Exploiting collisions for sampling-based multicopter motion planning

Jiaming Zha and Mark W. Mueller

***Abstract*— Multicopters with collision-resilient designs can operate with trajectories involving collisions. This paper presents a sampling-based method that can exploit collisions for better motion planning. The method is built upon the basis of the RRT* algorithm and takes advantages of fast motion primitive generation and collision checking for multicopters. It generates collision states by detecting potential intersections between motion primitives and obstacles, and connects these states with other sampled states to form collision-inclusive trajectories. We show that allowing collision helps improve the performance of the sampling-based planner in narrow spaces like tunnels. Finally, an experiment of tracking the trajectory generated by the collision-inclusive planner is presented.**

Fig. 1: The collision-resilient tensegrity multicopter (left) can still operate after colliding with a concrete wall with speed up to 6.5m/s (right). We use it as a test platform for the collision-inclusive planner.

I. Introduction

Autonomous systems often need to find feasible trajectories between desired states. One effective approach to the problem is to sample states in the state space and connect them with feasible (in terms of both input-feasible and collision-free) trajectory pieces. Methods using this approach, such as rapidly exploring random trees (RRT) [1] and probabilistic road maps (PRM) [2], are referred to as sampling-based methods. In particular, RRT*, a variant of RRT, has gained great popularity due to its unique asymptotic optimality characteristic [3]. Researchers have extended the RRT* algorithm to work with dynamic systems with differential constraints [4] and have developed different sampling methods [5] and heuristics guiding the sampling process [6], [7] to increase RRT*'s rate of convergence.

One key step of sampling-based motion planning is checking collisions. Trajectories colliding with obstacles are usually discarded by planners. However, in recent years, many autonomous systems that can survive collisions have been developed, such as the collision-resilient aerial vehicles in [8] and [9]. As these vehicles can fly trajectories that are not collision-free, they can plan their motion in an extended feasible state space, which usually leads to better trajectories with less cost or duration time. This idea of exploiting collisions for better trajectories is discussed in [10], which also proposes a method to find collision-inclusive optimal trajectories with mixed integer programming. The method is later experimentally verified in [11]. Furthermore, collision is shown to be beneficial to stochastic optimal steering problems, as contact with environment helps decrease the uncertainty of the state estimation [12].

We are interested in generating trajectories in the extended collision-inclusive state space with a sampling-based method for multicopters. Our method is based on RRT* algorithm and takes advantage of the differential flatness of multicopter dynamics, which enables rapid generation of minimum jerk motion primitives [13] and efficient collision detection [14]. Via detecting collisions and predicting the states of the vehicle after the collisions, the method can generate collision states and potentially connect them with other sampled states to form collision-inclusive trajectories.

Planning with collisions introduces an interesting trade-off. On one hand, more computation time is required for detecting and generating states with collisions. On the other hand, the planner will no longer discard samples due to collisions. Hence, the planner can add samples to its exploring tree more efficiently. We find that the benefit of planning with collision is likely to outweigh its computation cost in narrow environments such as tunnels, where collision is likely to take place and collision-exclusive planners are forced to discard most of their samples.

The contribution of this paper is as follows. We present a sampling-based motion planner that can find collision-inclusive trajectories for multicopters. We demonstrate that the planner is likely to generate better result than collision-exclusive planners in narrow spaces with a Monte Carlo test on an example environment featuring a tunnel. We have also experimentally validated the planned collision-inclusive trajectory with our tensegrity multicopter [15], shown in Fig. 1. Our method can also be adapted to work with other collision-resilient multicopters by replacing the collision model in this paper with other vehicle-specific models.

II. Motion primitive and collision detection

This section introduces two important building blocks of the collision-inclusive motion planner, the motion primitive generator for multicopters and the rapid collision detector for such primitives. These tools are originally presented in [13] and [14]. Here we only introduce the key results and explain how they are adapted to work with our motion planner.

The authors are with the HiPeRLab, University of California, Berkeley, CA 94720, USA. {jiaming_zha,mwm}@berkeley.edu

A. Multicopter motion primitive generator

The multicopter motion primitive generator is a computationally efficient tool that can generate and check the input-feasibility of about one million motion primitives in a second on a modern computer. This enables us to rapidly connect sampled states to form feasible trajectories.

The generator computes a thrice differentiable trajectory which guides the multicopter from an initial state at time t_0 to a final state at time t_f, while minimizing the cost function

$$J = \int_{t_0}^{t_f} ||\boldsymbol{j}(t)||^2 dt \tag{1}$$

where $\boldsymbol{j}(t)$ is the jerk of the multicopter at time t and is defined as the third order derivative of the position. The cost J can be interpreted as the upper bound of the product of the norm of inputs (in terms of total thrust and angular velocity) to the multicopter system [13]. Hence, a trajectory with lower cost tends to be less aggressive and is more likely to be input-feasible.

