Trajectory Planning and Control for Nonholonomic Mobile Robot among Obstacles

Zhuokai Zhao[1,2], Changxin Yan[1,3] and Mengdi Xu[1,3]

***Abstract*— Trajectory generation and tracking, which involve path planning, obstacle avoidance, and nonlinear control, have become a hot research area for the last two decades. EduMIP is a widely-used nonholonomic mobile robot which moves with two parallel wheels connected by a single axle. It is an underactuated system and its dynamics is under nonholonomic constraints. The paper focuses on both the motion planning and trajectory tracking in 2D space with obstacles for the EduMIP robot. The authors propose a modified RRT*-based trajectory planning algorithm with customized heuristic function. A feedback linearization controller is proposed to ensure EduMIP's accurate trajectory tracking. Experiments have been performed in both MATLAB and ROS Gazebo simulation environment.**

I. INTRODUCTION

Mobile Inverted Pendulum (MIP), as an extension of the inverted pendulum system, has been widely explored in recent research topics due to its unique characteristics that combine both the inverted pendulum system as well as a wheeled mobile robot. EduMIP, as shown in Figure 1, is one of the MIP systems that is capable of both self-balancing and moving under the nonholonomic constraints. In this paper, EduMIP was regarded solely based on its wheeled mobile robot characteristic, while its self-balancing function is disabled. This makes the robot be able to move and rotate around the center of its rear-wheel axle.

Fig. 1: EduMIP robot

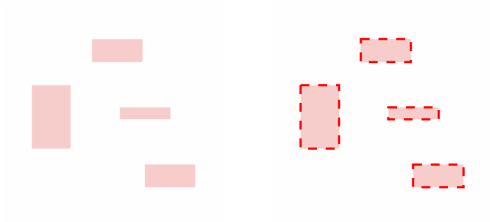
Many researches about trajectory planning and control of underactuated systems have been conducted in the recent years. Such systems are particularly attractive because they have fewer control inputs than generalized coordinates, which causes constraints on their dynamics. More specifically, these constraints can be represented by second-order differential equations [2]. Various approaches from both the planning and control perspectives have been explored. Kim, et al. [3] developed the motion planning algorithm of an airship using rapidly-exploring random trees (RRT). Spong et al. [4] implemented a controller for the acrobot swing-up. Nagarajan et al. [5] proposed an offline trajectory planning algorithm of the ballbot, which is particular relevant to this paper.

In this paper, a novel offline trajectory planning algorithm that provides the testing EduMIP robot a sub-optimal trajectory from start position q_{start} to end position q_{goal} is proposed. A customized controller which utilizes feedback linearization method is also designed to track EduMIP's movements.

The paper is organized with the following sections: Section II discusses the detailed approaches in path planning and trajectory generation; Section III describes the feedback linearization in general as well as our customized trajectory tracking method; Section IV presents the simulation results in both MATLAB and ROS Gazebo environment, along with comments on the performance; Section V discusses some potential future work and draws the conclusion.

II. TRAJECTORY PLANNING

The program starts with taking map images in either JPEG or PNG format, automatically analyzing the image and recognizing the obstacles, as shown in Fig. 2. The loaded map is then fed into the path planning algorithm.



(a) Sample map image (b) Detected obstacles

Fig. 2: Sample map with detected obstacles highlighted by red dash-line boxes

A. Path Planing with RRT*-based algorithm

The path planning algorithm presented in this paper, Customized-RRT* (C-RRT*), is built upon Rapidly-exploring Random Trees Star (RRT*) [1] algorithm. RRT* is

[1]These authors contributed equally to this work

[2]Zhuokai Zhao is with Laboratory for Computational Sensing and Robotics, Johns Hopkins University, Baltimore, MD 21210, USA zzhao30@jhu.edu

[3]Changxin Yan and Mengdi Xu are with the Department of Mechanical Engineering, Johns Hopkins University, Baltimore, MD 21210, USA cyan9@jhu.edu, mxu34@jhu.edu

an asymptotically optimal method which was an extension of the classic Rapidly-exploring Random Trees (RRT) [6]. C-RRT* sustained the basic skeleton and the advantages of RRT*, but was modified to fit the unique characteristics of EduMIP by changing existing methods and adding customized heuristic function to speed up the convergence rate. C-RRT*, as shown in Algorithm 1, takes the map with width x_{max} and height y_{max}, along with the starting and ending configuration q_{start}, q_{goal}, as the input and returns a graph that contains V and E, where V includes all the vertices (visited nodes) and E contains all the edges (paths between nodes). The algorithm uses several helper functions, whose main purposes are being introduced here but explained in detail later in this section.

