EECS 106b Project2 Report

Nonholonmic Control with Turtlebots

Anxing Xiao(SID: 3034333046)
Wenzhe Tong(SID: 3035444052)

March 8th, 2020

Contents

1 Video and Code

1. Video Link: Video
2. Github repository: https://github.com/threefruits/Path_planning_project/invitations

2 Abstract

In this project, we implemented three path planning algorithm and a close loop controller: optimization-based planner, Nonholonmic RRT, sinusoidal planner[2]. In optimization-based planner, we had discussed how to determent N and dt with different goal point and task. In Nonholonmic RRT planner, we come up with a cartesian-like distance which measure the distance in SE(2). Also, a greedy sample-based local planner is used to satisfy the nonholonmic motion of bicycle model. In sinusoidal planner, we added constraint checking method into planner to avoid impossible motion of our bicycle model. In close loop control part, we design a PD-MPC-like controller, which could auto tune the velocity and steer based on state error and minimized the error between predict state and desird state.
Then We experimented each planner and our PD-MPC-like controller in different task: simple manipulation in empty map and navigation in complex environment. Results shows that our controller works well in complex tasks. And we discuss the result of different planner in different environment and conclude the best using environment of each planner.

3 Methods

3.1 Sinusoidal planner

1. Input constraints
   For input constraints, we using a function *check_limit* to check the inputs in our ode of $a1$ and $a2$. If the inputs are out of the boundary, we set the variable $flag$ to False. We have the ode solver function $find\_root$, if the variable $flag$ is False when solving the ode, the solver will retry another value for $a1$ and $a2$ until the input constraints satisfied.

2. state bounds
   Similar to checking input constraints, we have another while loop for checking state bounds by calling the function $v\_path\_to\_u\_path$, which is modified to check the state first. If the function returns the state is not satisfied, we rerun the while loop to solve the ode again for another feasible $a1$ and $a2$ values until all the state bounds satisfied.

3. singularity[2]

We have two models, the original one is substituting dynamics model by:

$$v_1 = u_1 \cos\theta$$

$$v_2 = u_2$$

$$\alpha = \sin\theta$$

And the alternative one is substituting dynamics model by:

$$v_1 = u_1 \sin\theta$$

$$v_2 = u_2$$

$$\alpha = \cos\theta$$

As the document[4] said, we implemented the alternative model for the singularity of the original model which is $\pi/2, -\pi/2$. And also the alternative model have the singularity of $0, \pi$. We have to switch the model depend on the state θ, we divide the plane into 4 portions,$(-\pi/4, \pi/4), (\pi/4, 3\pi/4), (3\pi/4, -3\pi/4), (-3\pi/4, -\pi/4)$. In the first and third portions we use the original model to avoid the singularity of $\pi/2, -\pi/2$. Second and forth portions for alternative model to avoid singularity of $0, \pi$.

3.2 RRT planner

1. Local planner

In our planner, we generate a path with 50 timesteps. In each timesteps, we use sample-based greedy planner, which choose the best action from sampled action set in 0.01s.

Define a cost of action a in $state_{now}$

$$J(state_{now}, a) = cost_{distance} + cost_{steering}$$

where

$$cost_{distance} = distance(goal, state_{now} + Dynamics(state_{now}, a))$$

$$cost_{steering} = 0.15 * steering$$

So we choose the action which minimize the cost:

$$a^* = \arg\min_{a \in A} J(state_{now}, a)$$

After we get the best action a, we can estimate the state with the action by dynamics model. Then reset the $state_{now} = state_{now} + Dynamics(state_{now}, a*)$. Then repeat this 50times until we get a path with states and inputs.

In our work, we choose our A = [[0.5,0.5],[0.5,0.3],[0.7,0.0],[0.5,-0.3],[0.5,-0.5],[-0.3,0.5],[-0.3,0.3],[-0.5,0.0],[-0.3,-0.3],[-0.3,-0.5]]

2. Distance metric

   We define the distance by a cartesian-like method.

   $$Distance(c1, c2) = \sqrt{(x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2 + 0.15 * (distance_{theta})^2}$$

   where,

   $$distance_{theta} = pi - |mod(theta_{c1} - theta_{c2}, 2 * pi) - pi|$$

   By this way, we define the distance in (x,y,theta) space.

3. Sampling heuristics [3]

   We use a Goal-bias method. With probability of 0.4, choose the goal point as our sample point. Else, sample the point randomly in C-space.

3.3 Optimization planner

1. Setting N and δt

   For setting N and δt, we modify the code *optimization_planner.py*. To be specific, we design our distance similar as the distance metric as we talked above.

   $$Distance(c1, c2) = \sqrt{(x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2 + 0.15 * (distance_{theta})^2}$$

   And we set our N as

   $$N = K * Distance(start, goal)$$

   Where K is a coefficient we should tune for less computational complexity but high accuracy[1]. Also, we noticed that, running time is $N * \delta t$, so we have to tune our K carefully by measuring the minimum time to get the goal approximately by using the Manhattan distance.
   For setting δt, we choosing the same dt of other planner which is 0.01s.

2. Implemented for arbitrary goal state

   As we sets our coefficient K as we talk above, we corresponding our configuration space to our distance metric. So we can make sure our inputs are in the feasible set.

