Lab #6 Report: Path Planning

Team #8

German Espinosa
Andrew Manwaring
Habeeb Salau
Izzie Torres
Angelina Zhang

6.141: RSS

April 16, 2022

1 Introduction - Habeeb Salau

Path planning allows a robot to plan a viable trajectory between two locations on a map and follow the trajectory to the destination. This is an important property of any autonomous vehicle because it allows the vehicle to move between its current configuration and a goal configuration which is a precursor to higher-level tasks. In lab 6 we implemented the algorithms that would allow our robot to path plan.

Labs 3 and 4 focused on building autonomous motion guided by external cues in the robot's environment such as a wall or a line on the ground. This lab builds on these previous labs conceptually by continuing to equip the robot with autonomous motion. However, it is distinct from the other autonomous control labs because the robot no longer needs an external cue to automate its steering. Instead, it can internally decide two points on a map to navigate between and execute on this path. Lab 5 focused on localization which allowed the robot to have an internal model of the external world. This lab extends the results in lab 5 by equipping the robot to move based on its internal model of the world. The algorithms developed in this lab are especially important for the final challenge. The robot will need to be able quickly and accurately plan and optimize trajectories to successfully navigate.

The path planning execution is split into three parts. In the first, we developed algorithms for planning trajectories in a given occupancy grid map. We implemented and compared the trajectories produced by both A* and RRT search

algorithms and discuss our learnings in the paper. In the second part, we developed a pure pursuit algorithm that programs the car to follow a prefined trajectory. In the last part, we integrated the trajectory and the pure pursuit model to enable real-time path planning execution in the racecar simulation and robot. This culminated with an important milestone when the robot could navigate between any two locations in the Stata basement.

With the conclusion of this lab, we built an algorithm to plan and execute our robot's transition between any two positions on the map. In future sections, we further elaborate on our technical approach and experimental evaluation methods.

2 Technical Approach

2.1 Map Processing - Andrew Manwaring

In order to conduct a successful search for a path in the Stata basement, conversions to the map were made to account for physical constraints and improve runtime. While the path planning algorithms are run for a point-like racecar, the reality is that the racecar had the potential to hit obstacles that were up to 0.40 m away from the lidar, the simulated point of search for the car.

To account for this discrepancy, the obstacles needed to appear at least 0.4m larger than their actual size. In practice, a more conservative 0.46 m was chosen to enlarge the objects by. This was done using the scikit-image module's erosion function in python. The original map data, a 1D array of cells with values ranging from 0 to 100, each denoting the probability of occupation. This array was converted into a numpy 2D array of dimensions height x width of the stata basement map. To ensure no probable collisions would happen, only cells with a value of 0 were considered to be free. All other valued cells were expanded using the above mentioned erosion function by the appropriate distance, converted to pixels.

With the map ready for a path planner to plan a trajectory, initially this was enough to run a path planner. Our main path planning method, ended up being A* search, where each cell in the map is a node. At full resolution, this took prohibitively long to run over long trajectories and returned far more points than necessary. A scaling factor of 4 was introduced using Python Imaging Library's thumbnail function. This allowed the map to be scaled down in a factor while maintaining a similar aspect ratio, even if the width and height were not divisible by the number. The new cells could be converted back to coordinates by differentiating the x and y resolutions, since the cells were slightly rectangular.

Eroding and lower the resolution of the map took negligible time and is only

computed initially in the script. After the process above was complete, a map was returned for the trajectory planner to use.

2.2 Path Planning: A* Search - Andrew Manwaring

To find a trajectory with search based methods, we used A*, a method that searches based on "cost" of a path. This cost is generally a cost incurred so far + heuristic that estimates the cost to the goal.

Our A* implementation was as follows, following the standard A* algorithm:

1. Initialize start and end points. Place the start point into the queue of paths.
2. Remove the minimum cost path from the queue. If that path is the goal, return the path. The last node in this path is marked so that it will not be searched again.
3. Find all of the neighbors, ie all the nodes that can be reached from the last node in the path. For our implementation, this was the 8 cells surrounding a node on a 3x3 grid. If any of these cells were marked as occupied by an obstacle, there were not considered in the search.
    a If the neighbor is already in the queue, then assign it the minimum cost of reaching it, reassigning the path to it path if necessary. This step guarantees optimality by requiring the minimum path to the goal be the sum of minimum paths to each point along the trajectory.
    b Otherwise, add the neighbor path to the queue with a cost associated with it.
4. Steps 2 3 are repeated while there are nodes in the queue or until a goal is returned. If the queue is empty, then no path was found and an empty list is returned.

