RRT* Path Planning and Trajectory Optimization for a Quadrotor

Portia Gaitskell
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
portia@mit.edu

Christian Viteri
Department of Mechanical Engineering
Massachusetts Institute of Technology
cviteri@mit.edu

***Abstract*—Quadrotor manipulation through cluttered environments requires efficient evaluation of safe regions and reliable control. This paper presents our methods for combining sample based planning methods with trajectory optimization to find the optimal trajectory for a quadrotor through an obstacle field. We used RRT* to find a path between a start and end point given the preset obstacles. The path was then used as an initial guess in a direct collocation trajectory optimization in Drake with collision avoidance constraints to find the optimal trajectory through space. The dynamics were stabilized by a time-varying finite horizon linear quadratic regulator. The optimization problem successfully avoids spherical and rectangular obstacles in space and is stabilized by finite-horizon LQR in response to random perturbations.**

I. Introduction

Reliable navigation of unmanned aerial vehicles (UAVs) is integral to the further development of the aerospace industry. As the demand for quadrotors and UAVS usage in highly dense environments increases, such as in disaster response scenarios or autonomous package delivery, efficient path planning and collision avoidance becomes incredibly important. However, the effectiveness of these systems relies on utilizing optimal and collision-free trajectories. Traditional trajectory optimization methods which take into account quadrotor dynamics have been sufficiently explored but often require a robust initial guess in order to converge to an optimal solution. Our approach to this problem is utilizing RRT and RRT* algorithms as path planners for a quadrotor given a starting state, goal, and set of obstacles in space. The RRT and RRT* paths are then used within direct collocation trajectory optimization to satisfy the goal of collision avoidance while also adhering to the dynamics of the quadrotor system. The final trajectory is then stabilized locally utilizing a finite-horizon linear quadratic regulator to provide feedback control.

II. Related Work

A. RRT and RRT* for Motion Planning

The Rapidly-exploring Random Tree (RRT) algorithm is a probabilistically complete sample based planning algorithm, initially presented in 1998 [1]. The optimal variation of RRT called RRT* was then introduced in 2011 with proven asymptotic optimality [2]. Gaining optimality in RRT*, however, comes with a cost of execution time and slower path convergence rate. Both algorithms are still frequently used for

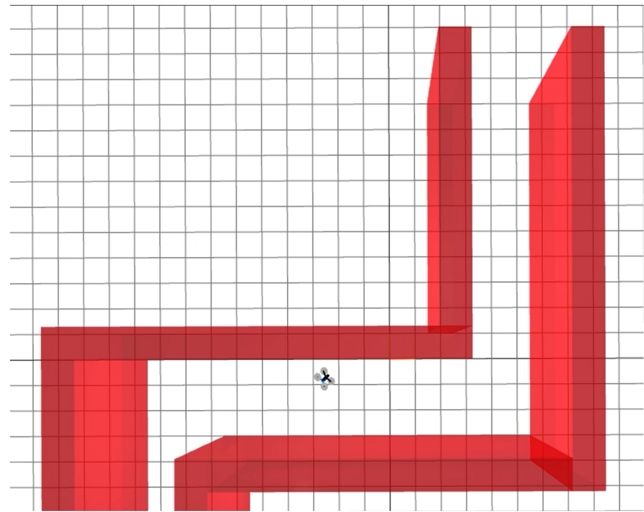


Fig. 1: Quadrotor navigating a maze environment in Drake.

path planning of mobile robots, including quadrotors, cars, and AUVs. In particular, RRT* has been used to generate optimal quadrotor paths through cluttered indoor environments as it can factor in static and dynamic constraints into the path [3]. However, its applications in real-time complex environments are limited due to slow convergence times in high dimensional space.

B. Trajectory Optimization of Quadrotors

There have been several different approaches for trajectory optimization of quadrotors. Diets and Tedrake utilized Mixed-Integer Planning to navigate a quadrotor through cluttered environments [4]. This approach separated the environment into convex, obstacle-free regions and then assigned polynomial trajectories within each region. This method found feasible and optimal trajectories through complex 3D environments. The primary limitation of this approach is that it is dependent on the set of convex regions which are generated from a random seed. This is where a sample-based motion planner such as RRT could more effectively find key waypoints through non-convex space without being dependent on the specific regions.

