LINEAR QUADRATIC REGULATOR MINIMUM SNAP QUADROTOR PLANNING IN ROS

AJAY ANAND [ANANDAJ@SEAS.UPENN.EDU], TAYLOR SHELBY [TESHELBY@SEAS.UPENN.EDU],

ABSTRACT. The primary objective of this work is to use a Linear Quadratic Regulator (LQR) (Bryson & Ho, 1975) to generate minimum snap trajectories for a quadrotor(Mellinger & Kumar, 2011). Due to the difficulties in training and testing quadrotors in the real world, a simulated ROS gazebo environment is set up and a quadrotor model is used (Meyer, Sendobry, Kohlbrecher, Klingauf, & Von Stryk, 2012) to simulate the dynamics of real quadrotor in flight. A variety of generated environments are then used to test the robustness and efficiency of the paths generated by our algorithm.
Key Words: Linear Quadratic Regulator (LQR) , ROS Melodic, trajectory planning, Hector Quadrotor.

1. INTRODUCTION

Quadrotors are one of the fastest growing and well-known fields of robotics today, however in many ways their design and control are still nascent technologies. Being state of the art technologies, there consequently exists room for improvement in their modelling and control. The focus of this work in particular will be improvements in trajectory generation. In the process of controlling a quadrotor from the start to a goal position, there exist multiple processes that need to be successfully completed in order to obtain a good, safe movement response. The first of these processes is obtaining an obstacle free path, which can be carried out in variety of ways using different path planning algorithms. For our project, we have opted to use the A* search algorithm to collect such waypoints. Next, it is necessary to convert discrete waypoints into a trajectory which is feasible for the actuators at every time instant in the trajectory. This is the process of trajectory generation, which converts a set of waypoints (either sparse or dense) into a continuous trajectory. The goal of trajectory generation is to compute the trajectory that passes within a minimum tolerance of all the waypoints without colliding with any of the obstacles in the environment, while at the same time minimizing the run time cost (be it in time or energy) for the quadrotor to take the path. However, dense trajectories are difficult to fit and resource intensive to work with. Hence we convert the dense waypoint set to a sparse set using the Douglas–Peucker algorithm or iterative end-point fit algorithm which dynamically converts the curve composed of the dense way points into a similar curve with fewer points (Douglas & Peucker, 1973). The sparse path is passed into the aforementioned trajectory generator, LQR in our case, then the last step of the process is to pass on the trajectory to a controller that models the dynamics of the quadrotor to control the state.

2. CONTRIBUTIONS

We implemented an A* path planner to collect dense waypoints, used Douglas-Peucker to reduce them to a sparse set, then used these waypoints to set up constraints for a quadratic program with a cost of the integral of snap squared, which was solved by cvxopt. This min-snap trajectory was then tested in two different simulators, one in Python (Welde & Paulos, 2020) and one in ROS Gazebo (Meyer et al., 2012). Both achieved good results, with no collisions and smooth trajectories, but were not particulary fast.

3. BACKGROUND

The motivation for this project came from our prior work with the control of quadrotors. To carry out control of quadrotors in 3D space, three different tasks needed to be carried out:

- Modelling dynamics and control
- Path creation from start to goal pose
- Trajectory generation

3.1. Quadrotor Dynamics.

Quadrotor Dynamics. The dynamics of the quadrotor are simplified in most approaches by assuming a state near a hover to give us a set of flat outputs which are then used to easily model the dynamics of the quadrotors. In this work we choose to follow the model described in Minimum Snap Trajectory Generation and Control for Quadrotors (Mellinger & Kumar, 2011).

