Implementing Drone Motion Planner using Two-sided Kinodynamic RRT

Dor Israeli

Tel Aviv University
Dor.Israeli@gmail.com

Drone motion planning problem

Problem at hand

In the last few years we have seen a rise in the usage of robotic drones in many fields, from professional filming and oil excavate monitoring, to shipping goods, all autonomously. Thus, there is an increasing interest in researching the area of path and motion planning for improving the autonomous uses of drones.

The problem facing while trying to plan a drone motion is not trivial, as it needs to address several aspects, specifically but not limited to:

1. Obstacles avoidance-
   be them static as trees and walls, or dynamic as birds
2. Physical forces and constraints-
   it needs to account for inertia and external forces, and inert constraints
3. Optimality-
   be it flight time, fuel usage or some other metrics

Although not the focus of this paper, this problem can be generalized, as a lot of robotic motion planning problems suffer by these aspects, like autonomous cars, but surely not all do, like a slow moving cleaning robot which does not mind being nonoptimal and hit a wall from time to time.
All of the source code and artifacts (scenes, snippets and more) can be found in my public github repository[1].

Assumptions

In this work I would like to address this problem of drone motion planning, while assuming these work assumptions:

1. The obstacles are known and static
2. The forces that affect the drone are inertia and gravity
3. The drone is simplified as a 2D point (x,y) and is represented in 4D (x,y,Vx, Vy)
4. The drone is accelerating using a motor
   a. thrust is limited
   b. instantaneous control over thrust and angle

[1] https://github.com/ohmydayum/Kinodynamics

Physical equations

I represent the state of the drone by [x, y, Vx, Vy], and assume that newton's laws of motion apply, and also regard gravitation. We get these simulation equations:

$$\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}_{n+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}_{n} + \frac{\Delta t}{2m} \cdot \begin{bmatrix} \Delta t(THRUST * sin(ANGLE)) \\ \Delta t(THRUST * sin(ANGLE) - g)) \\ THRUST * cos(ANGLE) \\ THRUST * sin(ANGLE) - g \end{bmatrix}$$

Where these controls are drawn uniformly in these ranges:

$$THRUST \in [0, MAX\_THRUST], ANGLE \in [0, 2 * \pi], \Delta t \in [0, MAX\_DURATION]$$

As we can see, it is not linear, and this is not fixable by adding the acceleration to the representation of the state, as the thrust component will still exist. Moreover, this does not take into account nonlinear forces like air drag ($F_{drag} \propto v^2$).

Probabilistic Road Maps

Published in 1996 by Karvaki et al[2]., the most basic and go-to nondeterministic algorithm is the Probabilistic Road Map algorithm, or PRM for short.

Algorithm

We develop a graph of valid states randomly selected from the configuration space. Each time we use a local planner to try and connect the new state with the already existing nodes. After some predefined limit, we stop and query the graph by adding the initial state and goal state to the nodes and connecting them to the others using the local planner. Then, by running some path finding algorithm, e.g. Dijkstra, we can discover a valid motion.

Caveats

Non holonomic spaces

In Layman's terms, we call a system in which the number of controlled degrees of freedom is less than the total degrees of freedom, nonholonomic. For example, a

[2] Kavraki, Lydia & Svestka, Petr & Latombe, J.C. & Overmars, M.H.. (1996). Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. Robotics and Automation, IEEE Transactions on. 12. 566 - 580. 10.1109/70.508439.

regular car cannot move sideways, although there are no obstacles, simply by system constraints.

Steering function

In order to connect two neighbouring states, the local planner needs to have what is called a steering function, that outputs the trajectory from one state to another state exactly. Generally, the steering function is not known or very expensive, time-wise.

Rapidly exploring Random Trees

Although PRM is very popular and simple, it lacks in some cases and thus unfitting to our need. The Rapidly exploring Random Trees algorithm, also known as RRT, was introduced in 2001 by LaValle and Kuffner[3], and since then adopted into many variants (most known is the RRT*). This nondeterministic algorithm enables us to address the kinodynamic nature of our problem. It Is important to note that RRT is probabilistic complete[4], while nonoptimal, meaning that if some valid motion exists, it will find a valid motion, and in some cases it won't be the shortest (according to some pre-chosen metrics).