Another approach is using differential flatness in conjunction with direct collocation [5]. Differential flatness is a clever method for turning a nonconvex trajectory optimization into

a convex problem using a change of variables. If a trajectory can be found by using the same output dimensions as the number of actuators in the system, then one can back-solve to find the inputs to each actuator. This approach has shown to produce feasible trajectories with respect to the system dynamics and input constraints as well as solving in lower computation time than state-of-the-art. However, this approach did not use an initial guess due to the computational challenges of solving for an initial guess for each specific environment. Their methods are also limited as increasing the number of obstacles reduces the efficiency of the algorithm. Sampling based path solvers in conjunction with their methods would resolve these limitations.

Finally, there has been some recent work combining sample based motion paths as the initial guess for direct trajectory optimization methods. [6] found feasible trajectories through using RRT* and further smoothing the path by B-spline for a quadrotor. They then solved for the required dynamics to carry out the proposed trajectory. [7] continued this work to incorporate the RRT path with kinodynamic motion planning. They utilized Kinodynamic-RRT, resulting in explicit evaluation of the forward dynamics. However, there was no sense of optimality in this approach. Zhang et. al utilized RRT as the initial guess to find the optimal trajectory using nonlinear optimization methods and sequence quadratic programming for unmanned helicopters [8].

III. Methods

In order to carry out this optimization problem, we broke the project into several sub components. RRT and RRT* algorithms were developed in 2D and 3D based on preliminary obstacle fields while the direct collocation on the quadrotor was developed separately, using simple initial guesses for the trajectories. The RRT and RRT* outputs were then converted to evenly-spaced vectors mapping a changing position in time to act as an initial guess for the trajectory optimization. The resulting state and input trajectories were then stabilized using a finite-horizon linear quadratic regulator.

A. Simulation Environment

We chose to simulate our system in Drake, which uses Meshcat as a visualizer. We built our trajectory optimization and finite-horizon LQR controller using the quadrotor plant, direct collocation object, and finite-horizon LQR object within Drake. We used multibody plants to visualize both obstacles and the quadrotor in simulation to display the collision avoidance capabilities of our method.

B. Quadrotor and Object Models

Success in navigating cluttered fields required an understanding of the quadrotor plant used within Drake as well as proper modeling of obstacles to develop good collision avoidance constraints. The quadrotor state, dynamics, and object representations are as follows:

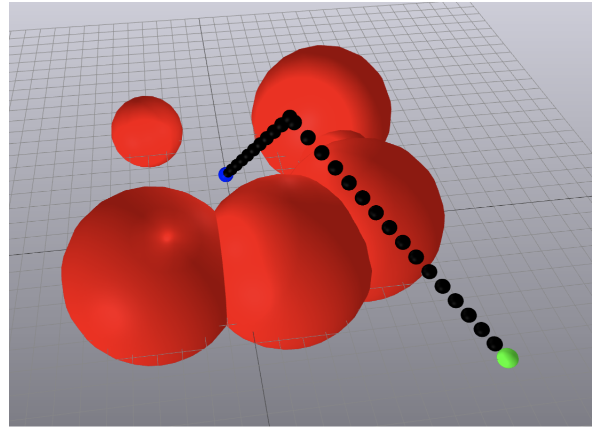


Fig. 2: RRT* Path from start (green) to goal (blue) over spherical obstacles.

State Coordinates:

$$q = [x,\ y,\ z,\ \phi,\ \theta,\ \psi]^T$$

$$\phi = roll$$

$$\theta = pitch$$

$$\psi = yaw$$

Input Thrust Forces:

$$u = [f_1,\ f_2,\ f_3,\ f_4]^T$$

Dynamics and State Vector:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = \tau_g(q) + Bu$$

$$x = [q, \dot{q}]^T$$

$$\dot{x} = f(x,u)$$

Spherical Obstacles:

$$o = [x,\ y,\ z,\ radius]$$

Box Obstacles:

$$o = [x,\ y,\ z,\ width,\ height,\ depth]$$

We chose to represent the quadrotor with cartesian position coordinates and three rotational coordinates rather than using a deferentially flat model as the ability to develop state constraints within the trajectory optimization could allow us to avoid dangerous positions for the quadrotor without a differentially flat model. Additionally, the input vector is represented by four forces corresponding to the the output thrust force from each rotor on the quadrotor. The dynamics are thus represented, as an affine system, by $\dot{x} = f(x,u)$. The objects we used are characterized by their center point and other defining characteristics.

