Finding a good path plan in arbitrary Cartesian coordinates is a requirement for any truly autonomous system. Currently the Rapidly-exploring Random Tree (RRT) algorithm has been shown to perform quickly at finding solution paths to a goal destination while navigating around any obstacles that may be in the way. While this algorithm is quite good at finding a valid solution, it unfortunately will never converge to the optimal solution. However, while the RRG and RRT* extensions provide a way of finding an optimal solutions, the published performance penalty reduces it's feasibility for the lunar path planning algorithm with severely limited processing power.

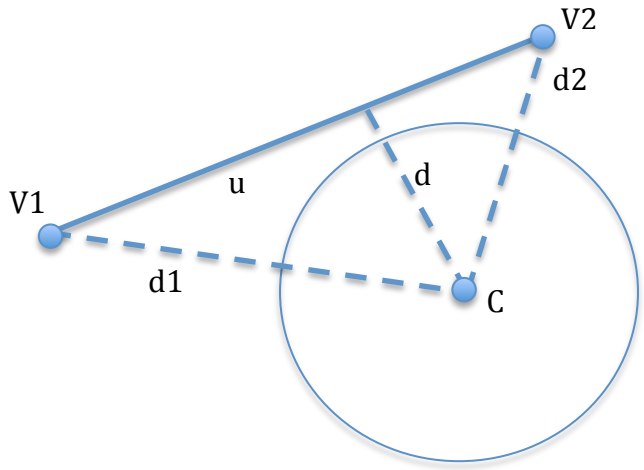
RRT searches the world for a valid path through the following algorithm. It starts with knowledge of the obstacles around it, its location, and the coordinates of the goal. To extend it's knowledge, it picks a random point in the entire space and determines the closest node that it knows how to get to. It then moves a small distance in that direction, and if that path is obstacle-free, it records this new vertex and new edge. The definition of the nearest neighbor node can be extended to include various metrics of path cost or weighted Euclidean space to achieve various tradeoffs. The algorithm terminates as soon as one path to the goal is found. Continuing computation may result in finding another route to the goal, however, it has been proven that it will never result in the optimal solution (except by very lucky chance).

I initially used brute force algorithms to path plan to provide a benchmark, reference implementation. Initial code profiling indicated the most expensive operation was nearest neighbor search. Enumerating every vertex that could be

extended at every time step was an expensive operation which scaled strongly with the number of known points and thus with the length / complexity of the path. I thus realized enormous timesaving here by using a kd-tree, in parallel with the edge / vertex tree used to store the node graph. The exact percentage depended upon the size of a step and the configuration of the area so I can't give exact descriptions, but the change from O(n) scaling to O(log n) was very apparent. I frequently saw factors of 4 reductions for many sample problems with my typical desired step size of 20%. This both necessitated using Euclidean distance for the determination of the nearest neighbor, and made it efficient.

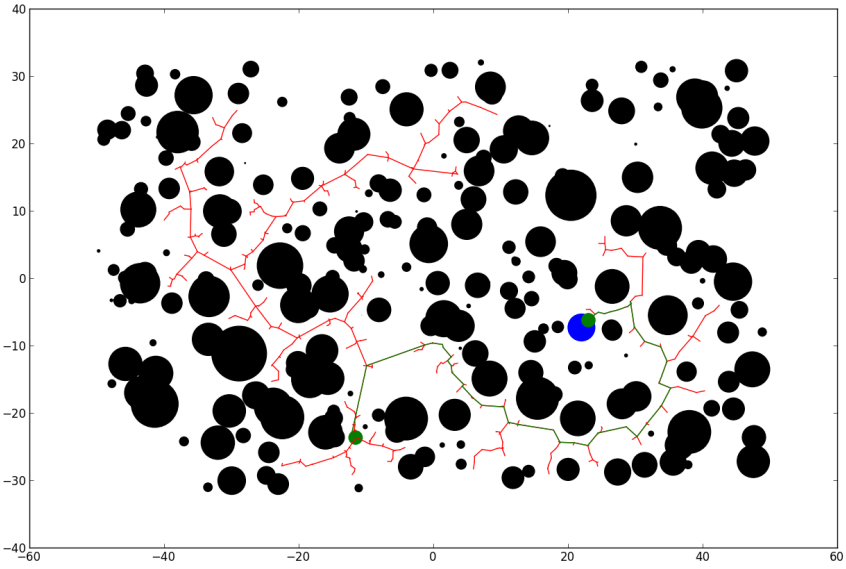
For collision checking, I assumed that obstacles could be of any size or location. This limited the amount that I could meaningfully optimize for speed. For instance, if obstacles could be approximated as points, the kd-tree used above would be very useful. However, representing an arbitrarily large circle as a point cloud would waste memory and still likely show little benefit in speed. Alternatively, if there are very few obstacles, discretizing the space can be very effective. For most general purposes, binning can also be effective, if the approximate obstacle size is known. Otherwise the algorithm may spend a considerable amount of time trying to determine which bins its path intersects with, and which obstacles in those bins it hasn't already tracked. Instead, I iterated over a simple linear list of all obstacles. I also found that up to a moderate number of obstacles (250+), the algorithm still performed very quickly. Each collision check involves the following steps: First, I do a quick check against the square circumscribed by the obstacle to determine if the line lies entirely to one side. This step saves considerable time as it involves only a

few linear operators and comparisons. Second, I check whether the new point is inside of the obstacle. Finally, I check that the point of minimum distance to the circle is between the endpoints and that this minimum distance avoids the obstacle. I add the radius of the robot to that of every obstacle when I load the environment, to allow me to treat the robot as a point mass here. Together, these checks ensure that the robot has a conflict free path from the current vertex to the new one. If this check passes for all obstacles, I add the new vertex to the current knowledge.



