Motion Planning for Quadrotors

Group 28

Xianzhong Liu(5500559) Shaohang Han(5448557) Marco Schouten(5352908) Chenghao Xu(5266068)
MSc Robotics *MSc Robotics* *MSc Computer Science* *MSc Robotics*

***Abstract*— In this project, two pipelines are implemented to fulfill the motion planning of quadrotor model. One applies k-PRM* for waypoints generation, Minisnap for trajectory generation, Waypoints Insertion method and Corridor Bounding method for collision avoidance. The other pipeline directly generates trajectories using simplified Kinodynamics RRT*. Other functionalities such as dynamic window approach and dynamic obstacle avoidance are also implemented.**

Index Terms: motion planning, PRM*, minisnap, kinodynamics RRT*

I. Introduction

A. Goal and the task

This project's scope is to plan the motion of a quadrotor moving between a predetermined starting and endpoint in a three-dimensional world filled with static and dynamic obstacles.

B. State of the Art

Over the last decades, optimization-based methodologies tackling motion planning have been developed. They can be summarized into sampling and combinatorial motion planning. The most well-known sampling ideas good for high dimensions are Probabilistic Roadmaps (PRM), Rapidly-Exploring Random Trees (RRT). Combinatorial methods have the best results for low dimensional spaces. However, the solution often depends on the map's resolution, and they might still miss some corridors between obstacles.

C. Our Contribution

We used an open-source quadrotor environment [2]. Based on this environment, we developed our code in `https://github.com/MarcoSchouten/Planning_Project`. Our implementations are explained in detail in the following sections.

II. Model

A. System Dynamics

A free body diagram of a quadrotor is shown in fig.1 [6]. For each motor with angular speed ω_i, a vertical force and moment are produced by

$$F_i = k_F \omega_i^2, \quad M_i = k_M \omega_i^2 \tag{1}$$

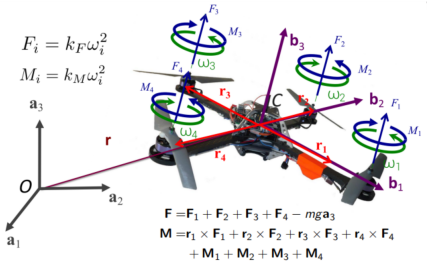

Fig. 1. Free body diagram of quadrator model

where k_F and k_M is constant given by experiment.

The total thrust force generated by motors is represented as $\mathbf{u_1}$. The moment vector in the body-fixed frame is defined as $\mathbf{u_2}$.

The equation of motion for a quadrotor are derived via Euler's equation:

$$\begin{aligned} m\ddot{\mathbf{r}} &= \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ u_1 \end{bmatrix} \\ I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} &= \mathbf{u_2} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix} \end{aligned} \tag{2}$$

where m is the mass of the quadrotor. $\mathbf{r}$ is the position vector of centre of gravity (COG) in the inertial frame. R is the change of bases matrix from the body frame to the inertial frame. I is the moment of inertia expressed along principle axes. p,q,r represent angular velocities in the body frame. The state of the dynamics system can be given by:

$$\mathbf{s} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, p, q, r]^T \tag{3}$$

B. Work Space and Configuration Space

The workspace of a quadrotor is $\mathbb{R}^3$. The configuration space is $\mathbb{R}^3 \times SO(3)$. The system state contains 12 variables, but many of them can be expressed by selected *flat outputs* and their derivatives. One possible choice of flat outputs is given by [5]:

$$\sigma = [x, y, z, \psi]^T. \tag{4}$$

This means it is sufficient to plan a trajectory with desired σ for the controller to follow. The generated smooth trajectory $\sigma(t)$ would then be in $\mathbb{R}^3 \times SO(2)$. In our

implementation, the the output of planner is $\mathbf{r}(t)$. Then, $\psi(t)$ is derived from $\dot{\mathbf{r}}(t)$.

III. Motion Planning

In III-A and III-B, we applied a classical motion planning pipeline, i.e. getting waypoints from a path planner (PRM*, RRT*, etc.) , then generating trajectory separately by connecting all waypoints. However, this pipeline does not consider kinodynamics during the path planning (waypoints generating) stage. Thus, we also implemented kinodynamics RRT* in III-C.

A. Path Planner

For the global path planner, we implemented the k-nearest variant of Optimal Probabilistic Roadmaps (PRM*). Our implementation differs from from the standard version by two main differences:

Firstly, instead of finding neighbours within a radius, k-nearest PRM* choose k neighbours with the smallest distance to the randomly sampled point. This was easy to implement using R-Tree data structure. Secondly, the number k of nearest neighbors is changing dynamically. It keeps proportional to $log(n)$ [4], where n is the number of sampled points in total .