The trajectory generated is a fifth order polynomial:

$$\boldsymbol{x}(t) = \boldsymbol{a}_0 t^5 + \boldsymbol{a}_1 t^4 + \boldsymbol{a}_2 t^3 + \ddot{\boldsymbol{x}}_0 t^2 + \dot{\boldsymbol{x}}_0 t + \boldsymbol{x_0} \tag{2}$$

where $\boldsymbol{x}(t)$ is the position at time t and $\boldsymbol{x}_0$, $\dot{\boldsymbol{x}}_0$, $\ddot{\boldsymbol{x}}_0$ are position, velocity and acceleration at the start of motion primitive. Vector parameters $\boldsymbol{a}_0$, $\boldsymbol{a}_1$ $\boldsymbol{a}_2 \in \mathbb{R}^3$ describe the trajectory, and are solved as linear functions of the initial state and the final state.

Let $\mathcal{P}$ = MOTIONPRIMITIVE$((\boldsymbol{s}_0, t_0), (\boldsymbol{s}_f, t_f))$ be the function generating the motion primitive connecting state $\boldsymbol{s}_0$ and $\boldsymbol{s}_f$ between time t_0 and t_f. A state is defined as the combination of position, velocity and acceleration of the vehicle. We define $\boldsymbol{C}(\mathcal{P})$ as the function evaluating the cost of the motion primitive with Eq. (1). For each generated motion primitive, we can check if the inputs for the multicopter to implement the primitive satisfy bounds on the minimum and maximum total thrust and the magnitude of the angular velocity. We define INPUTFEASIBLE$(\mathcal{P})$ as the function checking the input feasibility of the motion primitive. A method for quickly implementing this check is given in [13], where we refer the reader for further reading.

B. Rapid Collision Detection

The rapid collision detection algorithm [14] checks if a generated motion primitive will collide with convex obstacles in the environment. Non-convex obstacles may be approximated by defining them as a union of convex obstacles.

For a given motion primitive starting at time t_0 and ends at time t_f, the algorithm first checks if its begin position $\boldsymbol{x}(t_0)$, end position $\boldsymbol{x}(t_f)$ and the mid-time position $\boldsymbol{x}(t_{split})$ is inside an obstacle. Here $t_{split} = (t_0 + t_f)/2$. If not, it then checks if the primitive crosses a separation plane between the obstacle and $\boldsymbol{x}(t_{split})$. This separation plane is defined as the tangential plane of obstacle surface that includes a point $\boldsymbol{p}$, which is located in the obstacle and has minimum distance to $\boldsymbol{x}(t_{split})$. If the whole primitive does not cross the separation plane, it is guaranteed to be free of collision. If the primitive crosses the separation plane, we can bisect it into two sections, $\boldsymbol{x}(t_0) \rightarrow \boldsymbol{x}(t_{split})$ and $\boldsymbol{x}(t_{split}) \rightarrow \boldsymbol{x}(t_f)$ and repeat the above checking process on the section(s) crossing the separation plane. An illustration of this idea is shown in Fig 2.(a) and Fig 2.(b).

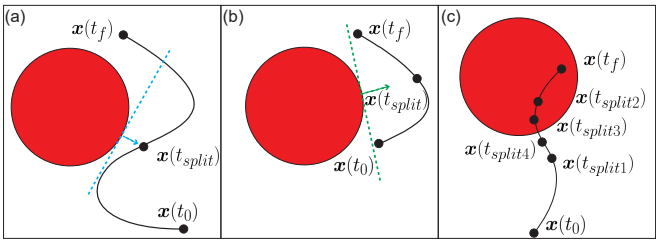

Fig. 2: A graphic illustration of key ideas of the collision detection process. (a). The detector checks if the primitive stays on the same side of the separation plane. If not, it bisects the primitive and checks either half that crosses the plane. (b). If the primitive piece stays on the same side of the separation plane, we know the primitive is free of collision. (c). We can estimate the collision time by keep bisecting the primitive until the section crossing the separation plane is shorter than a certain duration threshold.

In [14], the recursion terminates once any part of the primitive is found lying inside the obstacle. We modify the recursion termination rule to compute the time of the collision. When checking for collision, we can keep bisecting the primitive until the time of the checked primitive section crossing the separation plane is smaller than a certain threshold. We can thus estimate the time of collision as the beginning time of this primitive section, as shown in Fig. 2.(c). We define COLLISIONFREE$(\mathcal{P})$ as the function checking if the primitive collides with any obstacle in the space. We also denote COLLISIONTIME$(\mathcal{P})$ as the function finding the time of the first collision between a given primitive and the obstacles in the environment.