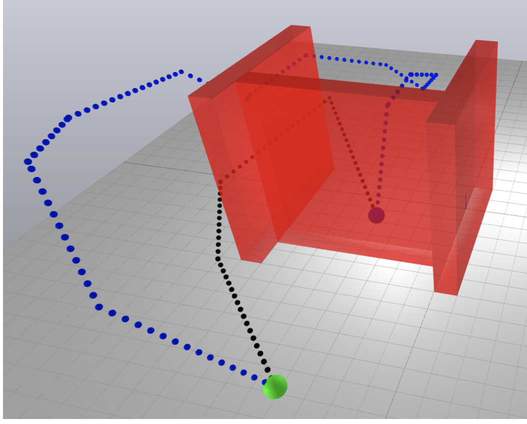


Fig. 3: RRT (blue) and RRT* (black) from start to goal around rectangular obstacles.

C. RRT* Path Finding

Optimal Rapidly-exploring Random Tree (RRT*) is a probabilistically complete sample based planning algorithm. RRT* is also asymptotically optimal and returns the shortest path to the goal. The pseudo-code for the implementation is as follows:

```
def RRT*:
   for itr in range(0... max_iters)
      rndNode = random_node()

      nearestNode = neaarest_node(rndNode
      cost = distance(nearestNode, rndNode)
      newNode = connect(rndNode, nearestNode)

      If in_collision(newNode, nearestNode):
          continue
      neighbors = get_neighbors(newNode)
      #improve the costs to each node using
          newNode
      rewire(newNode, neighbors)

   return path(bestGoalNodeIndex)
```

This approach was utilized for navigating both obstacle types. The sole difference was the collision checker. The collision checker verifies that the connection between the new node and its nearest node does not intersect an obstacle in the obstacle list.

1) RRT Collision for Spherical Obstacles:

The approach for spherical obstacles was to find the parametric line connecting the two nodes and find the closest point on that line to the obstacle center. Then, check if the distance between the closest point and the obstacle center is less than the radius. The code is documented in the Appendix.

2) RRT Collision for Rectangular Obstacles:

The collision checker for rectangular objects was a greater challenge as it required checking if the vector between the two nodes intersected the box on any face or edge. The implementation is in the Appendix.

D. Direct Collocation Optimization

The direct collocation algorithm finds a time step, state trajectory and input trajectory such that the cost function is minimized and all constraints below are satisfied.

$$\min_{x[\cdot],u[\cdot],h_n} l_f(x[N]) + \sum_{n_0}^{N-1} h_n l(u[n])$$
$$l(u[n]) = Ru[n]^2,\ R = 10$$

subject to

$$\begin{aligned} h_n &= constant \\ \dot{x}(t_c,n) &= f(x(t_{c,n}), u(t_{c,n})) \\ x[0] &= x_i \\ x[N] &= x_f \\ -\pi &< \phi < \pi \\ -\frac{\pi}{2} &< \theta < \frac{\pi}{2} \\ -\pi &< \psi < \pi \end{aligned}$$

We constrain the optimization to work on equal time steps, solving for the dynamics at each collocation point. Additionally, we constrain the roll, pitch, and the yaw of the quadrotor to avoid gimbal-lock and positions that cause infeasibility in the program. Although the initial guess from RRT and RRT* successfully avoids obstacles, there are also two additional constraints for obstacle avoidance dependent on whether the specific obstacle in question is a sphere or box. Spherical obstacle avoidance works as follows:

$$\forall n \in [0, N-1], \forall o \in O,$$
$$\sqrt{(x[n]_x - o_x)^2 + (x[n]_y - o_y)^2 + (x[n]_z - o_z)^2)} > o_r$$

Rectangular obstacles were first represented by ellipsoids according to the following parameter functions, causing elongation in the z-direction to ensure a tighter fit to the x and y dimensions, shown in Figure 3.