3.4 Close loop controller

In our close loop controller, we use combine the pd and mpc-like controller which only predict only one/two steps.
Firstly, we use simple pd control to tune the error of theta.

$$preinput_{velocity} = velocity_{ff}$$

$$preinput_{steer} = steer_{ff} + 0.5 * error_{theta}$$

Then define the sample tune action set T, which include all possible tune method. For all t in T, we calculate the cost of tune action as:

$$\begin{aligned} t^* = \arg\min_{t \in T} \quad & \sum_{i=0}^{N} (x_{desir\_state(i)} - x_{pred\_state(i)})^2 + (y_{desir\_state(i)} - y_{pred\_state(i)})^2 + 0.15 * (d_{theta})^2 \\ \text{s.t.} \quad & pred\_state(i+1) = desir_s tate(i) + Dynamics(desir_s tate(i), t) \\ & Inputlimit_{low} \le input + t \le Inputlimit_{up} \\ & Statelimit_{low} \le desired_s tate(i) + Dynamics(desired_s tate(i), t) \le Statelimit_{up} \end{aligned}$$

In our real implement, we choose N=1 or 2. And tune action set T=[[0.15,0],[-0.15,0],[0,0.15],[0,-0.15],[0.1,0.1],[0.1,-0.1],[-0.1,0.1],[-0.1,-0.1],[0,0]].
Then we define our final controller as:

$$Input_{velocity} = preinput_{velocity} + t^*_{velocity}$$

$$Input_{steer} = preinput_{steer} + t^*_{steer}$$

This is our final close loop controller.

4 Results

4.1 Sinusoidal planner

1. Simple motion task
   In our simple motion task, we want our robot go from (1, 1, 0, 0) to (2, 1.3, 0.7, 0). You can see from the figure below. Our planner goes x direction first, and then tune it's θ to around 0.7. And back up then tune it's y.
   There is another problem we haven't overcame yet, that is it cannot tune it's all parameters at the same time, and get an smooth curve (just like the OPT planner).

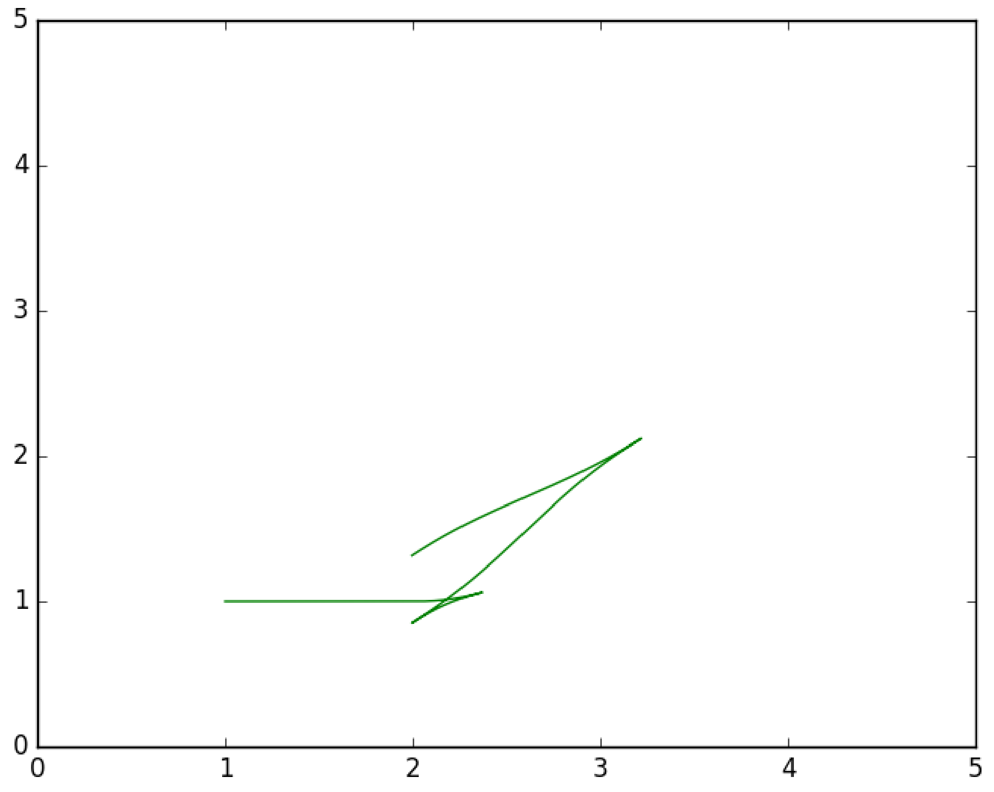

Figure 1: Sinusoidal planner simple motion

2. Parallel parking task
   For the parallel parking task, our aim is parking our car form state (1, 1, 0, 0) to state (1, 3, 0, 0). It act as I thought. Go forward and shift little bit left, and back up and shift left to our goal position. The steering angle ϕ touches the upper bound when steering, which is exactly 0.6 rad. If the upper bound changes, the distance of x

direction will change.
However, we comes the problem that if we change our goal state to (1, 3, 0.1, 0). The trajectory goes wired. I'm not sure what's the problem here, but hopefully our task achieves.