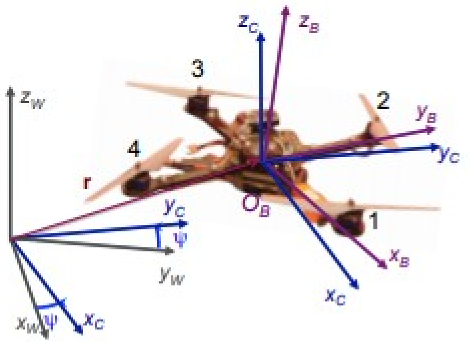


FIGURE 1. Quadrotor Model (Mellinger & Kumar, 2011)

The coordinate frames, including the world and body frames, as well as the naming convention for the propellers are depicted in figure 1. Rotation matrices are used to represent the orientations of the different frames with respect to one another to prevent singularities caused by large deviations from the hover position. Z-X-Y Euler angles are then used to define roll, pitch and yaw(ϕ,θ,ψ) angles as a local coordinate system. The angular velocity of the robot is denoted by ω_{WB} denoting the angular velocity of the frame in B in frame W, with orthonormal components p,q,r in the body frame. These values are directly related to the derivatives of the roll,pitch and yaw angles. Each of the four rotors has an angular speed ω_i, producing a force F_i and a moment M_i according to the relations

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

The motor dynamics, due to their fast response can be considered instantaneous compared to the rigid body dynamics and aerodynamics of the quadrotor.Therefore the control input to the system can be written as **u** where u1 is the net body force and u2, u3, u4 are the body moments which can be expressed according to the rotor speeds as

$$\mathbf{u} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & k_F * L & 0 & k_F * L \\ -k_F * L & 0 & k_F * L & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} * \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}$$

where L is the distance between the axis of rotation of each of the rotors to the center of the quadrotor.
While the position the of the center of mass of the quadrotor is denoted by **r**, the only forces acting on the system are gravity in $-z_W$ direction and the net force due to the quadrotors in the z_B direction. Newton's equations of motion for the system then are

$$m * \ddot{r} = -m * g * z_W = u1 * z_B \tag{2}$$

The angular acceleration can then be determined by the Euler equation as

$$\dot{\omega}_{BW} = I^{-1}[-\omega_{BW} \times I\omega_{BW} + \begin{bmatrix} u2 \\ u3 \\ u4 \end{bmatrix}] \tag{3}$$

The state of the system is then given by $\boldsymbol{x} = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, p, q, r]^T$

3.2. **Dijkstra and A*.** A collision free path between the start point and a desired goal point can be defined as a set of waypoints. There exist many search algorithms with varying speed, optimality and computational requirements. For the purposes of this work we focus on discrete search algorithms in the form of Dijkstra(Dijkstra, 1959) and A*star. Dijkstra's algorithm is a form of dynamic programming in which first the environment is discretized into the form of grid cells, these discrete cells henceforth referred to as nodes are iteratively expanded starting from the start position, each node is assigned a cost to come value which is the sum of all the costs of its parent nodes. In each iteration the node which is expanded is the node with the lowest cost to come. This process repeats until the goal node is added to the path. Dijkstra is guaranteed to find a path if one exists as it will keep expanding nodes ad infinitum until it finds a path, however it is computationally inefficient as it lacks directionality; ie it expands equally in all directions. A*star(Hart, Nilsson, & Raphael, 1968) is a improvement to Dijkstra that incorporates prior information about the direction from the start to the goal position, a heuristic measure is incorporated into the cost to come for each node to determine which node should be expanded next.

3.3. **LQR.** After having reduced the waypoints from the dense path provided by A* to a much sparser path, we sought to describe our LQR system as a quadratic program (QP), which can be solved via cvxopt when structured as shown by equation 4

$$\begin{aligned} \text{minimize } & \frac{1}{2}x^T Hx + q^T x \\ \text{subject to } & Gx \leq h \\ & Ax = b \end{aligned} \tag{4}$$

(*Cone Programming: Quadratic Programming*, n.d.)
This is a dynamic programming problem, with a quadratic cost to minimize and linear constraints. We think of each pair of sparse waypoints as endpoints of a trajectory subject to polynomial constraints. We are using cvxopt to solve for the coefficients of $n = (\text{num waypoints} - 1)$ polynomials, with $d = 8$ terms in each polynomial. (Posa, 2019) With this 7^{th} order polynomial for the position, we can just calculate derivatives as a function of our state position, $\begin{bmatrix} x & y & z & yaw \end{bmatrix}$. Thus x in the equation above is $4 * d * n$ long, representing the coefficients for the 7th order polynomial for n waypoints for each of these 4 state variables. Since quadrotors are differentially flat, we know that whatever smooth trajectory we generate will be feasible to control (Mellinger & Kumar, 2011). Rather than minimize control effort directly, we minimized the integral of snap squared as described by (Mellinger & Kumar, 2011). G and h were used to implement the corridor constraints, also as described by (Mellinger & Kumar, 2011). The details of this setup will be explained further in section 5.2.