The cost was first calculated as the sum of distance so far and a distance heuristic. This heuristic is the linear distance between a cell and the goal and is admissible because the actual distance to goal will always be greater than or equal to the linear distance. This produced a path that would work in an ideal world, but still did not steer clear enough of walls; the pure pursuit can't be on the line 100% of the time. More conservative erosion values were attempted, but these closed off tight areas, blocking off clearly traversal area.
The solution that worked best was a wall cost addition to each path. Using the scan simulator from Lab 5, at each point the minimum distance to the wall was estimated. A cost of $\frac{1}{\text{distance to wall}}$ was added to the distance so far and linear distance heuristic. It is worth noting that this is no longer admissible, because the cost my overestimate the actual cost, leading to a loss in optimality. The wall cost was enough to keep the trajectory away from walls, but did not steer the trajectory far from the optimal path in practice because its value was comparatively low to the linear distance or distance so far.

2.3 Path Planning: RRT - Izzie Torres

As an alternative to A*, we also implemented an RRT path planning algorithm. The goal was to create an algorithm that would generate trajectories faster than A*, although the trajectories may be less optimal.

In this algorithm, we would randomly generate points on the map and connect them to the nearest nodes of a growing path tree. These points were stored in a numpy array with four columns; two columns held the x and y values, one recorded which row the parent node was in, and one recorded the distance between nodes for evaluation purposes. As each point was generated, it was matched to the nearest node by calculating the distance to every single previously recorded node, then finding which point had the minimum distance. This point was recorded as the parent, and a branch was created.

For each branch, we also had to make sure no obstacles were obstructing the path. This was done by finding the slope and intercept of the line connecting the two endpoints of the branch. Another 50 evenly-spaced points were then generated along that line. Those points' x and y values were then rounded down to get integers that could be used as indices in the occupancy map. The branch would then be discarded if the probability of an obstacle in any one of those points was greater than 0.

These steps would continue until a new branch was generated within a certain distance of the end goal. Some final data processing would then reconstruct the path from the recorded points, convert from pixels into meters, construct a final trajectory, and publish the results.

2.4 Pure Pursuit - Habeeb Salau

Once a robot has a candidate trajectory to follow generated from methods like RRT and A*, it needs to have an efficient method to drive along the path. This is the problem of converting a trajectory – a series of x,y coordinates in the map frame – into steering commands for the robot. Pure pursuit is the algorithm we used to solve this technical problem.

In the simplest implementation of pure pursuit, we can visualize an airplane with engines that can swivel in any direction to provide lateral acceleration. Once a point on the plane's trajectory is identified as a goal point, the radius of the arc connecting the location to the goal point is determined. Using the equations for centripetal acceleration, the necessary acceleration to keep the plane on the arc is calculated with the arc's radius and the plane's velocity. This basic implementation extends to our car with some modifications to account for the car not having lateral accelerators and the constraints on turning provided by the car's axles.

2.4.1 Trajectory Detection

The first step in our implementation of pure pursuit is to find the point on the trajectory that is closest to the car's current position as determined by localization. We decided to represent the trajectory as an array of points because the points would be ordered (from start point to endpoint) and we'd use the indexing operation frequently. Since the trajectory is made of line segments, and the car's position is a point we decomposed this problem into one for finding the closest point on a line segment to a position point. We searched through all of the line segments on the trajectory and found the closest point as well as the index for the point that started the line segment.

```
closestpoint ← None
closestdistance ← Infinity
closestlinesegment ← None
for line segment in trajectory do
    calculate distance between pose and segment ;
    if calculated distance < closest distance then
        closestdistance ← calculateddistance
        closestlinesegment ← linesegment
    end
end
```

Algorithm 1: Pseudocode for function that calculates the point on the trajectory that is closest to the pose point.

2.4.2 Look-ahead Point

The second step involved finding a goal point for the robot to drive towards on the trajectory. Here we defined our lookahead distance to be 2 meters (rationale explained in the technical tradeoff section) and found the intersections with the circle centered on the car and the lookahead radius. This problem's geometric analog is the intersection of a circle and a line segment. We used optimization code for finding the line segment - circle intersections for each line segment and only searched with line segments downstream of the line segment found in step 1.

