotted at the end in real-time). In the case of the controller developed, the figure is updated every 1000 iterations of the main loop.

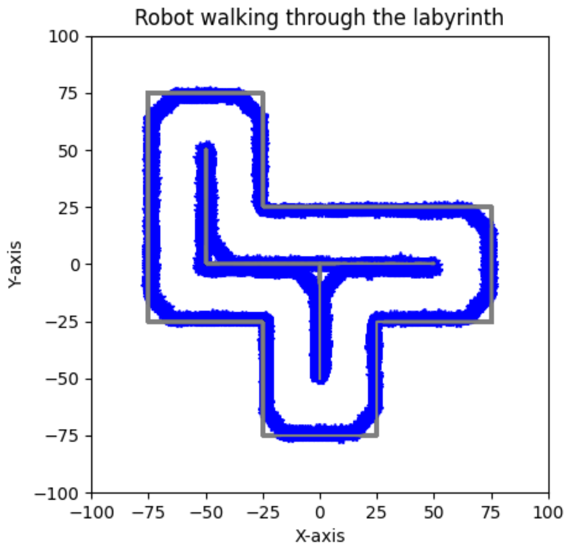
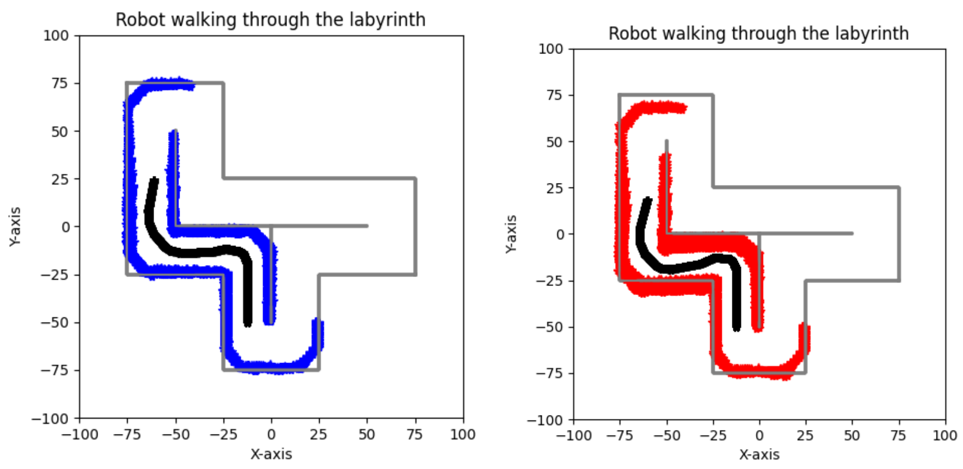


Figure 1: Laser telemeter points on the global map.

3 Laser rangefinders and localisation

We aim now to combine odometry-based localisation, from the previous practical report, with the use of the laser rangefinder. Using the localisation algorithm based on the integration of odometric data, we can apply the same principles previously explained to plot Figure 3b. Figure 3a represents the reference for the same performed simulation time. Figure 2 can be used to observe a gradual drift in the point cloud concerning the error in Figure 3b, which can be explained by the conclusions in the last report, in which some points that make odometry inaccurate were discussed.



(a) Reference demonstrative path. (b) Measured demonstrative path.

Figure 2: Demonstrative path.

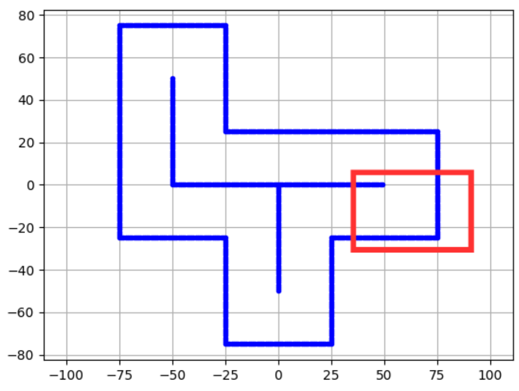
Considering our aim of being able to realign the point cloud using the map of the environment and then realign the location, some functions were suggested for applying the Iterated Closest Points (ICP) technique in the Point cloud re-calibration step. In the following code snippet, it is possible to see the ICP function using a Singular Value Decomposition (SVD), which we will explain in more detail:

```python
def indxtMean(index,arrays):
    indxSum = np.array([0.0, 0.0 ,0.0])
    for i in range(np.size(index,0)):
    indxSum = np.add(indxSum, np.array(arrays[index[i]]), out = indxSum ,casting = 'unsafe')
    return indxSum/np.size(index,0)

def indxtfixed(index,arrays):
    T = []
    for i in index:
    T.append(arrays[i])
    return np.asanyarray(T)

def ICPSVD(fixedX,fixedY,movingX,movingY):
    reqR = np.identity(3)
    reqT = [0.0, 0.0, 0.0]

    fixedt = []
    movingt = []
    for i in range(len(fixedX)):
    fixedt.append([fixedX[i], fixedY[i], 0])
    for i in range(len(movingX)):
    movingt.append([movingX[i], movingY[i], 0])
    moving = np.asarray(movingt)
    fixed = np.asarray(fixedt)
    n = np.size(moving,0)
    TREE = KDTree(fixed)

    for i in range(10):
        distance, index = TREE.query(moving)
        err = np.mean(distance**2)
        com = np.mean(moving,0)
        cof = indxtMean(index,fixed)
        W = np.dot(np.transpose(moving),indxtfixed(index,fixed)) - n*np.outer(com,cof)
        U , _ , V = np.linalg.svd(W, full_matrices = False)
        tempR = np.dot(V.T,U.T)
        tempT = cof - np.dot(tempR,com)

        moving = (tempR.dot(moving.T)).T
        moving = np.add(moving,tempT)
        reqR=np.dot(tempR,reqR)
        reqT = np.add(np.dot(tempR,reqT),tempT)
```

