Introduction to Robotics (236972)

Course project

Final Report

April 2020

Submission by:

- Alexander Shender 328626114
- David Alpert 201462538

Abstract

This report summarizes the work done in the scope of the final project for the Introduction to Robotics course. The report presents visually only some of the results, out of the complexity to perform the real tests in the current situation for better visualization.

The algorithms can be verified with the open-source github repository located at:

https://github.com/aka-sova/236927_project

The README.md file gives a brief explanation on the operation of the program.

Contents

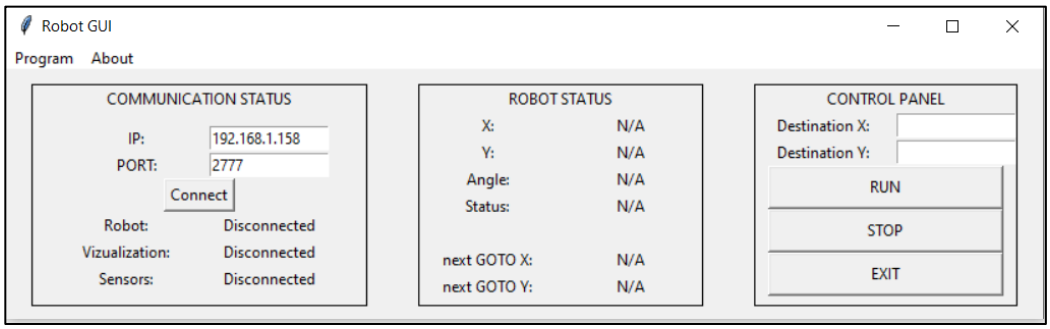
1. Brief overview

The program for navigation in the presence of obstacles consists of 2 separate GUIs: robot GUI and supplementary GUI for visualization.

a. Robot GUI

This GUI incorporates the access to the main program, which handles the communication with the robot, receives the commands from the user, and performs the algorithm calculation.

The general overview:



3 separate panels are described:

i. Communication status

Here, the user specifies the IP, PORT, connects. The bottom 3 lines indicate the connection status between the current program and (1) the robot (2) the secondary visualization program (3) the sensors. The 'sensors' status indicates the correctness of the sensor readings, when invalid readings are received, an "Error" indication appears

ii. Robot status

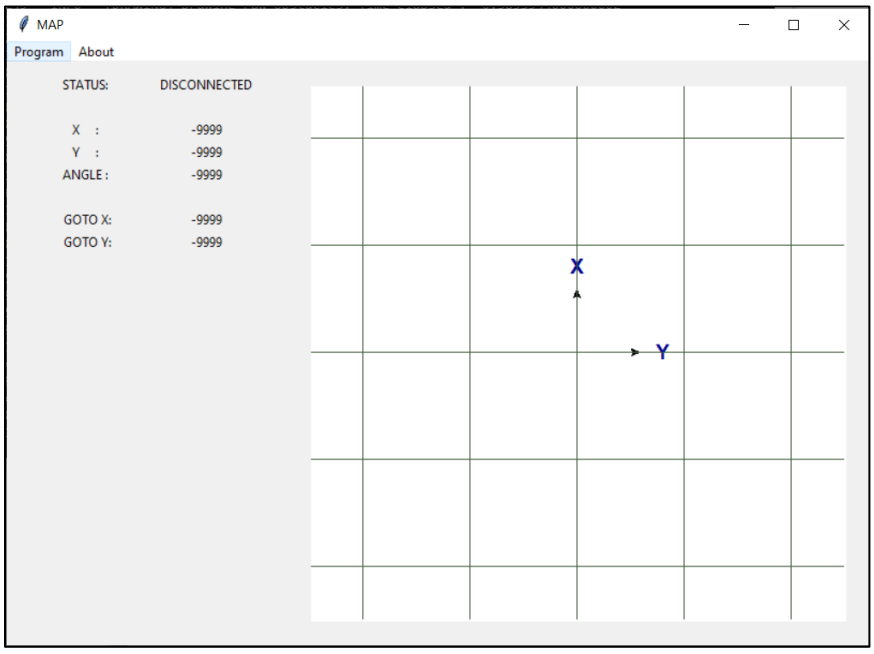
This pane indicated current robot pose (location, orientation), current 'status' (its current action), and the location of the next GOTO point (the robot may follow various GOTO locations of a way to its final destination)

iii. Control panel

The user gives the robot the X,Y location of the target here as well.

b. Visualization GUI

This supplementary GUI incorporates the access to the supplementary program, which handles the visualization task, which is to show the user the current available data (obstacles), draw the current chosen path, draw the current robot location and orientation. The grid is in units of 1 [m].



The user has no control buttons here. The main program identifies automatically that the supplementary GUI is connected and starts transmitting the data to it.

