UVIC - EURECAT
CURS 2019-2020

MASTER'S DEGREE IN ROBOTICS
GROUND ROBOTS

Path Planning and Collision Avoidance Assignment

Autor

Júlia Marsal Perendreu

Supervisor

Carlos Rizzo

May 31, 2020

Contents

1 Exercise 1: dijkstra Global Planner **2**
- 1.1 Run the code: . . . 2
- 1.2 Playing with the code the code: . . . 2
- 1.3 Understanding the code: . . . 2
  - 1.3.1 Run a_star.py and repeat exercises 1.1 and 1.2. Compare it with Dijkstra. Is it possible to easily convert the a_start script to dijkstra? Implement it . . . 3
  - 1.3.2 Compare it with Dijkstra. Is it possible to easily convert the a_start script to dijkstra? . . . 3

2 Exercise 2: dynamic window approach local planner **5**
- 2.1 Run the code: . . . 5
- 2.2 Playing with the code: . . . 5
- 2.3 Understanding the code: . . . 6
  - 2.3.1 Locate the objective function in the code. Is it the same as in the original paper? . . . 7
- 2.4 Read the ROS code of the dynamic window approach at ros-planning . . . 8

3 Global + local planner integration **8**
- 3.1 Before integrating the DWA planner, create/modify an environment in dijkstra.py: . . . 8
- 3.2 Calculate the global path . . . 9
  - 3.2.1 Integrate the DWA . . . 9

4 Repository **12**

5 Images **12**

1 Exercise 1: dijkstra Global Planner

1.1 Run the code:

The output of the code can be seen in image 3

1.2 Playing with the code the code:

- Locate start and goal. Change them. Run the script: the start is changed to (5,30) and goal to (50,10), the result is shown in image 4
- Locate the grid size. Change it. Run the script: the grid size is changed to 0.5. The result is shown in image 5
- Locate the robot size. Change it. Run the script: the robot size changed to 5.0. The result is shown in image 6
- Extract the path. E.g. Print the path in the screen (x-y positions). The result is shown in images 6 and 7
- Locate the obstacles. Add some obstacles inside the map, at coordinates previously given. The result is shown in image 8. If this image is compared with image 6 it can be seen that the path changed a lot.

1.3 Understanding the code:

Explain with your own words the algorithm in the code. The algorithm code is based on the **dijkstra_planning** function, this makes the following:

- 0. It obtains the unitarian *nodes* of start and goal positions. A node contains its position (x,y,cost(default:0.0),pind(default:-1)).
- 1. An obstacle map is created: based on a Boolean matrix, (T: means obstacle, F: means not an obstacle).
- 2. Obtains the motion model.
- 3. Adds the first node to the *openset*. This contains all the untreated computed positions.
- 4.1 The minimum cost position is located and added as the current position
- 4.2 It plots the current position.
- 4.3 If the current position is the goal position, the code will be finished (and will proceed with step 6).
- 4.4 The current position is deleted from the *openset* dictionary, and the current point is added to the processed position dictionary *closedset*).
- 5. For each position prevoiusly defined in motion algorithm:

- 5.1 Computes the next point as (current+motion_position).
- 5.2 Verifies that the next point is inside of region where we are working. If not, goes to 4.1.
- 5.3 Verifies if the next point is previously processed. If its, goes to 4.1
- 5.4 If the point is not previously processed (exists in *openset* dictionary) and if its cost is higher than the existing point in *openset*, it uploads its cost. If not the point is added to the *openset* dictionary.
- 6. It computes the path points, from the start to goal.

1.3.1 Run a_star.py and repeat exercises 1.1 and 1.2. Compare it with Dijkstra. Is it possible to easily convert the a_start script to dijkstra? Implement it

- Run the code *python a_ start.py*. The result is shown in image 9.
- Playing with the code:
  - Locate the start and goal. Change them. Run the script. Start is changed to (5,5) and goal to (50,40) m, the result is shown in image 10.
  - Locate the grid size. Change it. Run the script. The grid size has been changed to 0.5 m. The result is shown in image 11
  - Locate the robot size. Change it. Run the script. The robot size has been changed to 3.0 m. The result is shown in image 12
  - Extract the path. The path is shown in image 13
  - Locate the obstacles. Add some obstacles inside the map, at the given coordinates.If image 14 is compared with image 12, image without obstacles, it can be seen that the path changed.

1.3.2 Compare it with Dijkstra. Is it possible to easily convert the a_start script to dijkstra?

The code is almost the same. It only adds to the existing cost an heuristic parameter that depends on the goal node and the current processing position. This could easily convert to dijsktra, changing the weight parameter to 0.0.

- If the weight is set to 1.0, image 15 shows the output path that results.
- If the weight is set to 3.0, image 16 shows the output path that results.

– If the weight is set to 0.0, image 17 shows the output path that results.If we compare this with image 14, it can be seen that the result is the same. So, when the weight is 0.0, a_start can be easily convert to *Dijkstra*.

2 Exercise 2: dynamic window approach local planner

2.1 Run the code:

If the code is ran, we obtain what is shown in the image 18.

2.2 Playing with the code:

- Locate start and goal. Change them. Run the script.As it can be seen in image 19 The start position is changed to position (-1.0,1.0) and the goal position is changed to (11,9)
- If more obstacles are added, the path obtained is which shows image 20
- Change the DWA parameters. Vary one parameter at a time.
  - *max_ speed*: If the max speed is changed to 5.0 m/s the local path planer moves faster when the path is evicted. This velocity was too high that the robot destabilized and not reach the goal.It can be seen in image 21.
  - *min_ speed*: If the min speed is changed to -0.1 m/s the local path planes moves faster in overall execution. The final path can be seen in image 22.
  - *max_ yawrate*: It's the angular velocity around its vertical axis. If the max yawrate is changed to 60.0 degrees/s the local path planner moves faster when there exist a rotation component. The final path can be seen in image 23.
  - *max_ accel*: If the linear maximum acceleration is changed to 1 m/s^2. The planner moves slower when the maximum acceleration increases. The final path can be seen in image 24
  - *max_ dyawrate*: If the maximum angular acceleration around the vertical axis increases to 60 degrees/s^2, the local path planner moves faster when it has a rotational component and, its local trajectory spins with higher curvature. This means that, this growth of the curvature can make that maybe the local path planner cannot reach the desired target, as it can be seen in image 25.
  - *v_ reso*: If the resolution velocity is increased, the trajectories evaluates faster, but, the local planner cannot find a local plan to move to the target. The result can be seen in image 26
  - *yawrate_ reso*: If the yaw rate resolution velocity is increased to 0.5 degree/s,the local planner goes faster and can find a local plan to move to the goal. The result can be seen in image 27
  - *dt*: If the time differential is increased to 0.5 s, the local planner goes faster and reaches the target faster. The result can be seen in image 28

- *predict_ time*: If the predict time decreases to 1s, the local planner trajectory decreases and the local planner does not plan well. As it can be seen in image 29. If the predict time increases the predic time takes more time to predict, so it goes slower.
- *to_ goal_ cost_ gain*: To goal cost gain, if the gain is increased to 1.5, the target position reached is closer to the desired target position. As i becomes smaller, less cost are computed and less paths are planned.Sometimes, it cannot reach the target position as it can be seen in image 30.
- *speed_ cost_ gain*: If the speed cost gain is changed to 1.2, as closer is to the goal it goes faster.
- *robot_ radius*: If the robot radius decreases to 0.5 m, the robot reaches a closer position of the target,as it can be seen in image 31.

2.3 Understanding the code:

Explain with your own words the algorithm in the code. The aim of this algorithm is the generation of a dwa local planner.