4. Related Work

The main purpose of this project is to build upon the work carried out in (Mellinger & Kumar, 2011) to create minimum snap trajectories using a similar approach to trajectory generation using a Linear Quadratic Regulator and QP. A primary difference in testing methodologies however exists in the fact that all our code was tested in simulation environments (one formulated in Python and another in ROS Gazebo). We referenced (Meyer et al., 2012) to set up and run our simulated environment in ROS Gazebo with the Hector Quadrotor UAV modelling suite. We referenced (Dijkstra, 1959) and (Hart et al., 1968) to implement the A star algorithm.This algorithm was used to obtain a path from the start to the goal positions. After obtaining the set of dense waypoints we referenced (Douglas & Peucker, 1973) to implement the Douglas-Peucker, which reduced the waypoint set to a sparse one. The QP set up was also based on (Mellinger & Kumar, 2011), and differs primarily in the way we selected times for our polynomial segments. Namely, we just estimated a constant velocity and used the linear distance to guess the time, while (Mellinger & Kumar, 2011) improves upon this guess with several iterations.

5. Approach

5.1. **Waypoint finding and reduction.** Both A * and Douglas-Peucker were implemented as discussed in section 3.

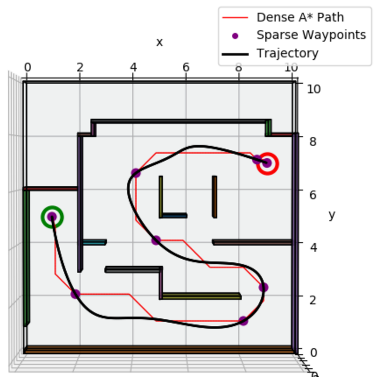
5.2. **LQR.** Δt for each of the various polynomial segments was estimated based on assuming a constant velocity, taking the euclidean distance between the two waypoints and dividing the distance by the constant velocity. In our initial non-Gazebo simulations, a velocity of 2.65 m/s was selected, while in Gazebo due to the larger quadrotor and less well-tuned PID controller a velocity of 1m/s was deemed more safe. The dynamics constraints, A and b, are derived from the passed in waypoints and desire for continuity. We chose to enforce continuity on derivatives up to the 4th order. We also added constraints on the endpoints, forcing the quadrotor to start at the 0 velocity and acceleration and end at 0 velocity and acceleration.(Posa, 2019)

The corridor constraints were mostly created as described by (Mellinger & Kumar, 2011), selecting sub-points along the trajectory and enforcing inequality constraints at each keeping the polynomial trajectory within a pre-selected distance of the straight line between the two waypoints. Currently the number of sub-points per pair of waypoints is just 1, representing the midpoint. However, this is set up in code as an argument to the create corridor constraints function, so it could be experimented with for future work.

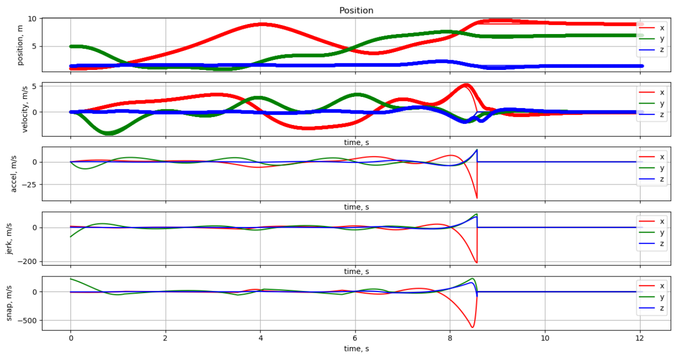
5.3. **ROS implementation.** Using the hector_quadrotor library for the quadrotor dynamics, models, and local controller, we applied our path planner to a simulation in Gazebo via ROS. After creating our virtual environment following (Navalgund, 2020) in ROS Gazebo and installing all the libraries, we set up a ROS node which ran A* to collect waypoints, Douglas-Peucker to thin the waypoints, then used cvxopt to solve the QP and create a smooth trajectory between them. This node then published the desired pose based on the current simulation time to the hector_quadrotor PID controller. We also had to write a script to take our .json map files and convert them to an SDF which could be used by Gazebo, but the map conversion occurred only once per map. The results were saved to the /.gazebo/models/mymap folder so they could be automatically loaded with the roslaunch files. Code for both the LQR and ROS is linked here.

