Assignment 1: Vishwajeet Karmarkar

2) A* is a part of the family of informed search algorithms. A* evaluates nodes based on two costs, the cost to reach the node g(n) and the cost to reach the goal from the node h(n) [1].

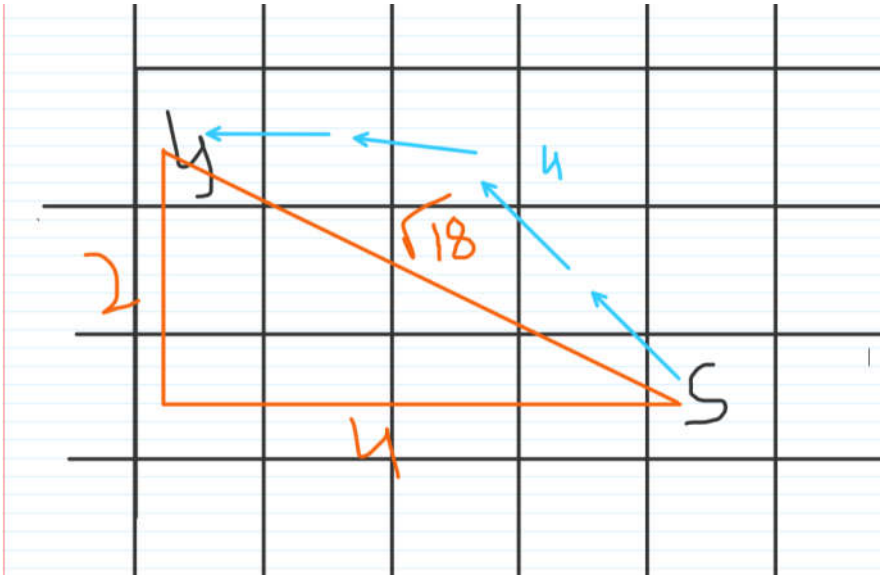
$$f(n) = g(n) + h(n) : f(n) = estimated\ cost\ of\ cheapest\ solution\ to\ n$$

Definition of admissibility: An admissible heuristic is such that it never overestimates the cost to the goal. Thus it is always an optimistic guess of the actual cost. Since heuristic cost from each node is optimistic, it provides a uniform way to estimate closeness to the goal.

Condition of admissibility: $h(n) \leq h^*(n)$

$$h(n) = heuristic\ function\ estimate\ to\ reach\ goal\ from\ node\ n$$

$$h^*(n): optimum\ cost\ to\ reach\ the\ goal\ from\ node\ n$$



In a non-constrained non-grid environment straight line distance is the shortest path to the goal, and hence can be used as the admissible heuristic (since the true cost will never be lesser than that value). However as we are restrained to move to only the neighbouring 8 grids, straight line distance is not the shortest path. This can be seen in the case show in the image above where the shortest distance is 4, while straight line distance is $\sqrt{18}$. However if we subtract 1, which is the minimum cost to move from the straight line distance, we get an admissible heuristic. In the worst case, where the node is the goal, the cost of going to the goal is 0, while our heuristic estimates a -1 cost, hence being optimistic. Thus the heuristic is the (Euclidean distance from node to goal -1).

$$h_{node} = \left(\sqrt{(x_{node} - x_{goal})^2 + (y_{node} - y_{goal})^2}\right) - 1 \leq h_{optimum}$$

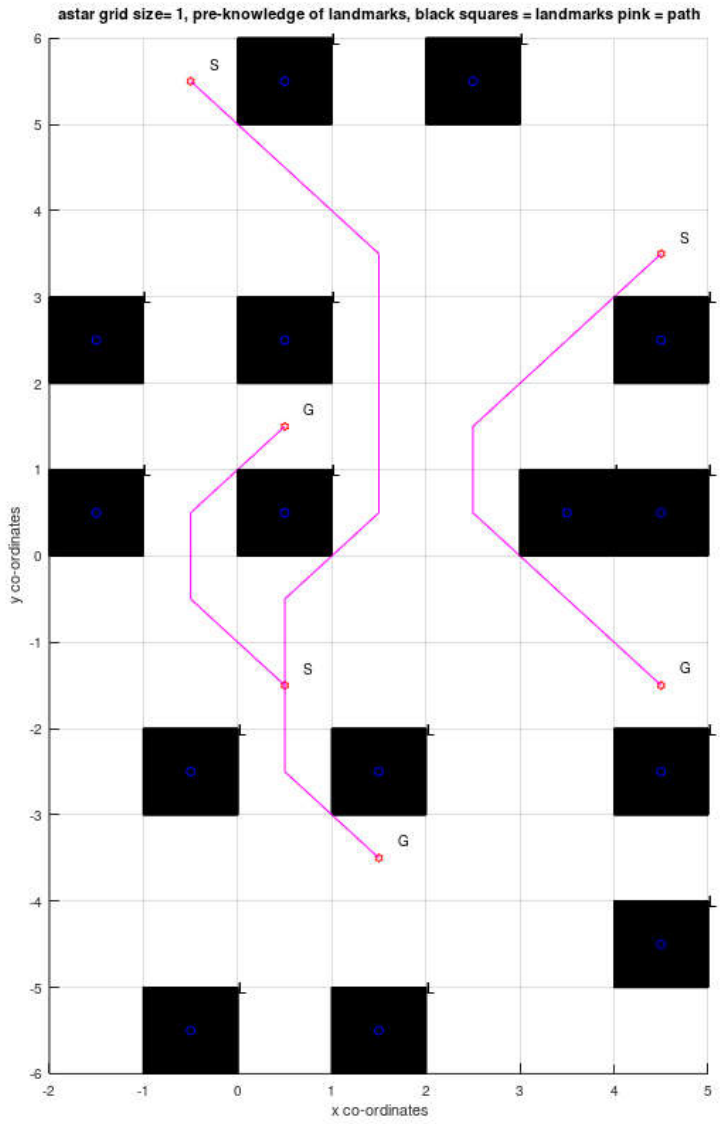
3) The above image shows the planned paths for the three starting conditions. White cells are unoccupied and free to move in, while the black ones are occupied. This version of the A* takes place in a virtual manner, and also knows all the landmarks in advance. Hence it is free to explore various paths, and then once an optimum path to the goal is found, can backtrack from the goal to the start position, giving the paths shown in pink in the image. The Algorithm used is as follows:

Open_list: list of all open nodes with their costs and their parent node

current_list= current node with its costs and its parent node

child_list= expanded nodes in neighbourhood of current node and their parent node

closed_list = visited nodes and their parent node



open_list_tracker= a grid with size of the total grid map which has all elements as 0 initially. Used to store open_list additions. When node is added to open_list, its position on open_list_tracker is marked 1.

move_grid: a grid with size of total grid map which has all elements as 0 initially. Used to store close_list_additions. When node is added to closed_list, its position on move_grid is marked 1.