SampleBiased() randomly (with bias) generates a new node $q_{rand} = (x_i, y_i, \theta_i)$ inside the map, where $x \in [0, x_{max}]$, $y \in [0, y_{max}]$, and $\theta \in [0, 2\pi]$. $Near(G, q_{rand})$ finds the nearest node in the current graph, G, to the input node q_{rand}. $Steer(q_{near}, q_{rand})$ steers the path from q_{near} to q_{rand}, and stops at the returned node q_{new} if the distance between q_{near} and q_{rand} is larger than the user-defined maximum step size. $Line(q_{near}, q_{new})$ connects a straight line between the two input nodes. $Cost(q_{near})$ returns the cost going from the start position q_{start} to q_{near}. Similarly, $c(q_1, q_2)$ returns the cost between two connected nodes. $NearNeighbor(G, q_{new}, R)$ finds all the nodes in G that are less than or equal to R distance away from q_{new}. $CollisionFree(q, q_{new})$ checks if the straight path between q and q_{new} will cause any collision, also if there is enough free space around both q and q_{new}.

The general disadvantage of RRT* is that its convergence to the optimal solution is very slow. Therefore, bias was added as the heuristic function in C-RRT* to accelerate the process. More specifically, function *SampleBiased*() is introduced to replace the original *SampleRandom*() in RRT* when randomly generating new node in each iteration. Inside *SampleBiased*(), the relative position between the previous node x_{prev} and x_{goal} is first analyzed. Then the function tends to provide higher bias toward the correct direction when generating the new node. For example, if $x_{prev} = (0,0)$, $x_{goal} = (10,10)$, $SampleBiased(x_{prev}, x_{goal})$ will have γ chance generating a new node $q_{rand} = (x, y, \theta)$ with $x \in [0,10]$, $y \in [0,10]$ and $\theta \in [0, \frac{\pi}{2}]$, where $\gamma \in [0,1]$ and is predefined to suit different situations. The function is explained in more details in Algorithm 2. Note that $rand(1)$ simply generates a random float number between 0 and 1.

One of the most important functions is to determine if there would be any collision between the two states. $CollisionFree(q_1, q_2)$ checks collision between q_1 and q_2 if they were going to be connected by a straight line. Note that this question could be transformed to checking whether the connected path will have intersections with any part of the boundaries of the obstacle. Moreover, since the goal of our customized algorithm is to design a path for EduMIP robot, the physical dimension of the robot also needs to be considered. Therefore, considering that EduMIP rotates around the center of its rear wheel axle and has a 40cm body length, along with requiring the path to have no intersections

Algorithm 1 G = C-RRT*(q_{start}, q_{goal})

while No Node ∈ V is near q_{goal} **do**
  $q_{rand} \leftarrow SampleBiased(i)$;
  $q_{near} \leftarrow$ Near($G = (V,E)$, q_{rand});
  $q_{new} \leftarrow$ Steer(q_{near}, q_{rand});
  if CollisionFree(q_{near}, q_{new}) **then**
    $V \leftarrow V \cup q_{new}$;
    $q_{min} \leftarrow q_{near}$;
    $q_{min} \leftarrow$ Cost(q_{near})+c(Line(q_{near},q_{new}));
    $Q_{neighbor} \leftarrow$ NearNeighbor($G = (V,E)$, q_{new}, R)
    for Each $q \in Q_{neighbor}$ **do**
      if CollisionFree(q, q_{new}) **then**
        if Cost(q)+c(Line(q, q_{new})) $< c_{min}$ **then**
          $q_{min} \leftarrow q$;
          $c_{min} \leftarrow$ Cost(q)+c(Line(q, q_{new}))
        end if
      end if
    end for
    $E \leftarrow E \cup \{(q_{min}, q_{new})\}$
    for Each $q \in Q_{neighbor}$ **do** ▷ Rewire the tree
      if CollisionFree(q_{new}, q) **then**
        c_q = Cost(q)
        if Cost(q_{new})+c(Line(q_{new}, q)) $< c_q$ **then**
          $q_{parent} \leftarrow$ Parent(q);
          $E \leftarrow (E \setminus \{(q_{parent}, q)\}) \cup \{(q_{new}, q)\}$
        end if
      end if
    end for
  end if
end while
return $G = (V,E)$

with obstacle boundaries, any node that is being checked must not collide with any obstacle in a 40cm radius in order to be determined as collision free. The more detailed steps are shown in Algorithm 3.