III. ALGORITHM DESCRIPTION

This section introduces the algorithm of the sampling-based method for collision-inclusive motion planning. The algorithm searches for a potentially collision-inclusive trajectory connecting the start state $\boldsymbol{s}_0$ and the goal state $\boldsymbol{s}_f$.

The algorithm is based on the RRT* [3], but with three key differences. First, a collision state generation step is introduced to allow states involving collisions to be considered. Second, instead of connecting sampled states with lines, we connect them with motion primitives described in Section II. Third, we pair each sampled state with a sampled time. The corresponding time difference between two states is the duration of the primitive connecting them, which determines the input-feasibility of the primitive. Within a given computation time limit, the algorithm keeps sampling and connecting state-time pairs to generate trajectories in the class of concatenated minimum jerk primitives. The fastest candidate among all generated trajectories connecting $\boldsymbol{s}_0$ and $\boldsymbol{s}_f$ is then returned as the best solution found by the planner. For the rest of this section, we will first introduce how key steps of the original RRT* algorithm are modified to find collision-inclusive trajectories for multicopters. Then, we will present the whole planner and discuss the benefits and disadvantages of planning with collisions.

A. Collision model

We define the state of the vehicle by its position, velocity and acceleration, i.e. $\boldsymbol{s} = [\boldsymbol{x}, \dot{\boldsymbol{x}}, \ddot{\boldsymbol{x}}]$. Given a pre-collision state $\boldsymbol{s}$, the function CollisionModel($\boldsymbol{s}$) predicts the post collision state $\boldsymbol{s}^+ = [\boldsymbol{x}^+, \dot{\boldsymbol{x}}^+, \ddot{\boldsymbol{x}}^+]$. This model depends on the vehicle design and the material of contact surface and may vary among different multicopters. A model for the vehicle in Fig. 1 is given in Section V.

B. Connecting nodes as state-time pairs

We define a node $\boldsymbol{n}$ as a pair of vehicle state and end time, $\boldsymbol{n} = (\boldsymbol{s}, t)$. This indicates that it takes the vehicle time t to reach $\boldsymbol{s}$ from the start node $(\boldsymbol{s}_0, 0)$, where $\boldsymbol{s}_0$ is the start state of the planning problem. When a node represents a state at collision, it will be coupled with a post-collision node containing the same time and a post-collision state predicted by the collision model. Denote $\boldsymbol{PC}(\boldsymbol{n})$ as the function accessing the post-collision node of $\boldsymbol{n}$. Define Connect($\boldsymbol{n}_1, \boldsymbol{n}_2$) as the process of generating the motion primitive connecting two nodes, as shown in Algorithm 1. Moreover, we define a path between two nodes as a set of primitives connecting the nodes and the cost of the path as the sum of the costs of the primitives the path contains. We further define the cost of node, cost($\boldsymbol{n}$) as the lowest cost of the feasible path found connecting the start node to $\boldsymbol{n}$.

Algorithm 1 Connect nodes

```
1: function Connect(n_1, n_2)
2:     assert time of n_1 is before time of n_2
3:     if n_1 is a collision node then
4:         return MotionPrimitive(PC(n_1), n_2)
5:     else
6:         return MotionPrimitive(n_1, n_2)
```

C. Generating sample states

For each step, the planner generates a node sample $\boldsymbol{n}_s$ via a random process. With possibility η_f, the goal sampling rate, the state of $\boldsymbol{n}_s$ is set as the goal state and with possibility (1- η_f), the state of $\boldsymbol{n}_s$ is uniformly sampled from the state space. The time t of $\boldsymbol{n}_s$ is sampled uniformly from $[0, t^*_{end}]$ where t^*_{end} is the time of the shortest feasible trajectory from the start state to the goal state the algorithm has found. This value is initiated with an overestimate of the shortest feasible trajectory time and then updated throughout the planning. We use function Sample() to denote the above process.

D. Collision node generation

After the sampled node is generated, we search for a node in $\mathcal{T}$ whose time is before the sampled node and can be connected to it with a primitive of the lowest cost. Define ClosestNode($\mathcal{T}, \boldsymbol{n}_s$) as the function of finding such node.

Then, we check if the primitive connecting the closest node to the sampled node collides with any obstacle in the environment. If the primitive is free of collision, we use the sampled node directly for future optimal connection attempts. However, if the primitive collides with any obstacle in the environment, we generate a collision node with the time and the state of the vehicle right before the collision. Meanwhile, we use the collision model to predict the post-collision state. This process is illustrated in Fig. 3 and we define the process as the GetCollisionNode($\boldsymbol{n}_s$) function, with its implementation detailed in Algorithm 2.