Algorithm

We maintain a tree where the nodes are states and the edges are trajectories already found. Then, each time we draw a random state from the configuration space and efficiently find the closest state from the existing tree. Then, we draw random control inputs within system limits (in our case: thrust percentage, thrust angle, duration), and propagate that closest state using these controls for some random simulation time. If the result state and the trajectory are valid, i.e. don't collide with the obstacles and don't break system constraints (e.g. maximum speed), we add them to the tree. We continue until we are close enough to the goal, or some time/iteration limit is reached.

Collision detection

In some cases it is possible to get an analytic representation of the trajectory, and then check if the path crosses an obstacle or breaks an inert constraint. In some cases it is

[3]LaValle, S. M., & Kuffner, J. J. (2001). Randomized kinodynamic planning. International Journal of Robotics Research, 20(5), 378-400. https://doi.org/10.1177/02783640122067453.
[4] M. Kleinbort, K. Solovey, Z. Littlefield, K. E. Bekris and D. Halperin, "Probabilistic Completeness of RRT for Geometric and Kinodynamic Planning With Forward Propagation," in *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. x-xvi, April 2019.

not feasible, and this time it is out of scope. Therefore, I've implemented a naive sampling based collision detection, by approximating each trajectory to small straight lines, and then checked crossing of obstacles' edges using CGAL.

Nearest Neighbour Search

In order to keep the implementation efficient, instead of naively done in O(n), the search of the closest state is done using Kd Tree in O(log n). And so, I've started by using CGAL's KD_Tree but it did not support 4D points,so I switched to SciPy's KDTree which worked, and afterwards to SciPy's Cython implementation, cKDTree, which was obviously faster. After researching benchmarks[5], I ended up switching to scikit-learn's KdTree implementation.
It is vital to mention that the cost of building the tree is O(n log n) which is not a problem as is, but currently there is no support for tree insertions yet[6][7], thus forcing me to rebuild the trees each time a new state is found, and making the operation of adding new states to the tree a bad O(n log n) instead of the theoretically good O(log n).
This could be a major speedup! Still, in my experiments, the benefit of searching in O(log n) time outweighs the bad insertion time of (n log n).

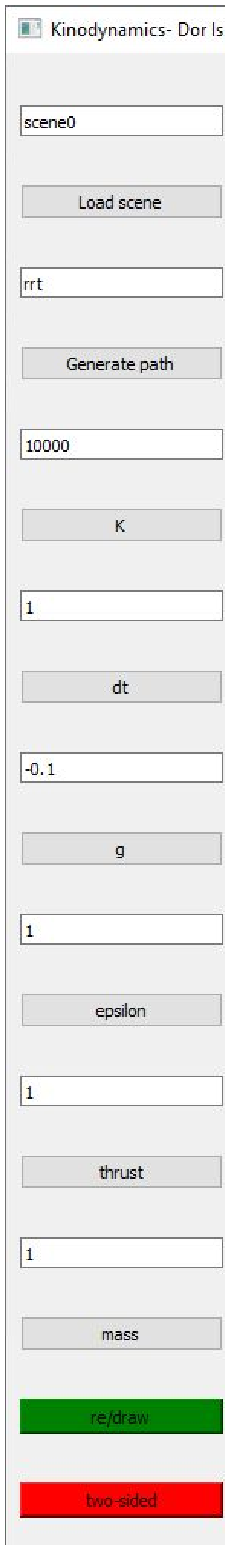
Two sided RRT

In the case the goal state is mostly blocked, the chances that the tree will grow close enough to the goal are slim. This is the logic behind fly traps- easy to enter, hard to escape. For this reason, it seems like a wise choice to not only propagate forward from the initial state, but also to propagate backwards from the goal. To do this, one needs to know how to "move backwards in time", or in other words, the inverse to the physical equations. In our case, it is simple and the equations are:

$$\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}_n = \begin{bmatrix} 1 & 0 & -\Delta t & 0 \\ 0 & 1 & 0 & -\Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}_{n+1} - \frac{\Delta t}{2m} \cdot \begin{bmatrix} \Delta t(THRUST * sin(ANGLE)) \\ \Delta t(THRUST * sin(ANGLE) - g)) \\ THRUST * cos(ANGLE) \\ THRUST * sin(ANGLE) - g \end{bmatrix}$$

Now, we just grow two RRTs simultaniasly, and each time we add a new state to one tree, we check if it is close enough to some state from the other tree, and not just the goal state. We will test this idea on a specially crafted scenario.