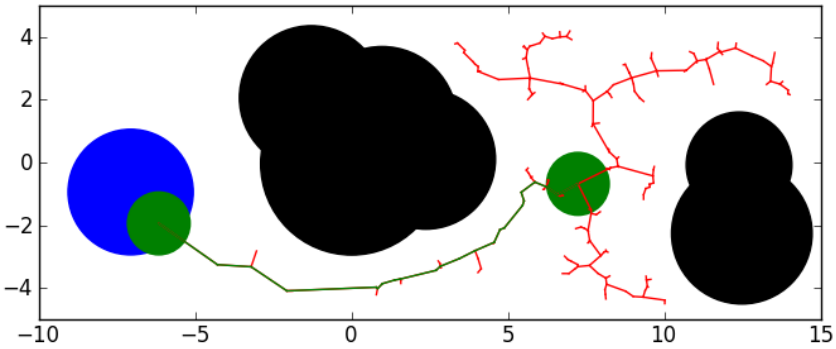
The final component of the algorithm is the sampling method used to extend the paths. For this, I used a uniform distribution over the entire box. But then, every 32 steps, I would take a sample just from the goal area. In a 4 sample trial runs of my large environment, I reduced the solution computation time from 967±837ms 204±52.9ms. Additionally, I have observed it can improve the quality of the path by making the path approach the goal more directly. See the plot on the next page for the output of a sample run of the large environment with the oversampling of the goal area every 32 steps.

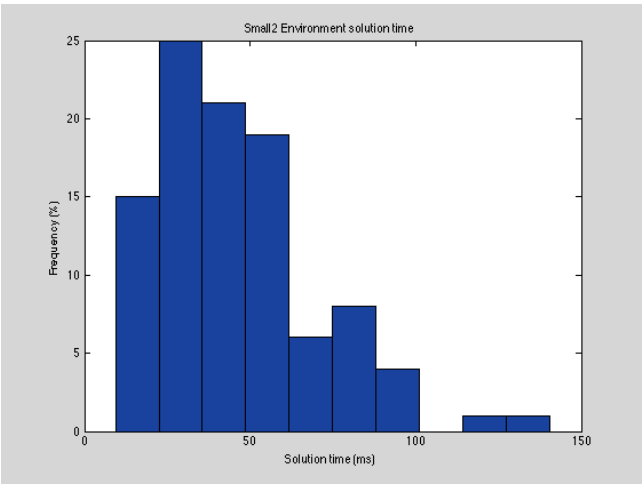
Large Environment

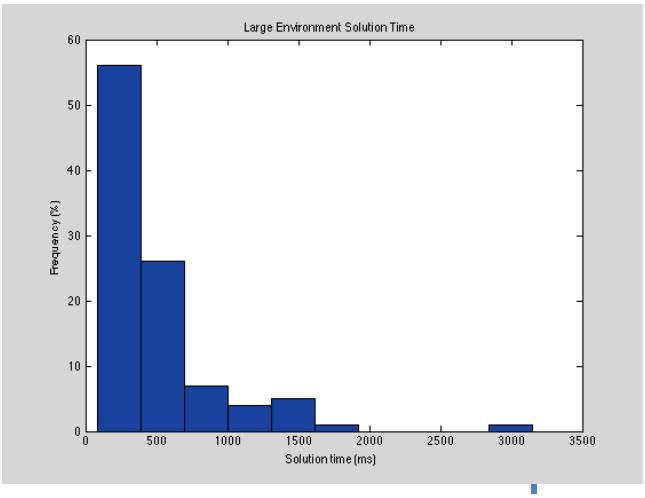


Too see the repeatable performance metrics, I got the following results for my “large” and “small2” environments. (See above and below for example solutions for each of these two environments, performance histograms follow.)