In the corner cases where there is more than one candidate intersection, we choose the point that is on the line segment furthest downstream. We break ties between intersections on the same segment by choosing the intersection closest to the downstream endpoint.

candidategoalpoints ← *array*
for *line segment in trajectory after closest segment* **do**
  calculate intersections between circle and segment ;
  add intersections to candidate goal points ;
end
if *len(candidate goal points) > 1* **then**
  choose most downstream point from candidate goal points ;
end

Algorithm 2: The pseudocode for finding the goal point once the closest point on the line is determined.

2.4.3 Controller Implementation

Finally, we use the goal point to inform the car's steering commands. We adapted the lateral acceleration model discussed earlier to a bicycle model and calculated - η - the angle between the direction of the pose and the vector from the position to the goal point. We used this to determine δ – the steering angle correction.

$$\delta = -\tan^{-1}\left(\frac{L\sin\eta}{\frac{L_{\text{fw}}}{2} + l_{\text{fw}}\cos\eta}\right)$$

Figure 1: The relationship dictating steering angle δ and the pose relative to the goal point η

2.4.4 Tradeoffs and considerations:

1. Lookahead distance: We decided to have a static look ahead distance of 2 meters over a dynamic one. We did this because the car performed well with corners in simulation with this value, and our attempts at creating a dynamic lookahead distance caused the car to react slowly to corners. The tradeoff here was that we gained coding simplicity while potentially sacrificing stability with a dynamic lookahead distance.

2. Position offset: The position of the car that we use in pure pursuit is 5 cm in front of the rear axle. This creates additional stability for the car and we varied this parameter while visually assessing pure pursuit in simulation. Varying this parameter had minimal impact on performance.

2.5 Integration - German Espinosa

Once we had a functional path planner and trajectory follower, we sought to tune the variable parameters in both algorithms in order to get the best overall

performance. From A*, we have to continuously alter the definition of the map which is given by the amount of erosion. The more we erode the map, the less the algorithm can see which might potentially result in "No path found" errors. On the other hand, the pure pursuit controller has two main variables: look-ahead distance and speed. When isolated, we found that a constant look-ahead distance performed well enough; nonetheless, we plan to implement a dynamic look-ahead as mentioned in Section 2.4.4. Along an open hall way, we would prefer higher speeds and look-ahead distances, but we would prefer lower values along the corners. This condition becomes even more important because our current A* implementation finds an optimal path closer to the wall. Therefore, large look-ahead distances near corners can potentially cause the car to hit the wall. Throughout the following two sections, we shall go over the challenges faced in both simulation and the real-environment.

2.5.1 Simulation

In order to easily visualize the performance of our algorithms, we added multiple markers within RVIZ which demonstrated our particle filter pose, goal point, trajectory, and end points. These markers can be seen in Figure 4. This approach allowed us to easily understand the scenarios in which our vehicle cut corners and whether or not localization was to blame. We noticed several scenarios where our particle filter pose left the car's position causing obvious collisions; nonetheless, we found that devoting time to optimize the staff solution was not in our best interest. However, our bugs were not limited to the localization module which we will now explore.

As mentioned above, we found that there were several issues with the path planner which led to the implementation of the wall cost mentioned in Section 2.2. Therefore, the optimal paths found from the algorithm caused several collisions within the simulator. Nonetheless, we were able to easily amend this by reducing our look-ahead distance from 10m to 1m. Since our optimal trajectory contains hundreds of points, having a small look-ahead constantly detects small lateral deltas between consecutive points which causes our car to pursue a point that quickly shifts between right and left. Therefore, we increased our look-ahead distance to 2m which solved any potential collisions within simulator testing. We tried our best to maintain high speeds in order to get the best completion times, but we understand that this may not translate well to the real environment.

2.5.2 Real Environment

Once we translated to the real environment, we found that many of the real-world delays caused our vehicle to perform well; however, we spent a large amount of time understanding how to use localization properly. Once we were able to localize the car correctly, we found that it drifted very little from the

intended path. Although we were able to localize the car, we were not able to use the convenient functionalities RVIZ had to offer because of rostopic confusions. As a solution, we manually sent out initial and end pose information to the relevant topics needed for our algorithms to run. After taking this approach, we found that we could easily evaluate the car as needed.