6. Experimental Results

6.1. **Python Proof of Concept.** First, since our code was written in Python, we tested it in a Python quadrotor simulator (Welde & Paulos, 2020) using a Crazyflie quadrotor. The results for this can be seen in figures 2 and 4. In both of these figures, graph b shows how our PD controller was able to closely follow the desired trajectory from the LQR results quite well. On the maze map, the sparse waypoints are nicely spaced and the trajectory is quite smooth. However, on the over-under map the many sharp turns required a smaller margin for Douglas-Peucker, resulting in a more jagged trajectory.

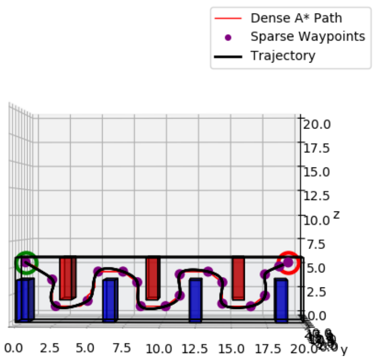


(A) Maze map results

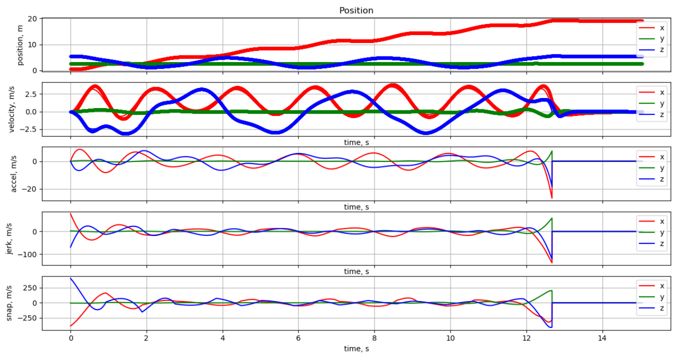


(B) Maze position, velocity, acceleration, and snap. Smooth lines are commanded trajectory, dots are actual trajectory (where tracked)

FIGURE 2. Maze results, with a Douglas-Peucker margin of 0.25



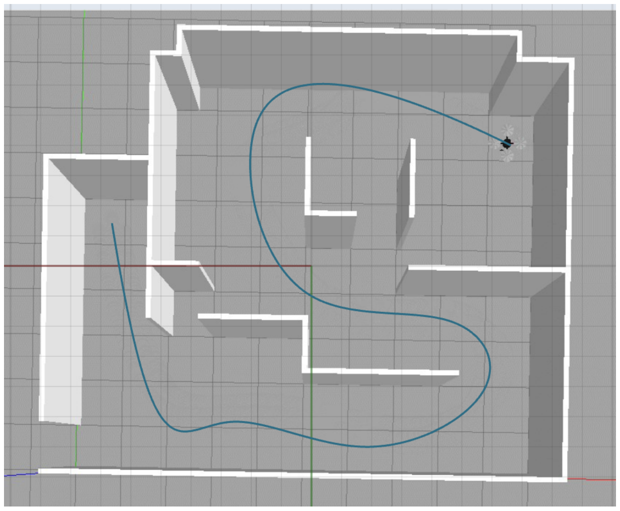
(A) Over-under map results



(B) Over-under position, velocity, acceleration, and snap. Smooth lines are commanded trajectory, dots are actual trajectory (where tracked)

FIGURE 3. Over-under results, with a Douglas-Peucker margin of 0.5

6.2. **Testing in ROS.** In ROS we used the hector_quadrotor default quadrotor, which is larger than the Crazyflie we tested with in Python. To deal with this, the estimated velocity used to choose Δt for each segment had to be decreased. However, with this decreased speed the quadrotor did successfully navigate the desired trajectory without collision.