$$ellipsoid = [o_x,\ o_y,\ o_z,\ a,\ b,\ c],$$
$$a = \frac{o_{width}}{2} + 0.1\sqrt{\frac{o_{width}}{2}},$$
$$b = \frac{o_{height}}{2} + 0.1\sqrt{\frac{o_{height}}{2}},$$
$$c = \frac{o_{depth}}{2}\max\left(o_{width},\ \ o_{height}\right)$$

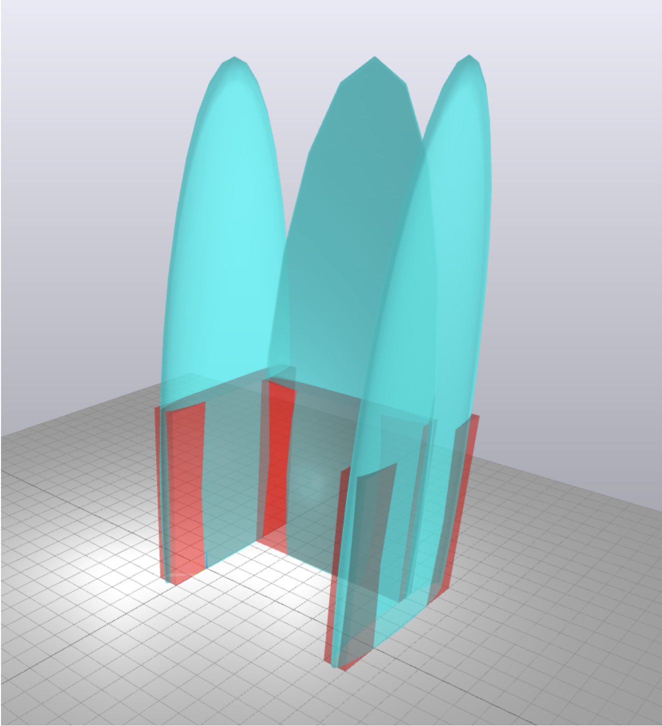


Fig. 4: Ellipsoid collision regions used to approximate the rectangular obstacles.

Ellipsoid avoidance is as follows:

$$\forall n \in [0, N-1], \forall o \in O,$$

$$\frac{(x[n]_x - o_x)^2}{a^2} + \frac{(x[n]_y - o_y)^2}{b^2} + \frac{(x[n]_z - o_z)^2}{c^2} \geq 3$$

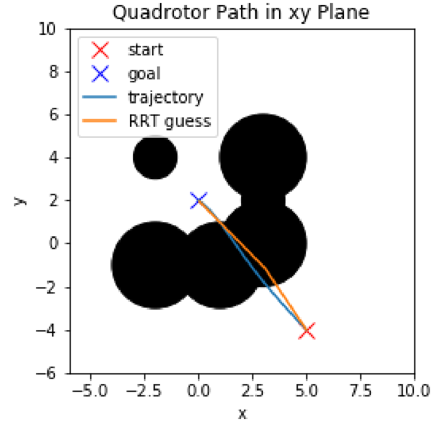
E. Trajectory Stabilization and Simulation

IV. Evaluation and Results

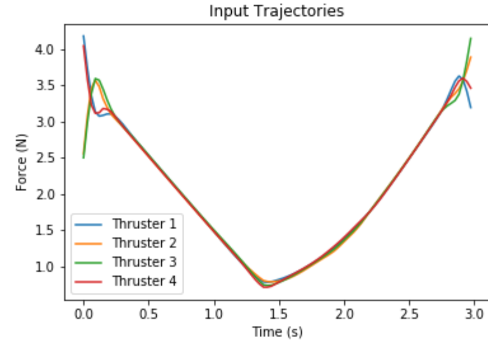
We tested our RRT* path finding and trajectory optimization on three different environments: one with spherical obstacles and two with box obstacles. The program's performance was thus determined by the optimality of both the RRT* solution and the trajectory optimization program. Performance was further determined by the result of the finite-horizon LQR stabilization and whether or not the same end goal was reached with a random perturbation about the start state.