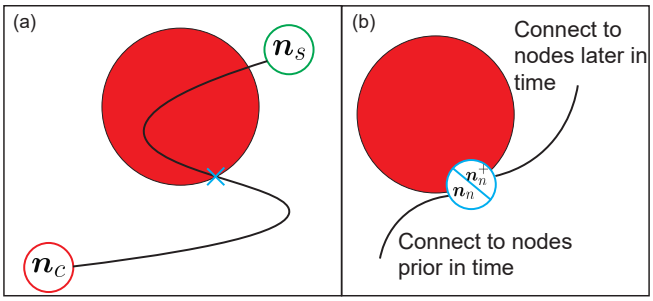

Fig. 3: A graphic illustration of the collision node generation process. After node $\boldsymbol{n}_s$ is sampled, we connect it with its closest node $\boldsymbol{n}_c$. If a collision takes place, we generate a pre-collision node with the time and the state right before the collision. We also predict its state after the collision and set it as the post-collision node for possible future connections.

Algorithm 2 Collision node generation

```
1: function GetCollisionNode(n_s)
2:     n_c ← ClosestNode(T, n_s)
3:     P ← Connect(n_c, n_s)
4:     if CollisionFree(P) then n_n ← n_s
5:     else
6:         t_n ← CollisionTime(P)
7:         s_n ← P(t_n)
8:         n_n ← (s_n, t_n)
9:         PC(n_n) ← (CollisionModel(s_n), t_n)
10:    return n_n
```

E. Connect along minimum cost path and rewire

After sampling and collision node generation, we want to connect the generated node to a best feasible parent node in $\mathcal{T}$ so that its cost is minimized. If such a feasible parent can be found, we will add the generated node to $\mathcal{T}$. Afterwards, we rewire the tree to ensure the optimality of all connections.

Notice that comparing to the original RRT*, we are not generating a "near neighbor" set to decrease the number of connections, because the metric of "distance" between two nodes is the cost of the primitive connecting them and such cost cannot be calculated before the connection attempt. However, for planning tasks that involve a large number of nodes, we can use a heap, $\mathcal{H}$ to track k-smallest costs of feasible primitives found so far. Following the suggestion of [3], we set $k = 2e \cdot log(|\mathcal{T}|)$, where $|\mathcal{T}|$ is the cardinality of $\mathcal{T}$. If a primitive candidate has a cost exceeding the largest value in the heap, it will be discarded without the feasibility check. This pre-screening helps decrease computation time and has an effect similar to the "near neighbor" screening in the original RRT*. The whole process of connecting and rewiring the exploring tree is shown in Algorithm 3.

Algorithm 3 Connect along minimum cost path and rewire

function CONNECTMINCOSTPATH($\boldsymbol{n}_n$)
  Initialize an empty heap $\mathcal{H}$.
  $k \leftarrow 2e \cdot log(|\mathcal{T}|)$
  PARENT($\boldsymbol{n}_n$) $\leftarrow$ *null*
  COST($\boldsymbol{n}_n$) $\leftarrow \infty$
  for $\boldsymbol{n}$ in $\mathcal{T}$ with time prior to $\boldsymbol{n}_n$ **do**
    $\mathcal{P} \leftarrow$ CONNECT($\boldsymbol{n}, \boldsymbol{n}_n$)
    if $\boldsymbol{C}(\mathcal{P}) > max(\mathcal{H})$ **then continue**
    if not INPUTFEASIBLE($\mathcal{P}$) **then continue**
    if not COLLISIONFREE($\mathcal{P}$) **then continue**
    if COST($\boldsymbol{n}$) + $\boldsymbol{C}(\mathcal{P})$ < COST($\boldsymbol{n}_n$) **then**
      COST($\boldsymbol{n}_n$) $\leftarrow$ COST($\boldsymbol{n}$)+ $\boldsymbol{C}(\mathcal{P})$
      PARENT($\boldsymbol{n}_n$) $\leftarrow \boldsymbol{n}$
    Push $\boldsymbol{C}(\mathcal{P})$ into $\mathcal{H}$
    if $|\mathcal{H}| > k$ **then** pop the largest value in $\mathcal{H}$
  if PARENT($\boldsymbol{n}_n$) is not *null* **then**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{\boldsymbol{n}_n\}$
  return
function REWIRE($\boldsymbol{n}_n$)
  Initialize an empty heap $\mathcal{H}$.
  $k \leftarrow 2e \cdot log(|\mathcal{T}|)$
  for $\boldsymbol{n}$ in $\mathcal{T}$ with time after $\boldsymbol{n}_n$ **do**
    $\mathcal{P} \leftarrow$ CONNECT($\boldsymbol{n}_n, \boldsymbol{n}$)
    if $\boldsymbol{C}(\mathcal{P}) > max(\mathcal{H})$ **then continue**
    if not INPUTFEASIBLE($\mathcal{P}$) **then continue**
    if not COLLISIONFREE($\mathcal{P}$) **then continue**
    if COST($\boldsymbol{n}_n$) + $\boldsymbol{C}(\mathcal{P})$ < COST($\boldsymbol{n}$) **then**
      COST($\boldsymbol{n}$) $\leftarrow$ COST($\boldsymbol{n}_n$)+ $\boldsymbol{C}(\mathcal{P})$
      PARENT($\boldsymbol{n}$) $\leftarrow \boldsymbol{n}_n$
      Update the cost of descendants of $\boldsymbol{n}$
    Push $\boldsymbol{C}(\mathcal{P})$ into $\mathcal{H}$
    if $|\mathcal{H}| > k$ **then** pop largest value in $\mathcal{H}$
  return

