Software Team

Selections - Task Round

Hello everyone, congratulations on making it this far into the selections of the AGV Software Team.

In this final phase of selections, you need to complete some tasks. You will need to understand the problem statement, read concepts from the internet (some resources are mentioned in this doc) and then implement (code) it.

Complete at least one of these tasks.

The task explanations are given at the end of this document in a separate section.

1. Make your programs Object Oriented if possible, it's not a requirement
2. The resources given in this document are not sufficient. You are expected to find more on your own.
3. You can discuss among yourselves and help each other, but sharing your code is a strict no. Use of plagiarized code will result in complete and immediate disqualification for both candidates, from whom the code/idea has been copied and who has copied.
4. The test data and their format for all tasks will be shared with you separately.
5. For the tasks that require it, you can visualise the path or data using Matplotlib or OpenCV.
6. Don't hesitate to contact any of us in case of any"obstacles".
7. We have allotted the time with sufficient consideration for various factors and hence cannot give any extension to anyone as it would be extremely unfair to the rest of the candidates

A word of advice - You are expected to do at least one task, but you are encouraged to do more - this will increase your chances of getting selected and will also help you explore your interests in different fields we work on and get a general idea about them.

Task 1

By now you all are familiar with the A* algorithm .In this task we will implement the same.
You are given a 100×100 image with a starting and end point along with obstacles and navigable path, colors of which are written below

Start point = rgb(45,204,113)
End point = rgb(231,76,60)
Obstacles = rgb(255,255,255)
Navigable path = rgb(0,0,0)

Your Task is to implement Dijkstra's algorithm and A* to find a path from start point to end point based on the given constraints and note the time required to find the path and the cost of the path found.

For A* use an admissible heuristic and a non admissible one along with Euclidean distance, Manhattan Distance and Diagonal distance. So in total we have 5 heuristic for A* and we will compare their results.
Note: You are free to choose the admissible and non admissible heuristic as per your choice but choose them wisely so as to see their effects on A* clearly in the results.

We will consider 2 cases
Case 1: Only left, right, top and bottom movement is allowed
Case 2: Along with left, right, top and bottom diagonal movement is also allowed.

Note: You are free to choose the cost function by yourself and your cost of final path should be in accordance with that

Case	Algorithm					
	Dijkstra	A*				
		Admissible Heuristic	Non Admissible Heuristic	Diagonal Distance	Manhattan Distance	Euclidean Distance
Case 1						
Case 2						

Make 2 such tables for **The time taken to find the path** and **cost of the path found**

Sample images are provided in the link below.
https://drive.google.com/drive/folders/1BYWoHXEzkERfXl3NcodYAKwfjVFEBZ2?usp=sharing

For displaying your results a 100px×100px resolution is too low so we will upscale it to 1000px×1000px In the form 10px×10px cells making a 100×100 grid which exactly represents our original image.
To make it more clear here's an image showing the grid for sample image of size 1000px×1000px.

Original Image

Image with Grid lines

Each cell of the grid represents a 10px×10px box in 1000px×1000px Image.

As for the result display the original image in 1000px×1000px form along with the final image along with a path found by the algorithms implemented and the nodes(cells of grid) visited during exploration(Use a color to distinguish them from the rest of the image),i.e in total 7 images per case including the original image

Sample Result with euclidean distance as heuristic for A* for Case 2.