[5] https://jakevdp.github.io/blog/2013/04/29/benchmarking-nearest-neighbor-searches-in-python/
[6] https://github.com/scikit-learn/scikit-learn/issues/11909
[7] https://github.com/scipy/scipy/issues/9029



Kinodynamics- Dor Is

scene0

Load scene

rrt

Generate path

10000

K

1

dt

-0.1

g

1

epsilon

1

thrust

1

mass

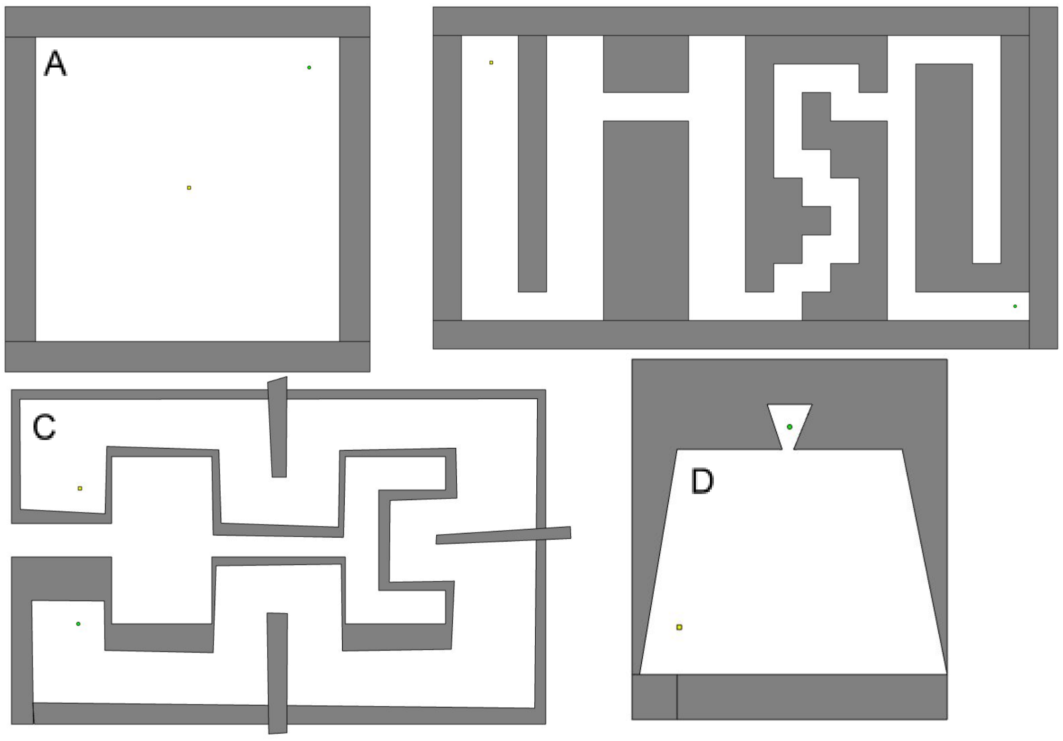
re/draw

two-sided

GUI: Controls and RTTs Visualization

In order to visualize and control the system, I have taken the basic PyQt5 GUI used in class for previous exercises and upgraded it in several ways. First, I have added a control bar that allows one to change the system options (max thrust, max duration, gravity, etc) with a click and now there's no need to change code and restart the program.
Second, I have added two extra switch buttons that reflect their state by their colors (red for "OFF", green for "ON") for: 1) should the two-sided extension be activated, and 2) should the program visualize all of the states tree. These switch buttons required me to alter the signature of the generate_path function, and also to redraw upon a click.

