Assignment 1

Conor Hayes

## Part A

**1. Build a grid with cell sizes of 1 x1 m , and a range of x: [−2,5 ] , y :[−6,6] . Mark each cell that contains a landmark as occupied.**

Done.

**2. Implement the A* algorithm. You can assume that you have knowledge of which cells are occupied. Design an admissible heuristic, given the true cost function described above. Write up all relevant equations (for your heuristic), and justify why your heuristic is admissible.**

A heuristic for A* is said to be "admissible" if it never overestimates the cost of reaching the goal. In the case of this 2D grid, a simple admissible heuristic is:

$$h(n) = \frac{\sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}}{d}$$

$(x_n, y_n)$: grid location of node $n$

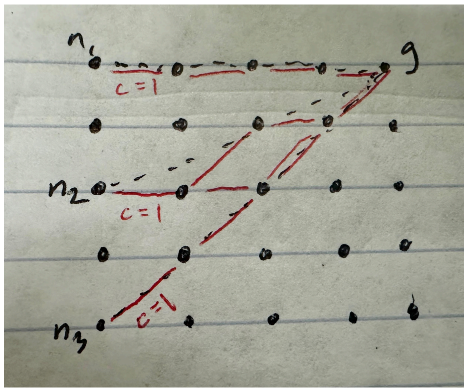
$(x_g, y_g)$: grid location of goal

$d$: cell diagonal length $= \sqrt{2}$

*Eq 1.1 - admissible cartesian distance-based heuristic for A* grid search*

That is–the cartesian distance to the goal, divided by the length of the cell diagonal. This is admissible by the following reasoning:

- The longest distance traveled for a cost of 1 is along the diagonal of a cell, i.e. to a corner neighbor
- The heuristic above gives an average edge cost of 1 in the case of a path composed entirely of diagonal steps, and an average edge cost of <1 for all other paths (i.e. those containing non-diagonal steps).
- Therefore, this heuristic never overestimates the path cost.



$$c(n_1),\ c(n_2),\ c(n_3) = 4$$

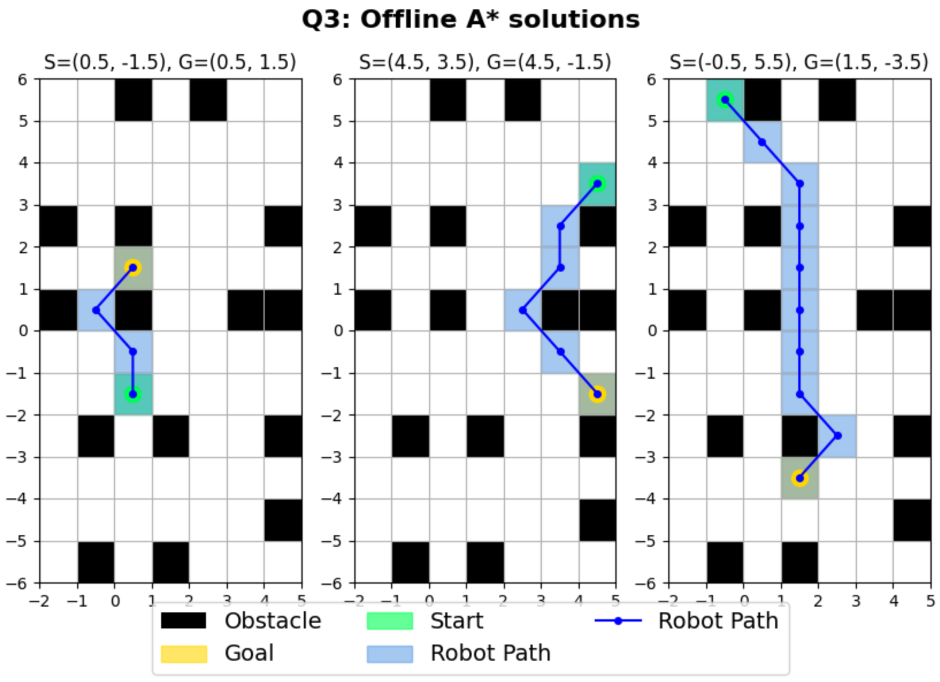
$$h(n_1) = \frac{4}{\sqrt{2}} \approx 2.83$$