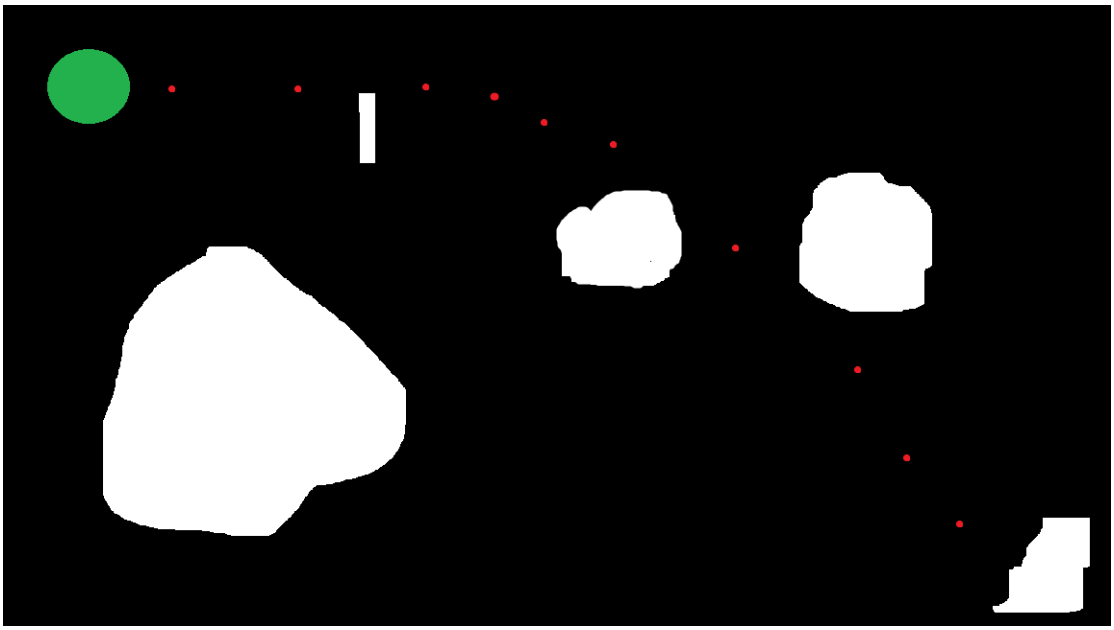
You should be able to explain the trends in time taken, cost of final path and number of nodes visited for different variations of algorithms

Task 2

Introduction

In this task, you will work on a local motion planner in the Frenet Frame. We will use this Frenet Planner.pdf paper as the base of the entire task. We strongly recommend reading this paper thoroughly and making your own notes. This task has multiple sub-tasks which build upon each other. Attempt as many as you can!

- Read the paper carefully before proceeding with the task.
- The paper is **self-sufficient** i.e. it has all the details necessary for you to code the planner. **You do not need to read up the references that are cited in the paper.**
- You are free to use libraries like Eigen in C++ or numpy in Python (and their equivalents)
- Try to modularize your code, it would help you to integrate your code
- In this task there is no concept of lanes, so all available space not occupied by obstacles can be traversed
- You can consider the unit of distance as the pixel size of the image and configure the use of distance, velocity and accelerations accordingly.
- For your ease, you may assume that your vehicle is a small spherical blob, but **not point sized**.
- You do not need to worry about *controllers* in this task, for the simulation assume that you can magically change the position of the vehicle in a single time step.



Reading Task (15%)

Read the linked paper, the paper has been annotated for your convenience, you can, however, ignore the annotations if you wish. If you attempt this task you will be asked questions to gauge how much you have read and understood the paper.

Spline (10%)

In this task for a given vector of points (x_i,y_i), you would have to generate **cubic (or any other) splines** between each (x_i, y_i) and (x_i+1, y_i+1). After doing this you would have to plot the generated curve which consists of individual splines using matplotlib/opencv. (ref 2D Spline Curves).

Polynomials (10%)

In this task, you would have to create a class for a 5-degree polynomial. This polynomial would define the equation of your path between the states a local start and end point. While forming this polynomial you have to consider the state of the car at the start and the end point(State of a car is given by (x, Vx, Ax)) and the time taken to travel between these two points. For each class implement functions to calculate f(x), f'(x) and all other non-trivial derivatives.

Detection (10%)

You will be given images, where you have to identify the ego vehicle (the green circle is your ego vehicle with the centre of the circle being considered as the position of the vehicle at any given point in time), waypoints(These are given by red points in the image. Consider the initial position of the centre of your ego vehicle as the start point and the last waypoint as the end point) and obstacles (Obstacles have been marked in the image as white). After doing so you would have to plot the global path by passing a curve (use the spline sub-task) through the waypoints.

The Planner (40%)

Prior to this task we would suggest to read the paper and then use the sub-tasks solved before to implement the paper. We expect you to implement the sampling process, wherein you will search all possible trajectories in the search space and choose the valid trajectory (should not collide with an obstacle, with the least cost.

In implementing the sampling process you are free to choose the lower limit, upper limit and the step size of all the parameters which have to be sampled. Make sure you use variables in the main process and these variables should be declared in the start of your code so that we can test it thoroughly.