1) Empty cells : open_list, current_list, child_list, closed_list
2) Add start position to open list

Until Current_list =! Goal or open list = empty, loop{

3) Sort open_list in ascending order of total cost, f
4) Assign first row of open_list (with lowest f) to current, and remove from open_list: add current to closed_list
5) Mark all positions of elements on closed_list as 1 on the move_grid. Mark all positions of elements on the open_list as 1 on the open_list_tracker
6) Expand the current list to get neighbour nodes. In a general case each cell has 8 neighbours. However if neighbour is on move_grid, do not expand that. Add all expanded neighbours to child_list
7) Find the g,h and f costs for all the elements on the child_list
8) If elements on child_list are already on open_list as tracked by open_list_tracker, check whether the g cost of that child is lower than existing g cost on the open_list. If this is the case then update that row on open_list with the child_list costs and source_element.This implies that the child_node is now coming from a different parent node.
9) If child_list is not on the open_list, add that child to the open_list }

Backtracking algorithm: Once the goal is found, use backtracking to trace back the optimum path

1) Choose goal as node
   Until node is not start: loop {
2) Find node on closed_list
3) Assign parent from closed_list to node}

grid location y	grid location x	g cost	h cost	f cost	parent node y	parent node x
8	3	0	0	0	0	0
7	3	1	2	3	8	3
7	2	1	2.23607	3.23607	8	3
7	4	1	2.23607	3.23607	8	3
6	2	2	1.41421	3.41421	7	3
5	3	3	0	3	6	2

parent node
node

This table shows how backtracking works for one case. The node (orange) starts from the goal, a parent is identified (purple), which then becomes the node. This continues until start position is acquired.

4) The algorithm described above is good for a virtual environment. This is true since the algorithm can explore many nodes and then backtrack from the goal to the start to choose the optimum path. However for a real-robot this isn't practical as backtracking is an added cost which is not accounted for in the virtual system. Moreover the robot cannot detect obstacles (with g cost 1000) until it reaches neighbour node.

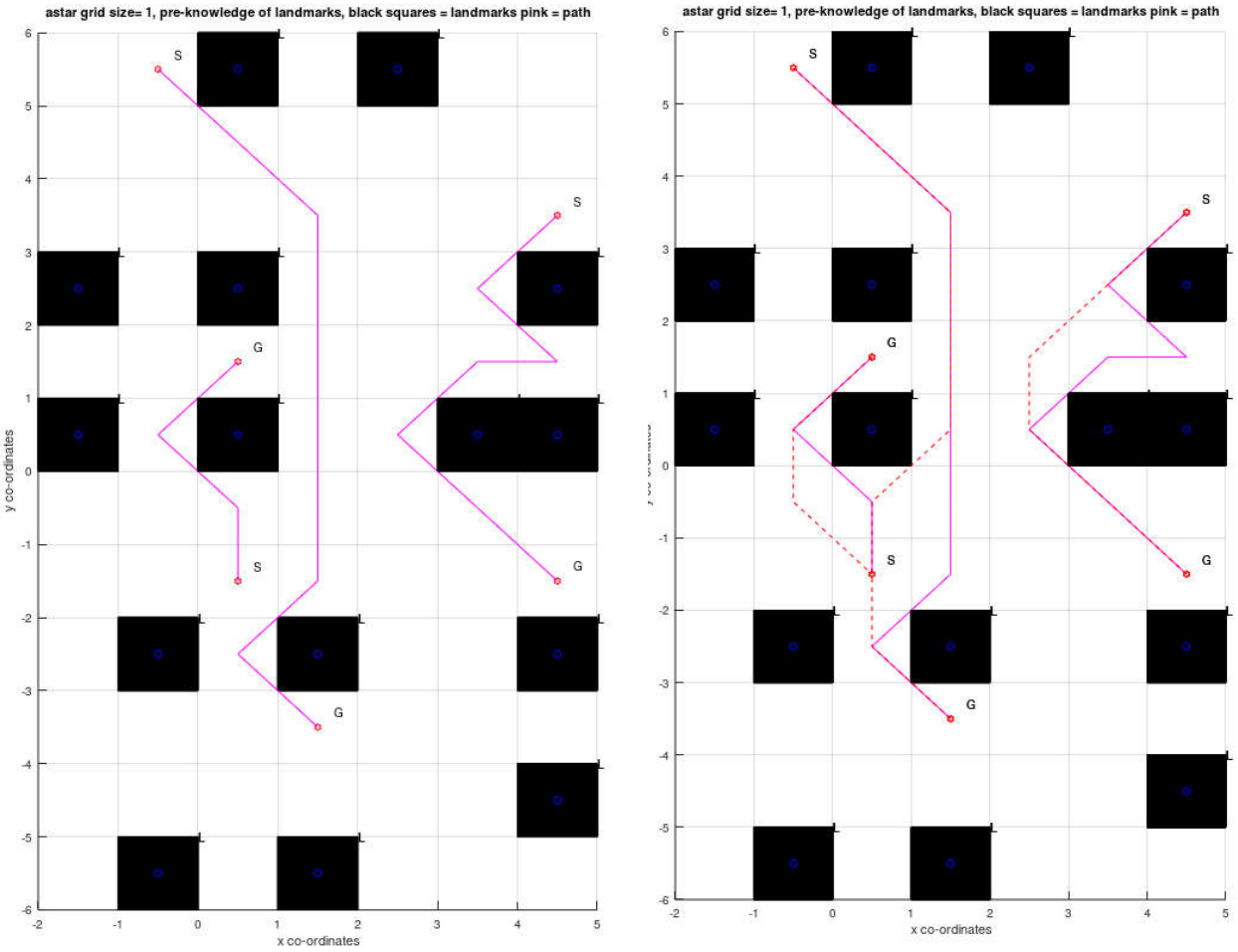
1) The list definitions are still the same as above.
2) The flow of the algorithm also remains almost the same. Two modifications are:
   1) The open_list is emptied after step2 is done, and first row of open_list is added to current_list and closed list. This ensures that information from nodes which are multiple cells away from the robots current position aren't still in the knowledge base. A virtual system can hop on to those nodes and start evaluating the neighbours from that node, however a robot cannot hop on to nodes which aren't its neighbours
   2) The backtracking part of the algorithm is removed.