- The main function *dwa_ control* needs the following parameters:
  - x: initial state - [x(m),y(m),yaw(rad),v(m/s),omega(rad/s)]
  - u: [v,y] minimum velocity and yaw
  - config: all the changeable parameters
  - goal: the desired goal
  - ob: obstacles previously defined
- First, the function *calc_ dynamic_ window(x,config)* obtains the dynamic window. First, it computes the *Vs*: the dynamic window from robot specification and the *Vd*, the dynamic window from motion model. Then, the dynamic window *dw: [vmin,vmax,yawratemin,yawratemax]* is obtained from the previous Vs and the Vd.
- Second, the function *calc_final_ input* does the following:
  - 0. For each velocity and yaw from the dynamic window obtained previously, it computes a trajectory:
    - 0. While the time is less than the *predict_ time parameter*, it computes the motion for its states and its saved to a trajectory array. The motion equations are:
      - linear velocity v: u[0]
      - angular velocity w: u[1]
      - yaw: $x[2] = x[2] + u[1] * dt$

- x: $x[0] = x[0] + u[0] * cos(x[2]) * dt$
- y: $x[1] = x[1] + u[0] * sin(x[2]) * dt$
- v: $x[3] = x[3] + u[0]$
- w: $x[4] = x[4] + u[1]$

* 1. It computes the *to_ goal_ cost* the cost needed to reach the goal, using the previous obtained trajectory. To obtain this, it needs to compute the *Error*, or *angle* between the Goal and the trajectory previously computed.
  - Goal Magnitude: $\sqrt{goal[0]^2 + goal[1]^2}$
  - Trajectory Magnitude: $\sqrt{traj[-1,0]^2 + traj[-1,1]^2}$
  - Dot product = goal*traj
  - Error = Dot product / Goal Magnitude * Trajectory Magnitude
  - Error_angle = acos(Error)
  - Cost = Error_angle**to_ goal_ cost_ gain parameter*
* 2. Then, it computes the *speed cost*, as the cost associated with the robot speed, as $speed\_cost = speed\_cost\_gain\_parameter * (max\_speed\_parameter - trajectory)$
* 3. Then, it computes the *obstacle cost*. For each obstacle declared previously, and for each trajectory previously computed, it computes the differential x and y (trajectory_x/y - obstacle_x/y) and then it obtains the distance ($r = \sqrt{dx^2 + dy^2}$). If this distance r is less or equal than the parameter *robot_ radius* then, the cost is infinite and there is going to be a collision. If not, the cost will be 1/(distance r).
* 4. It computes the final cost as the sum of the *to goal cost*, *the speed cost* and the *obstacle cost*.
* 5. It searches the minimum trajectory: if the *minimum cost* computed until that moment is higher than the *final cost* computed in step 4, this *minimum cost* will be uploaded to this *final cost*, and the *(linear, angular) velocities*, and the best trajectory will be the associated to this final cost.

- Then, the *linear,angular* velocities are used to obtain the next state x using the motion equations described in 0 and stored into the state history.

- Finally it shows the computed tarjectory in green, the computed state x in red, the obstacles in black, and the arrow with (x,y,yaw).

2.3.1 Locate the objective function in the code. Is it the same as in the original paper?

- The objective function is defined in the step 4. If we compare the original in the paper: $\sigma * (\alpha * heading(v,w) + \beta * dist(v,w) + \gamma * velocity(v,w))$

can be compared with $1*(to\_goal\_cost+obstacle\_cost+speed\_cost) \rightarrow 1*(to\_goal\_cost\_gain*error\_angle+1*obstacle\_cost+speed\_cost\_gain*velocity)$. Is it the same as in the origina paper.

2.4 Read the ROS code of the dynamic window approach at ros-planning

Try to undersand its implementation. Well, it is a dwa local planner implemented in cpp. It has the following functions:

- *Reconfigure*: reconfigure all the parametres needed and vx,vy and vth(yaw).
- *DWAPlanner*: calls the planner, the obstacle_costs, the path_costs,the goal_costs, the goal_front_costs and the alignment_costs. It initializes the ros node and search the *controller_frecuency* parameter. It also computes the simupate period, resets the oscillation costs,initializes the trajectory cloud and sets up all the cost functions that will be applied in order. Then it generate a trajectory and obtains its sample score.
- *GetCellCosts*: it implements the objective function, it sums all the costs (path_distance*path_cost + goal_distance_bias*goal_cost + occdist_scale_*occ_cost) in order to obtain the total cost. (Makes the same as step 4 of the python script).
- *setPlan*: from the original global plan, it sets the plan.
- *checkTrajectory*: this function is used when other strategies are to be applied, but the cost functions for obstacles are to be reused. It checks if the trajectory is a legal one.
- *updatePlanAndLocalCosts*: it updates the obstacle_costs, the path_costs, the global_costs, and make alignment costs with the goal pose. The robot nose wanted to be drawn to its final position. So, it computes the angle to goal in order to obtain the front_global_plan and the goal front costs. (Makes almost the same as the step 1 of the python script).
- *findBestPath*: given the current state of the robot (*previous x*),it finds a good trajectory.

3 Global + local planner integration

The goal of this exercise is to integrate both algorithms into one. It is recommended to integrate dynamic_window_approach into dijkstra.

3.1 Before integrating the DWA planner, create/modify an environment in dijkstra.py:

- Create/modify a map of 12 x 12

- The start position is (1,1)
- The goal position is (8,8)
- Only one obstacle (besides the walls) will be visible to dijkstra. Locate it at position (5,5), as shown in the following figure.
- More unmapped goals will be added later

The output image it can be seen in image 32 and the path's coordinates can be seen in image 33.

3.2 Calculate the global path

As it can be seen in image 32 the global path has been well implemented.

3.2.1 Integrate the DWA

Now, to integrate the DWA with the calculated path, the strategy is not to provide the final goal (8,8), but a series of sub-goals: the path calculated by dijkstra. You will need to implement a sub-goal manager in a form of a loop. This loop will assign the coordinates of the dijkstra path, one by one, to the DWA function. Assign one, and once the robot has reached that sub-goal, assign the next one... until the goal is reached. You can play with the trajectory resolution (give the DWA only some sub-goals, not all of them). You can also play with the DWA parameters (e.g. change the simulation time). At last, implement a variable that controls to follow or not the path (e.g. bool follow_dijkstra). If true, it should follow the path. If false, the loop should provide the algorithm only the final goal. See the desired behavior in the attached videos.

