Safe Return-to-Launch for ArduPilot

Sebastian Quilter <squilter@mit.edu>

April 2, 2017

1 Introduction

This project aims to develop a "safe" return-to-launch feature for ArduCopter. This feature allows a copter to autonomously return to its home location while attempting to avoid obstacles that may exist between itself and its home.

Such a feature can be used to prevent crashes caused by standard RTL. In many circumstances, setting a higher RTL altitude will allow RTL to be used safely. But in certain circumstances, this does not solve the problem. For example, an RTL might be triggered unexpectedly while a copter is being used to inspect a bridge, tower or other tall structure. If the vehicle is on the opposite side of the structure as the home point, there might not be a realistic altitude at which it could safely return as the crow flies.

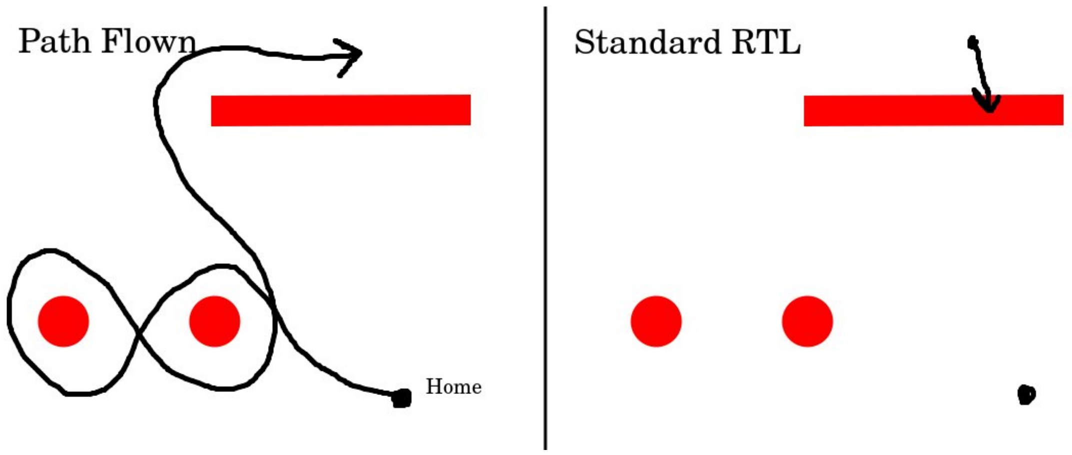

Figure 1: After flying a quadcopter through two pillars and behind a tall wall (Left), RTL is activated (Right). RTL will crash into the wall.

2 Possible approaches

2.1 Approach 1: Path finding

This approach would require the vehicle to record its location every X meters traveled, or every Y seconds. These waypoints will be recorded as "safe" points. When the RTL is triggered (by user, failsafe, etc.), a search algorithm will find an optimal path to home. This path will then be used to generate an auto mission, which will subsequently be run. This mission must not overwrite the primary mission created by the user.

This method might also require a clever data structure to store these points. Considerations must be made for the limited amount of memory available, and the limited amount of processing power. If the number of safe points is small enough, an unsorted linked list might be feasible. If there are too many, the path finding algorithm will take too long to find a suitable path. In this case, it would be better to use some sort of sorted data structure.

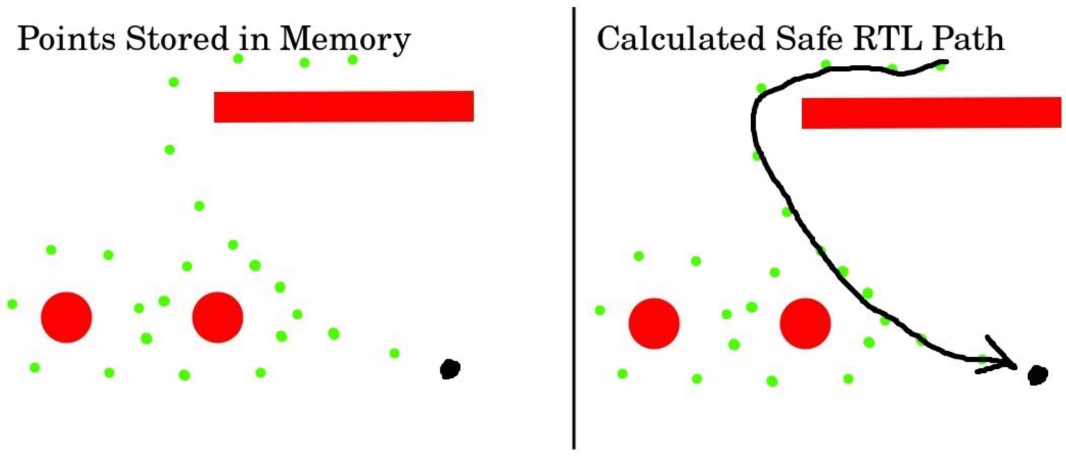

Figure 2: While flying, coordinate points are constantly being recorded in memory (Left). When Safe RTL is activated, a search algorithm finds the best path home through those points (Right).