After building the graph, we took advantage of the library NetworkX in Python to find the shortest path using Dijkstra algorithm. The pseudocode of k-PRM* is shown in algorithm 1.

Algorithm 1 k-PRM*

function kPRM(x_{init}, x_{goal}, W):
$V \leftarrow x_{init}, x_{goal} \bigcup Sample(W)_{i_{i=1,\ldots,n}}$
$E \leftarrow \emptyset$
for each $v \in V$ **do**
  $U \leftarrow Nearest(v, k = log(n))$
  for each $u \in U$ **do**
    if $CollisionFree(u,v)$ **then**
      $E \leftarrow E \bigcup (u,v),(v,u)$
    end if
  end for
end for
return $ShortestPath = Dijkstra(G = (V,E))$

B. Trajectory Generation

We follow the formulation in the lecture to generate minisnap trajectory [1] . But the original formulation does not consider obstacle avoidance. To generate collision-free trajectory, we implemented two methods. Given the collision-free waypoints, both methods generate collision-free minisnap trajectories but with different strategies.

1) Waypoint insertion method: This method firstly generates the minisnap trajectory using the formulation introduced in the lecture. The segment times are proportional to the distances between waypoints. If there is any collision between a particular segment of the trajectory, insert a new waypoint in the middle of the line segment connected by the two adjacent waypoints [7]. Then regenerate the trajectory again. This procedure is repeated iteratively until there is no collision on the trajectory. Adding more waypoints on the line segment gradually pushes the trajectory back to the original polyline path. Since the polyline path between each pair of waypoints is ensured to be collision-free, this method is guaranteed to find a collision-free trajectory.

2) Corridor bounding method: Instead of solving linear equations directly, we formulate the problem as a quadratic programming (QP) optimization problem. The cost function is the snap of the trajectory. We omit the continuity constraints on trajectory's higher-order derivatives. Instead of assigning time to each segment, from [7], it is possible to incorporate time allocation also into optimization. The final optimization problem formulation is:

$$
\begin{aligned}
\min_{\mathbf{c}} \quad & \sum_{i=1}^{m} \int_{T_{i1}}^{T_{i2}} P_{i,k}^{(4)}(t)^2 dt + \lambda_T \sum_{i=1}^{m} (T_{i2} - T_{i1}) \\
= \min_{c} \quad & c_k^\top Q_k c_k + \lambda_T \sum_{i=1}^{m} (T_{i2} - T_{i1}) \\
\text{s.t.} \quad & P_{i,k}^{(0)}(T_{i1}) = x_{i,k} \quad \forall i \in [1,m] \\
& P_{i,k}^{(0)} = x_{i+1,k} \quad \forall i \in [1,m] \\
& P_{i,k}^{(j)}(T_{i1}) = \begin{cases} 0, \text{ if } i = 1 \\ P_{i-1,k}^{(j)}(T_{i1}), \text{ otherwise} \end{cases} \\
& \forall i \in [1,m], j \in \{1,2,3\} \\
& P_{m,k}^{(j)}(T_{m2}) = 0 \quad \forall i \in [1,m], j \in \{1,2,3\} \\
& \mu_{i,k}^{\text{low}} \leqslant P_{i,k}(T) \leqslant \mu_{i,k}^{\text{up}} \quad \forall i \in [1,m]
\end{aligned}
\tag{5}
$$

where $\mathbf{c}$ is the column vector containing the polynomial coefficients to be optimized, m the number of segments, $P_{i,k}^{(j)}(t)$ the jth derivative of the ith polynomial segment of trajectory in dimension k. T_{i1} and T_{i2} are the time at the starting and end point of of the ith polynomial segment of trajectory. $x_{i,k}$ is specified waypoint that needed to be passed through. λ_T is a scalar representing penalty weight on the time term. In the inequality constraint, $\mu_{i,k}^{\text{low}} and \mu_{i,k}^{\text{up}}$ are the upper and lower bounds for each polynomial. Similar inequality constraints can also be imposed on the derivatives to limit velocity, acceleration and jerk,etc. To obtain bounds of the inequality constraints of the QP problem, We need to find collision-free corridors for each trajectory segment. The generation steps of such corridors are shown in figure 2, where the obstacles are

gray and corridors are green. We generate initial corridors by taking the every pair of waypoints as the corner points of a corridor. Then, to eliminate overlaps, we keep shrinking the corridors' size by iteratively bisecting. As shown in step 2, the collision free corridors are generated, but there are some small redundant corridors. They are then merged in step 3 by discarding unnecessary waypoints. The dimensions of corridors are dependent on the coordinates now. If the coordinates adjacent two waypoints do not change along one or two axes of the frame, then the corridor might collapse to a plane or line. So the corridors are then inflated, as shown in step 4.