The DWA has been implemented with a sub-goal manager in a form of a loop. The new code implemented merged with dijkstra code is the following:

```
a = len(rx)
    x = np.array([1.0, 1.0, math.pi / 8.0, 0.0, 0.0])
    i= 0
    ob = np.array([[4, 2],
                   [5, 4],
                   [5.0, 5.0],
                   [5.0, 6.0],
                   [5.0, 7.0],
                   [5.0, 8.0],
                   [5.0, 9.0]
                   ])
    u = np.array([0.0, 0.0])
    config = Config()
    traj = np.array(x)
    if show_animation:
```

```
            plt.plot(ox, oy, ".k")
            plt.plot(sx, sy, "xr")
            plt.plot(gx, gy, "xb")
            plt.grid(True)
            plt.axis("equal")
    while i < a:
            goal = np.array([rx[i], ry[i]])
            print("goal",goal)
            print("i",i)
            goal_reached = False
            follow_dijkstra = False
            while(not goal_reached):
                    u, ltraj = dwa_control(x, u, config, goal, ob)
                    x = motion(x, u, config.dt)
                    traj = np.vstack((traj, x))  # store state history
                    for o in ob:
                            if (o[0] == goal[0]) and (o[1] == goal[1]):
                                    print(goal)
                                    goal_reached = True
                    if show_animation:
                            plt.cla()
                            plt.plot(ltraj[:, 0], ltraj[:, 1], "-g")
                            plt.plot(x[0], x[1], "xr")
                            plt.plot(ob[:, 0], ob[:, 1], "om")
                            plt.plot(goal[0], goal[1], "xb")
                            plot_arrow(x[0], x[1], x[2])
                            plt.plot(rx, ry, "-r")
                            plt.axis("equal")
                            plt.grid(True)
                            plt.pause(0.0001)
                    if show_animation:
                            plt.plot(ox, oy, ".k")
                            plt.plot(rx, ry, "-r")
                    if follow_dijkstra:
                    if math.sqrt((x[0] -  goal[0])**2 +
                     (x[1] - goal[1])**2) <= config.robot_radius:
                            print("goal[0]",goal[0])
                            print("goal[1]",goal[1])
                            print("Goal!!")
                            goal_reached = True
                    else:
                            if math.sqrt((x[0] - rx[-1])**2 +
                            (x[1] - ry[-1])**2) <= config.robot_radius:
                                    print("rx-1",rx[-1])
                                    print("ry-1",ry[-1])
                                    print("Goal!!")
```

```
                                                    break
        i+=1
        goal_reached = False
        print("Done")
        if show_animation:
                plt.plot(traj[:, 0], traj[:, 1], "-c")
                plt.pause(0.0001)
        if show_animation:
                plt.plot(rx, ry, "-r")
        plt.show()
```

As it can be seen in image 1 it shows the local and global path to reach the goal not following the path and image 2 shows the local and global path following the path.

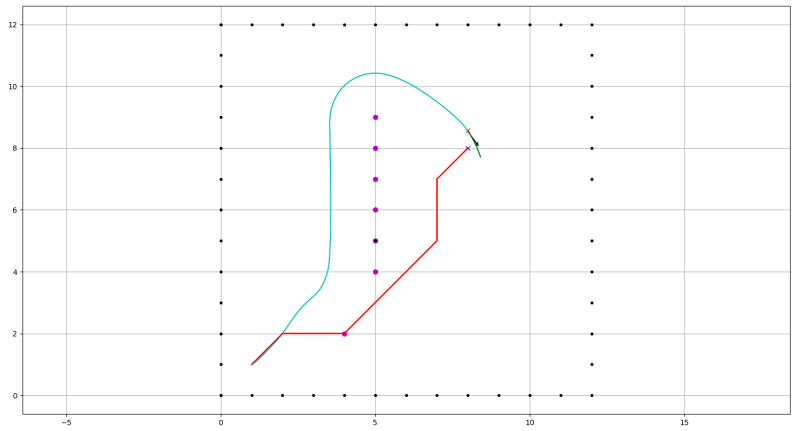

Figure (1) follow_dijkstra = false

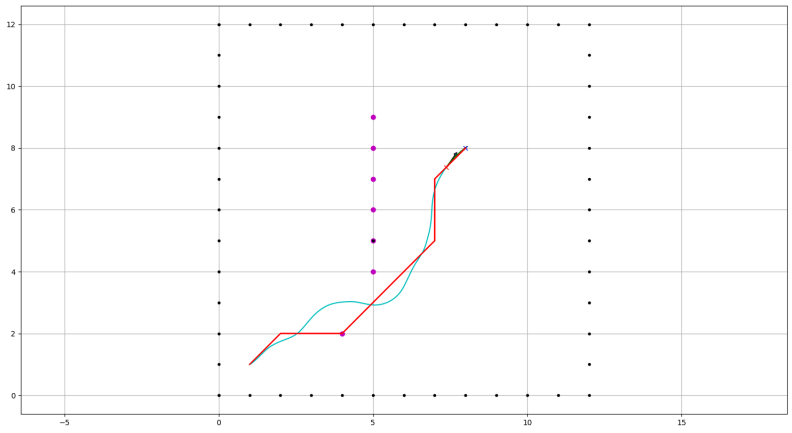

Figure (2) follow_dijkstra = true

4 Repository

All the practice information can be found in PathPlanningAssignmentJMA

5 Images

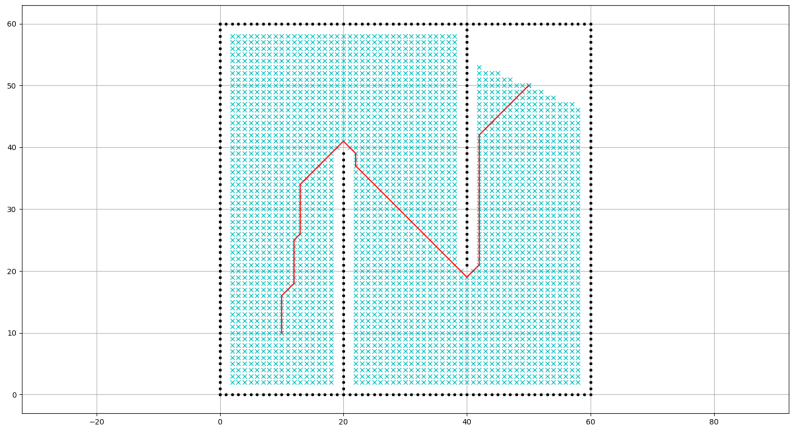

Figure (3) Result of dijkstra.py

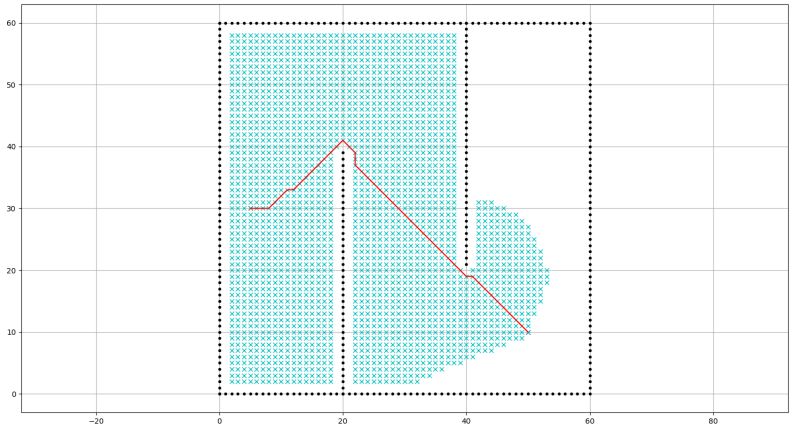

Figure (4) Start: (5,30) Goal: (50,10)