Thus as is clear from the above point, information of all expanded nodes is retained in the virtual system, however that information is removed in the robot having the mentioned conditions.

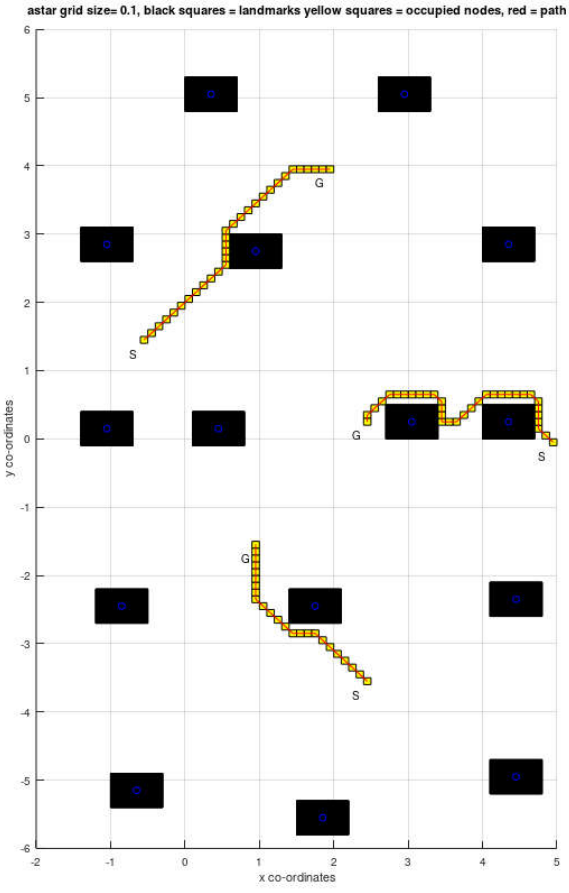
5)

The figure on the bottom left shows the planned paths when the system is considered to be 'online'; It can be seen that the paths are different in all the three instances. Take an example of the path on the right side of the figure, with the start node S in grid at X=(4,5) Y=(4,3). This path shows the robot going into a corner. This makes sense as the robot would have no way of knowing there is a blockage between its proposed path and the goal. The robot then turns west, and clears the two landmarks on its south, to reach the goal.

The virtual system on the other hand, has already evaluated this condition in one of its expanded nodes, and thus can just backtrack from the goal to start position to choose the optimum path that it took instead. An overlapped comparison of these cases is shown in the plot at the bottom right. The dotted red lines are the paths taken by the virtual system while the pink path is that taken by the robot. The virtual system can traverse the grid in a much optimized fashion or with a same overall cost as compared to the online robot A*.

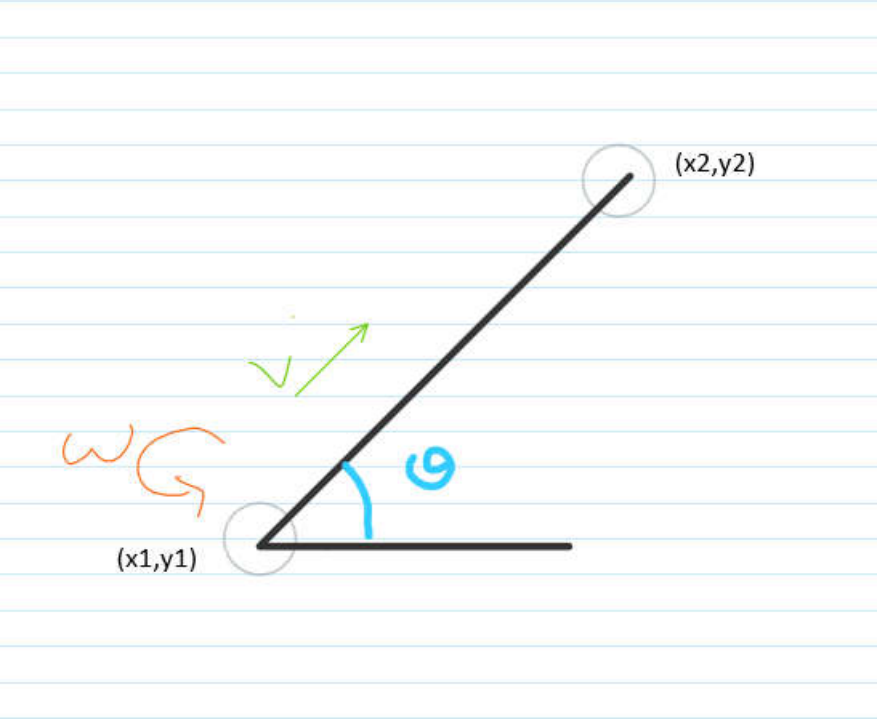


6,7) The next step decreases the grid size to 0.1m. To inflate the space occupied by landmarks a square shape with +- 0.3m dimension change is used.