The decision to main 2 programs comes from 2 reasons:

1. Supplementary program is not necessary for the main program to act, and is incorporated mainly to monitor the robot activity in real time.
2. Supplementary program can use resources which may be necessary for a quick work of a main program.

2. Algorithms details

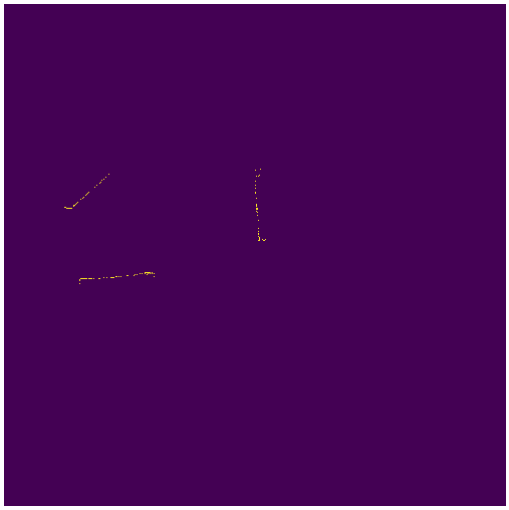
The algorithm implemented for the path planning is the RRT* algorithm. The description of this algorithm itself is present in many academic and other sources and is redundant in the scope of this report. The parameters were tuned to give an optimal solution to the problem. What will be described here are other algorithms for successful path finding

a. Collecting the obstacles data

The obstacles data is stored in 3 arrays, where the path finding algorithm uses the 3rd array.

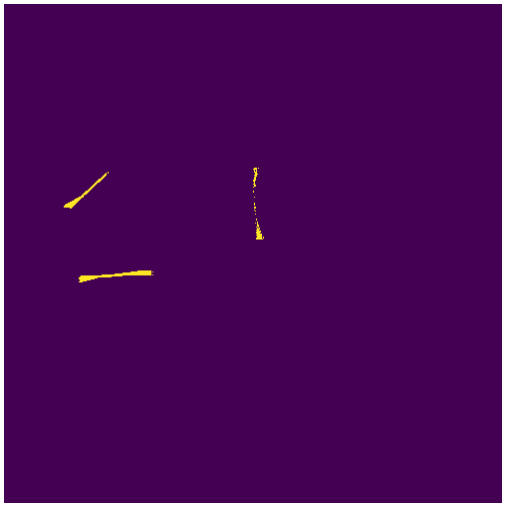
1st array:

The robot acquires the data from its sensors (note: we know that data is obtained through the camera, not 'sensors'), the orientation and the location of the robot is given. By knowing those, the exact location of the obstacle can be calculates using simple trigonometry rules. The obstacles POINT data is then stored in the first array. Example (yellow are are obstacles)



2nd array:

This array stores the obstacle data as does the first array, but has a preprocessing before entering a new obstacle point. It is obvious, that if we have 2 obstacle points, where the distance between them is 2 [cm], the robot will not pass between them. Thus, the second stage scans all the new obstacle points discovered, and checks if their distance from any existing point in the 2nd array is less than a certain margin (e.g. width of the robot). If it is – then all the points between those 2 obstacle points. If it isn't – the point is received as is. Example (continued from previous)



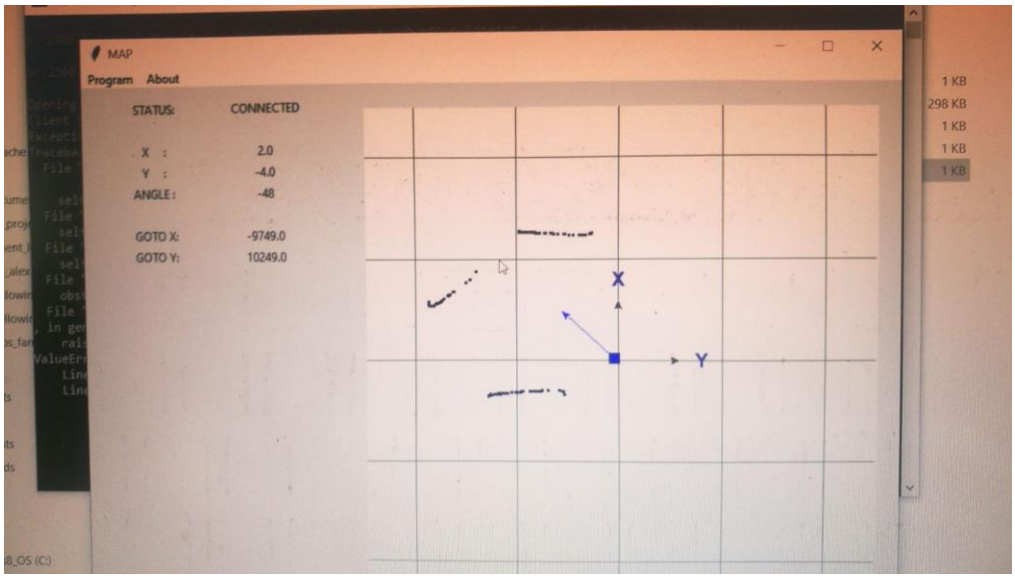
3rd array:

This array is created directly from the 2nd array. It ‘inflates’ all the obstacles, by a certain margin. This is done to give a safety margin to a robot, due to its actuator’s inaccuracies, and the sensors inaccuracies. Example:

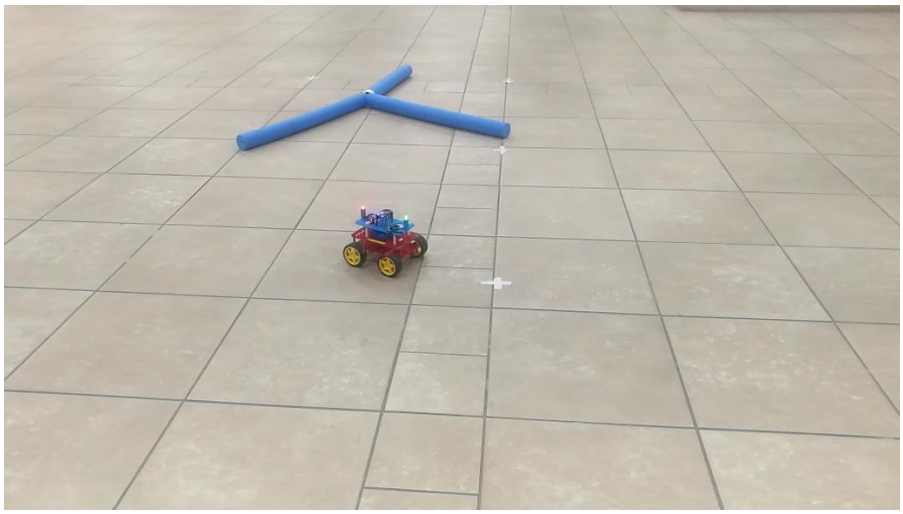
Real world example:

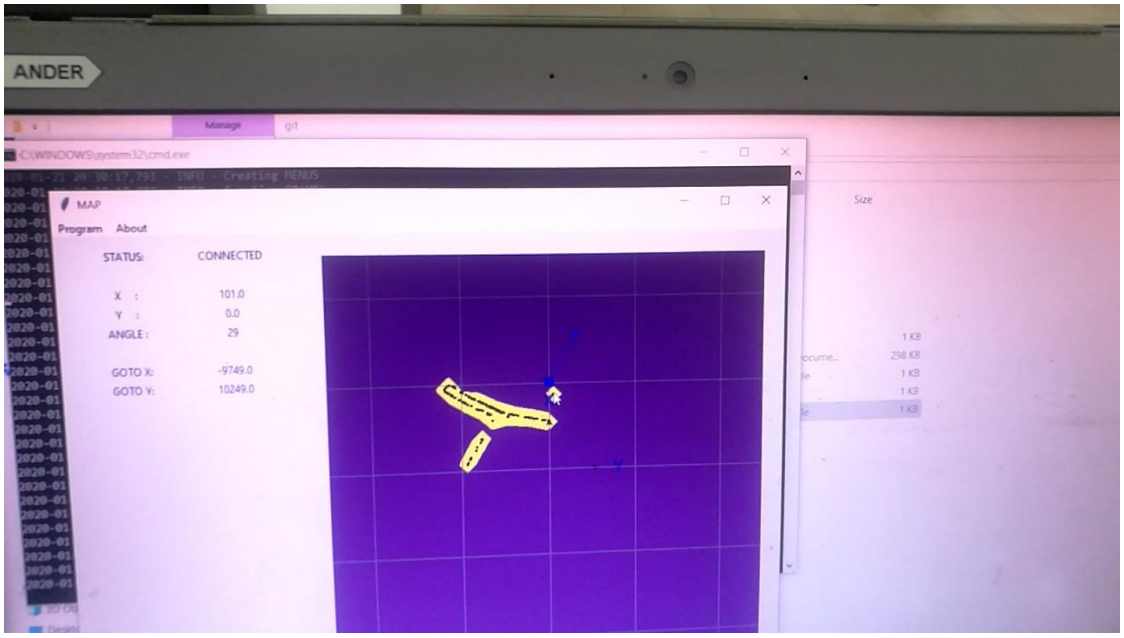
The screenshots below show the obstacles real location and corresponding 1st array.

We can also see how the visualization GUI shows the robot location, orientation, and the obstacles on the map.