Define points (x_1, y_1) and (x_2, y_2) to be the end points of line L_1 and points (x_3, y_3) and (x_4, y_4) to be the end points of line L_2. To determine if two line segments, $L_1 = Line((x_1, y_1), (x_2, y_2))$ and $L_2 = Line((x_3, y_3), (x_4, y_4))$ had any intersections, compute

$$d_1 = \det\left(\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix}\right) \cdot \det\left(\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_4 \\ y_1 & y_2 & y_4 \end{bmatrix}\right)$$

$$d_2 = \det\left(\begin{bmatrix} 1 & 1 & 1 \\ x_3 & x_4 & x_1 \\ y_3 & y_4 & y_1 \end{bmatrix}\right) \cdot \det\left(\begin{bmatrix} 1 & 1 & 1 \\ x_3 & x_4 & x_2 \\ y_3 & y_4 & y_2 \end{bmatrix}\right)$$

If both $d_1 \leq 0$ and $d_2 \leq 0$, it is determined that the two lines have an intersection. The corresponding pseudo code is illustrated in Algorithm 4, where q_1, q_2 and e_1, e_2 represent the end points of the two lines respectively.

The iterations of generating and processing new nodes will not stop until at least one node in V is within a certain user-defined distance to the goal state and the straight path

Algorithm 2 SampleBiased($q_{prev}, q_{goal}, \gamma$)

Initialize q_{rand} with $SampleRandom(q_{prev}, q_{goal})$;
$\gamma = 10\gamma$;
Initialize a length-10 zeros array with γ ones
Randomly pick a number, k, from the array
if k is 1 **then** ▷ Bias has been added
  if $x_{goal} < x_{prev}$ **then**
    if $y_{goal} < y_{prev}$ **then**
      $x_{rand} = x_{prev} - rand(1) \cdot x_{prev}$;
      $y_{rand} = y_{prev} - rand(1) \cdot y_{prev}$;
      $\theta_{rand} = rand(1) \cdot \frac{\pi}{2} + \pi$
    end if
    if $y_{goal} > y_{prev}$ **then**
      $x_{rand} = x_{prev} - rand(1) \cdot x_{prev}$;
      $y_{rand} = y_{prev} + rand(1) \cdot (y_{max} - y_{prev})$;
      $\theta_{rand} = rand(1) \cdot \frac{\pi}{2} + \frac{\pi}{2}$
    end if
  end if
  if $x_{goal} > x_{prev}$ **then**
    if $y_{goal} < y_{prev}$ **then**
      $x_{rand} = x_{prev} + rand(1) \cdot (x_max - x_{prev})$;
      $y_{rand} = y_{prev} - rand(1) \cdot y_{prev}$;
      $\theta_{rand} = rand(1) \cdot \frac{\pi}{2} + \frac{3\pi}{2}$
    end if
    if $y_{goal} > y_{prev}$ **then**
      $x_{rand} = x_{prev} + rand(1) \cdot (x_{max} - x_{prev})$;
      $y_{rand} = y_{prev} + rand(1) \cdot (y_{max} - y_{prev})$;
      $\theta_{rand} = rand(1) \cdot \frac{\pi}{2}$
    end if
  end if
end if
return $q_{rand} = (x_{rand}, y_{rand}, \theta_{rand})$

Algorithm 3 CollisionFree(q_1, q_2, *Obstacles*)

for Each Obstacle O in *Obstacles* **do**
  for Each Boundary p in Obstacle **do**
    if HasIntersection(q_1, q_2, p) == True **then**
      if CheckSurrounding(q_1, q_2) == True **then**
        return True;
      end if
    end if
  end for
end for
return False;

between them is collision-free. In other words, once the algorithm stops, it is guaranteed that enough nodes which would generate a path that is capable of going from start to the goal configuration have been obtained. Therefore, the next step is to find the minimum-cost path from these nodes. As shown in Algorithm 1, in every iteration, each new node q_{new} will be computed a cost associated with it, as well as a parent node q_{parent}. It is guaranteed that the cost going from q_{parent} to q_{new} is less than any other nodes reaching q_{new}.