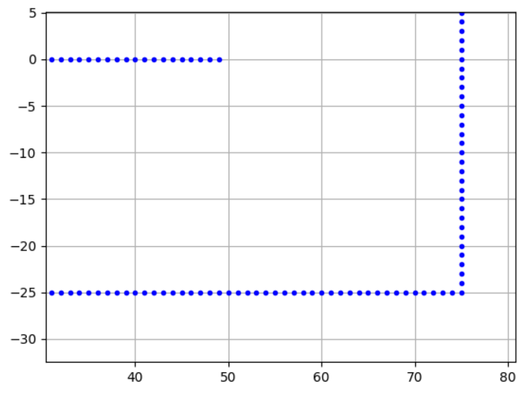
Focusing on the "ICPSVD" function, we have at the initialisation phase a variable called `reqR` initialised as the identity matrix, representing the accumulated rotations. Then, we can find the `reqT` variable which is initialised as a zero vector, representing the accumulated translations. When preparing the data, the input points "fixedX" and "fixedY" (representing the reference point cloud) and "movingX" and "movingY" (representing the moving point cloud) are converted into 3D points (with a Z-coordinate equal to zero). Subsequently, we create a KDTree, which is a data structure used to find nearest neighbors in the fixed point cloud for each point in the moving point cloud. Finally, we have a `for` loop in which 10 iterations are performed as follows:

1. `distance`, `index`: The nearest neighbors in the fixed point cloud are found for each point in the moving point cloud;
2. `err`: The mean squared distance error between corresponding points is computed;
3. `com`, `cof`: The centroid of the moving point cloud and the centroid of the corresponding points in the fixed point cloud are computed;
4. `W`: A weighted covariance matrix is computed based on the matched point pairs;
5. `U`, `V`, `tempR`, `tempT`: A Singular Value Decomposition is applied to W to estimate the rotation and translation that align the moving point cloud with the fixed point cloud;
6. `moving`: The moving point cloud is transformed according to the estimated rotation and translation;
7. `reqR`, `reqT`: The accumulated rotation and translation are updated based on the estimated transformation.

Before effectively using the `ICPSVD` function, we must know that it does not take wall segments as parameters. Because of that, we need to discretise the walls into the form of points in 1 cm steps. Figure 3 shows the result of creating this point cloud for the walls so we can apply ICP using them as fixed points.



(a) Discretised walls zoomed out.



(b) Discretised walls zoomed in.

Figure 3: Discretised walls.

Starting with the previously mentioned walls segments, already discussed over the second TP, the following function was used to generate the previous point cloud:

```python
def get_discretised_walls(walls):
    discretized_walls_x = []
    discretized_walls_y = []

    for wall in walls:
        wall_x = []
        wall_y = []

        for i in range(len(wall['x']) - 1):
            x0, y0 = wall['x'][i], wall['y'][i]
            x1, y1 = wall['x'][i + 1], wall['y'][i + 1]
            length = np.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)
            num_steps = int(length * 100)  # converting length to cm
            x_step = (x1 - x0) / (num_steps + 1e-100)
            y_step = (y1 - y0) / (num_steps + 1e-100)

            for j in range(num_steps):
                x_aux = round((x0 + j * x_step)*100)
                y_aux = round((y0 + j * y_step)*100)
                wall_x.append(x_aux)
                wall_y.append(-y_aux)

        discretized_walls_x.append(wall_y)
        discretized_walls_y.append(wall_x)

    return discretized_walls_x, discretized_walls_y
```