Lastly, I have crafted several scenarios to test the system in: (A) basic empty room, (B) obstacles course, (C) known paper example of double integrator robot[8], (D) fly-trap like scenario;

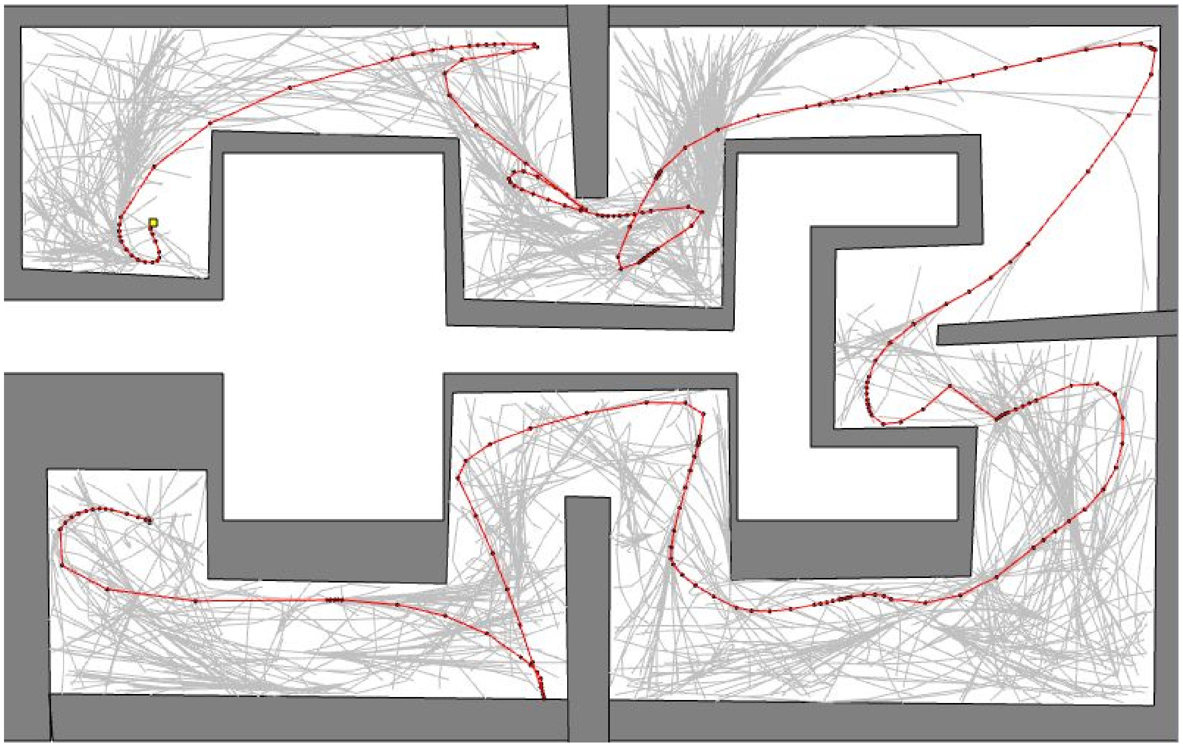


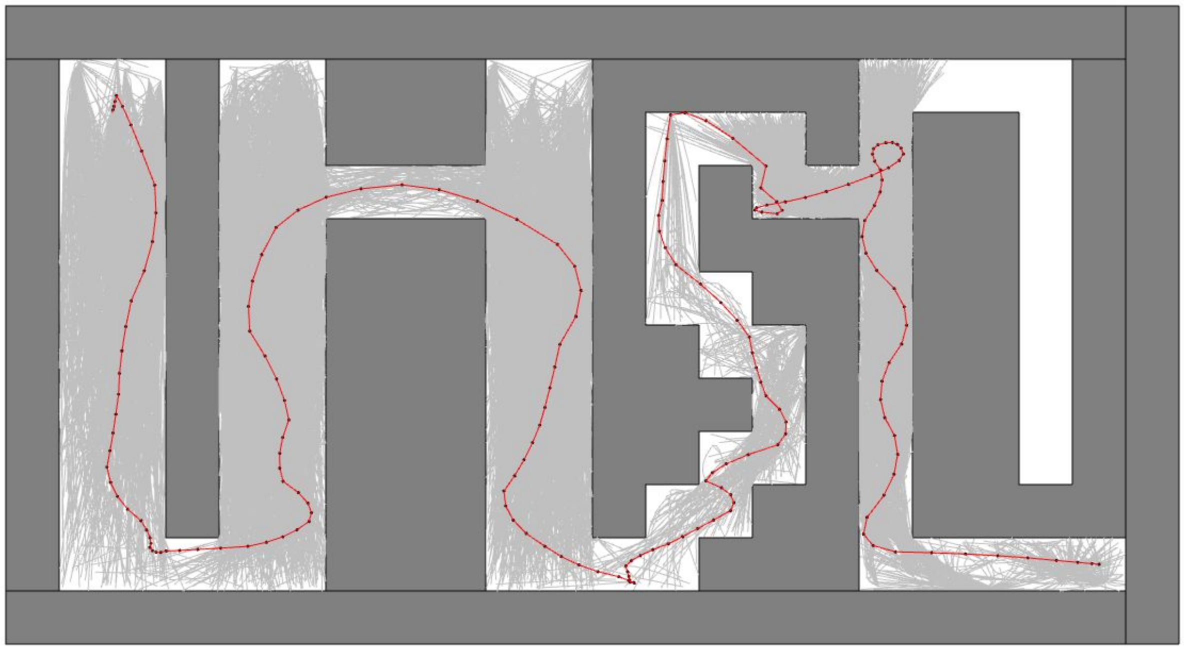