You have to run the sampling process and choose the frenet path for the particular situation. Your program should print the path along with velocity and angular velocity profiles at each point in the path.
You have to plot the following -

1. A velocity profile plot - it is a plot showing the velocity of the bot at each point along the path. For plotting this profile, take a certain scale on the X-axis which is a measure of how far you have moved along the path. Plot on the Y-axis the concerned velocity. (Note : This graph can be plot as an image in OpenCV/MATLAB/anyother thing according to your convenience).
2. Plot the expected frenet path for the first iteration of the frenet planner, on the given image itself assuming that the car is in an initial rest state. Make sure that you plot the global path also prior to plotting the frenet path.(Use Opencv for this plot)

Simulation (15%)

After you have obtained a frenet trajectory run the vehicle on the given trajectory for one timestep, after doing so the position of the ego vehicle would change however the waypoints and the obstacles would remain as they are. For the new position run the sampling process again and obtain a new trajectory. Repeat this process and you would end up with a 2-D simulation video of the planner. Refer to the window on the bottom right in this video (https://www.youtube.com/watch?v=BPBGQtXIR2Q). Note that the video referenced in does lane detection as well, you can ignore that.

Task Explanation

Relevant To Task 1 and 2

Here are a few definitions of terms we will commonly use: A cost map is a representation of the map of your environment stating which are the regions containing obstacles and which are free for your bot to go to. A global cost map is a map representation of a large area not looking into the immediate neighbourhood of your bot. A local cost map is a map representation of the immediate surroundings of your bot. For instance, if you want to go from the Mining Department to Nalanda, and given the fact that your bot can only use its sensors within 100 meters of its vicinity, the map from the Department to Nalanda would be your global cost map whereas the one containing all the moving objects and obstacles within a 100-meter radius of your bot is the local cost map. You can take the images that will be given to you to be your cost map for task 1 and 2.

1. Global Path Planning

Planning for ground vehicles is a fairly complicated task. It involves finding the shortest path to the destination avoiding obstacles and taking into account the kinematics and constraints of the vehicles.

Solving a simple path planning problem on an image using Dijkstra or A* may be pretty easy. But when you try to plan an optimal path for a real vehicle, it may not be so obvious. There are several constraints that one needs to look into.

The primary difference will be the bot dimensions. When you replace the point object with a bot that occupies a 2D space, it's not hard to see that the previously calculated optimal trajectory might not be

feasible. Further, when you add the factor that the bot can rotate about its axis, you increase the computations drastically, though you might get a shorter path. Even with all these constraints, your algorithm might result in a not-so-smooth path. Smoothening it is another important task.

So, you are asked to generate an optimal path keeping all of the above problems in mind and also taking care of computational efficiency. You will be given an image that will act as your cost map. A word of advice - try to generate the path first optimally and then apply smoothing rather than running both parallelly.

The resources given here are not sufficient. You are expected to identify certain keywords and look up stuff so that you know better about what you are working on. And make sure you comment and organise your code so that debugging will be easy. It goes without saying, your code should be object-oriented.

2. Local Path Planning

As already mentioned, path planning for real-life vehicles is a difficult job. Global motion planners have several issues, and local motion planners are designed to solve them. However, local motion planners take greedy decisions and may not be optimal. One major difficulty is that the obstacles are rarely static. At any point in time, it may so happen that the once optimal path gets obstructed by a barrier. Then, the planner may need to replan the complete path. However, this may be very inefficient.

Also, we might not always have a global cost map, i.e., the vehicle only knows about obstacles near it. A local planning algorithm is adopted to address the above issues. It is also easier to generate velocity profiles for the path using local planners.

One such simple and robust local planner is the Frenet frame planner described in the paper. It takes into account all the immediate surrounding of the bot and plans a path accordingly. Note that it also provides you the velocity and acceleration with which your bot has to move at every timestep. Your task is to implement the described planner efficiently. Like the previous task, you can consider the cost map as an image. Your code must be object-oriented.

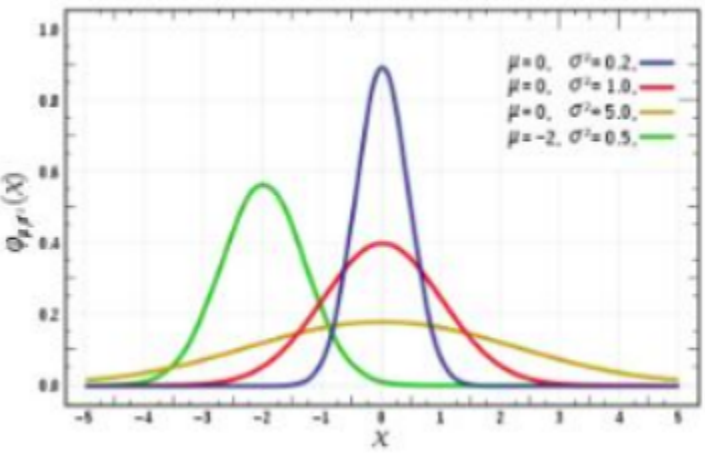
A general word of advice - Read the paper carefully before proceeding. Make sure you understand first what is required of you to code before actually just diving into implementation. It would be better to break down your planner into separate functions wrapped into multiple classes. You can even divide your code into header files or modules. That would modularize your code and also help debug, something that is a feature of a maintainable codebase.