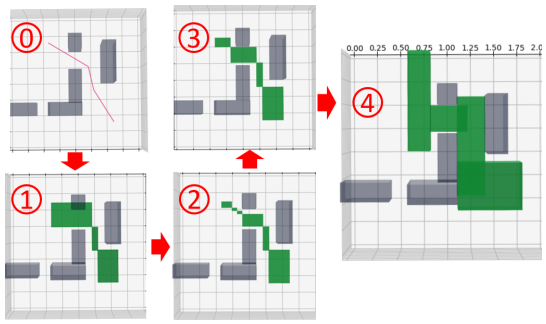

Fig. 2. Steps of generating corridors.

Now we can obtain the bounds from corridors. We impose the inequality constraint by taking many discrete points on the trajectory and limit them inside the corridors. The optimization problem is solved separately. It begins by guessing the segment times. Then, retrieves the cost by solving the minisnap problem with CVXOPT. By iteratively repeating this procedure, it refines the segment times using gradient descent.

C. Simplified Kinodynamics RRT*

The pseudocode of kinodynamics RRT* is shown in algorithm 2. We first sampled points with random position, velocity, and acceleration (line 3). Then we used high-order (10th in our code) polynomials as steer functions to connect the points. This was be achieved by formulating Quadratic Programming (QP) optimization problems and solving with CVXPY solver. Cost function was taken as the square of snap. In line 8, parent was chosen by selecting the one with lowest cost. We did not follow the method of solving *weighted controllability Gramian* [8], as it would be challenging to implement the optimal control problem formulation without using specifics solvers. We also neglected the forward/backward reachable set, and directly used Euclidean distance to find nearest neighbors instead.

D. Dynamic Obstacle Avoidance

Lastly, we need to prevent collisions against non-cooperative dynamic obstacles. To this end we designed

Algorithm 2 Simplified Kinodynamic-RRT*

1: **function** kRRT(x_{init}, x_{goal}):
2: **for** i = 1:n **do**
3: $x_{rand} \leftarrow Sample(\mathbf{X}, \dot{\mathbf{X}}, \ddot{\mathbf{X}})$
4: $x_{nearest} \leftarrow Nearest(Euclidean, x_{rand})$
5: $x_{new} \leftarrow Steer(x_{rand}, x_{near})$
6: **if** $CollisionFree(x_{new})$ **then**
7: $X_{near} \leftarrow k - Nearest(Euclidean, x_{rand})$
8: $x_{min} \leftarrow ChooseParent(T, x_{rand}, X_{near})$
9: $T.addNode(x_{rand})$
10: $T.rewire()$
11: **end if**
12: **end for**
13: return T // A trajectory T from x_{init} to x_{goal}

an additional module that gets activated after retrieving the desired trajectory for static environments.

- First, we sample a random velocity, from a triangular probability distribution slightly biased toward desired direction. In practice, this will lead to faster reaction times and keep the recovery overhead limited.
- Secondly we evaluate whether the sampled velocity will lead to a collision given the physical model of the drone and assuming constant velocity for the dynamic obstacle according to:

$$||(\mathbf{p_i} + t\mathbf{v}_i) - (\mathbf{p_j} + t\mathbf{v}_i)|| \leq r_i + r_j$$

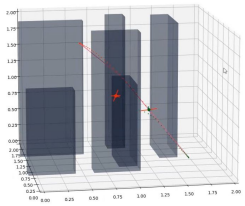

Fig. 3. Collison Avoidance

IV. Simulation and Results

We have tested both the k-PRM* pipeline (k-PRM*, minisnap, corridor bounding method) and the kinodynamics RRT* (kinoRRT*) method in random obstacle maps, as shown in figure 4. The blue lines denote a polyline connecting all the waypoints using straight lines. The red curves stand for the smooth trajectories. The green dots represent the real output trajectories of the nonlinear controller.

A. Random Map Generation

The random map is generated using Gaussian distribution over defined bounded area. Each grid value follows the standard normal distribution $\mathcal{N}(0,1)$. The grid value above the density threshold will be estimated as 1, while

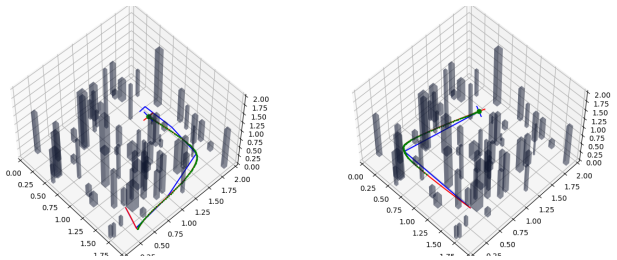

Fig. 4. Simulation Result of PRM* (left) and RRT (right) using random generated map.

others will be processed as 0. The density of obstacle distribution is determined by one density parameter with a randomized map generated every time. The size and altitude of each obstacle are also randomized to prove the generalizability of our implementations.