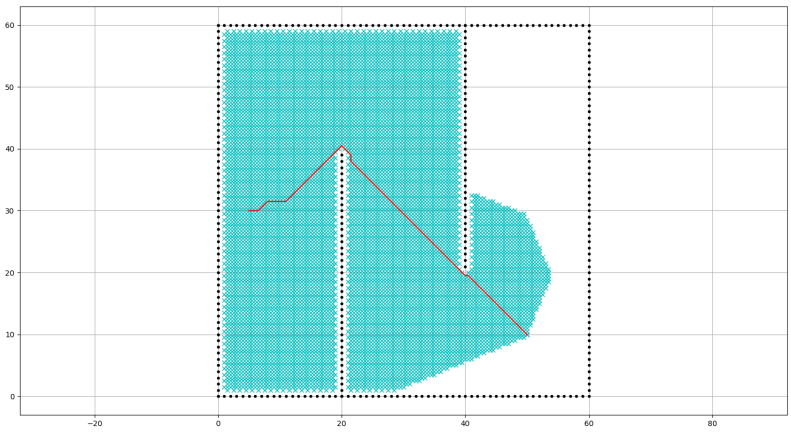

Figure (5) Grid size of 0.5 m

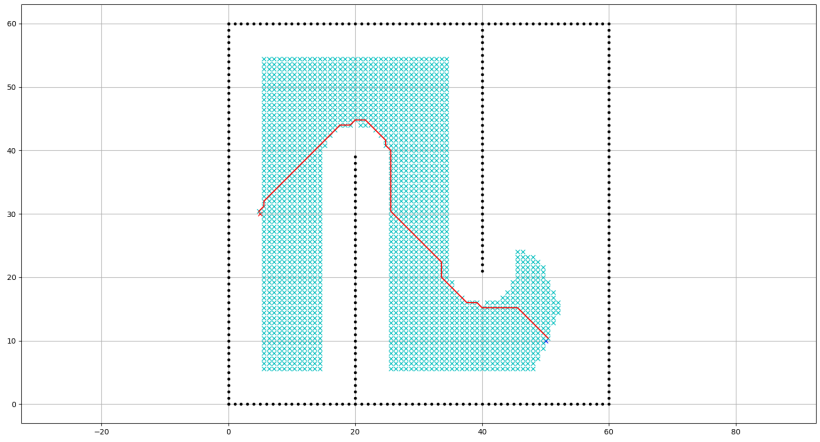

Figure (6) Robot size of 5.0 m

```
dijkstra.py start!!
Find goal
The path is:
('rx', [50.400000000000006, 49.6, 48.800000000000004, 48.0, 47.2, 46.400000000000006, 45.6, 44.800000000000004, 44.0, 43.2, 42.40000000
0000006, 41.6, 40.800000000000004, 40.0, 39.2, 38.400000000000006, 37.6, 36.800000000000004, 36.0, 35.2, 34.4, 33.6, 33.6, 33.6, 33.6,
32.800000000000004, 32.0, 31.200000000000003, 30.400000000000002, 29.6, 28.8, 28.0, 27.200000000000003, 26.400000000000002, 25.6, 25.6,
 25.6, 25.6, 25.6, 25.6, 25.6, 25.6, 25.6, 25.6, 25.6, 25.6, 24.8, 24.8, 24.0, 23.200000000000003, 22.400000000000002, 21.6, 20.8
, 20.0, 19.200000000000003, 18.400000000000002, 17.6, 16.8, 16.0, 15.200000000000001, 14.4, 13.600000000000001, 12.8, 12.0, 11.20000000
0000001, 10.4, 9.600000000000001, 8.8, 8.0, 7.2, 6.4, 5.6000000000000005, 5.600000000000005, 4.800000000000001])
('ry', [10.4, 11.200000000000001, 12.0, 12.8, 13.600000000000001, 14.4, 15.200000000000001, 15.200000000000001, 15.200000000000001, 15.
200000000000001, 15.200000000000001, 15.200000000000001, 15.200000000000001, 15.200000000000001, 15.200000000000001, 16.0, 16.0, 16.0, 16.8, 17.6, 18.40000
0000000002, 19.200000000000003, 20.0, 20.8, 21.6, 22.400000000000002, 23.200000000000003, 24.0, 24.8, 25.6, 26.400000000000002, 27.2000
00000000003, 28.0, 28.8, 29.6, 30.400000000000002, 31.200000000000003, 32.0, 32.800000000000004, 33.6, 34.4, 35.2, 36.0, 36.80000000000
0004, 37.6, 38.400000000000006, 39.2, 40.0, 40.800000000000004, 41.6, 42.400000000000006, 43.2, 44.0, 44.800000000000004, 44.8000000000
00004, 44.800000000000004, 44.0, 44.0, 44.0, 43.2, 42.400000000000006, 41.6, 40.800000000000004, 40.0, 39.2, 38.400000000000006, 37.6,
36.800000000000004, 36.0, 35.2, 34.4, 33.6, 32.800000000000004, 32.0, 31.200000000000003, 30.400000000000002])
```

Figure (7) X-Y path printed

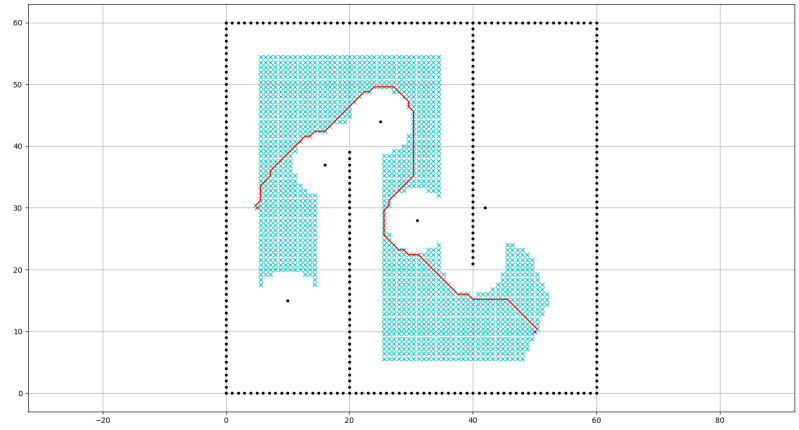

Figure (8) Adding obstacles inside map

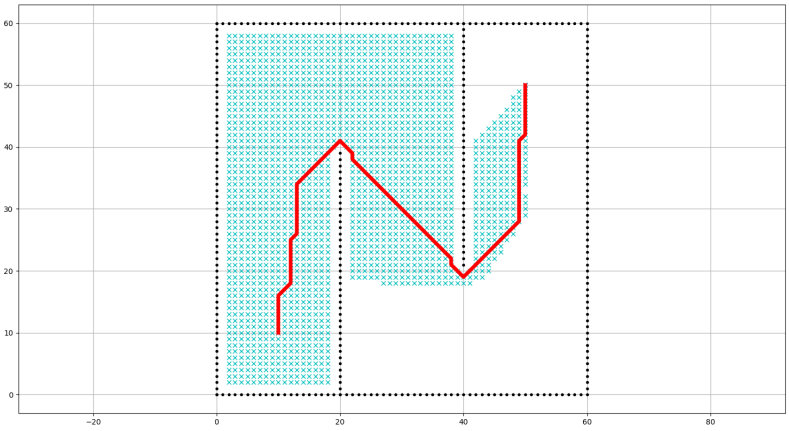

Figure (9) A_star algorithm execution

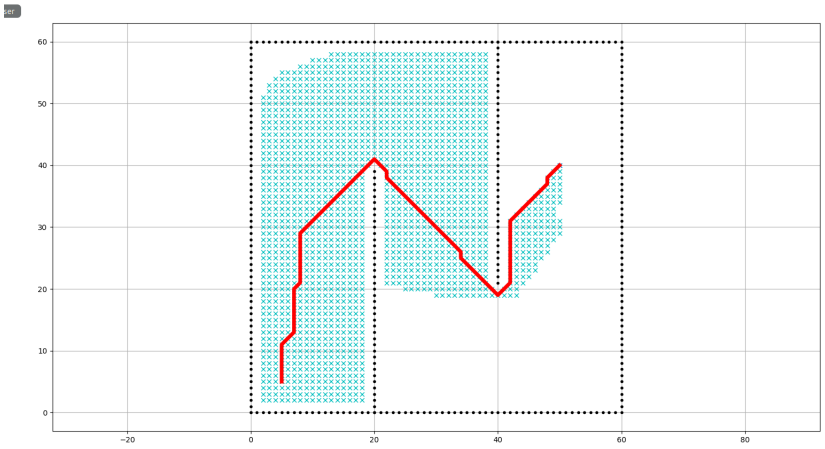

Figure (10) Start changed to (5,5) and goal to (50,40)