[8] Webb, Dustin & van den Berg, Jur. (2012). Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints.

Results

Planning a valid drone motion

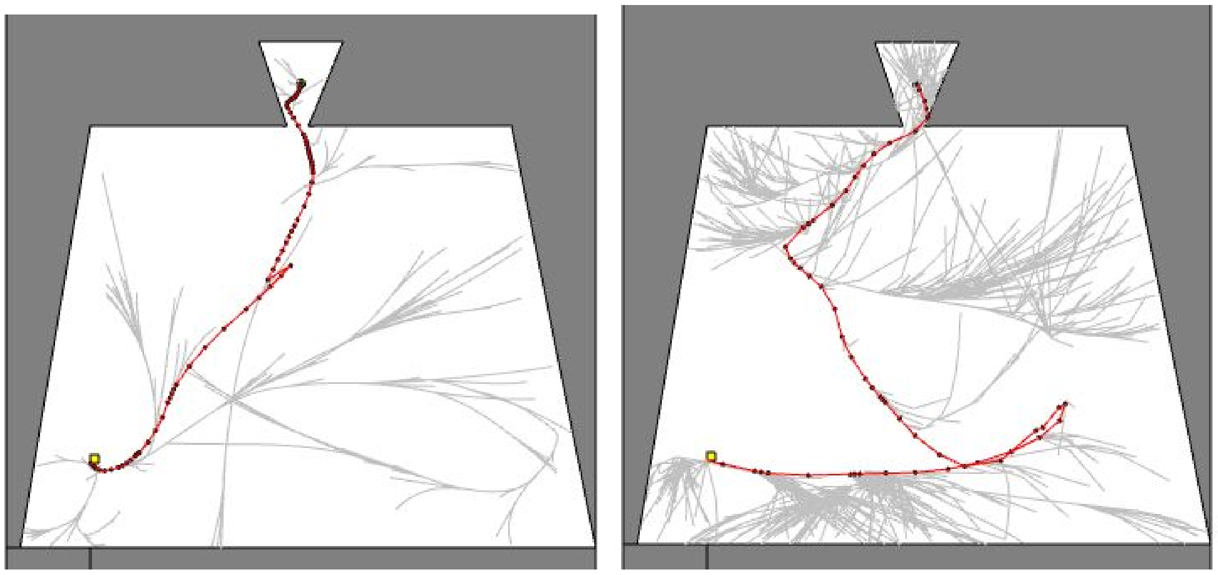
These are some example of valid (suboptimal) motion plans for our drone (<1 minute):