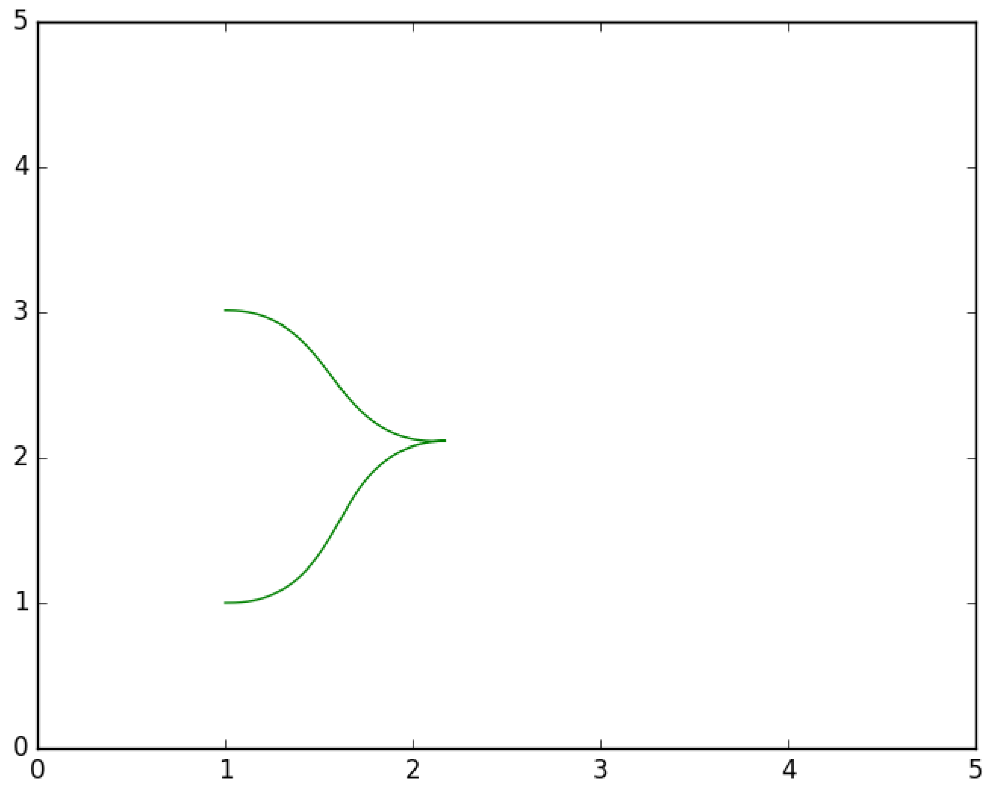

Figure 2: Sinusoidal planner parallel parking

3. Point turn task
   For this task, frustratedly we can not dealing with the singularity properly. We actually implemented two models given by the document, and also combined them together, but that cannot working well. For the first few trajectories it acts as normal, but when meet with the singularity it fails. I think the problem is the switching part, but as for now, I cannot debug it out.

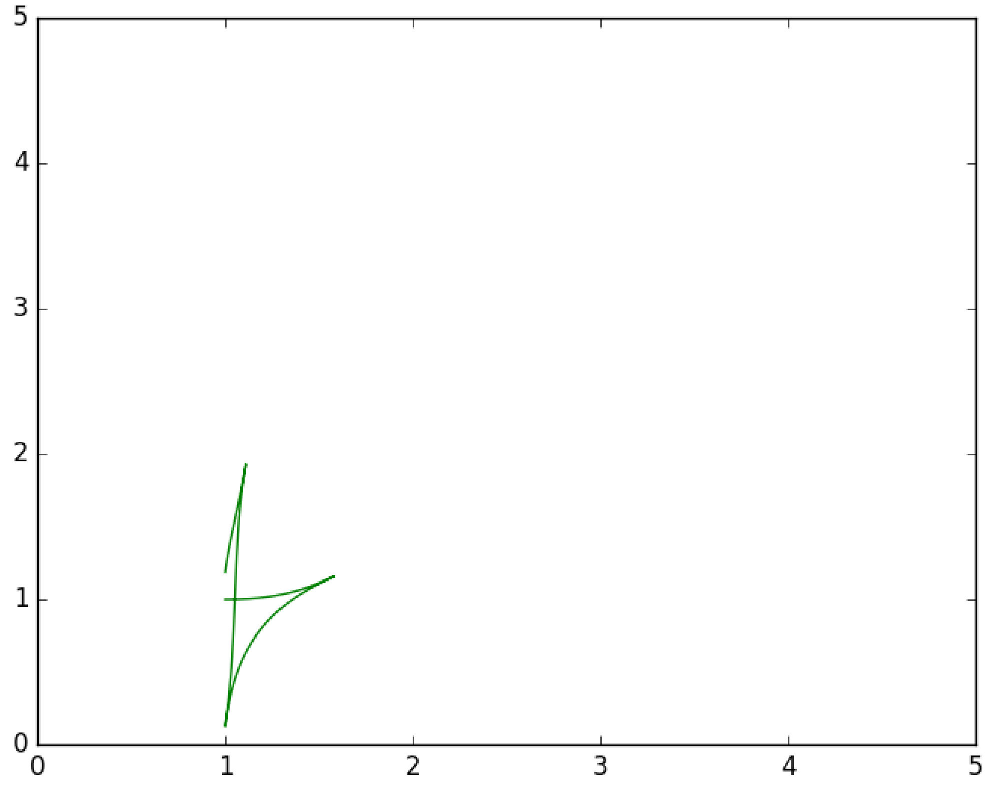

Figure 3: Sinusoidal planner point turn

4.2 RRT planner(3 tasks + navigation task)

See next section for summarize of results.