Algorithm 4 HasIntersection(q_1, q_2, e_1, e_2)

$x_1 = q_1(1)$, $x_2 = q_2(1)$, $x_3 = e_1(1)$, $x_4 = e_2(1)$;
$y_1 = q_1(2)$, $y_2 = q_2(2)$, $y_3 = e_1(2)$, $y_4 = e_2(2)$;
Compute d_1 and d_2
if $d_1 \leq 0$ && $d_2 \leq 0$ **then**
  return True;
end if
return False;

To ensure that the path is both the shortest and as far away from obstacle as possible, the cost function is designed as the linear combination of both the distance between two nodes and the distance to the nearby obstacle, which is defined as

$$\text{Cost} = \text{Dist}(q_{near}, q_{new}) + \rho \cdot e^{(-\sigma \cdot \text{Obstacle-Distance}^2)}$$

where Dist(q_{near}, q_{new}) is the Euclidean distance between q_{near} and q_{new}, ρ and σ are parameters that are tuned through experiments. It is not hard to see that the goal node q_{new} will have less cost if it is both closed to the starting node q_{near} and far away from any nearby obstacles.

To find the least-cost path between two configurations, the algorithm starts from the goal state and gradually moving backward to the starting state. It mainly utilizes the cost and parent information calculated in C-RRT* and always chooses the current node's parent as the previous step, until the starting position is reached, as shown in Algorithm 5.

Algorithm 5 GeneratePath(G, q_{start}, q_{goal})

while q_{goal}.parent != 0 **do**
  $q_{previous} \leftarrow q_{goal}.parent$;
  Path $\leftarrow$ {previous, Path};
  $q_{goal} \leftarrow q_{previous}$;
end while
return Path;

B. Trajectory Generation with Polynomial Fit

To generate trajectory from the path planning result, two key aspects have to be added on each position: time stamp and the corresponding velocity. To ensure that the velocity exists at any given position, the path has to be continuous and smooth. Therefore, the first step is using polynomials to fit the path generated by C-RRT* introduced in Section II(A). For different input maps, starting and ending positions, the path may be in very complicated shapes. Therefore, using only one polynomial, even with higher orders, may not be sufficient to provide an acceptable fitting result for the whole path. Thus, the authors proposed to divide the path into numerous segments, and to apply polynomial fit on each segment individually. One important issue to notice during this process is to make sure that both the positions and derivatives between the end of previous segment and the start of next segment are the same. This issue is solved by setting constraints on each segment when performing polynomial fit.

Assume C-RRT* provides a path with n points,

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

The goal is to find a k degree polynomial

$$P(x) = p_k x^k + p_{k-1} x^{k-1} + \cdots + p_2 x^2 + p_1 x + p_0$$

that fits the n points while passing through f fix points, given as

$$(x_{fix_1}, y_{fix_1}), (x_{fix_2}, y_{fix_2}), \ldots, (x_{fix_f}, y_{fix_f})$$

and satisfying d derivatives

$$\{\dot{y}_1, \dot{y}_2, \ldots, \dot{y}_d\}$$

at points

$$\{x_{der_1}, x_{der_2}, \ldots, x_{der_d}\}$$

The idea is to first solve the polynomial coefficients that satisfies both the position and derivative constraints, then modify this polynomial to make it fit for all the points while satisfying the constraint at the same time. Suppose the polynomial that satisfies the constraints has coefficients $p^c = \{p_k^c, p_{k-1}^c, p_{k-2}^c, \ldots, p_1^c, p_0^c\}$. Solve the least square solution of equation