2.2 Approach 2: Retrace steps

This method works by recording safe points every few meters, just as above. Instead of requiring a fancy data structure, these could be stored in a linked list. When RTL is triggered, this list must only be reversed. At this point, it could be fed into the controllers, or it could be turned into a mission.

This method avoids the problem of having to sort the points in real-time. The disadvantage to this method is that if the operator chooses to fly a "messy" path, the copter would then waste lots of time retracing steps unnecessarily. This could be mitigated by pruning loops from the list of safe points, either in real-time or immediately after being triggered.

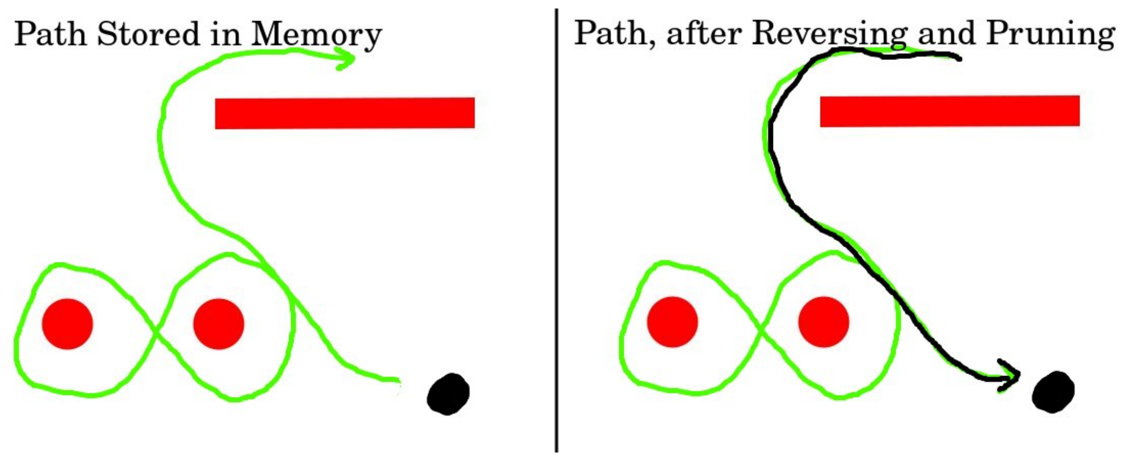

Figure 3: While flying, the path flown is being recorded in memory (Left). When Safe RTL is activated, the path is reversed and loops are removed (Right).

3 Possible Issues

3.1 Limited Memory

It is crucial that there is a cap on the amount of memory that can be used. Whether RAM or some other storage, it is important that there is a cap, because otherwise a very long flight would inevitable end with memory filling up and the system locking up. Even if such a hypothetical flight would need to be multiple days long before locking up the system, it would still be bad practice to knowingly leave a feature in the code which would put a time constraint on a flight.

This means that there must be a feature which deletes old points when appropriate. This will require some thought, and will vary depending on which method is chosen. For example, the "Retrace Steps" method could remove loops in real time, or when memory is close to full, rather than waiting until RTL is triggered. Still, a very long zig-zag path could potentially fill up memory. Eventually then, this method must either delete useful waypoints, or give up and declare that safe RTL is no longer available until reboot. For the "Path Finding" method, safe points that are close together can be deleted in real-time. Once again, this will fix the issue in most situations, but not all. One solution might be to run the search algorithm as soon as memory is close to full, and then delete any points that are not part of the solution. This will obviously delete some potentially useful points, but it guarantees that a solution can still be found after the cleanup is complete.

3.2 Scheduling

When safe RTL is triggered, an algorithm will need to calculate the return path, based on the information which it has stored. This algorithm must be designed within existing scheduling constraints, meaning that it must be able to run for a while, then pause, then keep running when execution is handed back, back and forth until the path has been fully calculated. If this algorithm ran continuously, it would likely prevent crucial system functions from executing in a timely manner which could result in a crash.

3.3 Tuning

Both methods listed above (and other methods not yet thought up) will require certain numbers to be tuned, which can only be done with thorough testing. For example, how often must safe points be recorded? The right balance must be struck between accuracy and memory-saving.

Another important variable for which a good value must be found is the minimum distance between points where it can safely be assumed that no obstacle is between them. For example, if two safe points are 1 meter apart, is it safe so assume that there is no obstacle between them? What about 5 meters or 10?

4 Roadmap

1. Write the entire feature with dronekit-python to run offboard. This will allow for easy debugging and visualization.
2. Collect some flight logs. Fly some "tricky" routes.
3. Feed these flight logs into the script, to visualize the flight and the resulting safe RTL path.
4. Test many different algorithms in python and choose the best implementation.
5. Begin coding actual implementation for ArduCopter. Start by just implementing the part that stores waypoints to memory in background. Demonstrate in flight.
6. Finish up implementation and demonstrate entire procedure in flight.
7. Continue developing other features, time permitting.

5 Links

Resume - `http://web.mit.edu/~squilter/www/resume.pdf`
GitHub - `http://github.com/squilter`