(A) Maze Gazebo results, with a Douglas-Peucker margin of 0.5

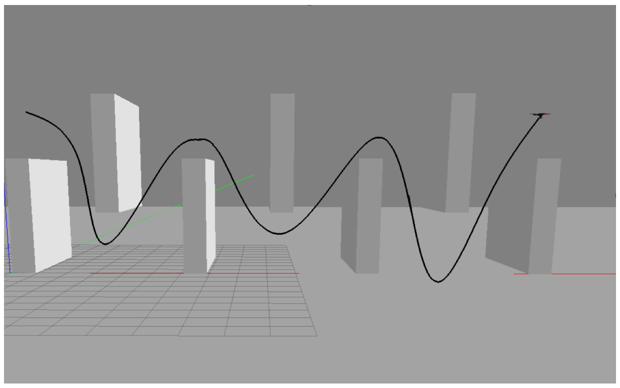

(B) Over-under Gazebo results, with a Douglas-Peucker margin of 0.25

FIGURE 4. ROS Gazebo results

7. DISCUSSION

7.1. **Results.** The QP min-snap planner functioned well both in the Gazebo simulator using ROS and in our Python quadrotor simulator. The QP planner was reasonably fast, taking less than 2 seconds to run while our A* took 30-60s, but this is not really fast enough to run as a real-time planner. The results both in the ROS Gazebo simulation and the Python simulation definitely have room for improvement, but they did manage to get through two tight courses with no collisions.

7.2. **Future Work.** Regarding the ROS version of our trajectory, a definite improvement could be found by applying a better local controller to follow the QP trajectory. Currently, we are using the hector_quadrotor inbuilt PID controller, just feeding it the position. However, to make this not occasionally hit obstacles the estimated velocity had to be slowed significantly, from 2.65 m/s to 1 m/s. In the Python based simulation where we had tuned the PID controller, the quadrotor was much better at keeping up with the rapid trajectory.

The trajectory planner also has room for improvement. With the current setup for point reduction, maps with tight corners can sometimes cut very close to the obstacles. While this may work tolerably in simulation, any additional noise in reality would be a risk for these close cut trajectories. Furthermore, it is difficult to pick good margins for point reduction in maps with a variety of terrain. In long open spaces, narrow margins can force the quadrotor to make unnecessary detours, while wide margins can cut off corners in tight spaces. Similarly, optimal margins for the corridor constraints vary drastically based on the local object density. A more local map based approach, whether hand tuned or machine learned, would likely provide more efficient and safe results.

The current trajectory generation set up relies on pre-selected time intervals assuming a constant velocity. The same framework could be improved by checking the length of the current segment and using faster velocities on longer segments. Alternatively, the time rather than the effort and snap could be selected for the cost, though that would be quite another problem.

References

Bryson, A. E., & Ho, Y.-C. (1975).
In *Applied optimal control: optimization, estimation, and control* (p. 128–167). Taylor et Francis.

Cone programming: Quadratic programming. (n.d.). `https://cvxopt.org/userguide/coneprog.html#quadratic-programming`.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. doi: 10.1007/bf01386390

Douglas, D. H., & Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, *10*(2), 112–122. doi: 10.3138/fm57-6770-u75u-7727

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100-107. doi: 10.1109/TSSC.1968.300136

Mellinger, D., & Kumar, V. (2011). Minimum snap trajectory generation and control for quadrotors. In *2011 ieee international conference on robotics and automation* (p. 2520-2525). doi: 10.1109/ICRA.2011.5980409

Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., & Von Stryk, O. (2012, 11). Comprehensive simulation of quadrotor uavs using ros and gazebo. In (Vol. 7628, p. 400-411). doi: 10.1007/978-3-642-34327-8_36

Navalgund, B. (2020). *Hector quadrotor implementation.* `https://github.com/basavarajnavalgund/hector-quadrotor`.

Posa, M. (2019). *Meam 517 homework 4.*

Welde, J., & Paulos, J. (2020). *Projects 1.1 - 1.3 quadrotor simulator.*