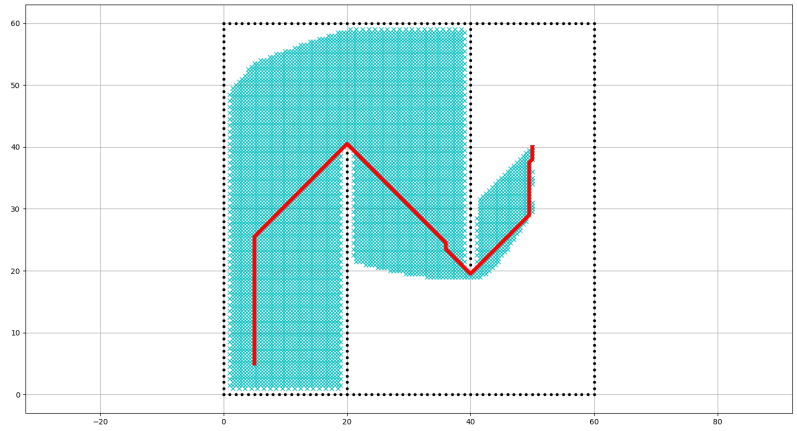

Figure (11) Grid size is 0.5 m

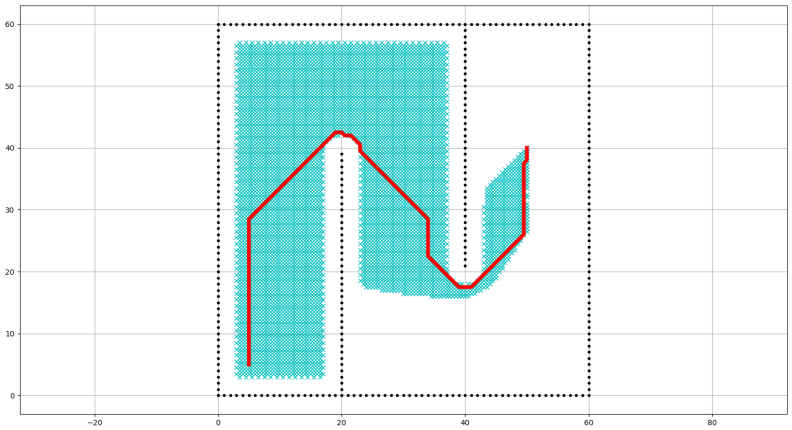

Figure (12) Robot size 3.0 m

```
a_star.py start!!
Find goal
Path:
('rx', [50.0, 50.0, 50.0, 50.0, 50.0, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 4
9.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.0, 48.5, 48.0, 47.5, 47.0, 46.5, 46.0, 45.5, 45.0, 44.5, 44.0, 43.5, 43.0, 42.5, 42.0
, 41.5, 41.0, 40.5, 40.0, 39.5, 39.0, 38.5, 38.0, 37.5, 37.0, 36.5, 36.0, 35.5, 35.0, 34.5, 34.0, 34.0, 34.0, 34.0, 34.0, 34.0, 34.0, 3
4.0, 34.0, 34.0, 34.0, 34.0, 34.0, 34.0, 33.5, 33.0, 32.5, 32.0, 31.5, 31.0, 30.5, 30.0, 29.5, 29.0, 28.5, 28.0, 27.5, 27.0, 26.5, 26.0, 25.5
, 25.0, 24.5, 24.0, 23.5, 23.0, 23.0, 23.0, 22.5, 22.0, 21.5, 21.0, 20.5, 20.0, 19.5, 19.0, 18.5, 18.0, 17.5, 17.0, 16.5, 16.0, 15.5, 1
5.0, 14.5, 14.0, 13.5, 13.0, 12.5, 12.0, 11.5, 11.0, 10.5, 10.0, 9.5, 9.0, 8.5, 8.0, 7.5, 7.0, 6.5, 6.0, 5.5, 5.0, 5.0, 5.0, 5.0, 5.0,
5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0,
5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0])
('ry', [40.0, 39.5, 39.0, 38.5, 38.0, 37.5, 37.0, 36.5, 36.0, 35.5, 35.0, 34.5, 34.0, 33.5, 33.0, 32.5, 32.0, 31.5, 31.0, 30.5, 30.0, 2
9.5, 29.0, 28.5, 28.0, 27.5, 27.0, 26.5, 26.0, 25.5, 25.0, 24.5, 24.0, 23.5, 23.0, 22.5, 22.0, 21.5, 21.0, 20.5, 20.0, 19.5, 19.0, 18.5
, 18.0, 17.5, 17.5, 17.5, 17.5, 17.5, 18.0, 18.5, 19.0, 19.5, 20.0, 20.5, 21.0, 21.5, 22.0, 22.5, 23.0, 23.5, 24.0, 24.5, 25.0, 25.5, 2
6.0, 26.5, 27.0, 27.5, 28.0, 28.5, 29.0, 29.5, 30.0, 30.5, 31.0, 31.5, 32.0, 32.5, 33.0, 33.5, 34.0, 34.5, 35.0, 35.5, 36.0, 36.5, 37.0
, 37.5, 38.0, 38.5, 39.0, 39.5, 40.0, 40.5, 41.0, 41.5, 42.0, 42.0, 42.0, 42.5, 42.5, 42.5, 42.0, 41.5, 41.0, 40.5, 40.0, 39.5, 39.0, 3
8.5, 38.0, 37.5, 37.0, 36.5, 36.0, 35.5, 35.0, 34.5, 34.0, 33.5, 33.0, 32.5, 32.0, 31.5, 31.0, 30.5, 30.0, 29.5, 29.0, 28.5, 28.0, 27.5
, 27.0, 26.5, 26.0, 25.5, 25.0, 24.5, 24.0, 23.5, 23.0, 22.5, 22.0, 21.5, 21.0, 20.5, 20.0, 19.5, 19.0, 18.5, 18.0, 17.5, 17.0, 16.5, 1
6.0, 15.5, 15.0, 14.5, 14.0, 13.5, 13.0, 12.5, 12.0, 11.5, 11.0, 10.5, 10.0, 9.5, 9.0, 8.5, 8.0, 7.5, 7.0, 6.5, 6.0, 5.5, 5.0])
```