Same from another scenario (we can see 1st Array and (2+) 3rd array)





By the way, we can see an erroneous measurement, which led to an unreal obstacle being present at the upper right corner. This may be perhaps the robot itself, since it is colored blue, as the obstacles themselves.

b. Work with threads

The whole work is based on threads in Python. Threading allows to have different parts of the program to run concurrently. For example, we want the robot to calculate movement, but still while he is doing it, our sensors data received to be updated. The function which is responsible to bring the robot to its destination is also a thread.

c. Calibration

Before the whole process, the robot performs the calibration once, calculating the value of a command it has to send to its actuators to move to a certain distance, or to rotate by a certain angle. Those tables are saved, and loaded each subsequent time. The whole process of calibration is 1-click process and automatic, the movement/rotation data is received from the sensors. Below are those generated tables:

Encoder_cmd	Movement
350	7.86
400	10.81
450	15.46
500	14.5
550	16.45
600	18.8
650	21.78
700	23.56
750	25
800	26.73
850	26.39
900	27.23
950	27.22
1000	29.73

Encoder_cmd	Rotation
350	20.19
400	28.06
450	36.8
500	47
550	55.06
600	66.2
650	74.6
700	76.28
750	87.46
800	91.84
850	99.43
900	99.88
950	109.35
1000	115.3

Those measurements were achieved using the sensor information through the camera, thus it is not accurate and may require several calibrations.

d. MAIN ALGORITHM

I will elaborate mainly on the main algorithm, which is responsible to bring the robot from Point A to point B. Other algorithms will only be described shortly.

In general, robot receives the [X,Y] coordinates of the destination. If the robot has no information on the surroundings, he rotates 360 degrees with a certain pace to obtain some map information around it. Given the current map, the Planner calculates the optional path (under given certain constrains, like no. of iterations), and returns it to the main function as the GOTO points list. The robot then follows each GOTO point. To follow each GOTO point, the robot calculates the distance, and the angle towards that point. Then the robot will divide the distance into a number of segments, and travel by those segments. Before driving a certain segment, the robot verifies that there's no obstacle in front of it (it waits until sensors data is being processed into the map). If there is an obstacle, the robot doesn't drive this segment, but instead rotates 360 degrees to obtain more information on the obstacle. The algorithm then calculates a new trajectory bypassing the known obstacles in the map, and returns a new GOTO list to follow. The process continues until the robot reaches the destination.

Important points:

- The obstacles information is collected and stored, and can be retrieved
- The subsequent path planning solutions will be better as the map of the environment is being updated with new obstacles information. When the whole map is explored, the solution is more likely to be optimal (under given algorithms constraints and parameters) and the solution (GOTO points list) will go around the obstacles in the first given solution.

The more detailed explanation of the main functions is given below.

i. Reach Destination Thread

This algorithm is a general algorithm which guides the robot. The easiest will be to paste my code here, which I have extended with more comments (in green).

```
def reach_destination_thread(self):
    """The main function which will look for a path to find to reach the goal"""

    # The flag which is triggered by the user. Check every 0.5 seconds
    while self.reach_destination_flag == False:
        time.sleep(0.5)

    # if location is invalid, wait
    target = self.current_destination
    while self.cur_loc == [-9999, -9999]:
        self.logger.info("Current location is invalid. Retry")
        time.sleep(1.0)

    # If the target is first target, get the map around the robot to see if there are obstacles
    # Rotate 360 degrees with a small pace
    if self.first_destination :
        self.gui.lbl_status.set("Get env. data")
        self.rotate_around(rotate_step = 45)
        self.first_destination = False

    self.logger.info('Initializing the Algorithm')
    self.dest_reached = False

    # will stay inside the loop until reached destination
    while not self.dest_reached:

        # Employ the RRT* algorith. Give as input current [X,Y], target [X,Y], and the map of obstacles
        # output is the list the GOTO points which robot has to path to reach destination
        self.gui.lbl_status.set("Calculating path")
        goto_list = self.planner.find_path(self.cur_loc, target = target, map = self.map)

        # Save the current target and the goto_list inside a file for the vizualiser (second program)
        # GOTO_LIST is a list of targets, each has X, Y coordinates
        self.update_path_file(goto_list)

        # Follow the list of the GOTO points. Return '0' if finished and didn't meet any obstacle on the way
        success_code = self.follow_goto_list(goto_list)   # 0 is success
```

```
if success_code == 0:
    self.logger.info('Algorithm completed')
    # check that the destination was indeed reached
    # 'calc_metrics' calculates distance and angle between robot and the target
    distance, _ = self.calc_metrics(target)
    if distance < C_CONSTANTS.REACH_MARGIN:
        self.dest_reached = True
        self.logger.info('Reached the destination')
        self.gui.lbl_status.set("Reached destination")

        # start the thread again, wait for new Destination
        self.reach_destination_flag = False
        self.current_destination = None
        self.reach_dest_thread=threading.Thread(target=self.reach_destination_thread)
        self.reach_dest_thread.start()
        break
else:
    # this will return to the beginning of 'while' loop and calculate the new path
    self.logger.info('Obstacle found on the GOTO path. Destination not reached')
```

ii. Follow GOTO list

This algorithm will follow the list of the GOTO points, will return ‘1’ is met obstacle on the way, else ‘0’.