$$h(n_2) = \sqrt{20}/\sqrt{2} \approx 3.16$$

$$h(n_3) = 4$$

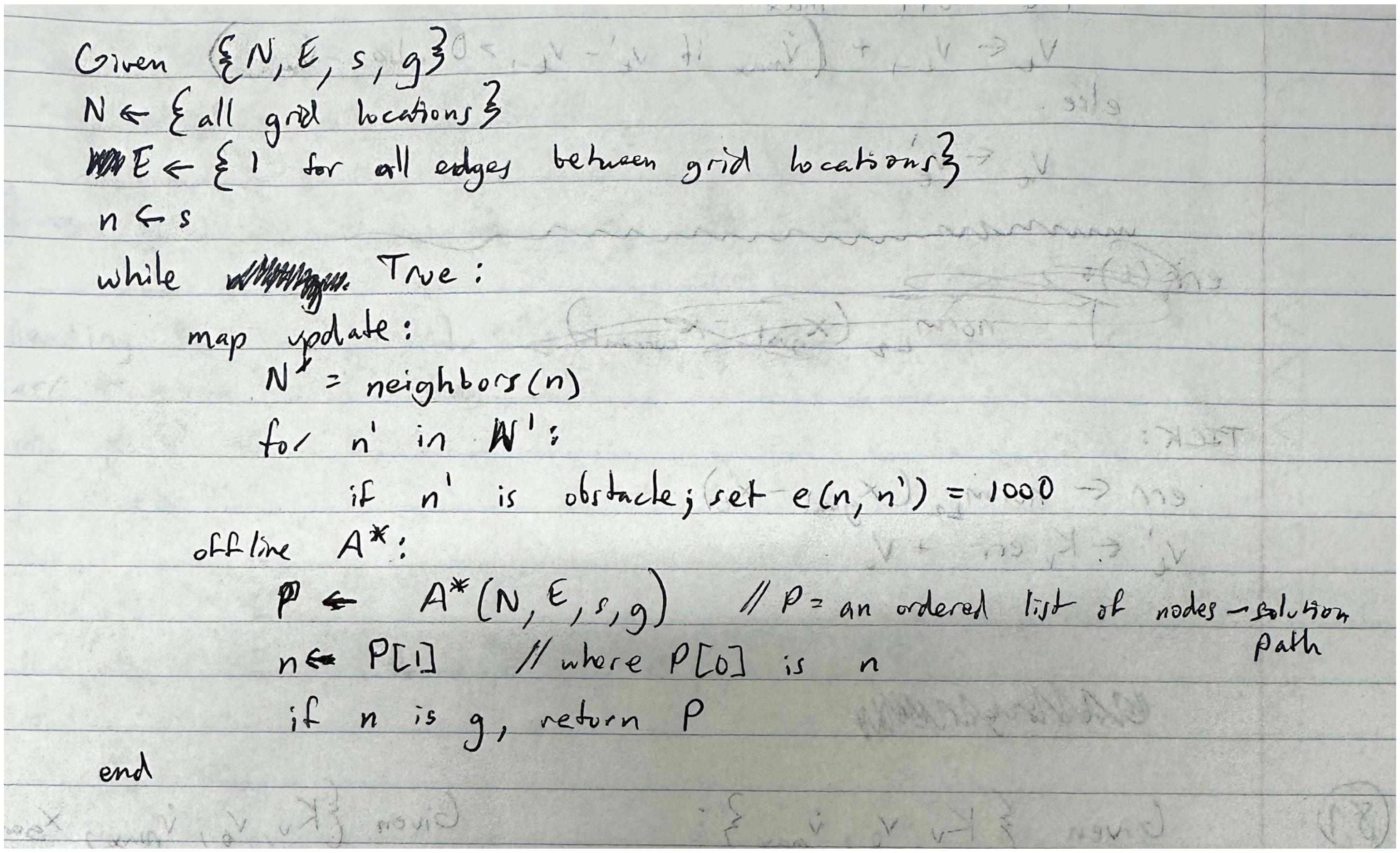
*Fig 1.2 – Left: Paths taken by the robot (pink), which are composed of diagonal & straight edges, are always at least the same length as straight-line paths (dotted black). They are only equal in the limiting case of all-diagonal edges. Dividing the cartesian distance by the cost of this limiting case, therefore, results in an admissible heuristic.*
*Right: costs & heuristic values for the paths $n_i \to g$ on the left figure.*

**3. Plan paths between the following sets of start (S) and goal (G) positions. Provide a visual display of your results, that shows each grid cell, distinguishes occupied cells, and also shows the planned paths.**



**4. Now modify your A* algorithm to be "online": that plans as the robot moves and does not have a priori knowledge of the obstacles (but rather, can observe them only when physically in a neighbor cell to the obstacle). What does this mean for the set of expanded nodes? Describe any modifications made to your A* algorithm from Step 2.**

This "online" version of A* is implemented as follows:



```
Given {N, E, s, g}
N ← {all grid locations}
E ← {1 for all edges between grid locations}
n ← s
while True:
    map update:
        N' = neighbors(n)
        for n' in N':
            if n' is obstacle; set e(n, n') = 1000
    offline A*:
        P ← A*(N, E, s, g)    // P = an ordered list of nodes - solution path
        n ← P[1]    // where P[0] is n
        if n is g, return P
end
```