Figure (13) X Y printed path points

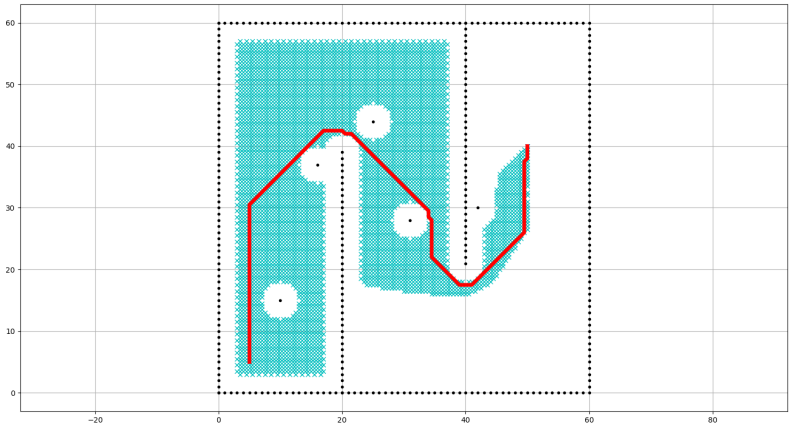

Figure (14) Adding obstacle avoidance

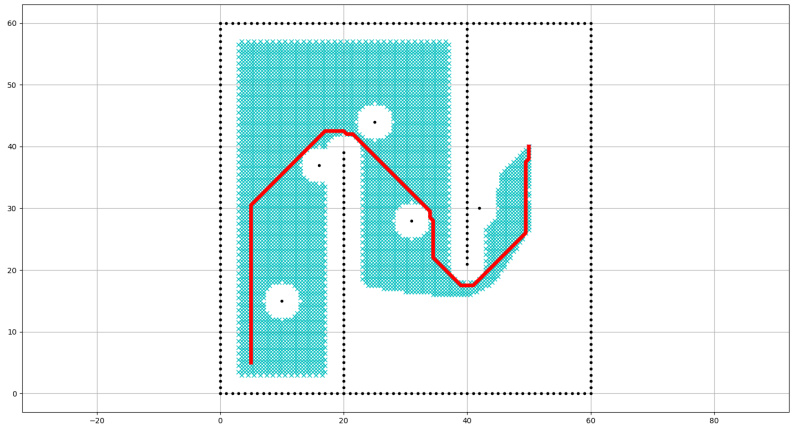

Figure (15) A_star with weight 1.0

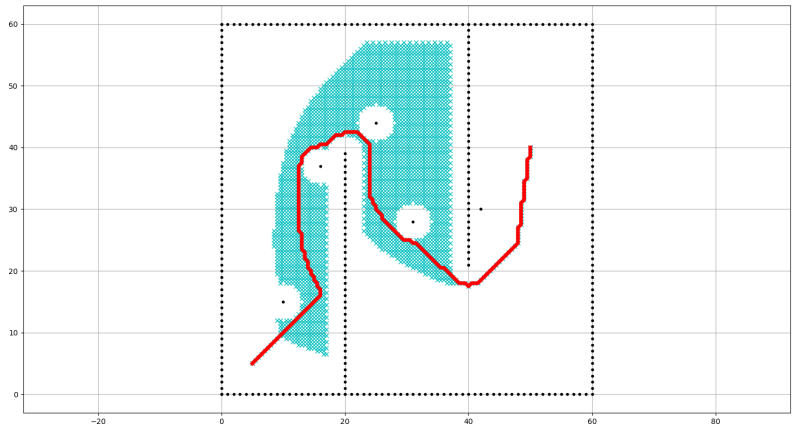

Figure (16) A_star with weight 3.0

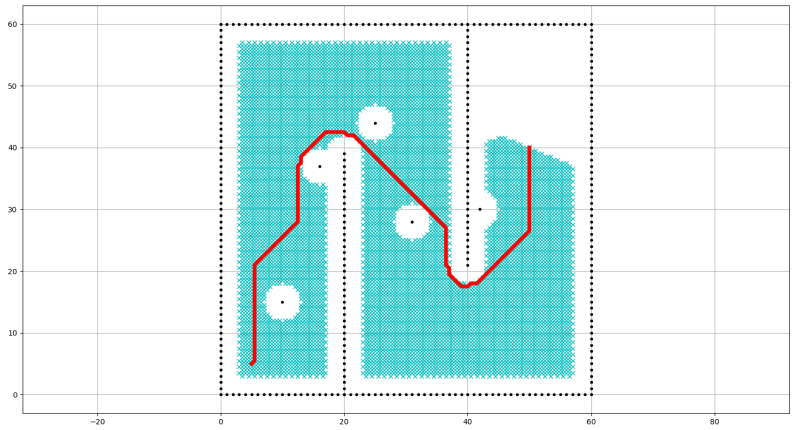

Figure (17) A_star with weight 0.0

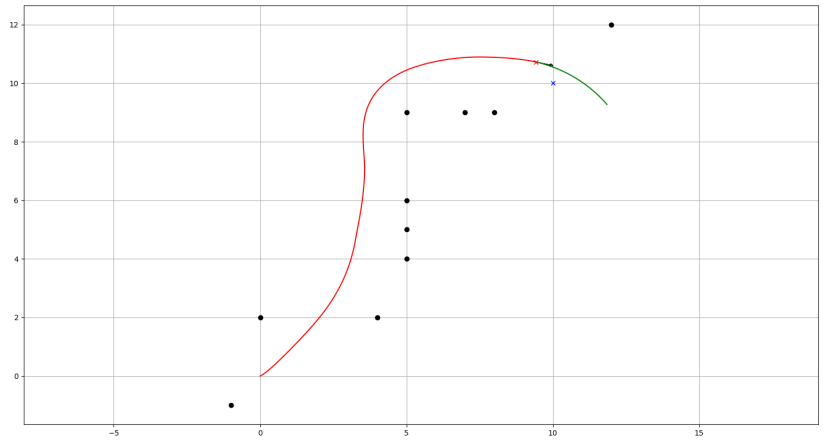

Figure (18) Local Planner output

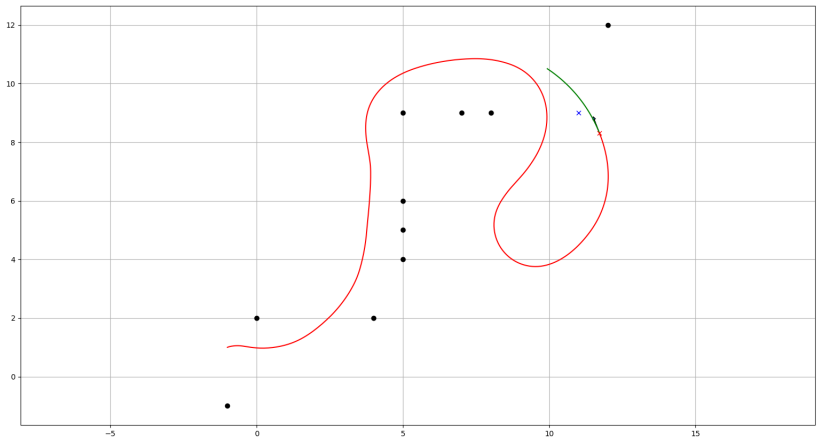

Figure (19) Start position is:(-1,1), goal position is (11,9)