For the image above, it can be seen from the path at the right, how the robot turns to head directly to the goal after crossing the first landmark in its path. It however encounters another landmark, and has to move around this obstruction to reach the goal.

8) Inverse Kinematics: The inverse kinematics is built on a simple model as shown in the image shown below. V is the straight line velocity of the robot along its x axis, ω is the angular velocity (counter-clockwise = positive) while θ is the heading angle of the robot. If a basic robot following this model starts from a position (x1,y1) and wants to go to (x2,y2), then it can simply rotate itself till it matches the required heading angle, and then move with only a linear velocity till it reaches the end destination. A slightly advanced can robot can do these actions simultaneously, and move in an arc to reach the goal position. Thus the two important parameters defining the robots ability to reach the goal is the heading angle and the distance between the two points.



This leads to the analytical inverse kinematic model.

$$\theta_{required} = \tan^{-1}(y_2 - y_1)/(x_2 - x_1)$$

$$\omega_{required} = \frac{(\theta_{required} - \theta_{heading})}{\Delta t}$$

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad V_{required} = \frac{Distance}{\Delta t}$$

Building on this model, an iterative process is used. A Proportional controller is employed to drive the robot to its intermediate goal state. These controllers come under the category of feedback controllers. It takes in a system error, which is the difference between desired state and measured state, and outputs a control signal proportional to this error.

$$P_{out} = K_p\, e(t)\ where\ K_p = Proportional\ Gain, e(t) = Setpoint - Process\ Variable$$

In this case two process variables are employed, and two control signals are calculated. One is for the linear velocity V, while the other is angular velocity ω.

$$for\ velocity\ controller\ e_v = Distance, V = P_v = K_v \times e_v(t)$$

$$for\ angular\ velocity\ controller\ e_\omega = (\theta_{required} - \theta_{heading}), \omega = P_\omega = K_\omega \times e_\omega(t)$$

At each time step t = 0.1, the inverse controller is fed the start and end positions. It outputs the necessary velocities, which are fed to the motion model as developed for last assignment. The output of the motion model is then fed as input position to the inverse controller, and the process repeats. This recursively guides the robot to the end position, until the distance value reaches an acceptable error value. At this point the system is cut-off.

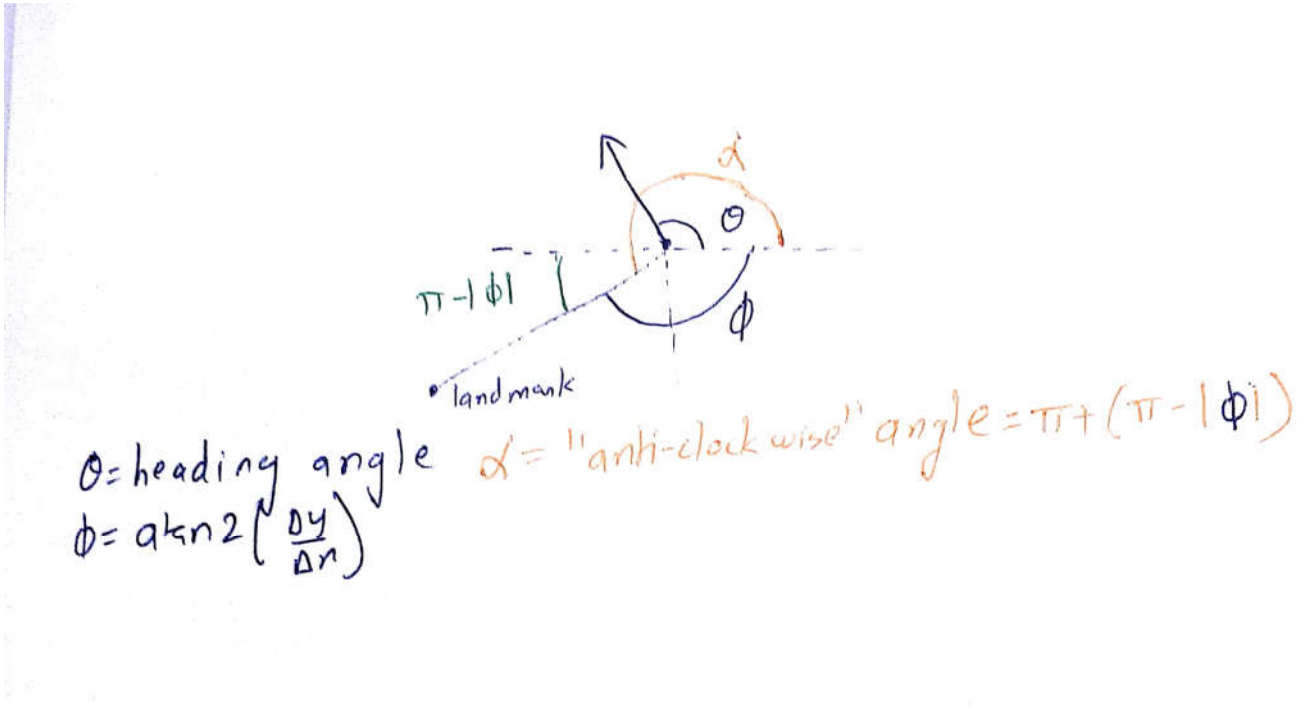
The advantage of using a P-controller is the automatic speed control built in the model. Since a robot at perfect heading will have no heading error, the angular velocity will be 0. More the error the faster the system tries to go back to the error free-state. This however can also lead to overshooting. There is also the possibility of error accumulating over time, giving a non-zero baseline error value (even at goal position there is an error). To tackle the overshooting a 'derivative' component can be added, while to tackle the drift an 'integral' component can be added, giving a PID controller. However the system seems to perform well for chosen parameter, hence the controller for this assignment is restricted to only 'P'.