Receive a list of the GOTO targets.

FOR goto_point IN goto_targets DO

    Obstacle_encountered = Follow_goto_point(goto_point)

    IF (obstacle_encountered) THEN

        Return 1 (met obstacle

RETURN 0 (success).

iii. Follow GOTO point

This algorithm will follow the GOTO point until reached under the margin of the 'reach_margin'.

Goal_reached = false

Receive GOTO point.

Distance, angle <- Calculate the angle and distance to the point.

If (distance < reach_margin)

    RETURN

WHILE (goal_reached = false) DO

    Correct_robot_angle *(adjust its orientation towards the goto point)*

    Drive_distance_long(distance)

    Distance, angle <- Calculate the angle and distance to the point.

    If (distance < reach_margin)

        goal_reached = True

iv. Drive distance long

The robot has to check constantly that its movement is in the correct direction and that its sensors have not detected any new obstacles on the way. This is the reason why the movement is performed in 'segments' of approximately 30 [cm], after which the robot checks if it has to readjust, or recalculate the route (if obstacle encountered).

```
 Receive the target, distance
Max_driving_distance <-  max(interpretation from calibtation)  // will be explained later
IF (distance < Max_driving_distance)
    Drive_distance_short(distance)
ELSE
    Drive_segments = int(distance / Max_driving_distance )
    Segment_path = distance / drive_segments
    FOR segment in RANGE(Drive_segments) DO
        Drive_distance_short(Segment_path)
        SLEEP(1.0 [s])  // until finished moving and obtaining obstacles data
        Current_robot_angle(current_pose, target)   // check if need to adjust.
```

v. Drive distance short

Driving the distance, which is in the reach of 1 single command to the motors. Before moving, validate that the path is clear. RAISE operation in this algorithm will return this error code directly to the “Follow GOTO list” algorithm.

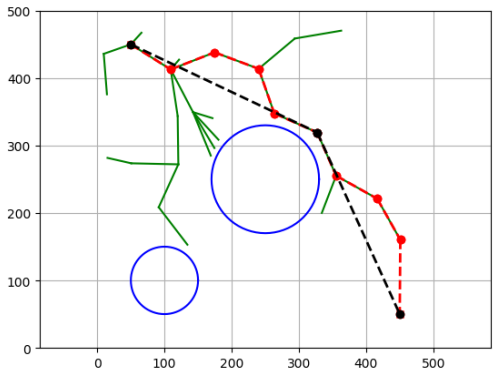
```
Receive current pose, target location
Max_driving_distance <-  max(interpretation from calibtation)
Min_driving_distance <- min(interpretation from calibtation)

Distance = min(Max_driving_distance, distance)  // safe check – distance to drive
Path_clear = check_clear_path(distance)   // using the obstacles map
IF (Path_clear = True)
    IF (Distance < Min_driving_distance)
            RETURN;   // distance too small to drive
    // get the right command value for the motors:
    Command_value = int(interpolate(calibration_data, distance))
    DRIVE (command_value, command_value)
ELSE
    RAISE “OBSTACLE_INTERFERENCE”
```

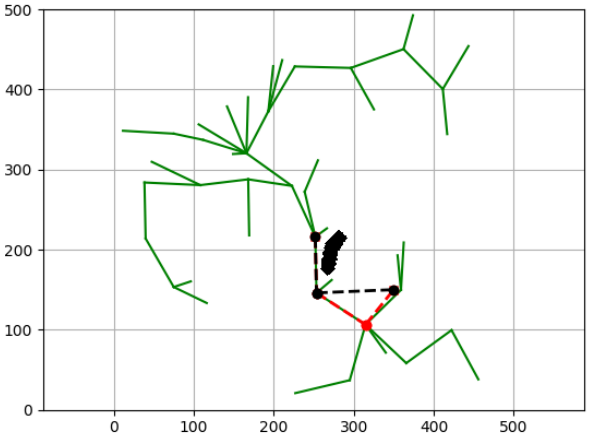
e. Path Planning tuning

The RRT* algorithm was implemented with all it's functions. It does not store the already developed graph, and creates a new graph every time. Since I cannot provide the latest data (I do not live near the Technion at the moment), the following images are made during the debug progress, and from the screenshots of occasional videos made during the development.