F. Full collision-inclusive sampling-based planner

Now, we combine the previous parts and present the full planner as Algorithm 4. After running the planner, we find the best end node as the node in $\mathcal{T}$ with an end state that has a smallest end time. After identifying the best end node, we can recover the best trajectory via backtracking from the best end node to the start node.

Planning with collision brings two benefits. First, allowing collisions extends the feasible state space that the sampling-based planner can search in. Second, sampled nodes will not be discarded due to infeasibility caused by collisions. Hence, the collision-inclusive planner may add nodes to $\mathcal{T}$ more efficiently.

However, planning with collision also comes with disadvantages. First, the process of generating collision nodes takes additional computation time. Second, due to the collision node generation process, sampled nodes may concentrate near obstacle surfaces facing the exploring tree. As a result, expansion of the exploring random tree towards the other side of the obstacle may be slowed down. An example is illustrated in Fig. 4. Nodes $\boldsymbol{n}_a$ and $\boldsymbol{n}_b$ are in $\mathcal{T}$ and a node $\boldsymbol{n}_s$ is sampled on the right side of the obstacle. In Fig. 4.(a), with the collision node generation process, $\boldsymbol{n}_s$ will be connected with its closest node, $\boldsymbol{n}_a$ and a collision node is generated on the left side of the obstacle. If the process is disabled, as in Fig. 4.(b), the sampled node will be connected with $\boldsymbol{n}_b$ and added directly to $\mathcal{T}$. We see that with the collision node generation, $\mathcal{T}$ can no longer reach the right of obstacle at this step and its expansion is slowed down in this example.

Algorithm 4 Collision-inclusive sampling-based planner

input: Start state $\boldsymbol{s}_0$, goal state $\boldsymbol{s}_f$, set of obstacles $\mathcal{O}$, state space $\boldsymbol{S}$, goal-sampling rate η_f
$\boldsymbol{n}_0 \leftarrow (\boldsymbol{s}_0, 0)$
PARENT($\boldsymbol{n}_0$) $\leftarrow$ *null*
COST($\boldsymbol{n}_0$) $\leftarrow 0$
$\mathcal{T} \leftarrow \{\boldsymbol{n}_0\}$
while computation time < planning time limit **do**
  $\boldsymbol{n}_s \leftarrow$ SAMPLE()
  if $\boldsymbol{n}_s$ is not a goal node **then**
    $\boldsymbol{n}_n \leftarrow$ GETCOLLISIONNODE($\boldsymbol{n}_s$)
  else
    $\boldsymbol{n}_n \leftarrow \boldsymbol{n}_s$
  CONNECTMINCOSTPATH($\boldsymbol{n}_n$)
  if $\boldsymbol{n}_n$ is added to $\mathcal{T}$ **then** REWIRE($\boldsymbol{n}_n$)

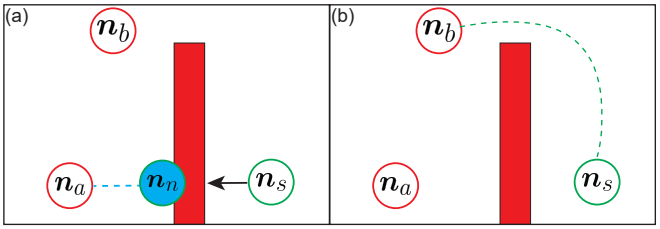

Fig. 4: Illustration of how the collision node generation process can slow down the expansion of the tree. $\boldsymbol{n}_a$ and $\boldsymbol{n}_b$ are nodes in $\mathcal{T}$. The planner generates a sampled state $\boldsymbol{n}_s$ that can be connect to $\boldsymbol{n}_a$ with a lower cost. (a). During the collision node generation process, a collision node $\boldsymbol{n}_n$ will be generated on the left of the obstacle and added to $\mathcal{T}$. (b). If the collision node generation is disabled, $\boldsymbol{n}_s$ will be added to $\mathcal{T}$ directly. As a result, $\mathcal{T}$ can expand to the right of the obstacle in this step.