1. Simple motion task

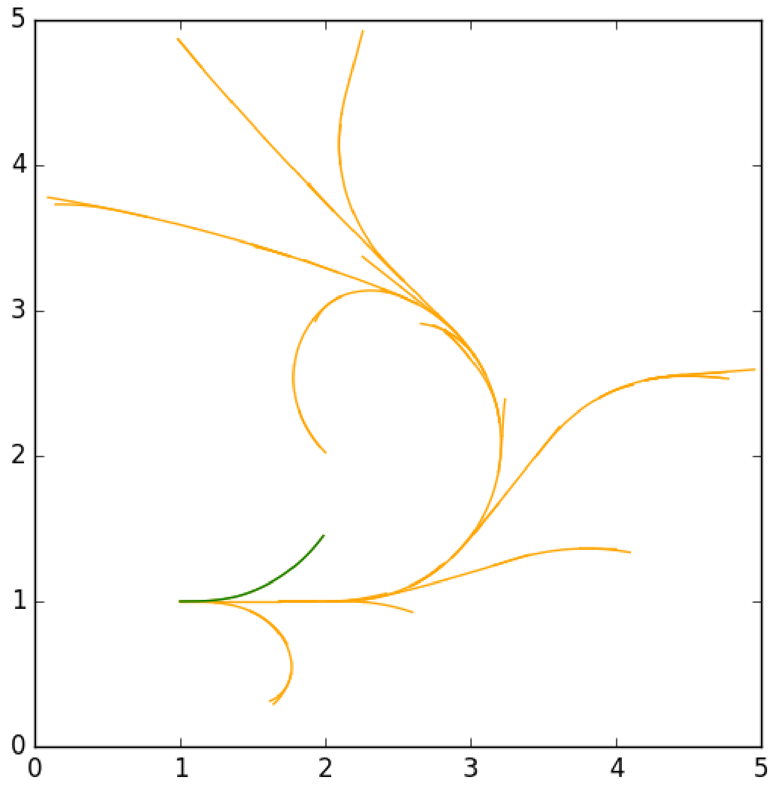

Figure 4: RRT planner simple motion

2. Parallel parking task

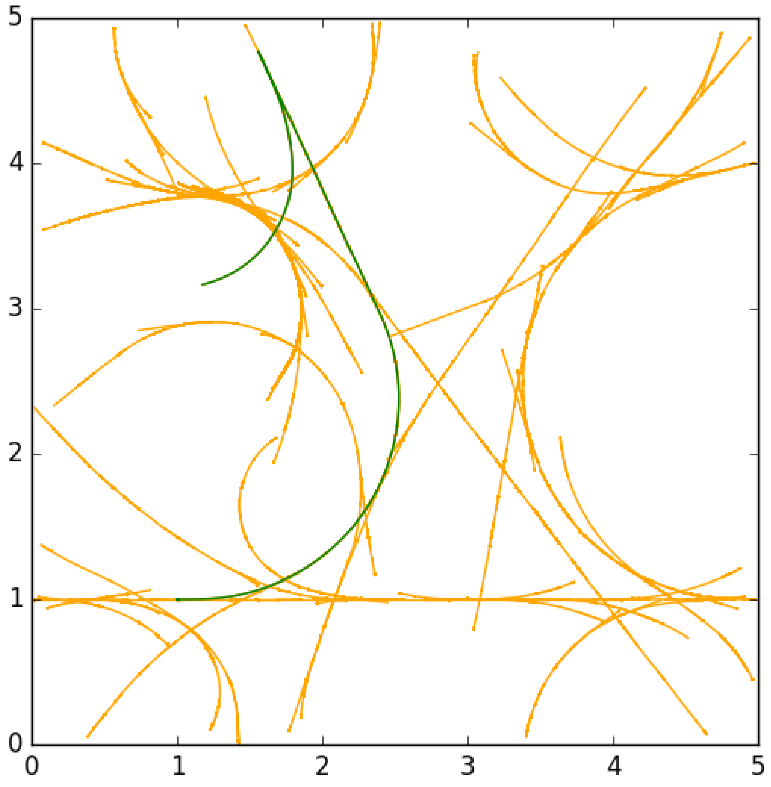

Figure 5: RRT planner parallel parking

3. Point turn task

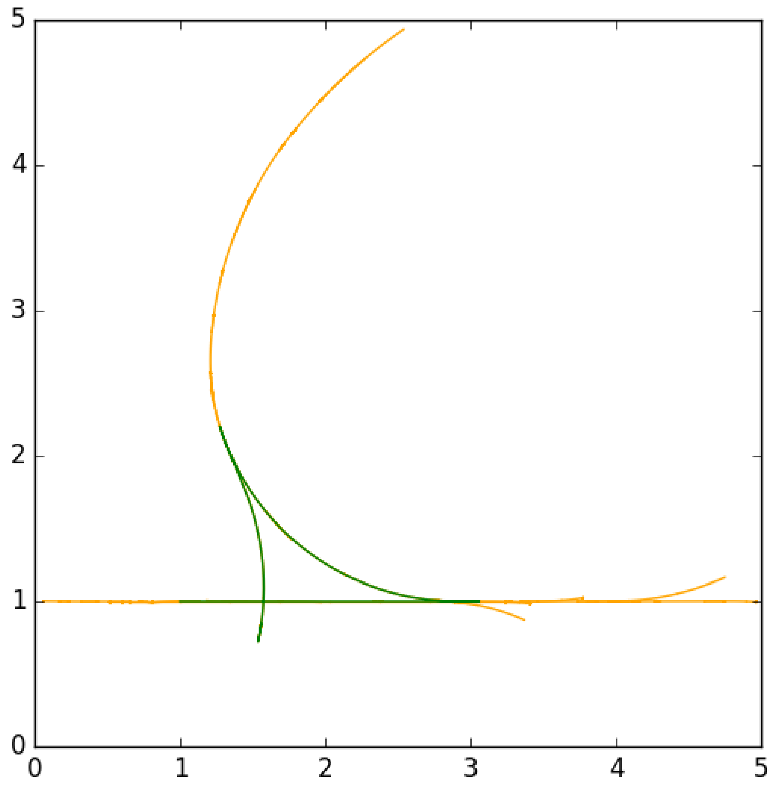

Figure 6: RRT planner point turn

4. Navigation task

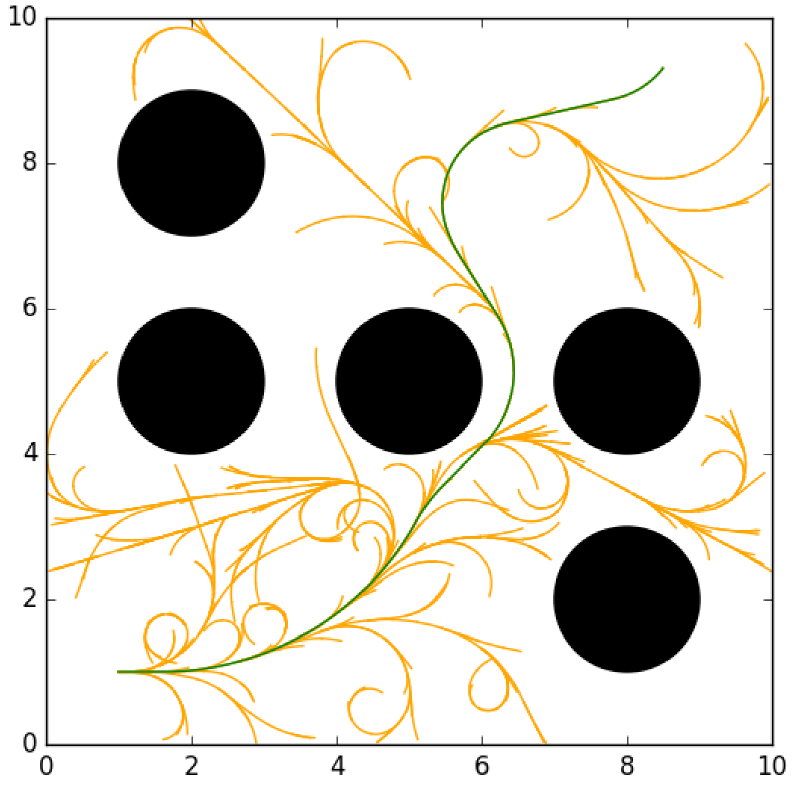

Figure 7: RRT planner navigation of map2

4.3 Optimization planner(3 tasks + navigation task)

1. Simple motion

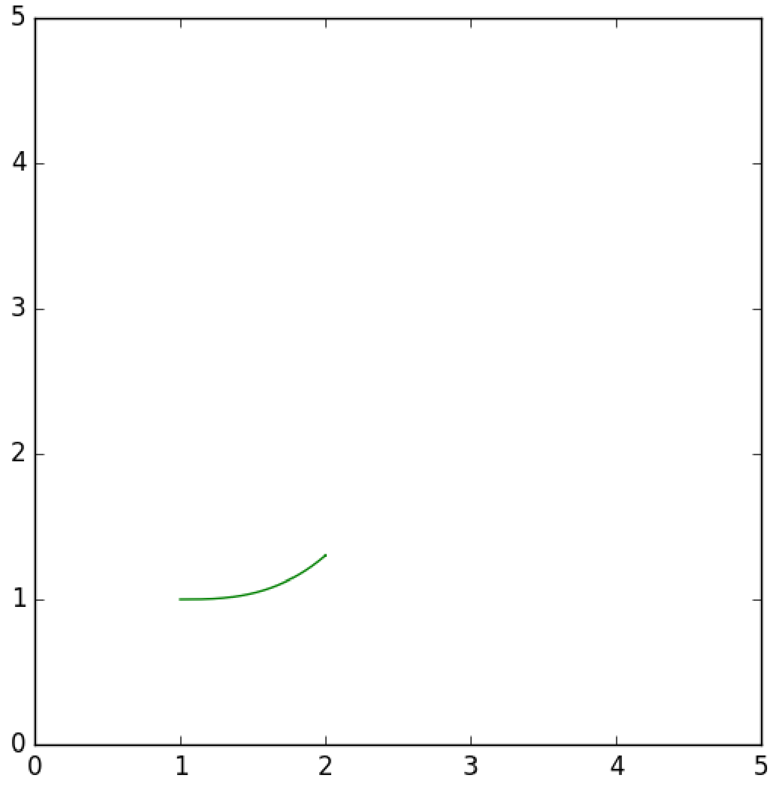

Figure 8: OPT planner simple motion

2. Parallel parking task

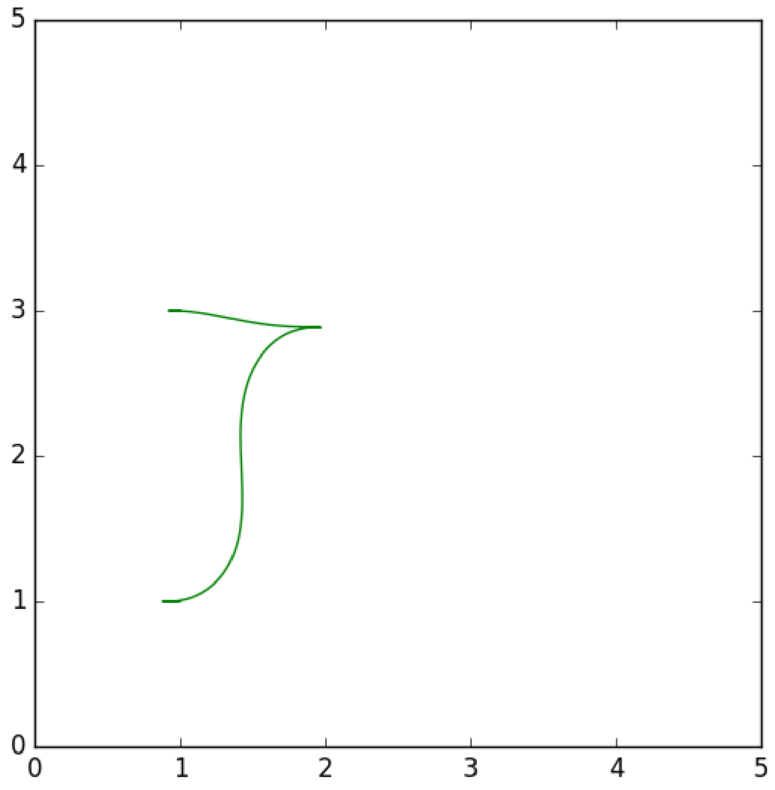

Figure 9: OPT planner parallel parking

3. Point turn task