$$A \cdot p^c = \begin{bmatrix} Y_{fix} \\ \dot{Y} \end{bmatrix}$$

that is,

$$\begin{bmatrix} x_{fix_1}^k & x_{fix_1}^{k-1} & \ldots & x_{fix_1}^0 \\ x_{fix_2}^k & x_{fix_2}^{k-1} & \ldots & x_{fix_2}^0 \\ \vdots & \vdots & \ldots & \vdots \\ x_{fix_f}^k & x_{fix_f}^{k-1} & \ldots & x_{fix_f}^0 \\ \hline k \cdot x_{der_1}^{k-1} & (k-1) \cdot x_{der_1}^{k-2} & \ldots & 0 \\ k \cdot x_{der_2}^{k-1} & (k-1) \cdot x_{der_2}^{k-2} & \ldots & 0 \\ \vdots & \vdots & \ldots & 0 \\ k \cdot x_{der_d}^{k-1} & (k-1) \cdot x_{der_d}^{k-2} & \ldots & 0 \end{bmatrix} \cdot p^c = \begin{bmatrix} y_{fix_1} \\ y_{fix_2} \\ \vdots \\ y_{fix_f} \\ \hline \dot{y}_1 \\ \dot{y}_2 \\ \vdots \\ \dot{y}_d \end{bmatrix}$$

where matrix A on the left hand side has $f+d$ rows and $k+1$ columns. By solving the least square solution for p^c, we have obtained the polynomial that would satisfy our position and derivative constraints. The next step is to find the compensation coefficients p^s such that $p = p^c + p^s$ will make the new polynomial, while still satisfying the constraints, also fit for all the points. To find such p^s, the y values needed, y^s, are defined as

$$y_i^s = y_i - (p_k^c x_i^k + p_{k-1}^c x_i^{k-1} + \ldots, p_1^c x + p_0^c)$$

Similar to the process solving for p^c above, p^s is obtained by finding the least square solution of equation

$$B \cdot p^s = y^s$$

that is

$$\begin{bmatrix} x_1^k & x_1^{k-1} & \ldots & x_1^0 \\ x_2^k & x_2^{k-1} & \ldots & x_2^0 \\ \vdots & \vdots & \ldots & \vdots \\ x_n^k & x_n^{k-1} & \ldots & x_n^0 \end{bmatrix} \cdot p^s = \begin{bmatrix} y_1^s \\ y_2^s \\ \vdots \\ y_n^s \end{bmatrix}$$

By solving the least square solution of the equation, coefficients $p^s = \{p_k^s, p_{k-1}^s, \ldots, p_0^s\}$ are obtained. Combine both p^c and p^s, the final polynomial has been obtained with coefficients

$$p = p^c + p^s = \{p_k^c + p_k^s, p_{k-1}^c + p_{k-1}^s, \ldots, p_0^c + p_0^s\}$$

After the smooth path is obtained, velocities could be assigned based on the curvature of each node. For every point in the polynomial fit, $P(i)$, the corresponding curvature, $k(i)$, is computed as

$$k(i) = \frac{|P''(i)|}{\sqrt{((1 + P'(i)^2)^3)}}$$

The velocity $v(i)$ associated with position $P(i)$ is therefore computed as

$$v(i) = \min(\sqrt{\frac{a_{max}}{k(i)}}, v_{max})$$

where v_{max} and a_{max} are the maximum velocity and acceleration of the EduMIP robot. Time stamps of each step is then assigned based on the average velocities. θ are re-computed based on the next position, that is,

$$\theta(i) = \arctan\left(\frac{P_y(i+1) - P_y(i)}{P_x(i+1) - P_x(i)}\right)$$

Therefore, the final generated trajectory is completed as

$$T(i) = \{t(i), P_x(i), P_y(i), \theta(i), v_x(i), v_y(i)\}$$

III. TRAJECTORY TRACKING CONTROL

The next step is generating the control input for EduMIP to track the desired trajectory. For the nonholonomic car-like EduMIP robot, we use dynamic feedback linearization to generate the control trajectory $u(t)$,

$$u(t) = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1}$$

where $u_1(t)$ is the linear velocity and $u_2(t)$ is the angular velocity.