Effects of two-sided variant

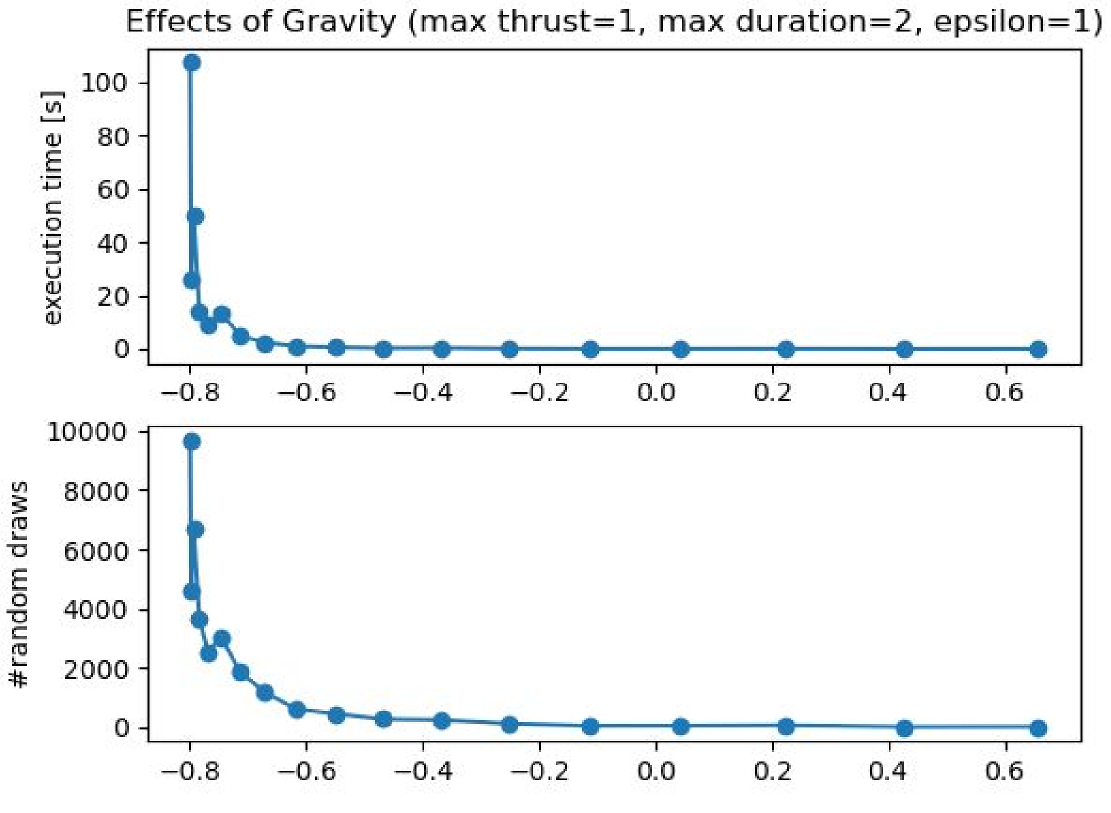
In the pathologic fly-trap scenario, the one-sided RRT takes too long to finish, while the two-sided RRT finishes trivially in a second (left: no gravity, right: strong gravity):



Another thing to note here is the “jump” in the middle of the left path, which is the connection of two RRTs with allowed error of ε=0.5.

Effects of intensifying gravity in empty room scenario

As the downward force of gravity intensifies, the planning becomes a challenging task:



Conclusion

In this work I ventured into implementing the elegante RRT algorithm, and its variant, the kinodynamic two-sided RRT, and experimented how inertia and gravitation affect the motion of a drone.
Also, I implemented a lean control and visualization program in PyQt5 and matplotlib, and compared several python libraries (CGAL+course python bindings, scipy, sklearn).

Future work

If one would like to extend this work, there are several immediate options:

- Improve tree update complexity to O(log n)
- Increase configuration space dimensions
- Upgrade to a 3D drone model
- Study effects of air drag on path and motion
- Implement RRT*