A. Spherical Obstacles

First, we tested the RRT algorithm and trajectory optimization with spherical obstacles in space. Here, we attempted to drive the quadrotor over a collection of spheres to land in between obstacles. Figure 5a displays the quadrotor's path which closely follows the RRT* guess. The trajectory shown highlights the capability of the trajectory optimization program to smooth out the RRT initial guess with the quadrotor



(a) RRT* guess (orange) and resulting xy state trajectory (blue).



(b) Resulting input trajectories for the four rotors.

Fig. 5: Trajectory outputs for the spherical obstacle environment.

dynamics. Additionally, the input trajectories are shown in Figure 5b. Here, the inputs are relatively smooth and, though we did not include input constraints in the direct collocation, the rotors all operated on below a thrust output of 5 newtons. This environment was successfully navigated by the program as RRT* found an optimal path and the direct collocation algorithm was able to find optimal input and state trajectories.

The state and input trajectories were then simulated in Drake and stabilized by finite-horizon LQR, yielding three trajectories starting from a random perturbation around the desired starting point used in the RRT algorithm. The trajectories are illustrated in Figure 6 and display the LQR stabilization effectively guiding the quadrotor along the proposed state trajectory. Despite the random perturbations introduced in the simulation, the system still converged to the same goal.

B. U-Shaped Barrier

The next environment we chose to test was a u-shaped barrier with a proposed path that would force the quadrotor to bank and move from outside to within the barrier. Similar to the first trial with spherical obstacles, there was an optimal path found with RRT* and optimal trajectories returned by the direct collocation optimization, both of which are displayed

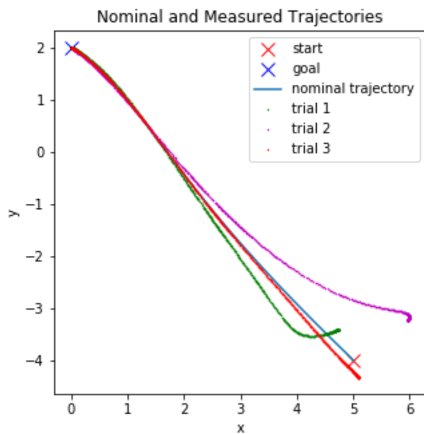
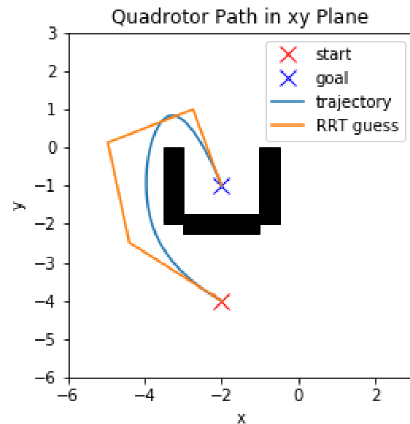
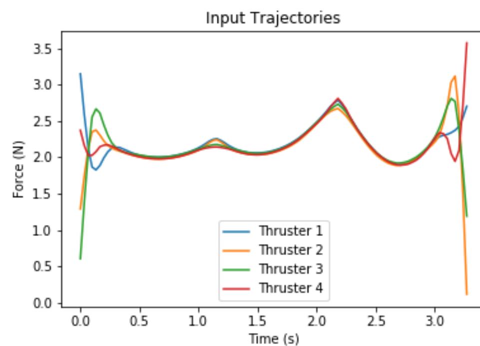


Fig. 6: Nominal trajectory and three random perturbations about the start state for the spherical environment.



(a) RRT* guess (orange) and resulting state trajectory (blue).



(b) Resulting input trajectories.

Fig. 7: Trajectory outputs for the u-shaped barrier environment.

in the xy plane in Figure 7a. The input trajectory, shown in Figure 7b, is well-bounded without input limits although it displays some possible numerical artifacts at the end of the trajectory as each thruster has a larger increase or decrease in output force.

This environment and maneuver was then simulated with the LQR stabilization and successfully converged to the set goal state from three random perturbations, indicating a well-defined feedback controller through simulation and good initial trajectories. The results are shown in Figure 8. Some trajectories simulated with the LQR regulator fly much closer to the corners of the boxes in comparison to the original trajectory. Due to our formation of obstacles in each environment, we do not account for the possibility of collisions with the quadrotor and simulate those dynamics. Rather, the obstacles are fed as data to the RRT* algorithm and trajectory optimization and the subsequent trajectories are a result of avoiding quadrotor states within the space inhabited by obstacles. Consequently, as tighter manuevers are simulated with some randomness in Drake, there is a possibility that the quadrotor would collide with obstacles given enough randomness.