The state variable for EduMIP system is $\vec{x} = (x, y, \theta)$. The output is $\vec{y} = (x, y)$. The system equations are

$$\begin{aligned} \dot{x} &= u_1 \cdot \cos\theta \\ \dot{y} &= u_1 \cdot \sin\theta \\ \dot{\theta} &= u_2 \end{aligned}$$

To use dynamic feedback linearization, we are interested in writing $u(t)$ in the from of

$$u = a(x) + b(x)\dot{v}(t),$$

where b(x) is a nonsingular matrix, and $v \in \mathbb{R}^2$ is virtual input. By differentiating y = h(x) enough times so that all controls appear in a linear, nonsingular relationship with the output and its higher derivatives.

The first order derivative is

$$\dot{\vec{y}} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{pmatrix} u_1 \cdot \cos\theta \\ u_1 \cdot \sin\theta \end{pmatrix}$$

It is not possible to track the desired trajectory since only u_1 could directly influence the output.

By differentiating both sides again, we have

$$\ddot{\vec{y}} = \begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} = \begin{pmatrix} \dot{u_1}\cos\theta + u_1 u_2 \sin\theta \\ \dot{u_1}\sin\theta + u_1 u_2 \cos\theta \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \dot{u_1} \\ u_1 u_2 \end{bmatrix}$$

Thus, we could get the acceleration $(\ddot{x}, \ddot{y})$ by controlling $(\dot{u_1}, u_2)$ instead of (u_1, u_2). The virtual input is defined as

$$\vec{v} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix}$$

Note that the system is determined only when $u_1 \leq 0$. We need to define a compensator $\xi = u_1$ which has its own dynamics that could affect the system.

Then we have the dynamic feedback as

$$\dot{\xi} = \cos\theta \cdot v_1 + \sin\theta \cdot v_2$$
$$u_2 = \frac{-\sin\theta \cdot v_1 + \cos\theta \cdot v_2}{\xi}$$

Qualified control inputs should be able to make the error state z asymptotically stabilized to zero where

$$z = \begin{bmatrix} \vec{y} - \vec{y_d} \\ \dot{\vec{y}} - \dot{\vec{y}}_d \end{bmatrix}$$

The control law in virtual input space is set as

$$\ddot{v} = \ddot{\vec{y}}_d - K_p(\vec{y} - \vec{y_d}) - K_d(\dot{\vec{y}} - \dot{\vec{y}}_d),$$

where K_p and K_d are some positive constants which mean position and derivative error gains respectively. This control law will result in the linear closed-form error dynamics

$$\dot{z} = Az,$$

where

$$A = \begin{bmatrix} 0 & 1 \\ -K_p & -K_d \end{bmatrix}$$

is a Hurwitz Matrix so that the controller is asymptotically stable.

Note that car-like robots including EduMIP have their own physical constraints. For example, the motors for the two wheels equipped in EduMIP have their own velocity and torque limits. In our case, we restrict the linear velocity to $[-0.8, 0.8] m/s$ and angular velocity to $[-0.5, 0.5] rad/s$. This constraint is necessary since u_1 could be near zero which will result in quite large u_2 based on the controller.

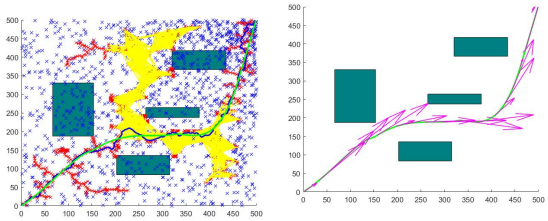
After getting the acceleration of u_1, we need to iterate at each time stamp to update u_1 as

$$u_1 = u_1 + \dot{u_1} \cdot \Delta t$$

IV. RESULTS

Trajectory planning results are shown in Figure 3. Figure 3a shows the path planning output from C-RRT* algorithm. The leftmost obstacle is numbered as O_1, the right three obstacles are numbered as O_2, O_3 and O_4 from top to bottom. It is clear that the path in Figure 3a successfully starts from position $q_{start} = (0,0)$ and ends at the goal position $q_{goal} = (499, 499)$. As the C-RRT* algorithm guarantees to output minimum-cost path from its current sampling points, we argue that from the number of points we have sampled, the shown path is the optimal solution. Better solutions might be achieved with longer sampling time, but it is a trade-off between performance and efficiency. Since our cost function is not only a minimization towards shortest distance, but also a maximization of distances from obstacles, during the process, especially when the path is passing through O_1, O_3 and O_4, the path very well keeps as far away to the obstacles on both sides as possible.