IV. ILLUSTRATIVE EXAMPLE: PLAN IN A TUNNEL

In this section, we present an illustrative example to showcase the trade-off between the cost and the benefits of planning with collisions and illustrate why the collision-inclusive planner may perform better than collision-exclusive planner in narrow spaces.

In this 2D example, the vehicle starts at position (1, 2)[m] and is in a 1m wide tunnel. It needs to move in the positive x-direction for 3.5 meters to leave the tunnel and then move in an open space to get to a goal position at (4, 5)[m]. We run both the collision-inclusive planner and the collision-exclusive planner on this problem for 10^5 times and compare the results generated.

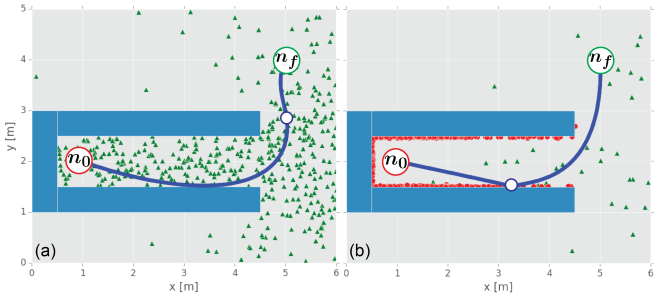

Fig. 5: Snapshots after (a). collision-exclusive planner and (b). collision-inclusive planner running for 0.1s. Red dots are the feasible collision nodes in $\mathcal{T}$ and green triangles are the feasible non-collision nodes in $\mathcal{T}$. The blue curve is the feasible trajectory connecting $\boldsymbol{n}_0$ to $\boldsymbol{n}_f$ with the shortest time found. The blue circle on trajectory is an intermediate node.

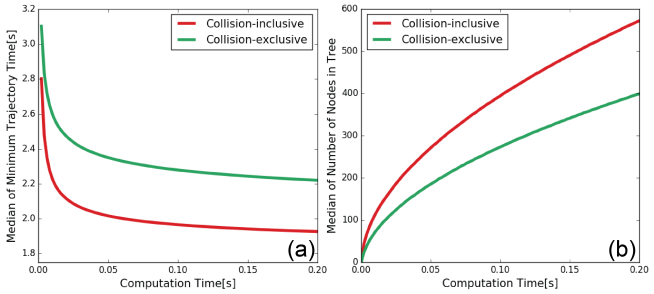

Fig. 6: Comparison of the performance of planners after running the problem for 10^5 times. (a). Median of the minimum trajectory time vs computation time. (b). Median of the number of the nodes in $\mathcal{T}$ vs computation time.

A snapshot of the planning result after computing for 0.1s is shown in Fig. 5. We observe that the collision-inclusive planner generates a trajectory colliding with the tunnel wall. Moreover, its nodes in $\mathcal{T}$ are mainly collision nodes. This phenomenon is expected. The tunnel is very narrow, so most of the primitives generated involve collisions. As a result, collision nodes are likely to be generated during the planning process. Moreover, if we compare the number of collision-free nodes in the open space right of the tunnel, we notice that the collision-exclusive planner generates more nodes in the open space than the collision-inclusive planner, indicating a faster expansion of the exploring tree in that region. This echoes the discussion in Section III that the collision node generation process can slow down the expansion of the exploring tree.

Now, we compare the performance of the planners with the median of the minimum feasible trajectory time the planners have found among all trials. We use median, instead of mean, to filter out the influence of outliers. Fig. 6.(a) plots the median of the best trajectory time found versus computation time. It shows that the collision-inclusive planner finds a better trajectory than the collision-exclusive planner under the same time limit. The relatively better performance can be credited to the two benefits of collision-inclusive planning. First, allowing collisions extends the feasible state space. Second, collision-inclusive planner can add nodes to $\mathcal{T}$ more efficiently, because no nodes are discarded due to infeasibility caused by collision. Fig. 6.(b) shows the median of the numbers of nodes in $\mathcal{T}$ versus computation time. The figure shows that collision-inclusive planner can add nodes to $\mathcal{T}$ with a higher speed. The relatively low efficiency of the collision-exclusive planner suggests that most of its sampled nodes are deemed as infeasible due to collisions and cannot be added to $\mathcal{T}$. This example shows that the collision-inclusive planner is likely to outperform the collision-exclusive planner when the vehicle is in narrow spaces, which are usually the scenarios that collision-resilient vehicles are designed for.

V. Experimentally Tracking Planned Trajectories

This section demonstrates the experiment of tracking planned collision-inclusive and collision-exclusive trajectories generated by our planner with a collision-resilient tensegrity multicopter [15]. The video of the experiment can be found in the attachment.