Limiting accelerations: Based on the dataset, accelerations are limited to specific values $A^v{}_{limit}\ and\ A^w{}_{limit}$ Let the robot move with velocities$V_{old}\ and\ \omega_{old}$. If the P-controller outputs new values $V_{new}\ and\ \omega_{new}$the robot has to accelerate or decelerate to these values. To check acceleration

$$if\ {}^{\Delta V}/_{\Delta t} > A^v_{limit}\ or\ {}^{\Delta V}/_{\Delta t} < -A^v_{limit}\ then, \Delta V = \Delta t \times A^v_{limit}$$

$$if\ {}^{\Delta \omega}/_{\Delta t} > A^\omega_{limit}\ or\ {}^{\Delta \omega}/_{\Delta t} < -A^\omega_{limit}\ then, \Delta \omega = \Delta t \times A^\omega_{limit}$$

Note: Borrowing from the last assignment, heading angles are again measured in an 'anti-clockwise' manner to maintain uniformity.



To find the angle of the line joining the robot co-ordinates with the goal coordinates, we can use the atan2 function. Atan2 gives the value of the robot in a 0: π , and 0: $-\pi$ radians range. Thus if a line is in the 3rd quadrant, the value will lie in $\frac{-\pi}{2}$ to -π, for the 1st quadrant between 0 to $\frac{\pi}{2}$ and so on. Thus to find the bearing w.r.t positive rotation of the robot, we change the angle values as follows. Refer the figure for visual illustration.

If $(\theta > 0)$

$$\theta_{anti-clockwise} = \theta$$

else

$$\theta_{anti-\ clockwise} = \pi + (\pi - abs(\theta))$$

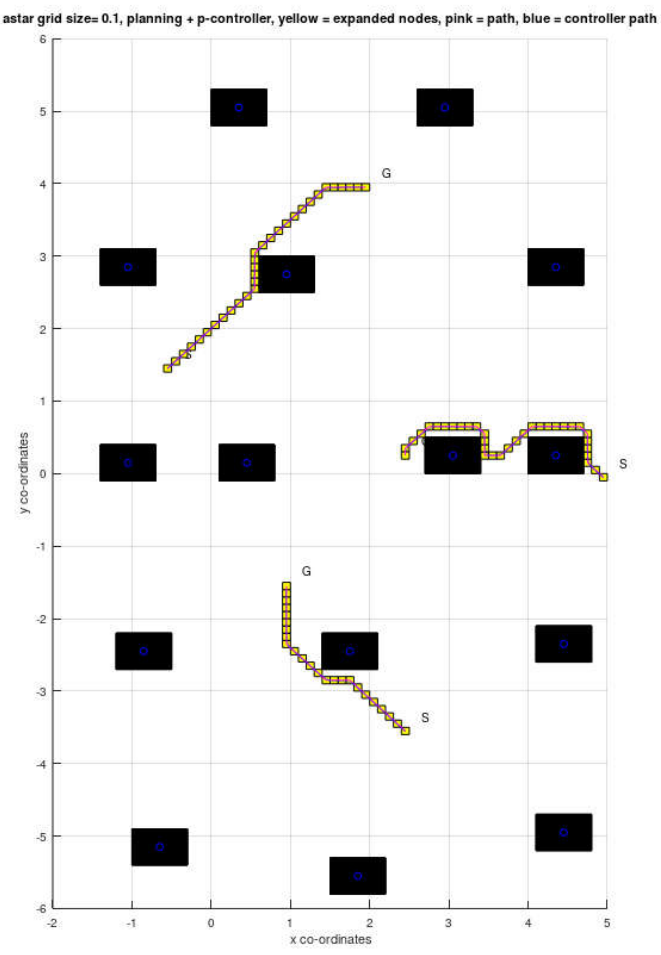
endif

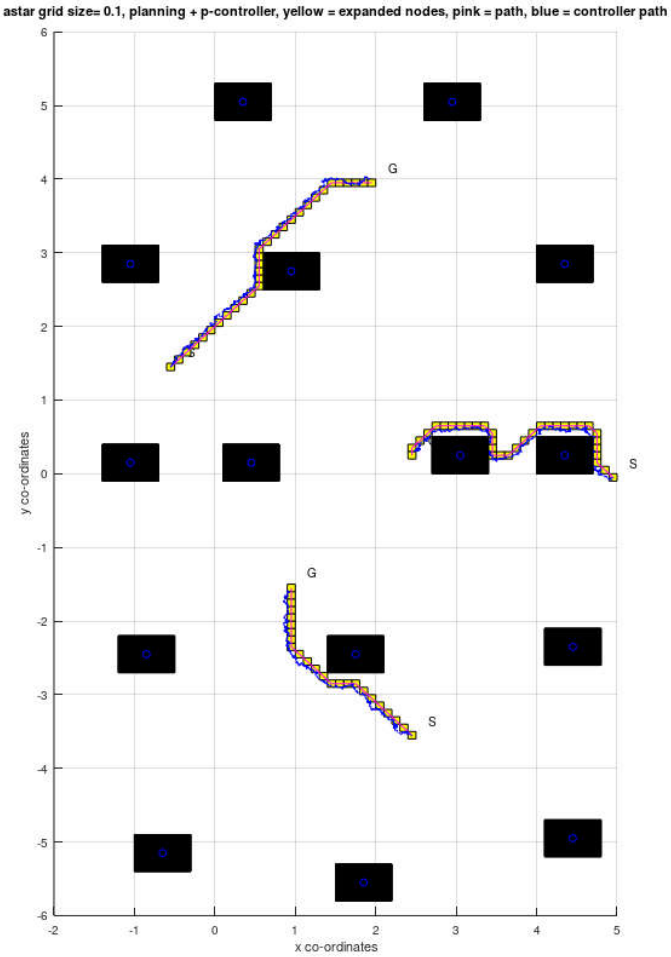
Once we have the 'anti-clockwise' angles we can calculate the bearing angle as follows