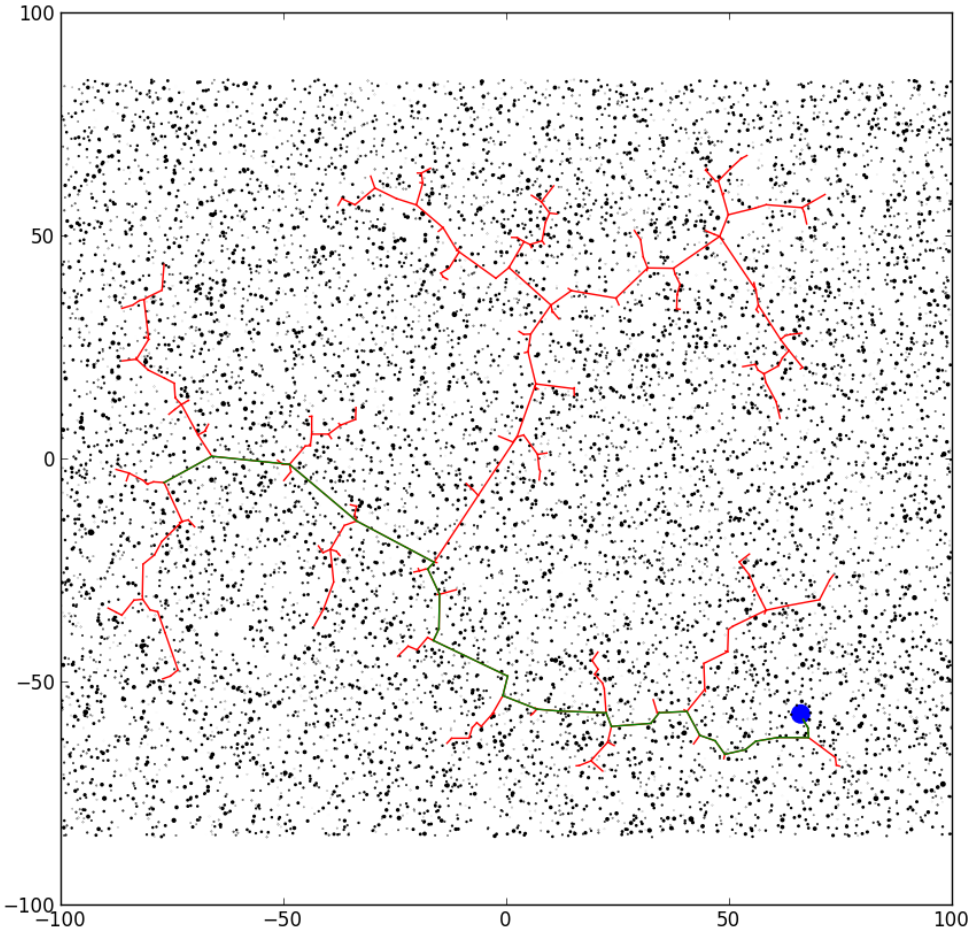
Small 2 Environment



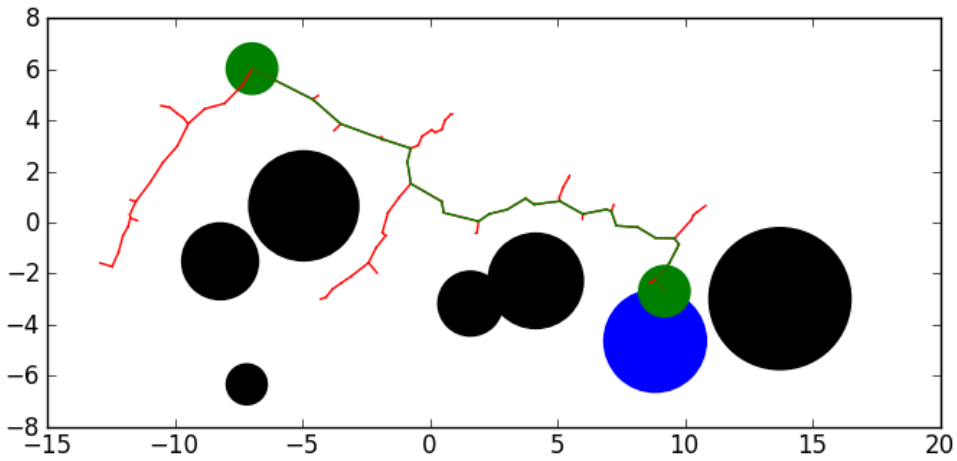




Finally, here are two additional problems. The first has an obstacle cloud of 10000 points. The robot is 1/10 the size of the obstacles to give it any chance of making it's way around. However, it takes only a couple of seconds to solve typically because, while the point density is very high, the path to the goal is fairly directly. Hopefully no real world robot is designed to require this much finesse in navigating the terrain!



On the other end, I made a very simple, but narrow hallway for the robot to navigate. The circles could represent bulk knowledge of doorways, tables, and expensive vases the robot must know to avoid. In many ground-based service instances, this is exactly the world formulation the robot is most likely to have.



The algorithm works quickly for problem instances with reasonable (small to medium) numbers of obstacles. The implementation does not handle exceptionally large numbers of obstacles very well. However, given the limited computational power of this lunar lander, I assume the obstacle detection algorithm will need to intelligently reduce this degree of the obstacles point cloud to large masses to be able to quickly update them and transfer them to the processor. In many cases, a mostly optimal solution can be found simply by smoothing out the path. As long as the smoothed path also avoids obstacles, this is probably a better solution. Thus, to avoid the complexity and time of using RRT, I propose that the driver controller on the robot generates progressively more smoothed approximations of the curve

between the points. This should allow the robot to gain additional drive performance without adding significantly to the computational cost. And in the event it cannot compute a smoother, conflict-free path, it can fall back to using the exact solution from the RRT solver.