B. Comparison of Different Planners

A comparison of various planners is shown in figure 5. The metrics we used are the total length of a trajectory, the flying time, the time needed to plan a trajectory. The blue line denotes k-PRM* pipeline and the orange lines stand for kinoRRT*. It is shown that k-PRM* pipeline has lower planning time and more stable trajectory quality.

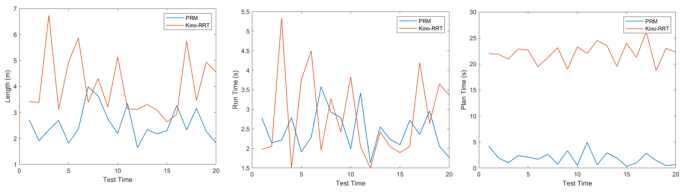

Fig. 5. Comparison of Different Planners, using Trajectory Length (left), Flying Time (middle), and Planning Time (right).

V. Discussion

A. k-PRM* pipeline vs. kinoRRT*

k-PRM* is asymptotic optimal and probabilistic complete [4]. It's time complexity is $O(nlog(n))$. However, k-PRM* does not have monotone convergence. This could be a disadvantage compared with other path planning algorithms, such as Informed-RRT*. As for Kinodynamics RRT*, it is unclear whether our implementation upholds the asymptotic optimal property of RRT*, as we used the polynomial steering function. Practically, k-PRM* with minisnap trajectory has better planning time and trajectory quality results. This could be because kinoRRT* samples randomly in velocity and acceleration, meaning many sample points are challenging to connect and thus useless. However, the best neighbour has to be found by solving multiple QP problems for each sample point. This makes our implementation of kinoRRT* inefficient. In contrast, k-PRM* pipeline plans waypoints using Euclidean distance, which is computationally inexpensive. Minisnap only has to solve optimization problems a few times. This could be the reason behind the result shown in figure 5.

B. Safety Corridor vs. Waypoint Insertion

Trajectory optimization: the discussion mainly focuses on the performance, computation time and robustness of the two methods mentioned in III-B.

1) Performance: The corridor bounding method generally gives the best performance with lower snap cost and shorter time to reach the goal. During the merging step of corridor generation, some unnecessary waypoints produced by the global planner can be discarded, making the trajectory smoother and more directly towards the goal. The waypoints insertion method has the result going through all waypoints, which yields wiggled trajectory and thus higher snap cost. It did not incorporate time optimization either, yielding longer flying time.

2) Computation time: The corridor bounding method requires a longer computation time during the simulation test. Because it solves the QP problem multiple times to iteratively optimize time allocation. The waypoint insertion method requires less time. But in some complex environments, where many waypoints need to be inserted along the path, it potentially takes the same or even more time than the corridor bounding method. A new waypoint is inserted because it solves more complex linear equations every time.

3) Robustness: The corridor bounding method is not robust since the magnitude of the constraint matrix differs by several orders of magnitude. This causes a problem of numerical stability for the CVXOPT solver. But several techniques are implemented to improve the numerical stability, such as scaling the map and the constraints. Limiting the velocity of the quadrotor also improves the robustness. The waypoint insertion method always finds a solution. Even in a bad case where many waypoints need to be inserted, it returns a solution that degenerates to the polyline path.

VI. Future Works

Our implementation of k-PRM* used R-Tree. However, other data structures, such as KD-Tree, and variants of PRM could be futher inspected.

The corridor bounding method now trivially imposes inequality constraints on many discrete points on the trajectory segments. From [3], we have learned that using B-spline is a better way because the inequality constraints can be imposed on just a few control points to generate a collision-free path.

For kinoRRT*, more optimal control methods could connect the sampled nodes, as well as developing reachable sets in the original paper [8].

REFERENCES

[1] Javier Alonso-Mora. ro47004, planning and decision-making, lecture 11: Trajectory optimization. 2021.

[2] Bharath Chandra. Quadrotor simulation. `https://github.com/Bharath2/Quadrotor-Simulation`, 2021.

[3] Fei Gao, William Wu, Yi Lin, and Shaojie Shen. Online safe trajectory generation for quadrotors using fast marching method and bernstein basis polynomial. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 344–351, 2018.

[4] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[5] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation*, pages 2520–2525. IEEE, 2011.

[6] Wei Pan. ro47001, dynamics and control. 2021.

[7] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. *Masayuki Inaba*, page 649, 2016.

[8] Dustin J Webb and Jur Van Den Berg. Kinodynamic rrt*: Asymptotically optimal motion planning for robots with linear dynamics. In *2013 IEEE international conference on robotics and automation*, pages 5054–5061. IEEE, 2013.