A. Collision model for the test platform

We predict that the post-collision position stays the same as the pre-collision position, $\boldsymbol{x}^+ = \boldsymbol{x}$. For post collision velocity, we use an empirical model similar to the one in [16] for the non-sliding case. The model predicts the velocity component normal to the obstacle with a linear function:

$$\dot{x}_n^+ = -e\dot{x}_n \tag{3}$$

Where e is the coefficient of restitution and $\dot{x}_n$ and $\dot{x}_n^+$ are velocity component normal to the obstacle before and after the collision. Moreover, the model predicts that the ratio between tangential impulse and normal impulse is proportional to the incidence angle, which is the angle between the pre-collision velocity vector and the normal vector of the obstacle surface. As a result, we have

$$\dot{x}_t^+ = \dot{x}_t + \kappa(-e-1)\arctan\left(\frac{\dot{x}_t}{\dot{x}_n}\right)\dot{x}_n \tag{4}$$

where κ is a constant. $\dot{x}_t$ and $\dot{x}_t^+$ are velocity component tangential to the obstacle before and after the collision. With experiments, we identify $e = 0.43$ and $\kappa = 0.20$ for our model.

The post-collision acceleration is dependent on the attitude of the vehicle after collision, which can be hard to predict due to the large torque disturbance the vehicle may experience during the collision process. As a result, we assume $\ddot{\boldsymbol{x}} = \boldsymbol{0}$, which corresponds to a hovering status, and treat the difference between the true attitude after collision and the hovering attitude as an initial attitude error for the trajectory piece after the collision. As multicopters have responsive attitude controllers, this error is expected to be corrected in a negligible time.

B. Improving the tracking of collision trajectories

Due to the torque disturbance during the collision, the vehicle can rotate with a large angular velocity after the collision. To mitigate the tracking error caused by this, we temporarily (for 0.3s) increase the gains of the attitude and angular rates controller of the vehicle after the collision.

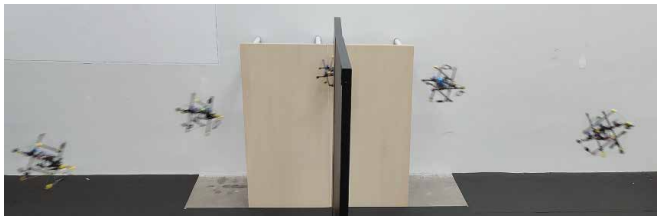

Fig. 7: Image sequence of the tracking experiment. The multicopter tracks a trajectory from left to right while avoiding the black obstacle in the middle of the space. For the collision-inclusive trajectory, the multicopter takes advantage of a collision with the yellow obstacle in the back. The distance between the shown left-most state and right-most state is about 2m.

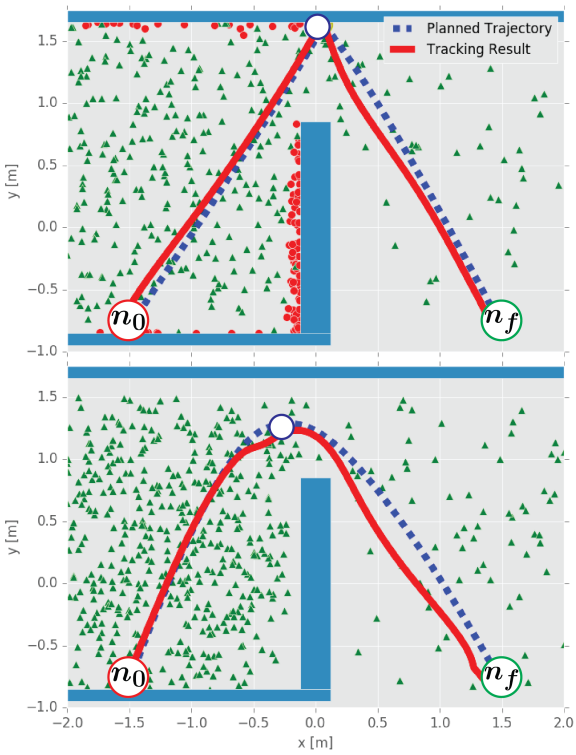

Fig. 8: Experimental result of tracking planned trajectories from $\boldsymbol{n}_0$ to $\boldsymbol{n}_f$. Top: generated and executed collision-inclusive trajectory. Bottom: generated and executed collision-exclusive planner. The blue rectangles represent the obstacles. Green triangles are non-collision nodes and the red dots are collision nodes in $\mathcal{T}$. The blue circle is an intermediate node.

C. Tracking experiment

An image sequence of the tracking experiment is shown in Fig. 7 and Fig. 8 shows the planned trajectories and tracking results for both collision-inclusive and collision-exclusive cases. The trajectories have U-shapes and avoid an obstacle in the center of the space. The obstacle separates the space into two parts, connected by a 1m gap. The planned collision-inclusive trajectory takes 2.34 seconds, whereas the collision-exclusive trajectory takes 2.39 seconds.