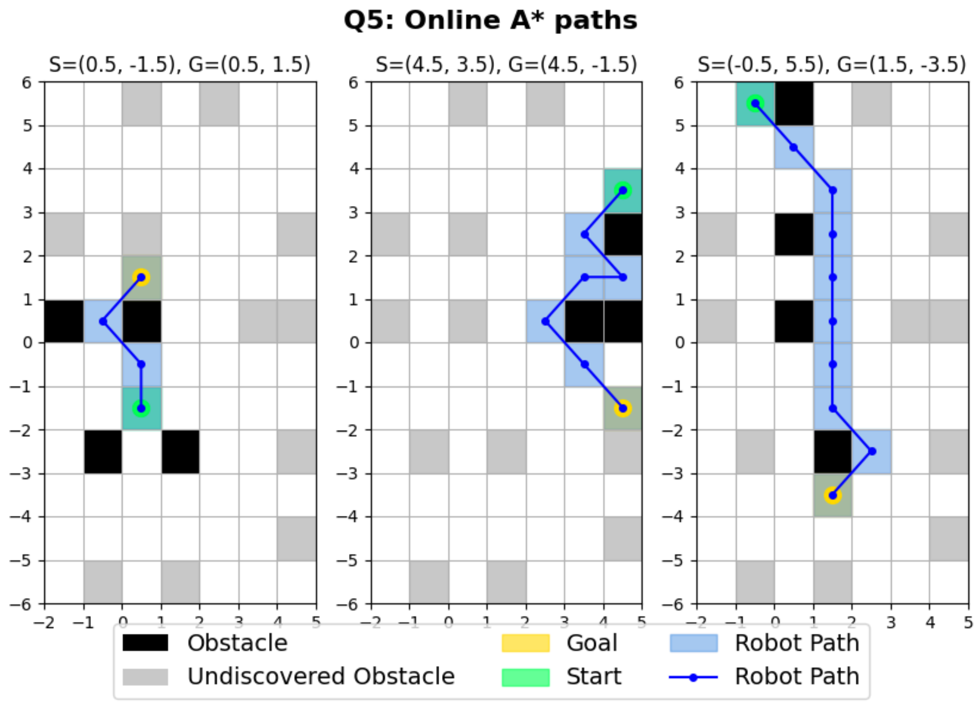
*Eq 4.1 - a pseudocode algorithm for Online A**

In short, in each loop:

1. the map is updated to include new obstacles observed from the current node *n*
2. A* search is run on the new map, returning an optimal path *P*
3. The current node n is updated to be the next node in *P*, and we exit the loop, returning *P,* if we've reached the goal

In this "online" modification, the set of expanded nodes for the top-level online algorithm (which is distinct from the expanded set for the *A** call in the "offline *A**" step in Eq. 4.1 above) only includes the nodes on the path the robot has taken so far. Furthermore, we are constrained to explore a node immediately adjacent to our current node for the next step in the loop, whereas in the offline *A** algorithm, we will select a node from an open set that's adjacent to *anywhere* we've previously explored. This means the order in which we add nodes to the closed set is different in online vs. offline A*.

**5. Plan paths between the start (S) and goal (G) positions from Step 3. Provide the same visualization.**



**6/7. Inflate the amount of space each landmark occupies by .3m in all directions (with the final result of a square, or an approximation to a circle – either is acceptable). P**

**Plan paths between the following sets of start (S) and goal (G) positions. Provide a visual display of your results, that shows the occupied cells and the planned paths. (It is okay to not display demarcations for the unoccupied grid cells.)**

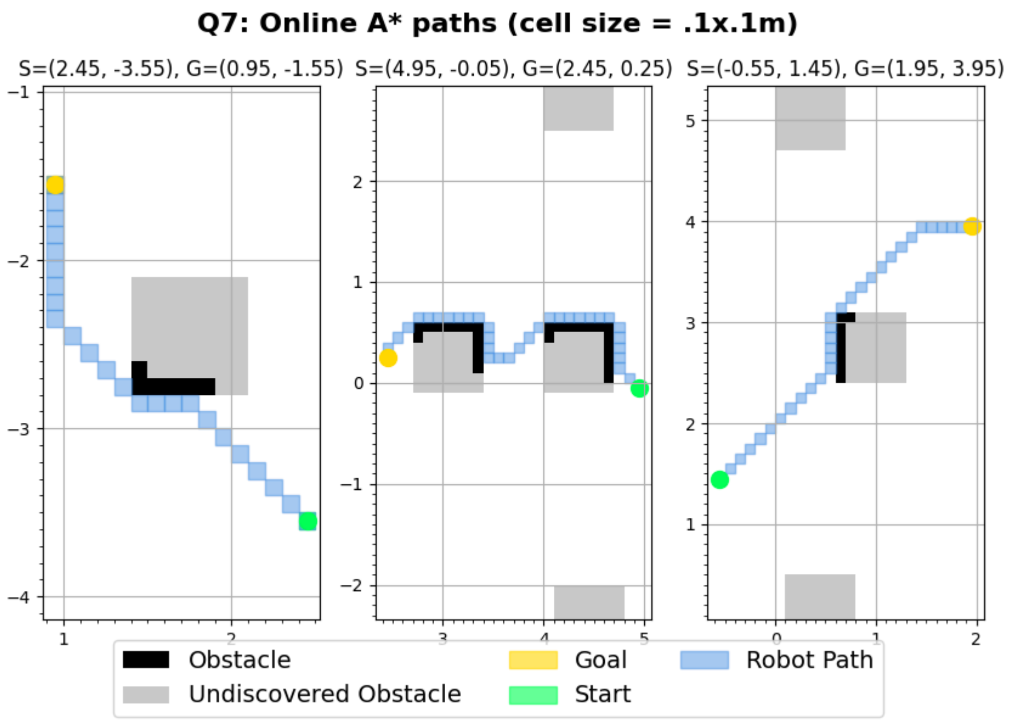
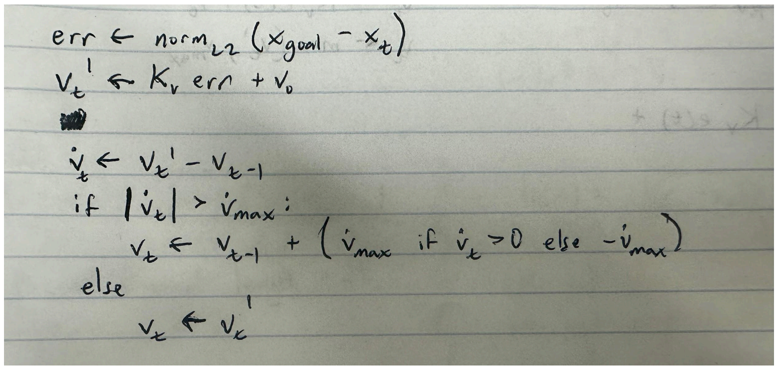