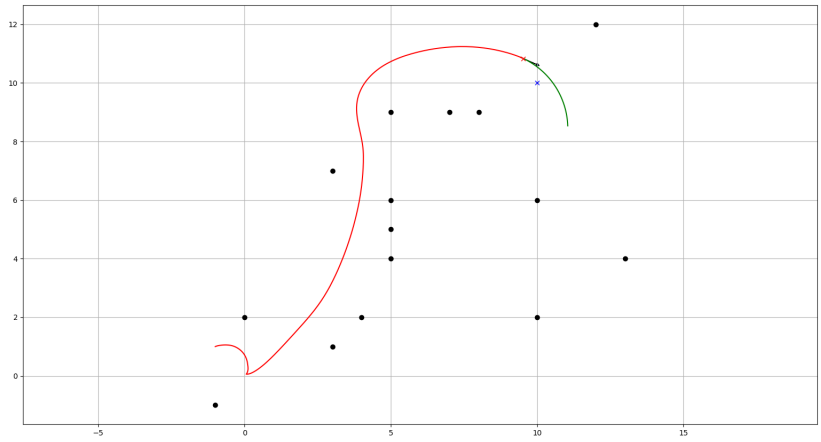

Figure (20) Local Planner with more obstacles

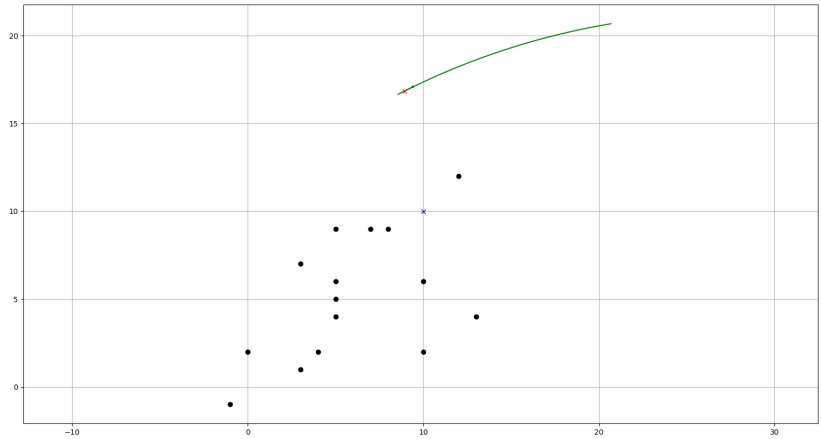

Figure (21) Max speed fixed to 5.0 m/s

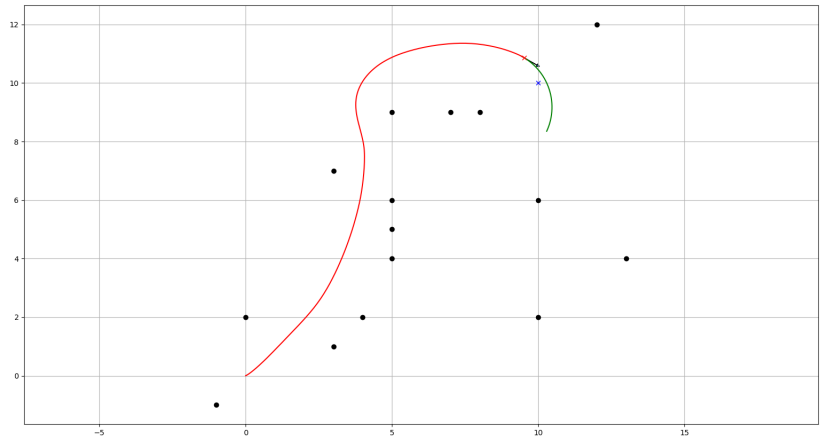

Figure (22) Min speed fixed to -0.1 m/s

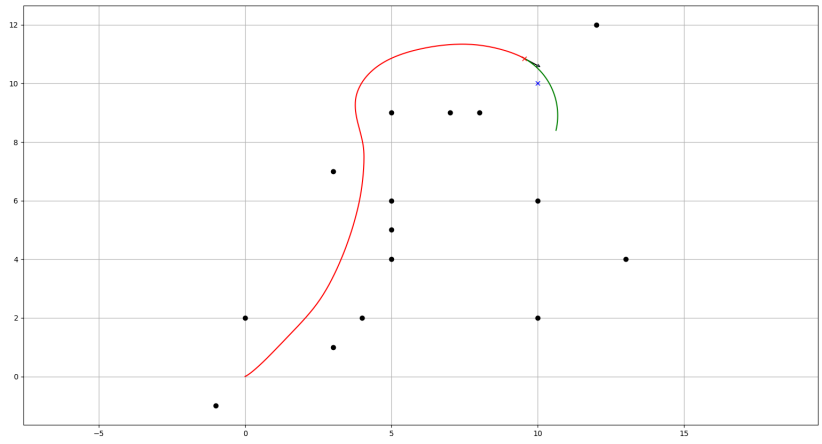

Figure (23) Max yawrate changed to 60 degrees/s

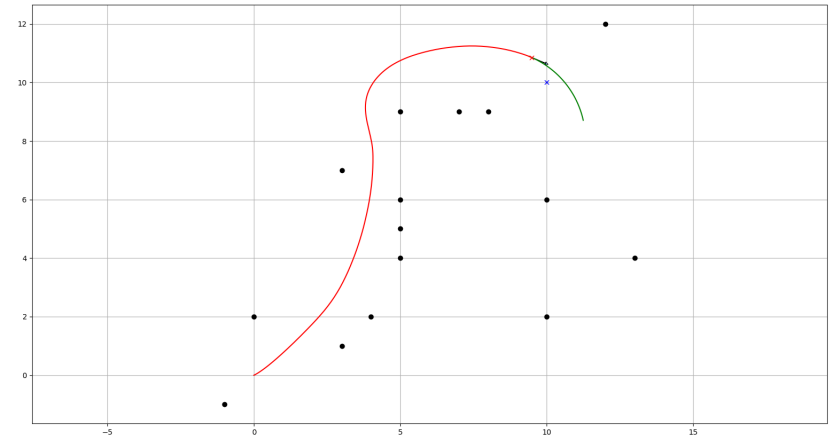

Figure (24) Max acceleration changed to 1 m/s^2

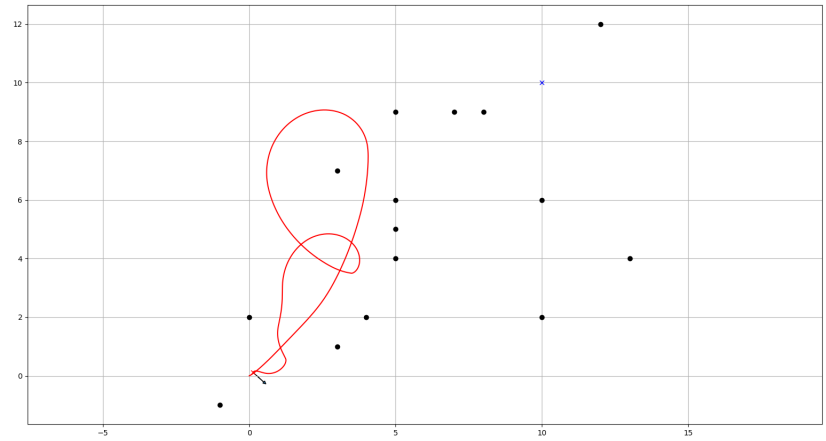

Figure (25) Max dyawrate changed to 60.0 degrees/s^2

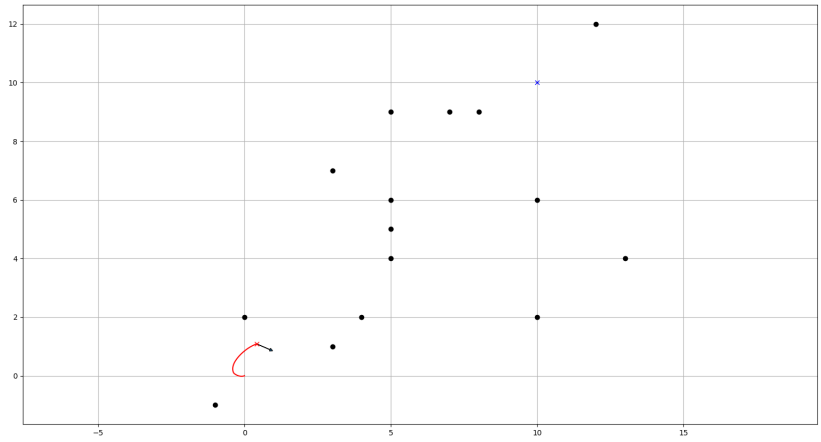

Figure (26) Resolution velocity changed to 0.03 m/s

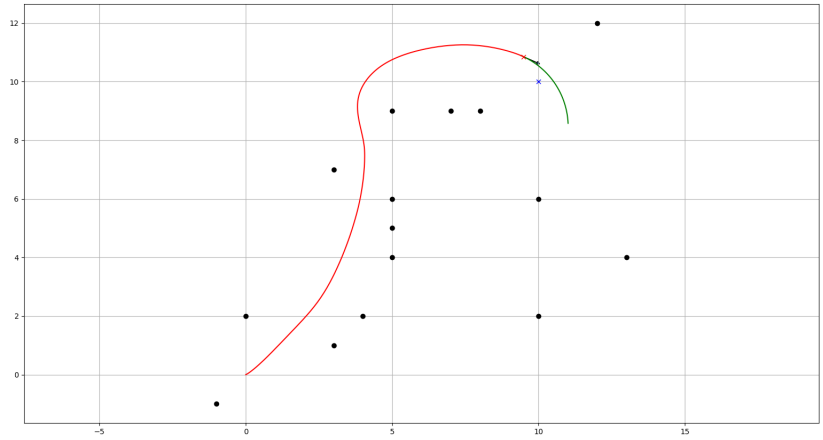

Figure (27) Yaw rate resolution velocity changed to 0.5 degrees/s