For both scenarios, the multicopter can successfully follow the reference to reach the end goal. However, we observe tracking error starting at the tip of the U-shape for both tracking attempts. For the collision-exclusive case, the error is caused by aggressive maneuver. For the collision-inclusive case, the collision introduces a torque disturbance that is not fully captured by the collision model. This makes the state of the vehicle deviates from the reference state and causes tracking error after the collision. This experiment verifies that trajectories generated by the collision-inclusive planner can be tracked successfully, and also suggests that better tracking of the collision trajectory can be achieved through decreasing the impact of torque disturbance on the system during collisions, either via physical designs or control strategies.

VI. Conclusions

In this paper we present a sampling-based motion planner that can exploit collisions to generate better trajectories for multicopters. The planner samples collisions as impacts between generated motion primitives and obstacles, and connects collision states with other sampled states to form collision-inclusive trajectories.

Planning with collisions has two benefits. First, allowing collisions extends the feasible state space. Second, sampled states are no longer discarded due to infeasibility caused by collisions. Thus, the rate of adding samples to the exploring tree may be increased. Collision-inclusive planning also comes with its disadvantages. The process of generating collision node requires additional computation time and it may cause sampled nodes to concentrate near obstacles and potentially slow down the expansion of the exploring tree. We illustrate with an example that the benefits of planning with collision are likely to outweigh the disadvantages when searching for trajectories in narrow environments, where most generated trajectory pieces involve collisions.

We have also experimentally tracked trajectories generated by our planner. Experiment result indicates that a major source of tracking error comes from the disturbance that is not captured by the collision model. This could be mitigated by designing a short and aggressive recovery trajectory piece to decrease the variance of the state after collision. Such trajectory piece can also make the planner less dependent on the accuracy of collision models and hence make it easier to apply the planner on different platforms.

Some possible extensions to this work are as follows. First, the algorithm efficiency can be increased via developing faster nearest neighbor search methods that can work with the primitive cost metric. Second, the vehicle can keep re-planning mid-flight to mitigate influences of error in dynamics model and disturbances the collision model fails to predict. Third, as the collision-inclusive planner is likely to find better trajectories in environments like narrow tunnels whereas collision-exclusive planner may perform better in more spacious environments, a decision making mechanisms can be designed to help the vehicle intelligently choose between the two planners. This can improve the vehicle's ability to operate in more general and complicated environments.

Acknowledgements

The experimental testbed at the HiPeRLab is the result of contributions of many people, a full list of which can be found at `hiperlab.berkeley.edu/members/`

REFERENCES

[1] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[2] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[4] D. J. Webb and J. Van Den Berg, "Kinodynamic rrt*: Asymptotically optimal motion planning for robots with linear dynamics," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 5054–5061.

[5] F. Islam, J. Nasir, U. Malik, Y. Ayaz, and O. Hasan, "Rrt*-smart: Rapid convergence implementation of rrt* towards optimal solution," in *2012 IEEE International Conference on Mechatronics and Automation*. IEEE, 2012, pp. 1651–1656.

[6] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 2997–3004.

[7] Z. Tang, B. Chen, R. Lan, and S. Li, "Vector field guided rrt* based on motion primitives for quadrotor kinodynamic planning," *Journal of Intelligent & Robotic Systems*, pp. 1–15, 2020.

[8] A. Briod, P. Kornatowski, J.-C. Zufferey, and D. Floreano, "A collision-resilient flying robot," *Journal of Field Robotics*, vol. 31, no. 4, pp. 496–509, 2014.

[9] C. J. Salaan, K. Tadakuma, Y. Okada, Y. Sakai, K. Ohno, and S. Tadokoro, "Development and experimental validation of aerial vehicle with passive rotating shell on each rotor," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 2568–2575, 2019.

[10] M. Mote, J. P. Afman, and E. Feron, "Robotic trajectory planning through collisional interaction," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 1144–1149.

[11] M. Mote, M. Egerstedt, E. Feron, A. Bylard, and M. Pavone, "Collision-inclusive trajectory optimization for free-flying spacecraft," *Journal of Guidance, Control, and Dynamics*, pp. 1–12, 2020.

[12] Z. Lu and K. Karydis, "Optimal steering of stochastic mobile robots that undergo collisions with their environment," in *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2019, pp. 668–675.

[13] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[14] N. Bucki and M. W. Mueller, "Rapid collision detection for multicopter trajectories," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 7234–7239.

[15] J. Zha, X. Wu, J. Kroeger, N. Perez, and M. W. Mueller, "A collision-resilient aerial vehicle with icosahedron tensegrity structure," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 1407–1412.

[16] J. Calsamiglia, S. W. Kennedy, A. Chatterjee, A. Ruina, and J. T. Jenkins, "Anomalous frictional behavior in collisions of thin disks," 1999.