C. Maze

The last environment we tested was a miniature maze made up of two 90-degree turns for the quadrotor to navigate, resulting in a more complex path required to avoid the walls. In similar fashion to the two previous tests, the RRT* algorithm and trajectory optimization both found optimal paths given our constraints, the results of which are shown in the xy plane in Figure 9a. The quadrotor performs two banks to complete the 90 degree turns and accelerate to the end of the maze. The trajectory seems to follow the end of the initial RRT* path more closely than the beginning, indicating how our RRT* algorithm lacks dynamical understanding and the trajectory has to find a different path close to the RRT* guess in order to satisfy quadrotor dynamics throughout the entirety of the optimization. Additionally, Figure 9b displays the thrust forces

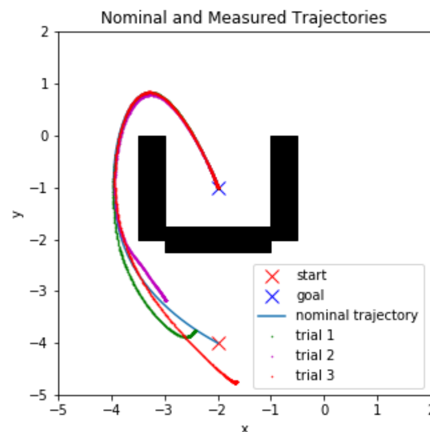
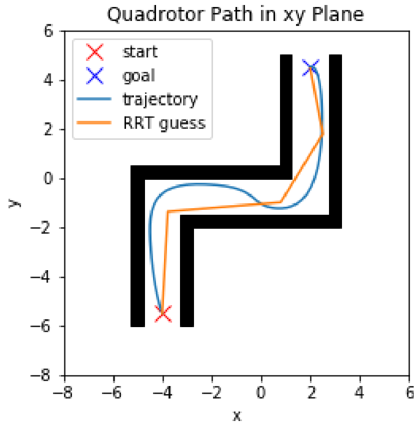
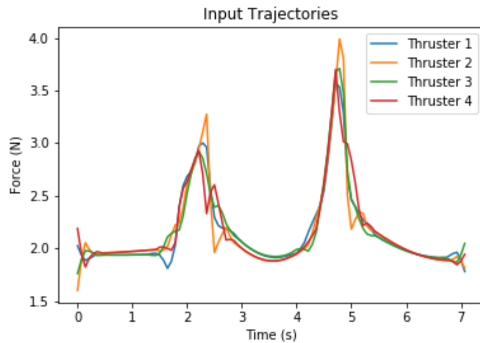


Fig. 8: Nominal trajectory and three random perturbations about the start state for the u-shaped barrier environment.



(a) RRT* guess (orange) and resulting state trajectory (blue).



(b) Resulting input trajectories.

Fig. 9: Trajectory outputs for the maze environment.

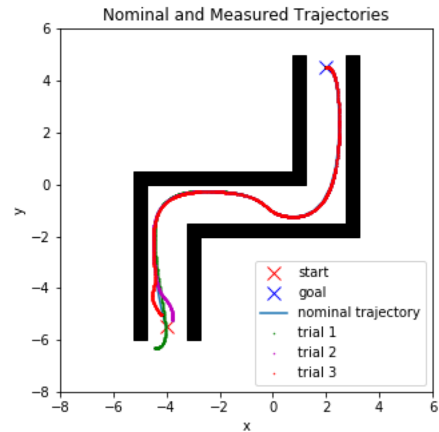


Fig. 10: Nominal trajectory (blue) and three random perturbations about the start state for the maze environment.

yielded by the trajectory optimization over time. In comparison to the spherical obstacles and u-shaped barrier, these input trajectories are much more discontinuous and jagged despite characterizing a relatively smooth trajectory.