To our surprise, we had to do very little tuning to get our car to perform well. The small look-ahead distance we used in simulation translated well to the real world. Our primary issue came from the A* algorithm being unable to find a trajectory when in small hall ways. We believed the use of erosion caused the robot to think that there was no viable space; hence, causing a "No Path Found" error. We were able to fix this error by tuning the functionality of the erosion.

Once we solved the above issue, we found that the car had no problem driving at speeds ranging from 1 m/s to 5 m/s. However, we noticed that the car behaved more erratically around corners at higher speeds due to the high density of points and small look-ahead. Nonetheless, we had to maintain a small look-ahead to prevent head on collisions. In conclusion, we were able to find the optimal trajectory between two points which we were able to follow efficiently using a pure pursuit controller.

3 Experimental Evaluation

3.1 A* in Simulation - Angelina Zhang

To evaluate our A* algorithm in simulation, we compared our planned trajectory with the staff solution on Gradescope and tracked its runtime to complete a lap around the Stata basement. Initially, our implementation of A* resulted in code timeouts, since the algorithm was often unable to find a path. After careful investigation, we noticed that this occurred because of our erosion method. As mentioned previously, erosion expands occluded areas on the map, such as walls and pillars, enabling the robot to steer clear of such obstacles. However, this eroded already-narrow hallways into spaces too narrow for the robot to navigate, preventing it from planning a trajectory. Thus, we decreased our obstacle enlargement factor to limit the amount of erosion. However, we found that this caused the robot to enter occluded spaces, crashing in simulation. After experimenting with different values, we found that an obstacle enlargement factor of 0.46 was most optimal, as it resulted in the lowest runtime and highest similarity with the staff solution (highest score on Gradescope). Our trajectory, compared with the staff solution, is displayed in Figure 1 below. We see that our implementation closely follows the staff solution, finding a direct path without entering occludedn spaces. We will discuss runtime further in the following section.

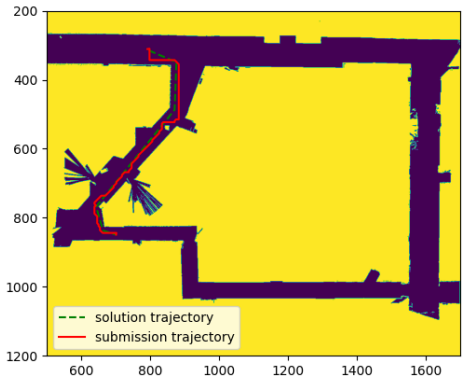

Figure 2: Our A* algorithm (red) compared to the staff trajectory (green)

3.2 RRT in Simulation - Izzie Torres

In order to determine which path planning algorithm to use on the robot, we compared the relative performances of RRT and A* in simulation. This was done by modifying the code to run both algorithms for each pair of start and end points, then publish the length of the path when it was finished planning. Thus, by recording a bag file of the published path lengths, we could compare both the path lengths and the time (based on how long it took between each published message).

We took 18 trials of different start and end points, then computed the relative time and path lengths (shown below). One trial was discarded due to bad recording. These results showed that, on average, RRT was a 47.90% improvement in time, but a 638.20% increase in path length. This increase in path length was so egregious that it far outweighed any time benefits of RRT, especially as that saved time would be lost in the extra time spent traversing an unnecessarily long path. Further, while the path length was longer for RRT in every single trial, the time RRT took to compute was not always faster than A*. The path length issue could be improved in the future by modifying the algorithm to use RRT*, but for the purposes of this lab we decided to stick with A*.

Trial	A* Time	RRT Time	% Diff	A* Dist	RRT Dist	% Diff
1	1206	48	-95.99	105.9	577.9	445.59
2	153	190	24.25	44.1	247.0	460.24
3	416	247	-40.69	66.4	334.1	403.25
4	450	146	-67.59	56.1	270.5	382.02
5	39	2	-94.94	33.6	317.9	847.01
6	27	7	-73.37	10.7	186.2	1638.15
7	23	1	-95.03	10.6	33.6	215.82
8	682	93	-86.36	68.5	444.7	549.24
9	137	253	84.43	30.3	645.7	2028.19
10	408	25	-93.81	46.6	510.8	995.57
11	68	13	-80.82	35.1	187.2	432.99
12	1105	228	-79.37	79.7	397.0	398.10
13	817	9	-98.86	73.2	377.9	416.10
14	87	301	245.35	25.7	114.2	343.80
15	117	25	-78.89	44.4	254.2	471.90
16	222	30	-86.32	44.7	236.7	429.71
17	1013	37	-96.32	80.0	393.6	391.78

Note that the distance units are in pixels, while the time units are in ROS-specific counts divided by 10^7. While the units make little sense in a physical context, the relative values work well for comparison, even without any conversions to meters or seconds.