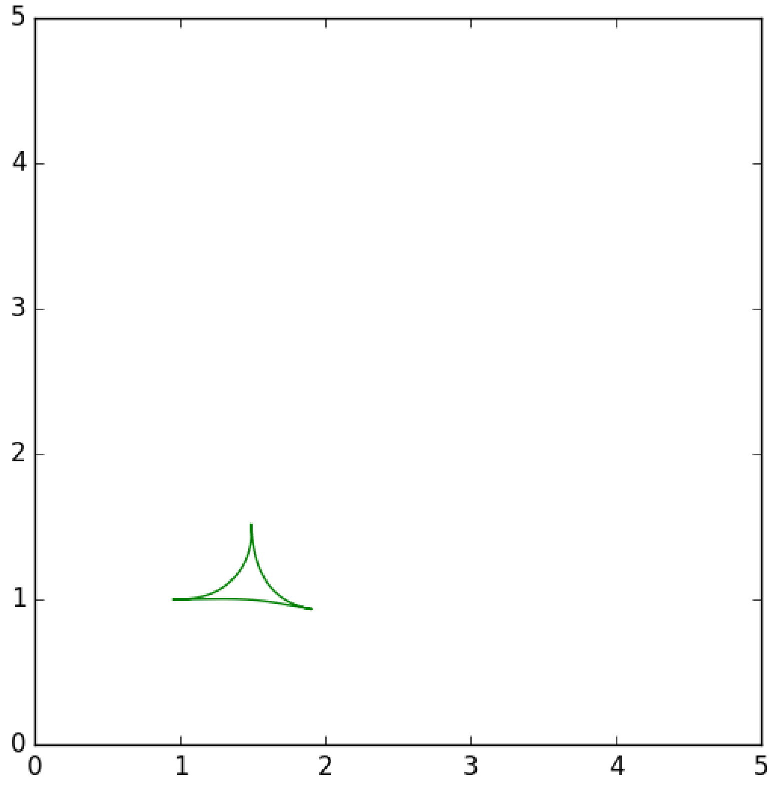

Figure 10: OPT planner point turn

4. Navigation task

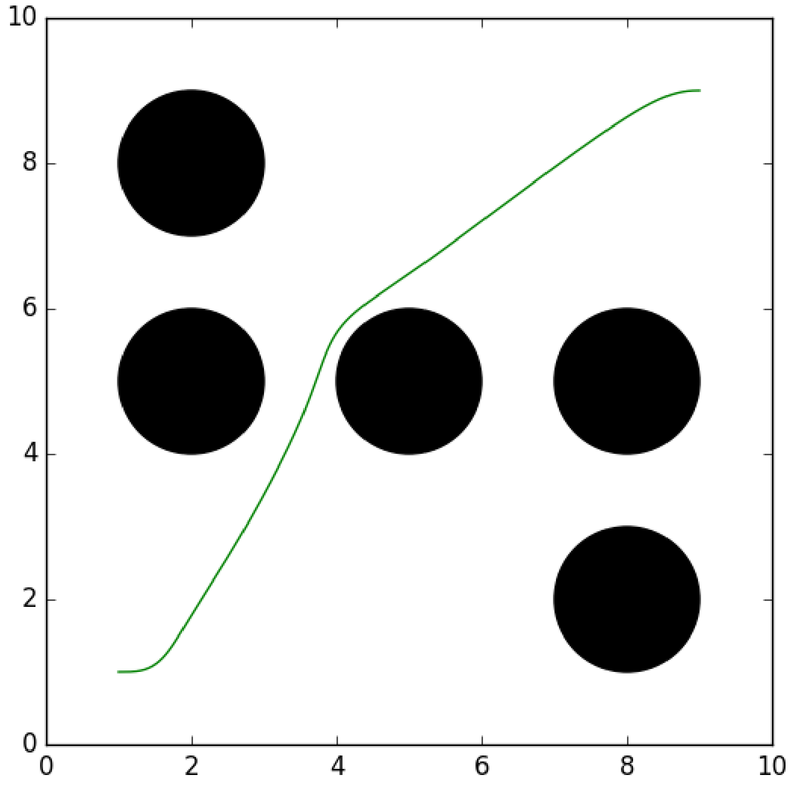

Figure 11: OPT planner navigation of map2

4.4 Close loop controller

By using Close loop controller we discuss above, we try it in our rrt planner in map1 and map2

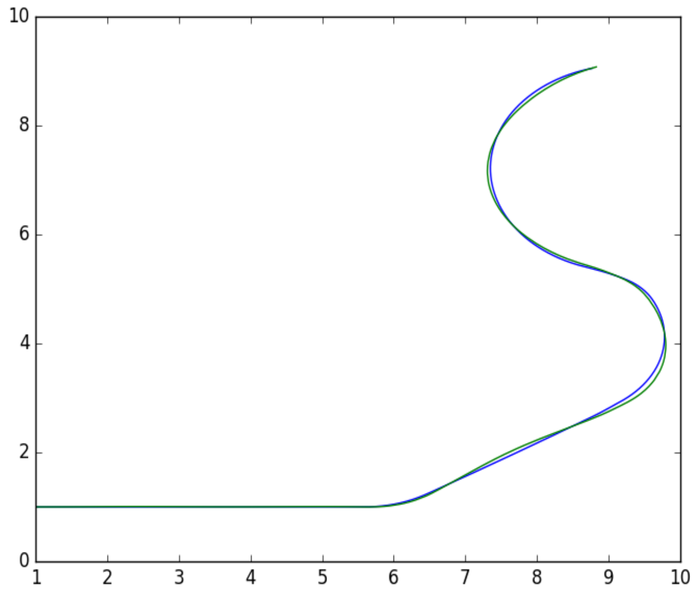

Figure 12: RRT planner navigation of map1 result

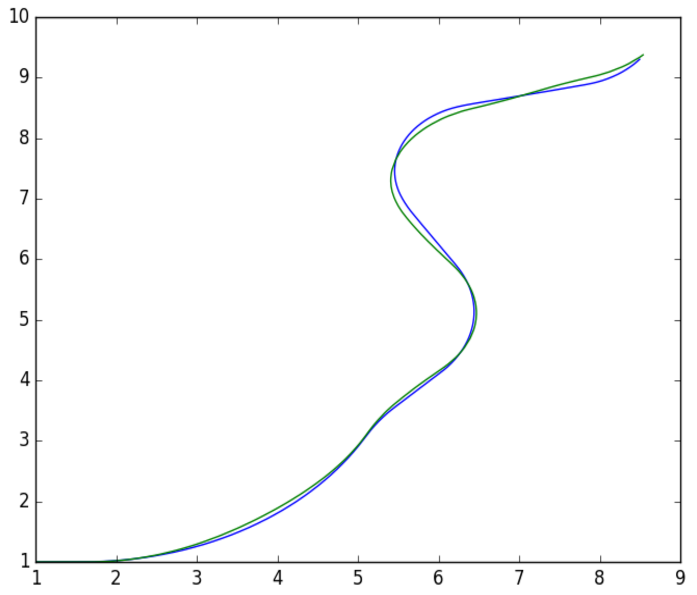

Figure 13: RRT planner navigation of map2 result

4.5 Discussions and comparisons

In our result above, We can easily conclude that:

1. RRT and Sinusoidal Planner can compute more fast than the optimize planner, which can be really helpful in real time tasks or dynamic environment planning.
2. In navigation tasks, optimize planner can generate the best path which can get the goal more fast and safe. However, the path generate by RRT sometimes have more dangerous action and long path compared with optimize planner. So we can use optimize planner when the navigation tasks has static environment.
3. In simple manipulation task, Sinusoidal Planner can generate the path fast and accurate. Optimize planner can generate the best path but waste lots of time. RRT can't deal with this problem well in most of time. Which means when our task space is small enough, we'd better use sinusoidal planner(without obstacle constraints) or Optimize planner (with obstacle constrains) rather than RRT planner
4. Sinusoidal Planner can't avoid obstacle sin we didn't add obstacle constraint in it. For further implementation, we can add the constraints on configuration states, and check the collision when solving the $a1$ and $a2$, if there is collision, we rerun the solver for another set of $a1$ and $a2$.
5. In real robot, we can't tracking the path well. Because there are lot's gap between simulations and real environment. The real robot model is different with the ideal robot. In real robot, there are acceleration limit and control delay. Besides, the state estimate isn't accurate enough compared with simulation, which will bring error into controller.