Figure 4 shows the result of applying the ICP technique to the LIDAR data. Compared to the Figure 3b it is possible to see a better adjustment of the point cloud to the walls, mainly in the upper area of the figure, where we had a completely out-of-wall piece of cloud. We apply the reqR and reqT values to the measured LIDAR data, point by point, so we can obtain such results. The return of the ICPSVG function (reqT and reqR) has the following format:

```
reqR:
    [ 0.99979192  0.02039896  0.        ]
    [-0.02039896  0.99979192  0.        ]
    [ 0.          0.          1.        ]]

reqT: [ 0.2115393   2.37057664  0.        ]
```

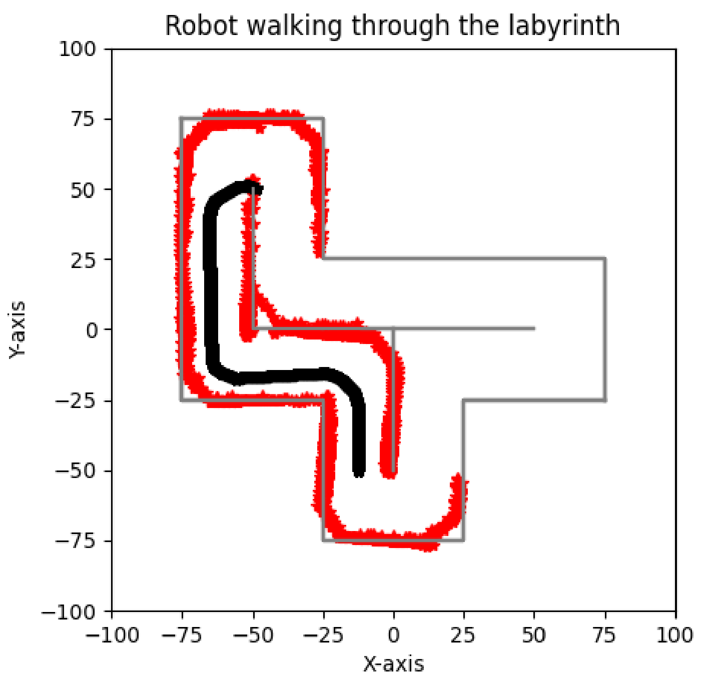


Figure 4: LIDAR data after applying ICP.

The transformation required to realign the two clouds corresponds to the error produced by the odometry algorithm. Because of that, we are going to change our controller to calculate the transformation every 5 seconds (the aim is to give the odometry time to drift). Then we apply the transformation to realign the robot's location. Table 1 shows a comparison between the reference taken from the simulation and the location with/without the transformation.

Real Position (x,y)	**Measured (x, y)**	**Measured with ICP (x, y)**
(0.125, -0.500)	(0.125, -0.500)	(0.083, -0.509)
(0.138, -0.206)	(0.142, -0.206)	(0.124, -0.217)
(0.209, -0.111)	(0.227, -0.125)	(0.200, -0.170)
(0.606, -0.048)	(0.626, -0.161)	(0.608, -0.221)
(0.656, 0.075)	(0.683, -0.039)	(0.682, -0.050)
(0.628, 0.513)	(0.643, 0.401)	(0.610, 0.345)
(0.358, 0.376)	(0.475, 0.207)	(0.358, 0.426)

Table 1: Comparing location with/without alignment over time.

4 Conclusions

In this study, we explored the implications of modifying our algorithm with the final aim of eliminating the use of odometric data, focusing on data provided by the laser rangefinder. This adaptation requires a significant revision of our localisation approach, highlighting the strengths and limitations of this new methodology. So unfortunately, the final goal wasn't achieved, due to different constraints involving the project deadline and other limitations discussed in our previous work. Moreover, we successfully integrated the use of the LIDAR into our localisation framework, achieving promising results in terms of data fusion and environment perception. Additionally, we took initial steps towards utilising Iterative Closest Point Singular Value Decomposition (ICPSVD), demonstrating its potential for improving localization accuracy and robustness in dynamic environments.

It's worth noting that a considerable amount of time was allocated to installing and adapting the tool to the Linux environment. As a result, the time available for the actual implementation and refinement of the algorithm was limited. Despite this constraint, we managed to make significant progress, laying a solid foundation for future development and optimisation efforts. The qualification of this algorithm relies on a comprehensive comparative analysis with our three previous localisation methods, showcasing the potential for great performance in terms of accuracy, robustness, and adaptability to complex and changing environments. This comparative evaluation will determine the relevance and effectiveness of a future laser-only-based approach, providing crucial insights for the continuous improvement of our localisation systems in real-world contexts.