3.3 Pure Pursuit in Simulation - Angelina Zhang

We evaluated our pure pursuit implementation based on how well the car followed the specified trajectory. We did this in two ways:

First, we used the Gradescope autograder to determine how often the car stayed on the trajectory. After several attempts, we determined that our car had 95.03% accuracy in following a given trajectory. We determined this was good enough performance in simulation, as integrating in the real world would require additional tuning. Figure 2 illustrates our car's trajectory, compared to the desired path.

Next, we tracked the error over time, the distance between the car's position and the closest point on the given trajectory. Our car had an average error of 0.09 meters, and was never over 0.3m from the planned path. We observed that the car had the greatest error when it completed turns, as seen through the spikes in the error plot. This may occur because the car is unable to change steering angles fast enough to accomodate the planned path. We determined this was good enough performance in simulation, as with a physical racecar, odometry measures are not perfect, and there is bound to be error from the planned trajectory. Our error plot is depicted in Figure 3, and the corresponding trajectory, a lap around the Stata basement, is depicted in Figure 4.

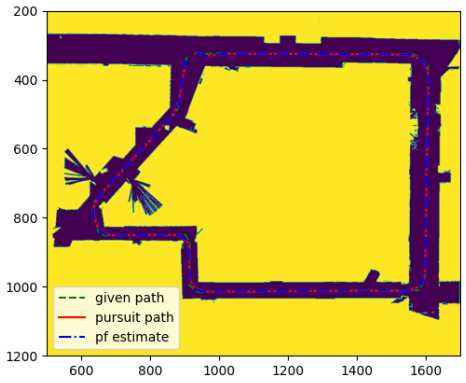

Figure 3: Our actual trajectory (red) compared to the desired trajectory (green)

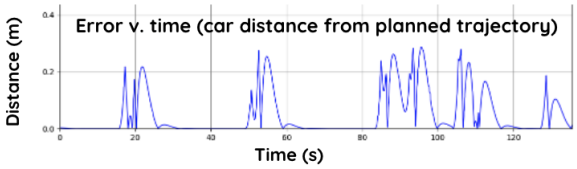

Figure 4: Our car's error over time

3.4 Integration in Real Life - German Espinosa

Using the same error metrics defined in Section 3.3, we were able to measure the performance of our integrated module in the real environment. From Figure 5, we can see that our max and average errors were far less than the examples used in simulation. The previous examples exhibited errors reaching nearly .35 m whereas we stay within .2 m of the intended path. After completing the given trajectory for this example, we found an average error of .10 m. We followed this process a few more times with look-ahead distances ranging from [1 m , 5 m] and speeds ranging from [2 m/s , 5 m/s]. As expected, we noticed that the car was getting dangerously close to corners when the look-ahead distance exceeded 4 m. To our surprise, the increase in speeds did not affect how close we got to the wall, but it reduced the smoothness of the followed path. Given that we prioritize faster completion times, we preferred higher speeds. Therefore, we found that the optimal look-ahead distance was 2 m and the optimal speed was 5 m/s.

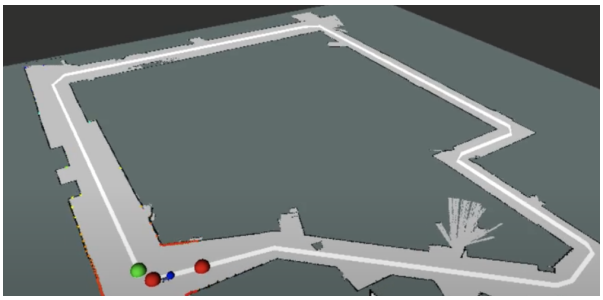

Figure 5: The planned trajectory corresponding to the error plot in Figure 3.