*Fig 7.1 - Paths between the listed start & goal position in online A*, given a grid size of 0.1m*

## Part B

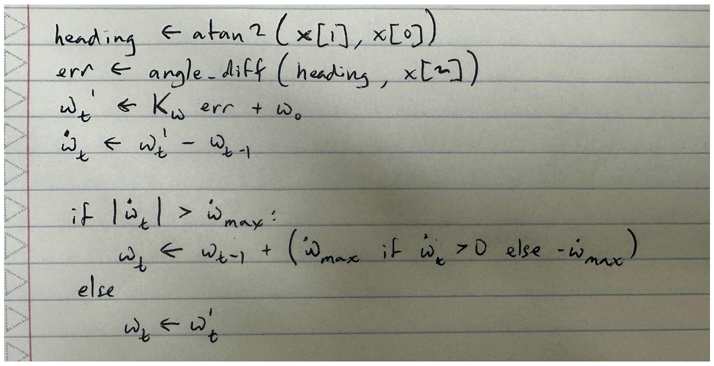
**8. [*] Design an controller able to drive a path generated by your online-A* implementation, based on the motion model you developed in Step 1 of Homework #0. That is, the function should output translational and rotational speeds [v ,ω ] that will allow the robot to achieve a target 2-D position [x T , y T ] , given the robot's current 2-D position and heading [x , y ,θ] . (The target positions will come from the paths generated by your online-A* implementation.) Make this controller more realistic by considering also the robot's current speeds, and restricting maximum acceleration to the following values $\dot{v}$=0.288 2 , $\dot{\omega}$=5.579. Report the maths of your formulation, with explanations as needed.**

A controller composed of 2 independent P-controllers, one for linear velocity and another for angular velocity, was implemented for this purpose. Each controller saturates at the given maximum acceleration values. The filter parameters were hand-tuned to produce sensible trajectories, as evaluated visually on the leftmost plot in Fig 8.2 (without noise).

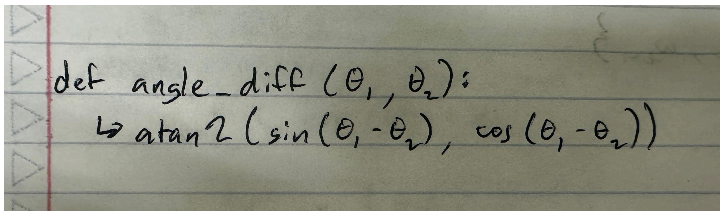


```
$err \leftarrow norm_{L2}(x_{goal} - x_t)$
$V_t' \leftarrow K_v \, err + V_0$

$\dot{v}_t \leftarrow V_t' - V_{t-1}$
if $|\dot{v}_t| > \dot{v}_{max}$:
    $V_t \leftarrow V_{t-1} + (\dot{v}_{max}$ if $\dot{v}_t > 0$ else $-\dot{v}_{max})$
else
    $V_t \leftarrow V_t'$
```

*Eq 8.1 – v controller equations*

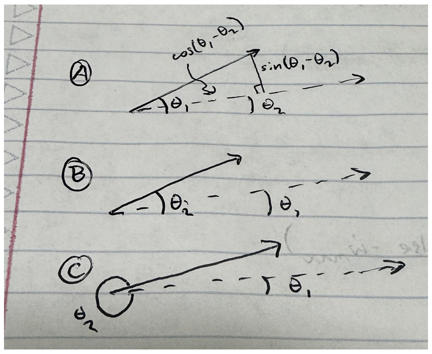


```
heading $\leftarrow$ atan2(x[1], x[0])
err $\leftarrow$ angle_diff(heading, x[2])
$\omega_t' \leftarrow K_\omega \, err + \omega_0$
$\dot{\omega}_t \leftarrow \omega_t' - \omega_{t-1}$

if $|\dot{\omega}_t| > \dot{\omega}_{max}$:
    $\omega_t \leftarrow \omega_{t-1} + (\dot{\omega}_{max}$ if $\dot{\omega}_t > 0$ else $-\dot{\omega}_{max})$
else
    $\omega_t \leftarrow \omega_t'$
```