A test of our algorithm was first made with some abstract obstacles:



As we can observe, blue are the obstacles, green is the developed tree. Red (sometimes overlaps with the black) is the found path before optimization, and black is the final path after optimization (removing redundant nodes). The list of the nodes on the black path is returned to the robot. An output from the real scenario during one of the runs:

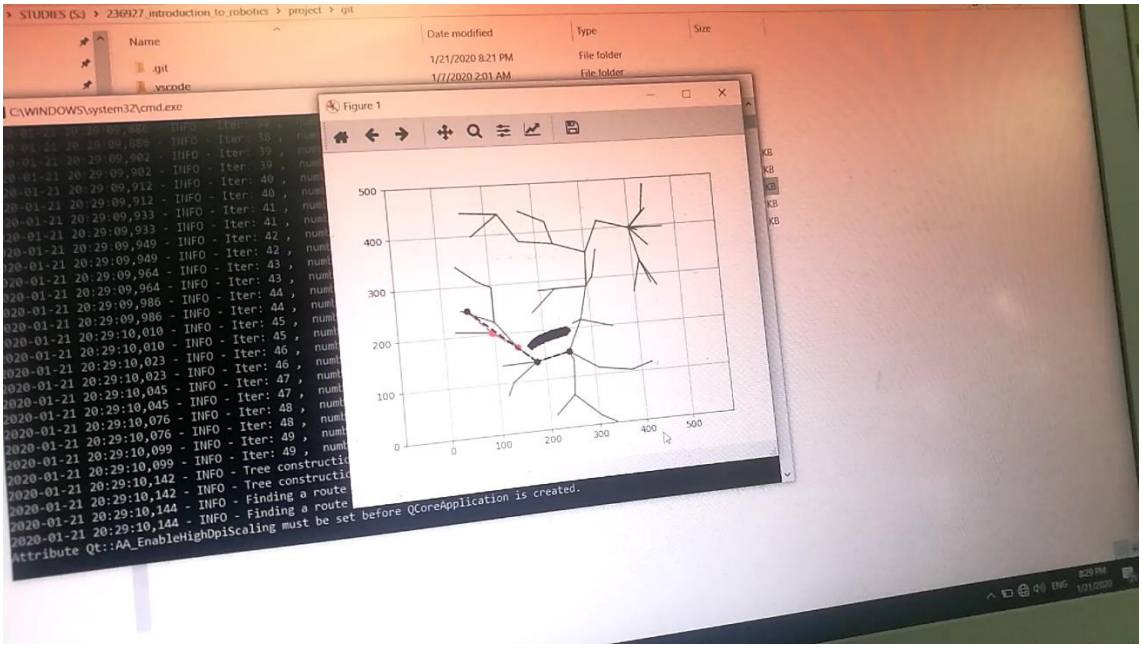


Difference: now the obstacles are the black bulky thing in the middle. Red and Black dotted lines are the result path before & after the optimization.

In general, we used 100 iterations, which would usually suffice to give an applicable path. This parameter can be increased to give potentially a better solution. A new Search Tree is generated every time the Path Planning algorithm is run.

Another example:

During one of the runs:



Future possible improvements:

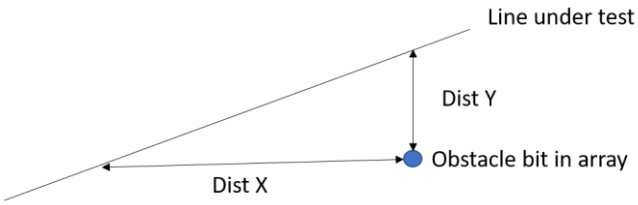
- Using a single Search Tree, updating it as new obstacles appear and as the robots moves, and using it to give an immediate new trajectory when the current trajectory is not collision-free.

f. Challenges

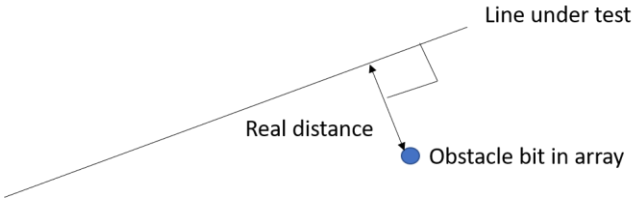
As in every project, additional challenges arose:

i. RRT* algorithm challenges

- Usually, the RRT* algorithm copes with the path finding problem by having a list of general obstacles, which are represented as circles or rectangles (as in the simulation in the Path Planning tuning section). In our case, the obstacles are presented as an array, in which cells with obstacles equal '1' value). The algorithm process includes the collision checks during various steps ('rewire', 'choose new parent', 'search for best goal node', 'path optimination'). Thus, this function is performed numerous times and needs to be optimal. 3 main veriations were tested:
    - Creating a line function for the line under test, and evaluate the perpendicular distance to the line in X and Y direction. Then take the minimum of those ditances. This does NOT return a true distance from a line, which is a perpendicular line, but gives an **approximation**. Was considered as being faster to other methods without big loss of accuracy:



    - Using the cross product. This function was tested with 2nd array (with connected obstacle point) and on the 3rd array (where the obstacles are 'inflated'):