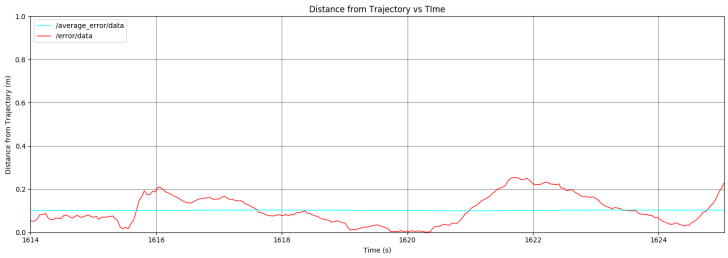

Figure 6: Example demonstrating error (instantaneous and average) versus time with look-ahead = 2 m and speed = 2 m/s following a path around a corner.

4 Conclusion - Angelina Zhang

Through this lab, we have successfully implemented path planning and pure pursuit control. We wrote two algorithms for path planning, A* and RRT, that enable us to create a trajectory, given a start and end point. During evaluation, we found that A* generally performed better than RRT–it created shorter trajectories and had more robust collision detection. Thus, we opted to use A* when integrating with the physical racecar. We used pure pursuit control, along with the particle filter from the localization lab, to follow the trajectory. Through integration, we were able to program a racecar that drives autonomously to a specified location.

Before the final challenge, we hope to improve upon our current work. In particular, we hope to make path planning more robust. In our current implementation, the published trajectory tends to drive close to walls. As a result, our car does not always take the most efficient path. Instead of driving diagonally, it will turn at right angles. We hope to fix this issue by changing the way we process the map and reassigning wall costs so that the trajectory does not directly trace against walls.

5 Lessons Learned

5.1 Andrew Manwaring

I learned about using many ROS nodes in series with eachother requires careful analysis of the topics they are publishing to and from. There was a careful order in this lab from the LiDAR to localization to path planner to the pure pursuit, so a few bugs were found in the topics these were publishing and subscribing to. Additionally, I learned about asking for help earlier because we had some issues that could not be debugged without support from teaching staff but still used a lot of time thinking over. Working with a group all trying to find bugs at the same time helped for solving these problems faster.

I also found with the admissible heuristic that the optimal path is not always practical in reality. This was the case for adding in the wall cost.

5.2 Angelina Zhang

I learned about the importance of debugging in parallel and asking for help. I was initially unable to get the staff solution for localization to run, and spent many days trying to debug. I later consulted Piazza and also found that other groups were having similar issues. I then was able to make the relevant changes fairly quickly and use localization during integration.

5.3 German Espinosa

I learned that it is important to trace publisher/subscriber rostopics in order to find an error. The base parameters for many of the variables were often intended for simulation which resulted in many errors when transitioning to the real world. As students, we generally believe our own implementation of our algorithms cause errors, but sometimes it is as simple as renaming the topic name. Therefore, I will make sure to start with the basics before jumping to complicated conclusions next time.

In addition, I learned that it is important to prioritize functionality versus visibility. Since I took the time to add markers in simulation to facilitate debugging, I planned to do the same thing with real environment testing. However, there were rostopic conflicts within the simulator which made this impossible without extensive knowledge regarding the simulator itself. Nonetheless, I made the decision to spend countless hours trying to figure out this process in order to match the simulator workflow. Eventually, I gave up and managed to use path planning manually within an hour. This experience is definitely something I will consider going forward.

5.4 Habeeb Salau

During this lab, I learned the importance of developing modular code for debugging. When writing the pure pursuit algorithms, I broke up the code into several helper functions to support each of the three sub problems. When it came time to test out the pure pursuit algorithm in simulation, it was significantly easier than in previous labs to isolate the cause of an issue from a helper function. Moving forward, I'll work on developing more modular code and rewriting some of our older libraries into helper functions. As we approach the final challenge, it'll help the team quickly debug issues saving us costly development time.

5.5 Izzie Torres

During this lab, I received yet another lesson in reading instructions carefully. When I first sat down to write the RRT function, I wrote a probabilistic map generator and basic search algorithm instead. I did not discover this until the next day, when I had to start completely from scratch. I then ran into an issue where I could not get the trajectory to show in rviz. It turned out that I was running the wrong launch file because I had not checked what the launch file was actually launching. Going forward, I recognize that I must be much more careful reading instructions, or I might waste even more hours confused or writing unnecessary code.