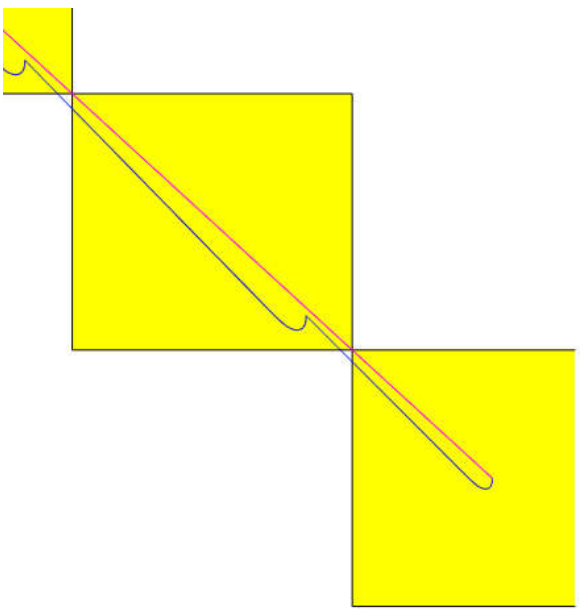
$$bearing = heading\_angle_{anticlockwise} - \theta_{anticlockwise}$$

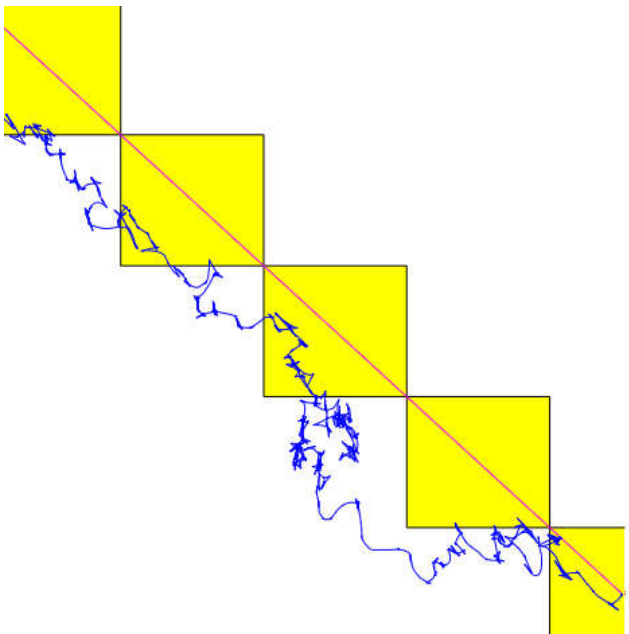
Parameter tuning: The Kv values are kept very small (0.05, so that the system moves in small increments. The Kw values are set a bit higher so that the system doesn't turn too slow.

9) The robot is now driven towards the path found by the A* algorithm. It starts with 0 velocity and heading of $-\pi/2$.

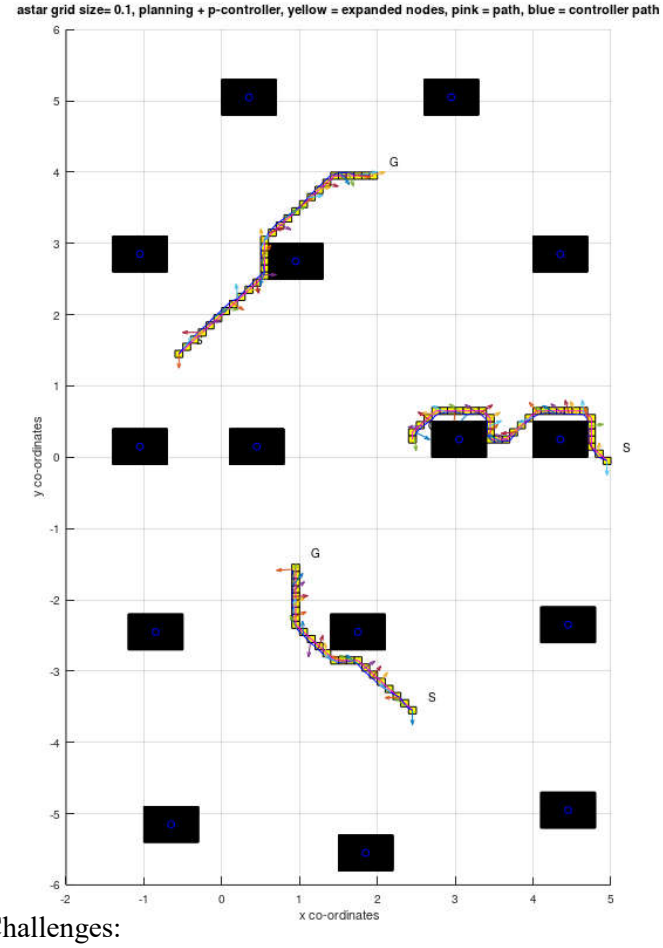


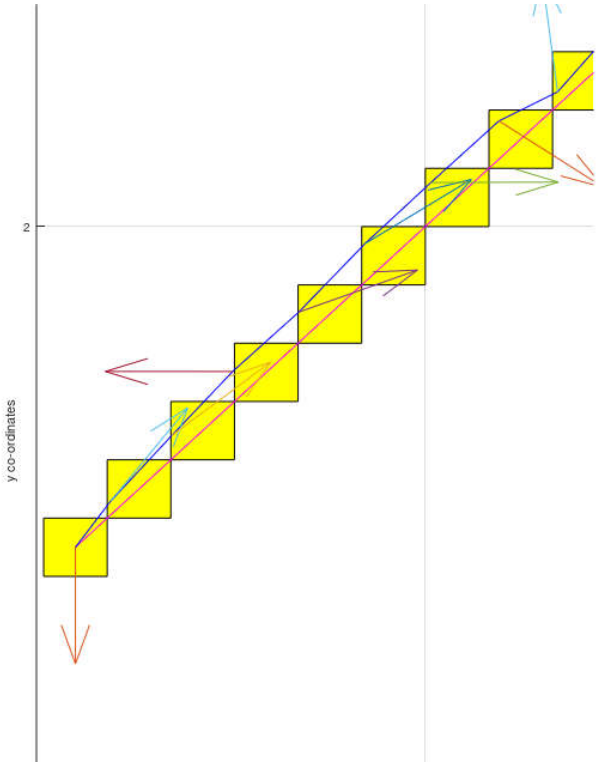






Figures at the start point above show the path given by the A*, the occupied cells according to A*; without and with added noise respectively. The figures below it show highly zoomed in images of one of the paths, without and with added noise respectively. Figure at the bottom left shows the heading angles at each intermediate goal points. If all heading angles are plotted the image becomes extremely unclear, hence only intermediate headings are plotted. Figure at the bottom right shows a zoomed in version of the heading angles.