Figure 3b shows that the velocity assignments of the path are also reasonable. The velocity directions very well align with the tangent of the path. The magnitudes (lengths of the arrows) of the velocities are also as planned larger during straight-line segments and smaller if the path is going through corners.



(a) Path planning output (b) Trajectory with added time and velocity

Fig. 3: Trajectory planning results

To show the trajectory control result, we simulate the whole system both in MATLAB and ROS-Gazebo. The simulation in MATLAB could show the trajectory both in workspace and control space clearly. However, the real process involving physical constrains as well as communication could not easily be added or verified in MATLAB. Gazebo in ROS gives us a better choice to visualize the interaction between our controller and EduMIP as well as EduMIP and the environment.

The planning and control task is to control the car-like EduMIP robot move from the start position $[0,0]$ to target final position $[5,5]$ in an optimal path. The optimal path has least cost value and will not collide with any obstacles.

A. MATLAB Trajectory Tracking Simulation

The generated trajectory data set that contains t, x_d, y_d, $\dot{x_d}$, $\dot{y_d}$, $\ddot{x_d}$, $\ddot{y_d}$ and θ is imported first. To get more accurate simulation, we interpolate $N = 10$ times between each time gap in the desired trajectory.

Figure 4 shows the tracked result.

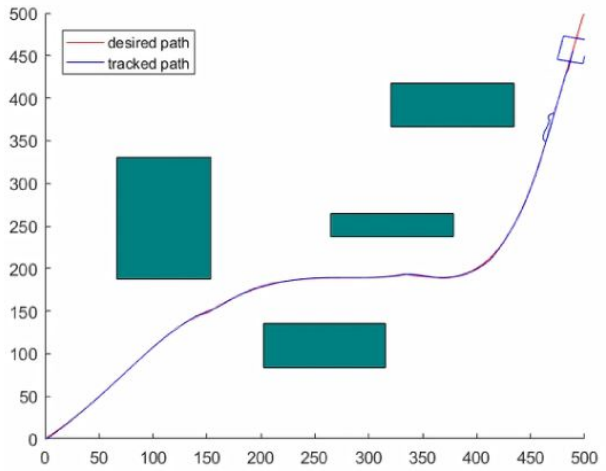
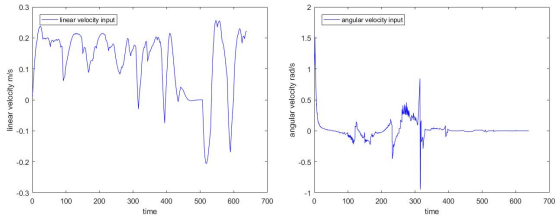


Fig. 4: Matlab trajectory trajectory simulation result.

Figure 5 shows the inputs versus time during the whole tracking process.



(a) Linear velocity input u1 (b) Angular velocity input u2

Fig. 5: Inputs varying with time.

B. ROS-Gazebo-RVIZ Simulation

Gazebo is a robust 3D dynamic simulator built on ROS which could accurately simulate robots in complex environments. It provides physics simulation at a much higher degree of fidelity than other simulators, such as a lot of sensors and interfaces for both users and programs. [7] Another useful platform which could show the movement of our car-like robot is RVIZ. It can show the position and heading of the EduMIP and stack them into a trajectory.

We first build the ”world” environment, which contains light source and obstacles with predefined shape and position. Note that the top view of the environment in Gazebo should be consistent with the map. The model of the EduMIP is modified into a three-wheel cart. Two out of three are parallel and could move clockwise and counter clockwise. The last passive wheel is placed on another side for support. A controller plugin ”differential_drive_controller” is added which could receive the linear and angular velocity command. To get the real-time pose, we add a ”p3d_base_controller” plugin which could read the position and orientation of EduMIP with respect to the fixed world frame. Figure 6 shows the simulation environment in Gazebo.