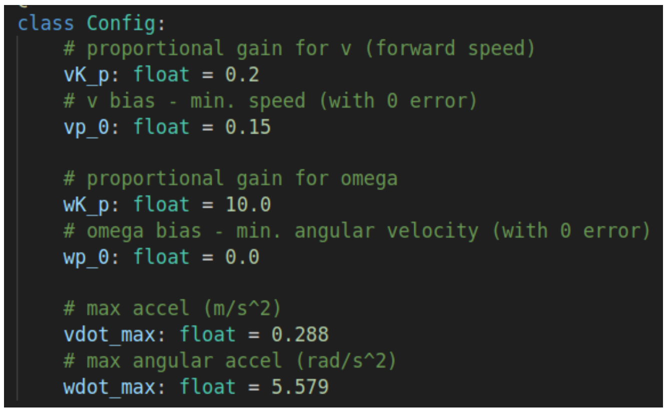


```
def angle_diff($\theta_1$, $\theta_2$):
    ↳ atan2(sin($\theta_1 - \theta_2$), cos($\theta_1 - \theta_2$))
```

*Eq 8.2 – w controller equations*



*Eq 8.3 The* angle_diff() *function is necessary (rather than simply taking $\theta_1 - \theta_2$) in order to elegantly handle case C in the diagram above, in which the shortest distance between the two given angles is not $\theta_1 - \theta_2$.*



```python
class Config:
    # proportional gain for v (forward speed)
    vK_p: float = 0.2
    # v bias - min. speed (with 0 error)
    vp_0: float = 0.15

    # proportional gain for omega
    wK_p: float = 10.0
    # omega bias - min. angular velocity (with 0 error)
    wp_0: float = 0.0

    # max accel (m/s^2)
    vdot_max: float = 0.288
    # max angular accel (rad/s^2)
    wdot_max: float = 5.579
```

*Eq 8.4: tuning parameters for the dual P-controller implementation*

The tuning parameters selected reflect sensible assumptions about the sort of trajectories we want; $w_0$=0 because we want to continue driving straight if we're pointed directly at the waypoint, and $K_w$ is relatively large because we want to turn as quickly as we can to face new waypoints without inducing large oscillations in heading angle. On the other hand, $v_o$ is fairly large in the context of the map because although we want to slow down as we approach the waypoint (to ensure we can continue to turn to face it), we never want to slow entirely to a stop.
As a high-level objective, we can evaluate the relative success of a trajectory by how long (i.e. how many timesteps) it takes for the robot to navigate from the start to the goal. By this token, an ideal trajectory moves the robot with as high a forward velocity as possible without missing waypoints. Therefore we must be careful not to set $v_0$ and $K_v$ to values which are too small. However, if $K_v$ is too large, we will slingshot rapidly past the objective before the angle controller has a chance to adjust our heading towards the waypoint.

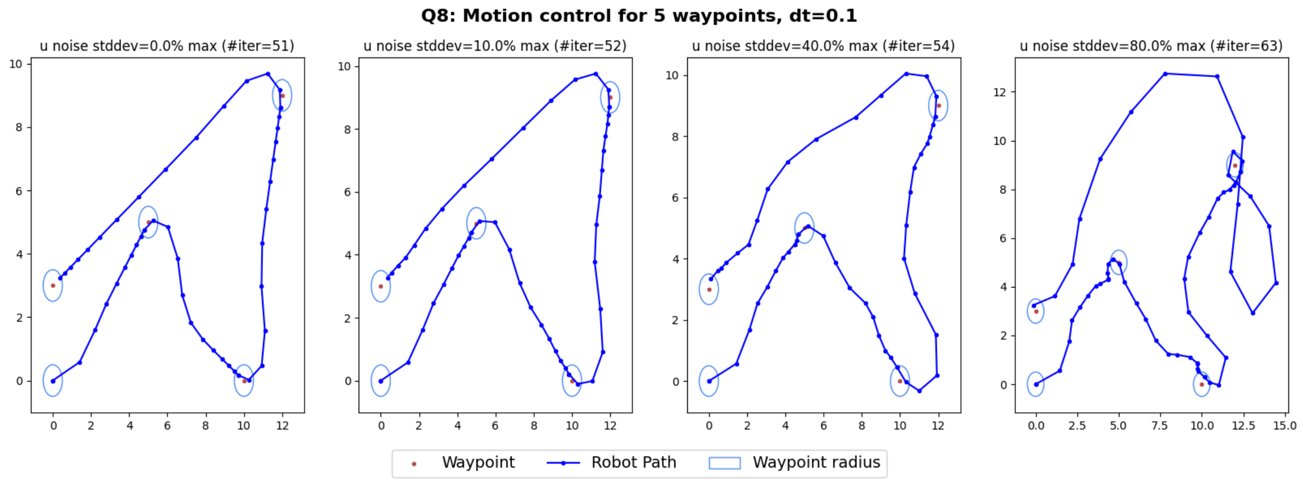


*Fig 8.5 - Motion control using the algorithm above for 5 test waypoints, subject to varying amounts of noise (see Q9 below). Noise is specified here as a percentage of $v_{max}$ and $w_{max}$ in the controller.*

**9. [*] Drive the paths generated in Step 7. When you execute the speeds output by your controller of Step 8, make things even more realistic by adding on some noise. You can assume that your robot always knows where it is. What you cannot assume, however, is that it gets exactly where you asked it to go. Provide a visual display of your results, that shows the occupied cells and the planned paths. (It is okay to not display demarcations for the unoccupied grid cells.) Display also the position and heading of your robot at each step of the execution. Discuss any challenges, inconsistent or surprising behavior.**