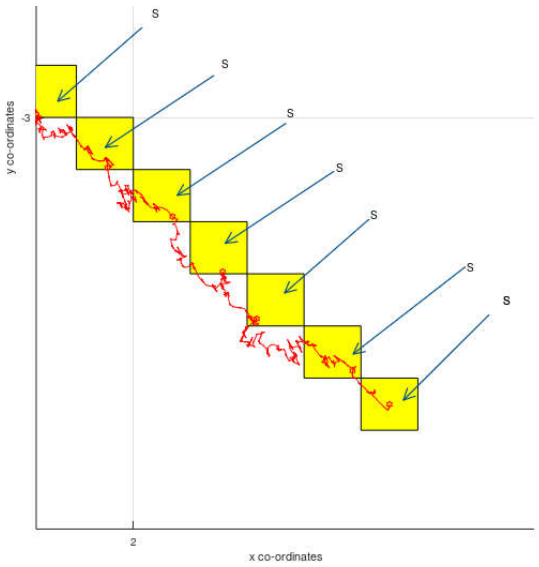


Challenges:

1) Using a p-controller only is bound to give issues with overshooting and drift errors over time in real-world robots. This is bound to reduce performance.
2) Increasing noise leads to a lot of variation in the plots. Thus it seems that in a real-world operation the robot will definitely start deviating with respect to the suggest p-controller. Adding a filter to the system as built in the last assignments would resolve this issue and lead to a good performance.

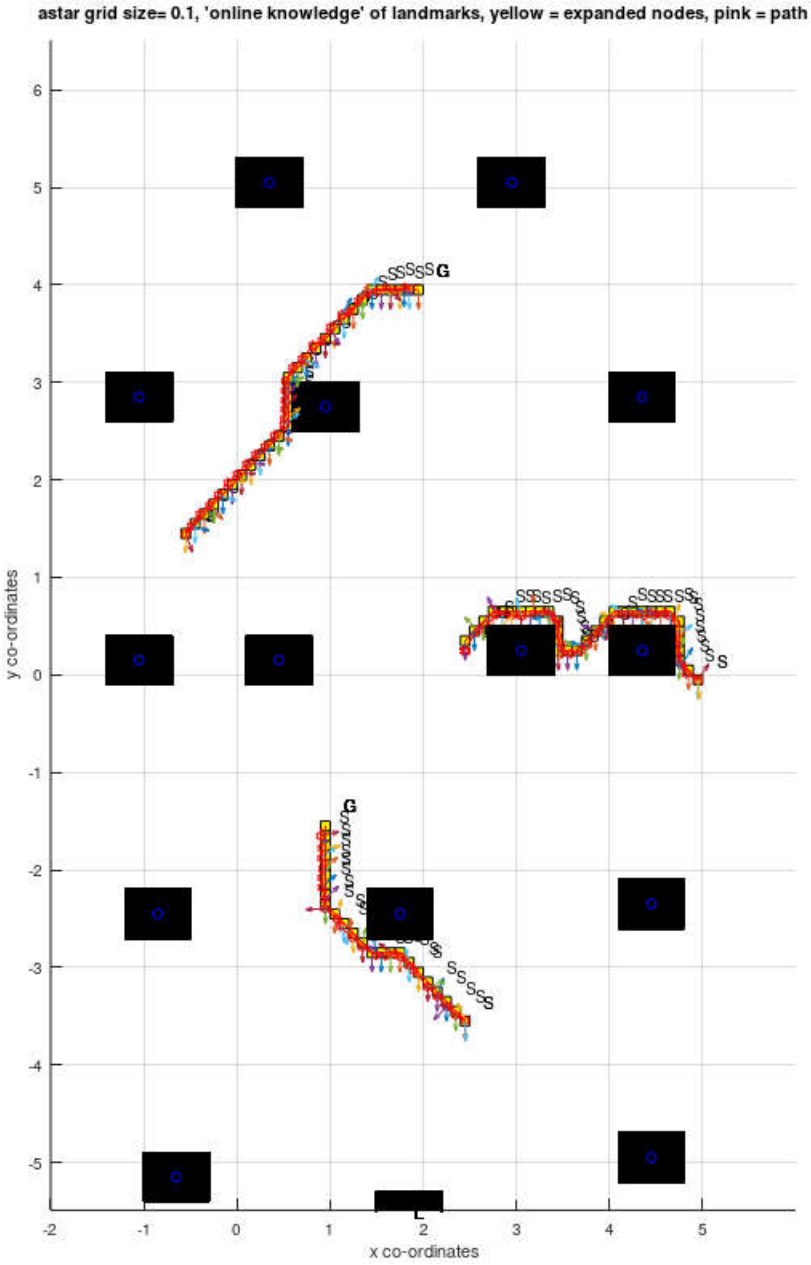
Surprising: The p-controller with the selected Kv and Kw values tracks the planned path quite accurately, and isn't very slow.

10)



Planning paths while driving leads to the system giving the co-ordinates achieved from the motion model which is driven by the p-controller, to the A* planner. This allows for an optimum path even if the robot deviates from the path, and hence is better than making it go to a pre-defined plan as achieved in a pre-planned environment. The image on the left shows a zoomed in view.

The S point are the new start points given to the A* planner at each instance. The image below shows the paths for all the cases along with the heading angles.