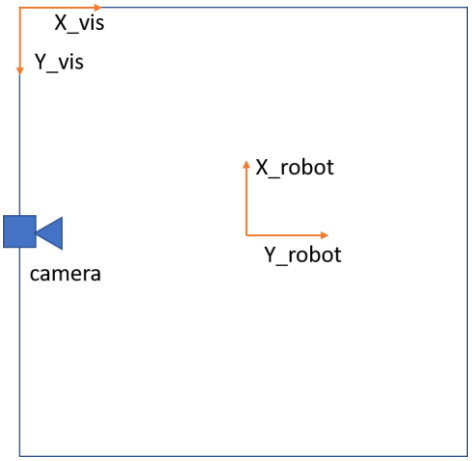


In the end, the Cross product function was used on the 2nd array with a collision margin of 3 [cm]

- A possible way to improve this algorithm in the future would be to represent the obstacles not as an array, where each bit represents a separate obstacle measurement (which is in the resolution of 1 [cm], but to create a lower resolution array, where each cell represents 10*10 [cm^2] area. If obstacle (from measurements) is spotted anywhere inside this cell, the whole cell is rendered as an obstacle. But this was not implemented.
- Other challenges may be considered as usual challenges of a creation process, including debug, logs creation etc

g. Visualization API

The second auxiliary program was mentioned in the beginning. It is worth mentioning that it uses a different coordinate system (as in usual Windows applications), and every time the data has to undergo this transformation to a different coordinate system.



It displays the following data on the map in real time and for each of the data it has a different transmission method:

1. **The data on the robot pose & current GOTO location:** is being transmitter through the localhost. The port is being opened in main program and constantly monitors for the visualization program to listen. Then, the connection is established, and the tuple of data (POS_X, POX_Y, ANGLE, CURRENT_TARGET_X, CURRENT_TARGET_Y) is being translated into a binary data and sent over the link.
2. **The 1st array obstacles data**: The data about the obstacles cannot be transmitted in the same way, since the amount of obstacles is constantly increasing. Thus, another method was chosen. The main program thread creates a binary file with new obstacles data with a certain frequency. The visualization app thread constantly reads and decodes the file, receiving the new obstacles data.
3. **The GOTO points list:** After the GOTO points list is being calculated, it is being displayed on the visualization map to constantly monitor the future robot path. Since its size isn't constant as well, it is being transmitted though a file as with 1st array of obstacles
4. **The 3rd array obstacles data:** as was demonstrated in the last image in the Collecting the obstacles data section, the 3rd array of the obstacles data can also be displayed. This is done though saving the array as an image in main program, and making it a background in the visualization app.

All those transmission functions are controlled by threads, of course.

3. Algorithm control details & debug

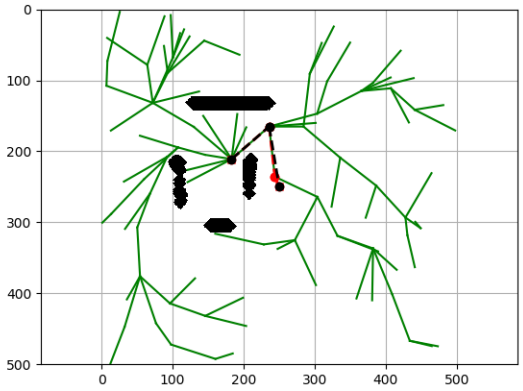
The constants which control the main functions parameters are all located in the C_CONSTANTS.py file.

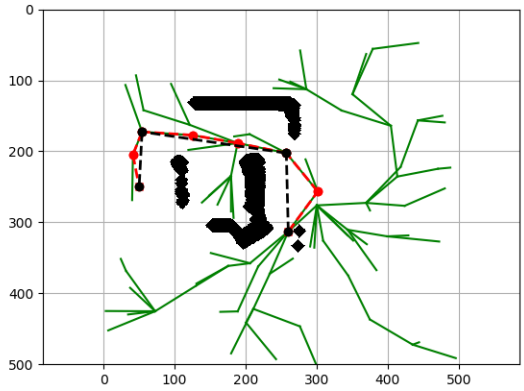
Each run creates an outputs folder, where are placed:

- RRT* algorithm path visualization – an image showing a calculated path for every time the algorithm is activated.
- Obstacles maps
- Logger for debug

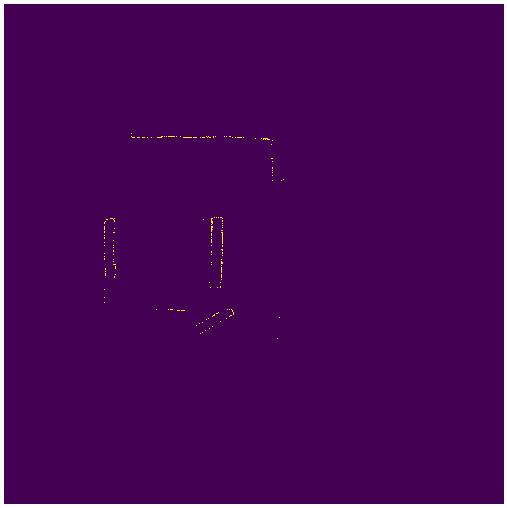
Several artifacts folders do exist for you to inspect. I will show here an example:

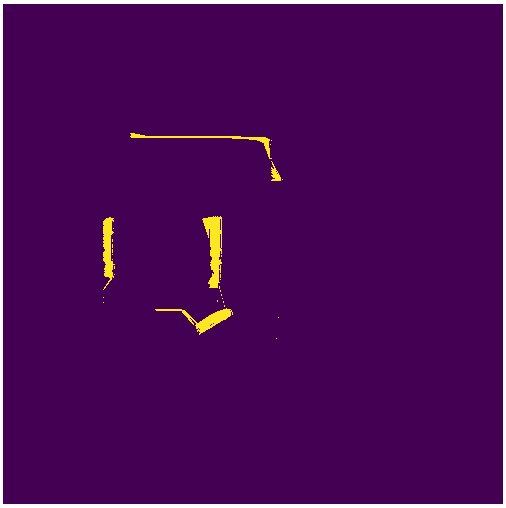
- RRT* paths (2 different runs):

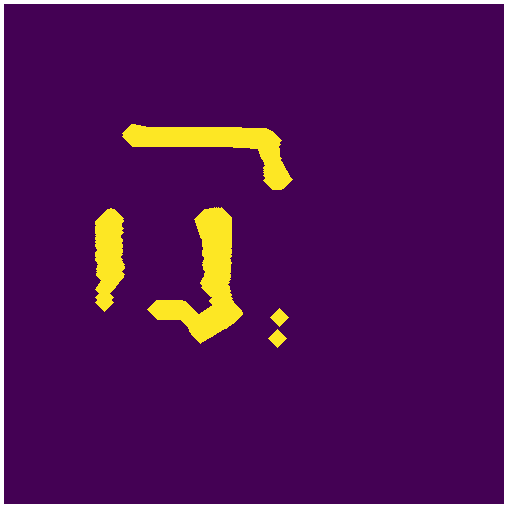




- Obstacles maps created:







The debug file shows the essential information to inspect that the program acts correctly:

Example:

```
    distance : 151.76000618792818
    angle: 123.45869151892003
03/16 10:37:58 - DEBUG - [def correct_robot_angle] Calculated:
    distance : 151.75656083820562
    angle: 123.45869151892003
03/16 10:37:58 - DEBUG - Performing turn of -110.57573625053186
03/16 10:37:59 - DEBUG - Turn performed. current angle 109.33221440016692,  discrepancy: 14.126477118753115
03/16 10:37:59 - DEBUG - Turn has too big discrepancy. Trying again
03/16 10:37:59 - DEBUG - Performing turn of 14.126477118753115
03/16 10:38:02 - DEBUG - Turn performed. current angle 125.24352229033491,  discrepancy: 1.7848307714148746
03/16 10:38:02 - DEBUG - [def drive_distance_long] NEED TO DRIVE THIS DISTANCE : 151.76000618792818
03/16 10:38:02 - DEBUG - [def drive_distance_long] max driving dist : 29.73
03/16 10:38:03 - INFO - Verifying path is clear before initializing movement
03/16 10:38:03 - INFO - Current loc: [84.2718 -126.704], dest loc : [65 -99]
03/16 10:38:03 - INFO - Path is clear: True
03/16 10:38:03 - DEBUG - [def drive_distance_short] Distance required: 29.73, Command to the motors: 1000
03/16 10:38:05 - INFO - Verifying path is clear before initializing movement
03/16 10:38:05 - INFO - Current loc: [62.0077 -95.3191], dest loc : [44 -70]
03/16 10:38:05 - INFO - Path is clear: True
03/16 10:38:05 - DEBUG - [def drive_distance_short] Distance required: 29.73, Command to the motors: 1000
03/16 10:38:07 - INFO - Verifying path is clear before initializing movement
03/16 10:38:07 - INFO - Current loc: [38.2615 -65.4287], dest loc : [19 -41]
03/16 10:38:07 - INFO - Distance from the obstacle : 0.0904755707297447 , smaller that the margin
03/16 10:38:07 - INFO - Path is clear: False
03/16 10:38:07 - INFO - Found obstacle on the way
03/16 10:38:07 - INFO - Rotating the agent to receive environment location
03/16 10:38:11 - INFO - Obstacle found on the GOTO path. Destination not reached
03/16 10:38:11 - INFO - Finding a path from [38.5445 -66.2777] to [0 0]
03/16 10:38:11 - INFO - Collision detected, initializing algorithm
03/16 10:38:23 - INFO - Tree construction completed
03/16 10:38:23 - INFO - Finding a route. Finding best node
03/16 10:38:23 - INFO - Best node found. Generating final course
03/16 10:38:23 - INFO - Route found. Optimizing
```

Similar log file exists for the visualization program.

4. Results

After numerous experiments and tuning, the program was acting in a predictable and reliable way, and could find a route in environments of different complexity, with challenging starting and ending points.