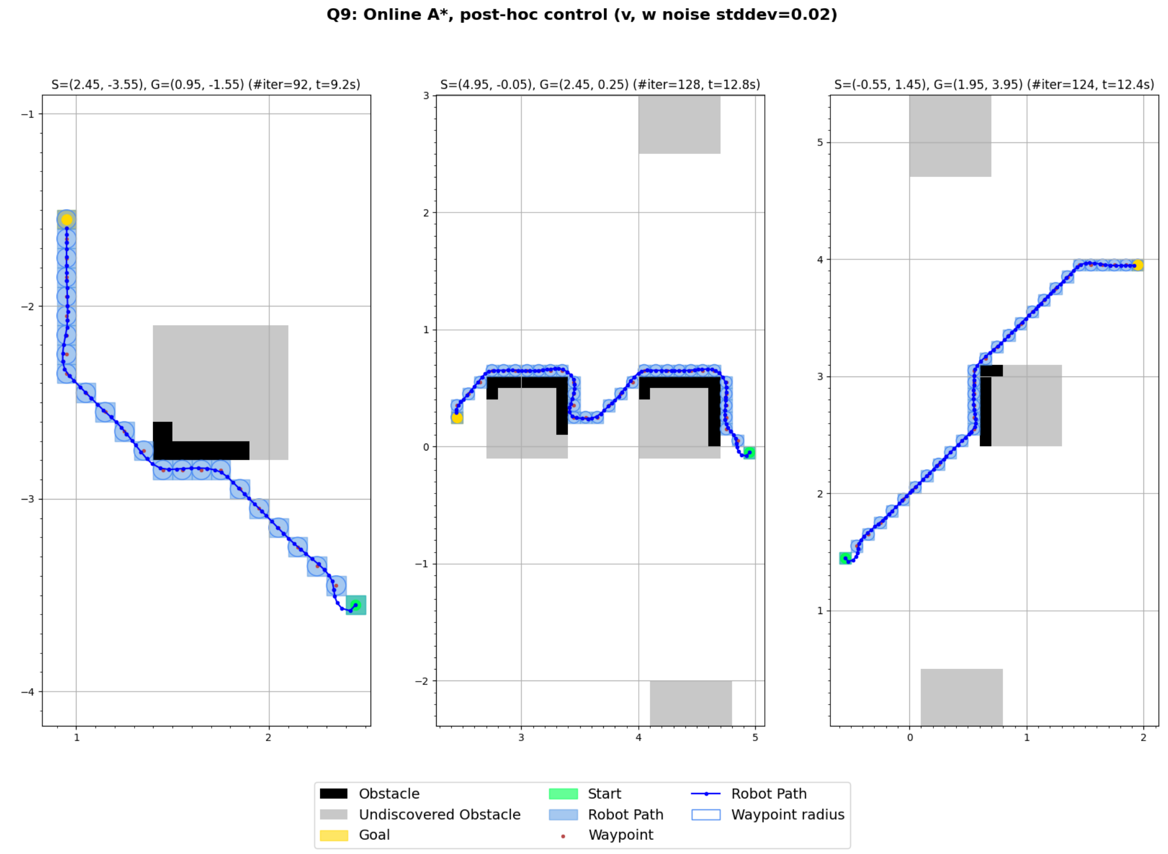
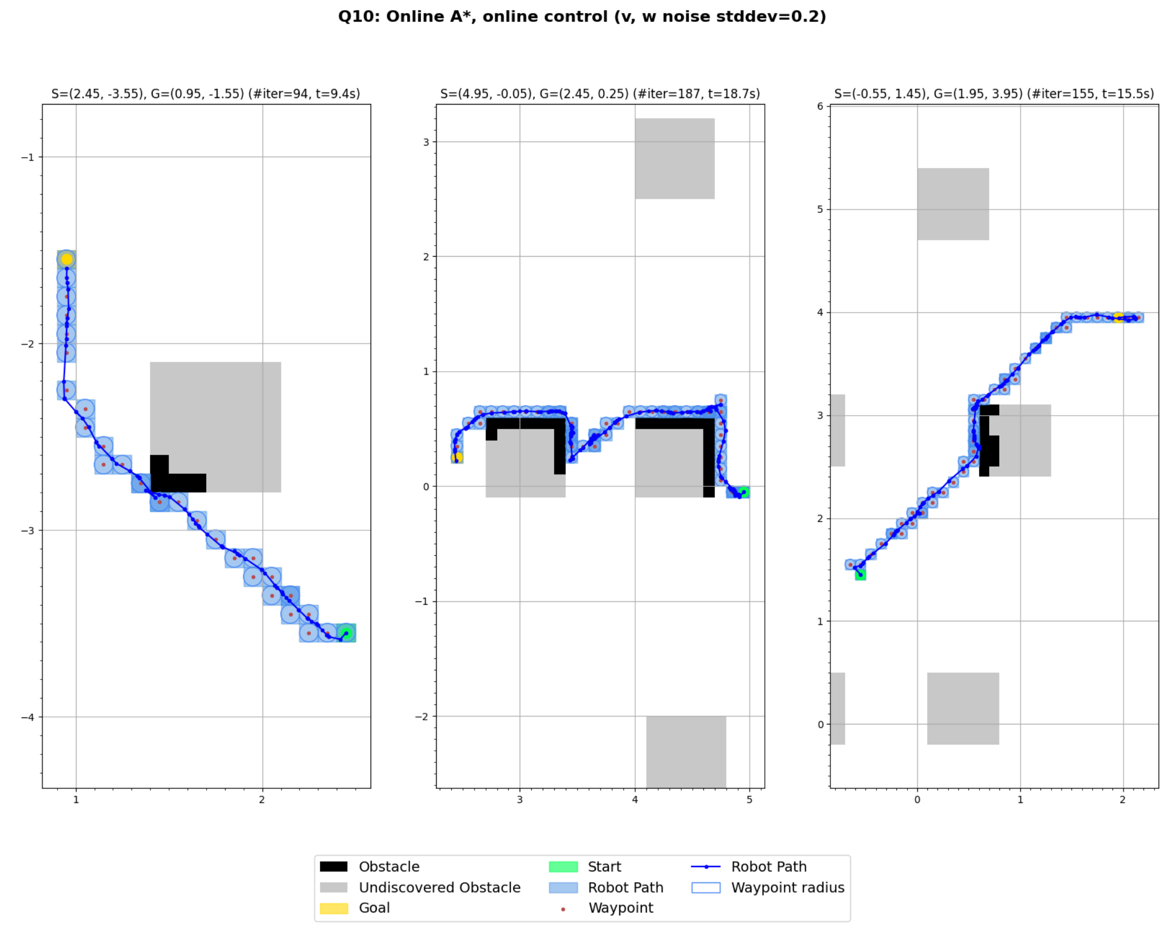