The trajectories found in the maze environment were simulated with the LQR regulator and again successfully converged to the correct end point dictated by the problem definition and desired trajectory. The three random perturbations about the start point can be seen in Figure 10. Here, the issue proposed by the u-shaped barrier persists. Lack of collision dynamics could result in the random perturbations about the start state forcing the quadrotor into positions that are impossible given the obstacles we have set in the problem. In addition to these impossible initial states, the path the quadrotor takes to reach the desired trajectory could collide with the walls, highlighting the limitations of our current implementation.

V. Discussion

Our methods prove that using optimal RRT paths as well-informed initial guesses to a direct collocation for a quadrotor yield not just feasible, but optimal trajectories given simple obstacle fields. For each environment, RRT* paths produced optimal input and state trajectories which were then successfully stabilized by finite-horizon LQR feedback control in simulation. Additionally, the introduction of random perturbations to the initial state displayed the ability of the simulated regulator to guide the quadrotor back to the desired trajectory and complete the desired maneuver. While the current implementation successfully avoids obstacles within the RRT* algorithm and direct collocation optimization, the methods here are limited in scope as they lack a formulation of collision avoidance within the LQR stabilization.

A secondary limitation is the rectangular obstacle collision constraints. The ellipsoid approximation worked well for the environments we selected by over constraining the boxes vertically as shown in Figure 4 to leave accurate approximate obstacle regions in the quadrotor air space. However, in order to expand to more complex environments, such as with holes through the obstacles, we will need to modify this pipeline to more accurately approximate the obstacles. Finally, the main bottleneck of our system is the computation time vs. the number of obstacles when conducting trajectory optimization. This is because we add a collision constraint to all knot points for every obstacle in the space, regardless of it's distance. Ideally, we would want to modify this to only check collision constraints within a reasonable spherical region in order to improve the efficiency.

VI. Conclusion and Future Work

The goal of this project was to create an end-to-end system for finding an optimal quadrotor trajectory through a cluttered environment. Our final implementation achieved this goal and was validated in several different environments, including navigating into a non-convex obstacle space and through a maze with 90-degree turns. The system was able to successfully navigate the specified environments and was robust to perturbations from the initial starting state.

In the future we would like to approve upon our limitations as specified above, by adding collision checking to the LQR stability and better representing rectangular obstacles. We also

would like to investigate the performance of other trajectory optimization pipelines, such as Model Predictive Control, using the initial RRT* guess. Finally, we would like to conduct further analysis along the entire trajectory to find regions of stability using Lyapunov analysis.

VII. Team Contributions

Portia predominantly worked on developing the path finding algorithms. This included implementing the RRT and RRT* algorithms in 2D and 3D and developing spherical obstacle models and cluttered fields for the path finding algorithms to work through. She also worked to develop visualizations of the obstacle fields within drake to work with our visualization of the quadrotor.

Christian mainly worked on implementing the direct collocation optimization for the quadrotor, implementing state constraints to avoid dangerous positions for the quadrotor and collision constraints to avoid spherical obstacles. He used the results of the path finding algorithms to provide better initial guesses to the optimization and subsequently simulated the quadrotor in drake, using finite horizon LQR to stabilize the quadrotor along the proposed trajectory.

Acknowledgment

Many thanks to Russ Tedrake, Alexandre Amice, Lujie Yang and the rest of the 6.832 course staff for their patience, guidance, and attentiveness throughout the semester.

References

[1] S. M. Lavalle, “Rapidly-Exploring Random Trees: A New Tool for Path Planning,” 1998.

[2] 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, Jun. 2011, doi: 10.1177/0278364911406761.

[3] B. Liu, W. Feng, T. Li, C. Hu, and J. Zhang, “A Variable-Step RRT* Path Planning Algorithm for Quadrotors in Below-Canopy,” IEEE Access, vol. 8, pp. 62980–62989, 2020, doi: 10.1109/ACCESS.2020.2983177.

[4] R. Deits and R. Tedrake, “Efficient mixed-integer planning for UAVs in cluttered environments,” in 2015 IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, USA, May 2015, pp. 42–49. doi: 10.1109/ICRA.2015.7138978.

[5] J. Zeng, P. Kotaru, M. W. Mueller, and K. Sreenath, “Differential Flatness Based Path Planning With Direct Collocation on Hybrid Modes for a Quadrotor With a Cable-Suspended Payload,” IEEE Robotics and Automation Letters, vol. 5, no. 2, pp. 3074–3081, Apr. 2020, doi: 10.1109/LRA.2020.2972845.