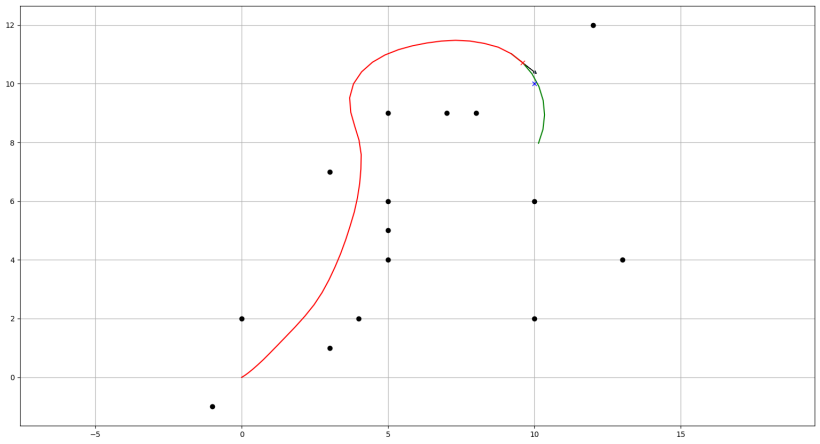

Figure (28) Time differential changed to 0.5 s

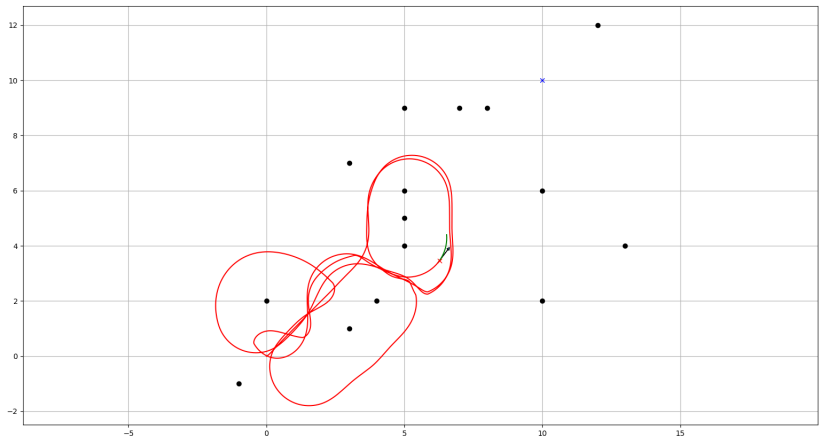

Figure (29) Predict time decreased to 3.5s

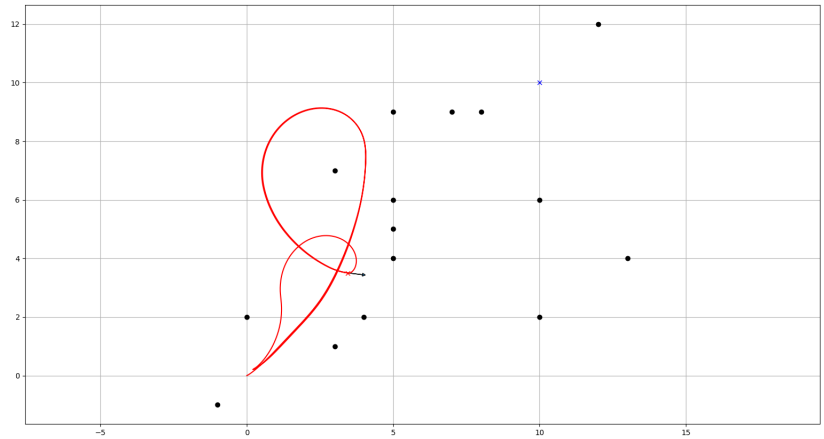

Figure (30) To goal cost gain is decreased to 0.8

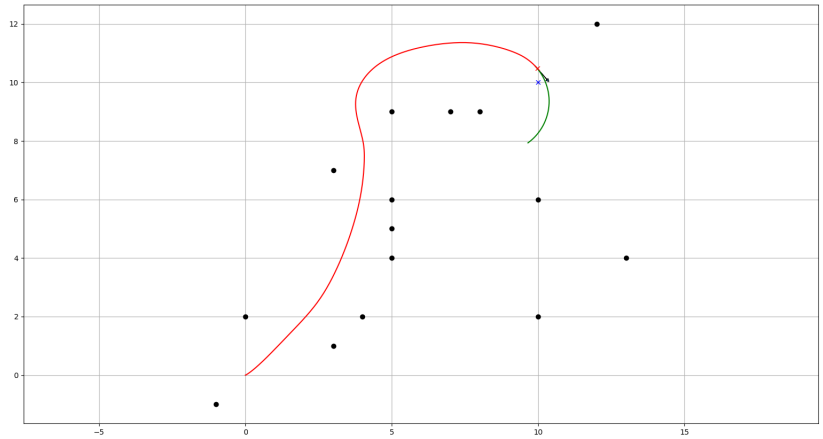

Figure (31) The robot radius is decreased to 0.5

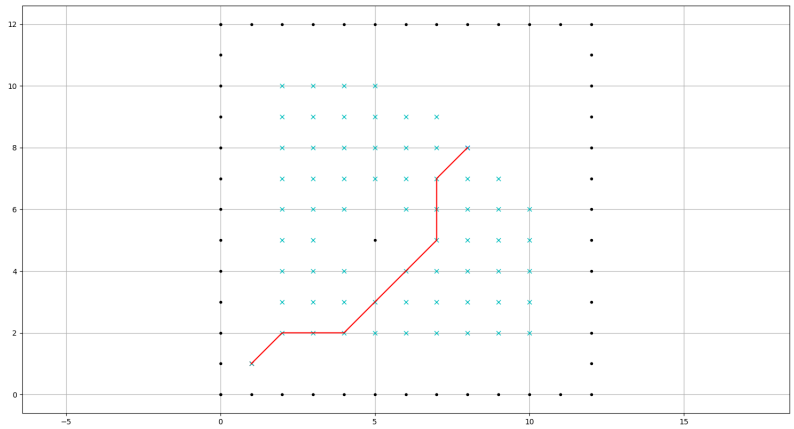

Figure (32) Dijkstra global planner over that environment path

```
the path coordinates are:
('rx', [8.0, 7.0, 7.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0])
('ry', [8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 2.0, 2.0, 1.0])
```

Figure (33) The path coordinates output