*Fig 9.1 - Online A* path evaluation, followed by a trajectory simulation using the controller and waypoints given by the A* path.*

An important feature of making this controller work in practice is a *waypoint threshold* $r_{waypoint}$ such that we reach the waypoint (AKA the center of the next grid square in the path) when we enter a circle centered on the waypoint with radius $r_{waypoint}$. In the successful implementation, $r_{waypoint}$ was set to a radius roughly equal to ½ of the cell size, such that by the time the robot crosses the border of a cell, it's already headed towards the next waypoint. This keeps the robot moving smoothly along the path, without requiring sharp-angled turns, slowing down near waypoints, or looping back to catch missed waypoints.
This latter issue (looping around to catch missed waypoints) is also a significant issue for large control noise values and poorly tuned control parameters. However, given a reasonable noise value and control values as specified above, such wasteful loopbacks present fairly seldom, especially since this A* implementation, given convex obstacles, rarely takes sharp corners.

Another issue in implementation is that, without integrating knowledge of the obstacles into the controller, it will occasionally navigate inside of obstacles. This issue gets worse as the trajectory planning gets more chaotic, i.e. for a poorly tuned controller and for high process noise values. There are a variety of strategies to adjust the controlled trajectory to avoid the obstacles entirely, but the simplest (assuming this issue is relatively minor or uncommon) is just to increase the amount of padding around the obstacles, thereby avoiding the need to adjust the controller. That is the approach assumed here. However, other strategies may also be adopted, ranging from simply adjusting controller output to take the nearest acceptable path after the core control logic has executed, to adopting a more complex control algorithm that more naturally incorporates obstacle avoidance, such as potential fields, flow fields, or model-predictive control.

**10. Putting it all together: Plan the paths while driving them. That is, no longer use your paths from Step 7 (which assumed perfect world dynamics). Instead, at each step of your online planning, execute the next chosen step using your controller. Provide a visual display of your results, that shows the occupied cells, planned paths, robot position, and robot heading. Discuss any challenges, inconsistent or surprising behavior.**



The primary benefit of this online, simultaneous A* and control evaluation seems to be in avoiding unnecessarily looping; this makes sense, as this implementation handles deviations from the trajectory due to noise or control problems naturally by adjusting the optimal path to match the new grid location, rather than requiring the low-level controller to double back to the original path.
As discussed above, there are issues with clipping into obstacles; however, these paths are short enough that we don't see it in this example.

**11. Drive while planning for the start/goal positions of Step 3, using your finer grid. Repeat this procedure, now using the coarse grid (of Step 3). Provide a visual display of your results, that shows both planned paths, as well as the position and heading of your robot at each step of the execution (for each path). Show also each grid cell and distinguish occupied cells (for this, choose a single grid resolution; either is fine). Discuss any behavior differences you see between operating on a coarser versus finer grid decomposition. What are advantages and disadvantages of each?**