[6] G. Kulathunga, R. Fedorenko, S. Kopylov, and A. Klimchik, “Real-Time Long Range Trajectory Replanning for MAVs in the Presence of Dynamic Obstacles,” 2020 5th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS), pp. 145–153, Jul. 2020, doi: 10.1109/ACIRS49895.2020.9162605.

[7] K. Albee and B. Coltin, “Kinodynamic-RRT for Robotic Free-Flyers: Generating Feasible Trajectories for On-orbit Mobile Manipulation,” p. 2.

[8] B. Zhang, Q. Zong, L. Dou, B. Tian, D. Wang, and X. Zhao, “Trajectory Optimization and Finite-Time Control for Unmanned Helicopters Formation,” IEEE Access, vol. 7, pp. 93023–93034, 2019, doi: 10.1109/ACCESS.2019.2927817.

VIII. Appendix

A. Collision Checker for Spherical Obstacles

```
def sphere_in_collision(p1, p2,
    obstacle_list):
   for o in obstacle_list:
      d12 # vector from p1 to p2
      d1c = o_center - p1 # vector from
          circle to p1
      t = d12.dot(d1c) / (d12.dot(d12) + 1e-7)
      t = max(0, min(t, 1)) # 0<=t<=1
      d = p1 + d12*t # point where line
          segment and o are closest
      if sum((o_center-d)**2) < radius**2
         return True # is in collision
   return False # is not in collision
```

B. Collision Checker for Rectangular Obstacles

```
def check_in_box(fDst1, fDst2, P1, P2, B1,
    B2, Axis):
'''
Checks if line intersects box face or edge
'''
   if ( (fDst1 * fDst2) >= 0.0):
      return False
   if ( fDst1 == fDst2):
      return False

   p = P1 + (P2-P1) * ( -fDst1/(fDst2-fDst1)
       ) #get intersection point

   if ( Axis==1 and p[2] >= B1[2] and p[2] <=
       B2[2] and p[1] >= B1[1] and p[1] <=
       B2[1]):
      return True
   if ( Axis==2 and p[2] >= B1[2] and p[2] <=
       B2[2] and p[0] >= B1[0] and p[0] <=
       B2[0]):
      return True
   if ( Axis==3 and p[0] >= B1[0] and p[0] <=
       B2[0] and p[1] > B1[1] and p[1] <=
       B2[1]):
      return True
   return False


def check_line_box_collision(B1, B2, L1, L2):
'''
B1: min x,y,z of box
B2: max x,y,z of box
L1 and L2: end vertices of the line
Main collision checker method. Returns true if
'''
   # check if both points are outside the box
       on the same side
   if (L2[0] < B1[0] and L1[0] < B1[0]):
      return False
   if (L2[0] > B2[0] and L1[0] > B2[0]):
      return False
   if (L2[1] < B1[1] and L1[1] < B1[1]):
      return False
   if (L2[1] > B2[1] and L1[1] > B2[1]):
      return False
   if (L2[2] < B1[2] and L1[2] < B1[2]):
      return False
   if (L2[2] > B2[2] and L1[2] > B2[2]):
      return False

   # Check if completely within
   if (L1[0] >= B1[0] and L1[0] <= B2[0] and
       L1[1] >= B1[1] and L1[1] <= B2[1] and
       L1[2] >= B1[2] and L1[2] <= B2[2]):
      return True

   if check_in_box(L1[0]-B1[0], L2[0]-B1[0],
       L1, L2, B1, B2, 1) or \
   check_in_box(L1[1]-B1[1], L2[1]-B1[1], L1,
       L2, B1, B2, 2) or \
   check_in_box(L1[2]-B1[2], L2[2]-B1[2], L1,
       L2, B1, B2, 3) or \
   check_in_box(L1[0]-B2[0], L2[0]-B2[0], L1,
       L2, B1, B2, 1) or \
   check_in_box(L1[1]-B2[1], L2[1]-B2[1], L1,
       L2, B1, B2, 2) or \
   check_in_box(L1[[0]]-B2[2], L2[2]-B2[2],
       L1, L2, B1, B2, 3):
      return True

   return False
```