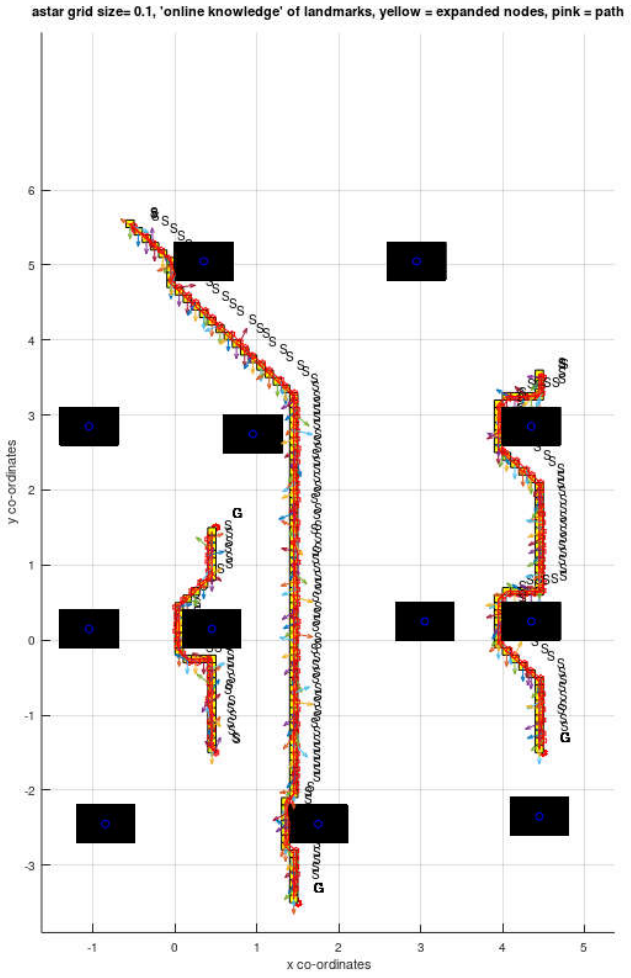
Challenges: Even though this system simulates a more realistic scenario it takes a longer time to compute.

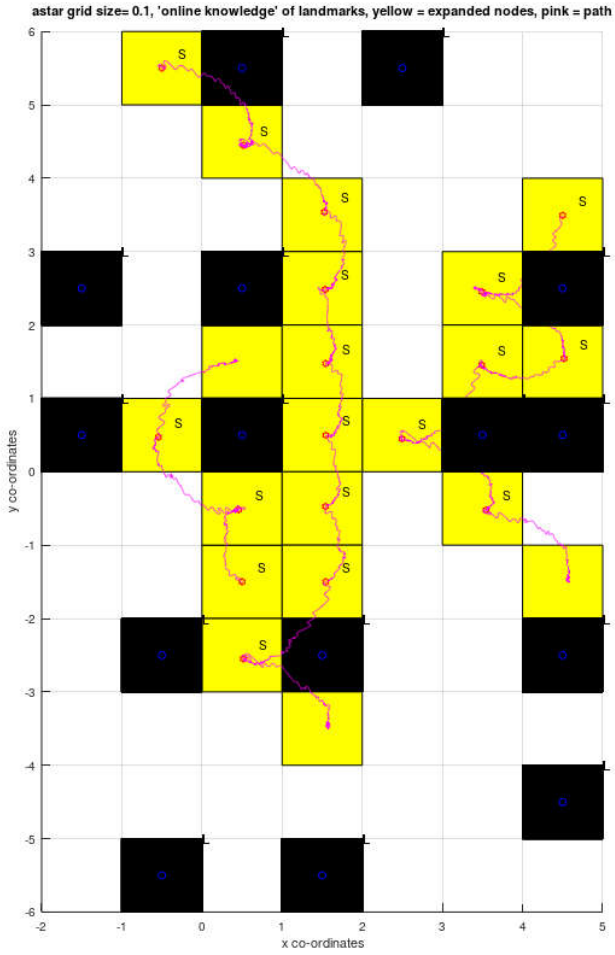
11) The two images shown below analyse the case in step 11, with the older list of points. The image on the left shows the path taken by the system for a finer grid of 0.1m, while the one on the right shows the result of the coarse grid of 1m, both planning the paths while driving. As can be seen the finer grid provides much structured path with less deviation from the expected line of motion. The coarse grid on the other hand, achieves the task of reaching the intermediate goal, however it deviates a lot while reaching this target. Both plots have the same amount of noise variances.

The coarse grid leads to deviation, because there are much fewer data points along the way to correct the course. The finer grid plans at a lot more instances, hence the p-controller only has to traverse paths between these finer data points, which it does very nicely.

There are instances in the coarse grid where it goes on to hit a landmark. This happens because there is no presence of the A* assessing landmarks, until an entire large grid move is done. However, barring these limitations the computational time for the coarse grid is drastically lesser. Thus it makes sense to use a coarser

grid when landmarks are not too densely aligned, when compared to the expected path that the robot is going to take. It can be concluded that a sweet spot for an optimum grid will give an optimum performance.





12)

1) Dynamically moving obstacles are bound to confuse the A* algorithm. If an obstacle lands in a neighbouring grid after the A* has decided to move into that particular grid, the robot will have no time to re-plan a new move. If we assume a grid with a wall spanning all the rows, barring 1 and this wall moves up and down, the robot will keep moving in arcs from one past opening in the map to another, without having the ability to predict the next possible opening.
2) As mentioned earlier, stochastic nature of the real world will lead to errors in planning and movement which will result in unfavourable behaviour. Equipping the robot with sensors and using filters to better track the state, will lead to a better real world performance of the planning system.
3) The controller makes use of a very simple motion model and inverse kinematics model to drive the system. There could be many factors (such as slip, misalignment etc) which will result in the actual model being a lot different than the model used. One way to tackle this is by using machine learning over test runs to get a better prediction of the real world dynamics governing the robotic system.
4) The heuristics used for this A* involves taking square- roots which can load the on-board computers of a real world robot. Thus using heuristics optimal in a real-world setting need to be explored.

Citations:

[1] : Stuart Russel, Peter Norvig: Artificial Intelligence a Modern approach

[2]: Sebastian Thrun, Wolfram Burgard, Dieter Fox Probabilistic Robotics