5 Difficulties

1. Sinusoids planner
   The switching of models is hard to implement. We have two separated models for the system, and both model have their singularities, so determined when and how to switch the model is not easy. Also we have some wired outputs of the states. You can see the figure below, at first, the states of θ and ϕ didn't move and in the last it moves quickly. It also exists in the simulator, it won't move for a moment and speed up suddenly.

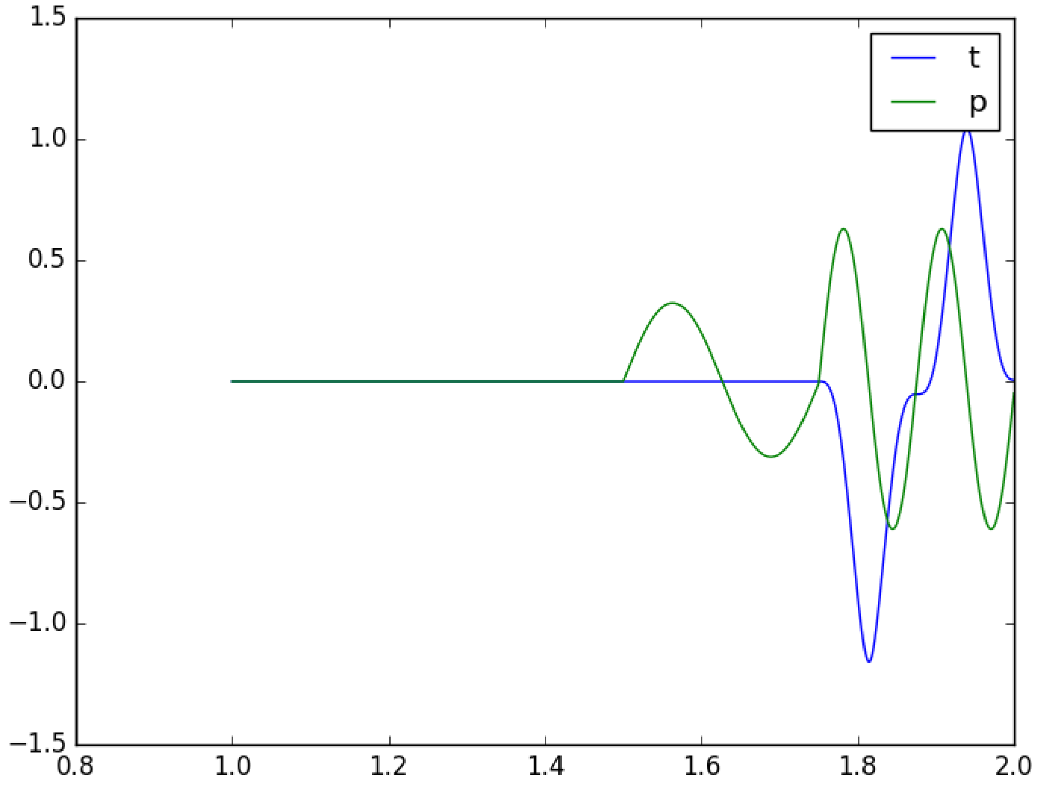

Figure 14: θ and ϕ in parallel parking by sinusoids planner

2. Optimization planner
   Firstly, when we implement the optimization planner for manipulation tasks, specifically parallel parking task. It takes quite a long time to solve the trajectory. So we considering the distance first, so modify N using the distance between start position and goal position. Then, when we facing the point_turn task, the distance is 0, so we modify our distance function. Taking angular distance of θ into consideration.

6 Documentation feedback

1. I meet some issues for install the dependencies and environments for package stdr-simulator. We can install ros-kinetic-navigation in the terminal first, and install stdr-simulator. That works for me.

2. For illustrating the relations between different python files, or some variable flows, we probably can use another method to demonstrate it. For example using the tree-like

graph to describe the relation ship between different of directories. Using rqt_graph-liked graph to describe the variable flows. That is more intensive than describing in words in Section 5.7.

3. We suggest to reduce the limit of velocity since the max velocity is too hard to control and works really bad in real robot since it need lot's time to accelerate.

4. For global planning, in the future, we can try to implement an A-star-like algorithms which can consider the nonholonomic model.

References

[1] Richard M Murray, Zexiang Li, S Shankar Sastry, and S Shankara Sastry. *A mathematical introduction to robotic manipulation.* CRC press, 1994.

[2] Richard M Murray and Sosale Shankara Sastry. Nonholonomic motion planning: Steering using sinusoids. *IEEE transactions on Automatic Control*, 38(5):700–716, 1993.

[3] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016.

[4] Valmik Prabhu, Chris Correa, Amay Saxena, and Tiffany Cappellari. Nonholonmic control with turtlebots. pages 1–19, 2020.