To show the pose of EduMIP in RVIZ, we wrote a TF broadcaster to publish the transformation between the

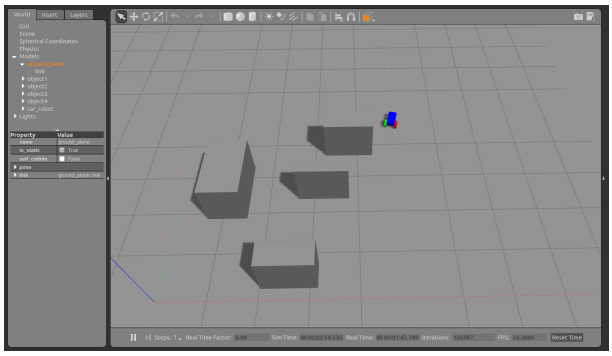


Fig. 6: Gazebo trajectory tracking simulation.

”world” frame and the ”car-body” fixed on EduMIP. The whole odometry path is shown in Figure 7.

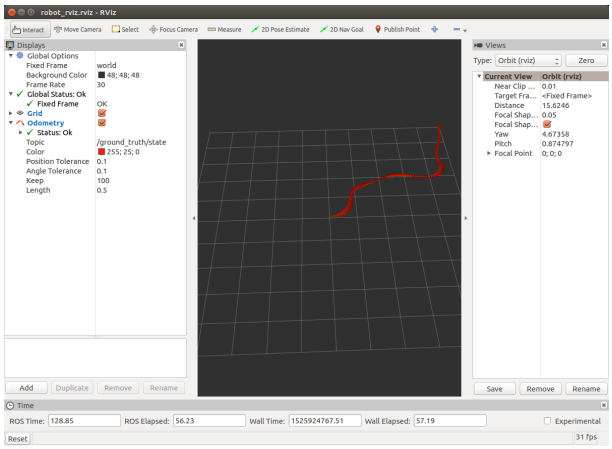


Fig. 7: Odometry path.

The communication between the trajectory tracking controller and differential drive controller is by topic. The desired trajectory is published at the time stamp according to t got from the data set via topic ”desired states”. The trajectory tracking controller will calculate and publish the control command at 10Hz to topic named ”cmd vel”. One important thing to note is that the time step of the desired trajectory set is various but the frequency of publisher is fixed. To synchronize publisher and the desired trajectory time, during each time period, we use unchanged desired point to calculate the control. Publishing the command in a smaller fixed time step could serve as a kind of interpolation that could help make the simulation more accurate.

The source codes are public on `https://github.com/ChangxinY/nonlinear_control`

V. CONCLUSION AND FUTURE WORK

As shown in Section IV, the ability to successfully plan a trajectory and perform tracking control has been demonstrated and experimentally verified.

Without an enormous amount of iterations, C-RRT* can only provide sub-optimal path planning results, even with

the added heuristic function that accelerates the process. Therefore, the authors wish to explore other path optimization methods to locally optimize the path in the future. The authors could use Cross-Entropy (CE), which is a gradient-free stochastic optimization method, or Stochastic Policy Optimization (SPO), which is, on the other hand, a gradient-based method.

ACKNOWLEDGMENT

The authors thank course instructor Prof. Marin Kobilarov and teaching assistants Gowtham Garimella, Matthew Sheckells of EN 530.678 Nonlinear Control and Planning in Robotics.

REFERENCES

[1] Karaman, S, Frazzoli, E (2011) Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research 30(7): 846-894.

[2] J. R. Ray, Nonholonomic constraints, Am. J. Phys., no. 34, 1966, pp 406-408.

[3] J. Kim and J. P. Ostrowski, Motion planning of aerial robot using rapidly-exploring random trees with dynamic constraints, in Proceeding IEEE International Conference on Robotics and Automation, 2003, pp 2200-2205.

[4] M.W. Spong, The swing up control problem for the Acrobot, IEEE Control Systems Magazine, February, 1995, pp 49-55.

[5] U. Nagarajan, G. A. Kantor and R. L. Hollis, Trajectory Planning and Control of an Underactuated Dynamically Stable Single Spherical Wheeled Mobile Robot? in Proceeding IEEE International Conference on Robotics and Automation, pp. 3743-3748, 2009.

[6] LaValle, S. M. 1998b. Rapidly-exploring random trees: A new tool for path planning. Report No. TR 98-11, Computer Science Department, Iowa State University.

[7] Gazebo Beginner Overview, `http://gazebosim.org/tutorials?tut=guided_b1&cat=`.