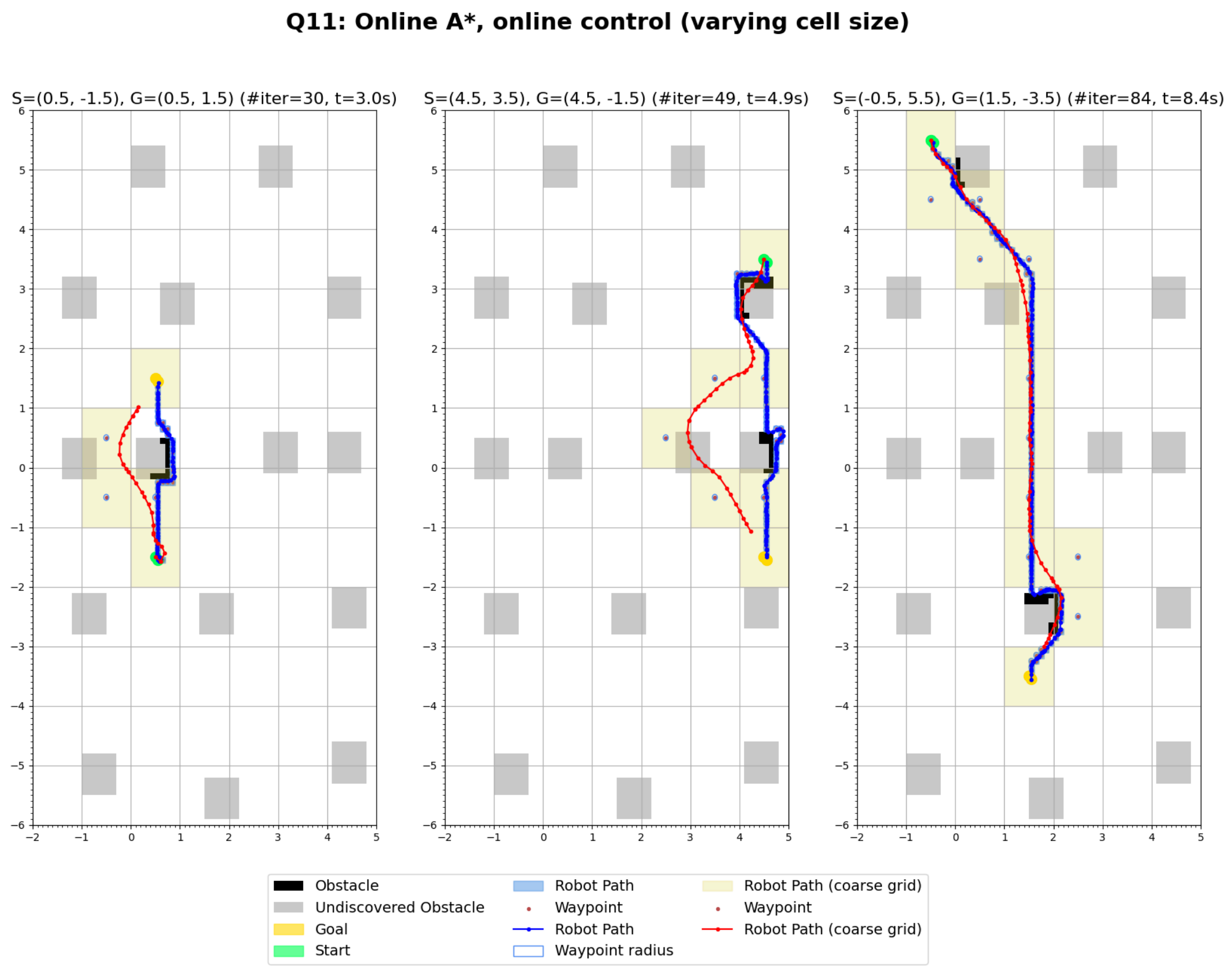


Fig 11.1 - course & fine grid trajectories for the same start & goal locations, superimposed on top of each other.

One note is that a direct side-by-side comparison isn't *quite* possible, because the coarse and fine grids construct slightly different obstacle shapes & locations due to the way they approximate the obstacles, which can be seen in Fig 11.1. However, as can be seen in the plot, the coarser grid results in a smoother trajectory with a faster average velocity, but the coarser grid results in a trajectory that more tightly hugs the obstacles. All in all, the relative efficiency of the trajectories followed are similar; however, this is mostly due to the controller being more performant on the coarser grid. The fine-grid path is more efficient in terms of straight-line distance between grid cell centers; therefore, if a controller that were able to handle higher speeds on the fine grid were implemented, the fine grid would result in more efficient paths in terms of trajectory time. As-is, the finer grid is slightly more performant in distance, while the coarser grid is slightly more performant in time.

**12. What additional factors would confound driving these paths if our robot was operating within the real world? (What sort of assumptions and simplifications do our simulated world and controller make?)**

We make several assumptions that may fail in real-world scenarios.
First, this implementation would perform more poorly for sharp, convex obstacles, in which A* might drive its way into a corner before observing a wall, causing the controller to have to make a sharp turn (which, as we've seen, it's not as good at. Note the coarse grid path in the right axes in Fig 11.1, at the grid cell centered at (2.5, 4.5)--the path enters an obstacle because it can't turn fast enough to escape it).
Second, we have assumed that we know where we are at all times–our control scheme may be noisy, but our localization (and mapping) is not. This is not true in the real world; we may even want to compute paths from multiple simultaneous possible start locations, if our probabilistic representations of our location on an unknown map are multimodal or very spread out.
We have also failed to take advantage of additional capabilities of a diff-drive robot; for instance, it can turn in place, and can also reverse–this controller supplies a lower limit to the forward velocity, so the robot never turns without moving forward. This might enable better escapes from obstacles as mentioned above.
We also have created a binary world, in which each grid square is either an obstacle (and thus impassible) or completely free. Real world environments may contain grey areas between the two; we might wish to supply a higher cost to driving on difficult terrain like sand, hills, mud, etc, and we also may not know how difficult such terrain is until we drive on it, requiring an online adjustment to our cost estimate. This is fairly easy to accommodate in our architecture by enabling variable costs in our map beyond just 1 or 1000; however, we did not attempt it in our simplified model.