Task 3

Introduction

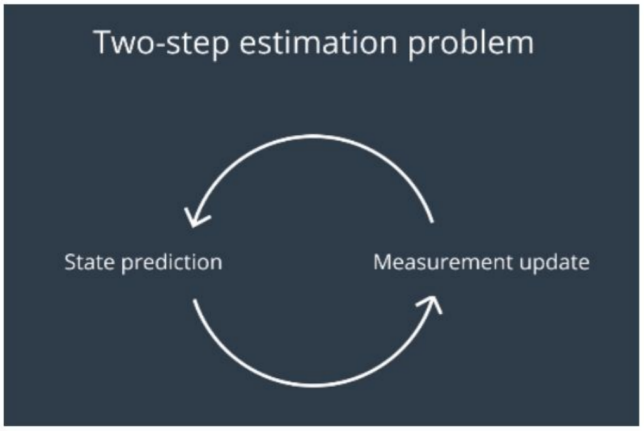
For a self-driving car, it is very important to know where you are and how fast you are moving at all times with high precision. So what we need for this is just your average motion equations and also measurement data. The motion and measurement steps occur iteratively. First, we get a state estimate with the help of our motion model and then we tally that to the measured data.

But wait which one to believe? Which one has more error - Our motion equations or measurement data? Hence to tackle this exact problem we use a very fundamental mathematical function and its properties - The Gaussian.



We consider each of our steps as an operation on Gaussians.

To do this task more effectively we use a mathematical technique known as the Kalman Filter. What Kalman Filter does is that it takes into account both the state estimate and the measurement estimate and gives out a very accurate estimate combining the above two estimates.



The filter loop that goes on and on.

Read about the Kalman filter and its various Matrices. Apply the motion update step and the measurement update step. And then print out the position, velocity and uncertainty matrix at each step.

Problem Statement:

1. You have to find the accurate position of your bot. You have data from two sensors - the GPS coordinates, and the bot's average velocity, given in a file. You have to read each state successively, but only after you have processed the previous one.
2. You need to implement a Kalman filter for a self-driving car travelling in a 2D world.
3. All the required parameters are to be read from a .txt file: link
4. At each step process the data by applying the Kalman Filter, and print the updated positions and their uncertainty.
5. Visualise the processed points using OpenCV or matplotlib.
6. Read about the Extended Kalman Filter (EKF) and Unscented Kalman Filter. (Optional)

Resources:- PROBABILISTIC ROBOTICS

Hint:- You can use the Eigen library for matrix multiplication in C++ and numpy for Python.

Explanation of Input Data: (link)

The format goes something like this:

Line 1: Initial pos x, Initial posy
From Line 2: Pos x, Pos y, vel x, vel y

You can open the file and have a look too. The velx, vely is the average velocity between the previous and current state.

As for the covariances, we have decided to leave it up to you. Well, it goes without saying that you can't choose some random values. The values should go hand in hand with the data such that you end up as close to where you started (Looks like life to me xD).

Task 4

Introduction

In this task, you will work on obstacle segmentation as well as lane detection. Navigating your vehicle through the road is a difficult task and involves many components, the most important of which is obstacle detection. Once we know where the obstacles are we need to navigate the vehicle in the available region. But how will we know where the vehicle can traverse? Lane detection comes into play here. It is a vital component for autonomous vehicles and helps in path planning.

Here is a sample video with which you'll work with: link. You have to perform the following tasks on the video, in order:

1. Obstacle Segmentation:

Using image processing operations, you have to segment out the obstacles present in the video. As shown above, you have to create a mask and blacken the obstacles detected. Be careful, as to not accidentally detect the lanes along with the obstacles. Also, take care of the noise present in the mask.

For this task, you will have to Google and read research papers, articles and any other resources you may find. Try to be as innovative as possible because you'll be evaluated on your final approach and not so much on the final output.

2. Lane Detection:

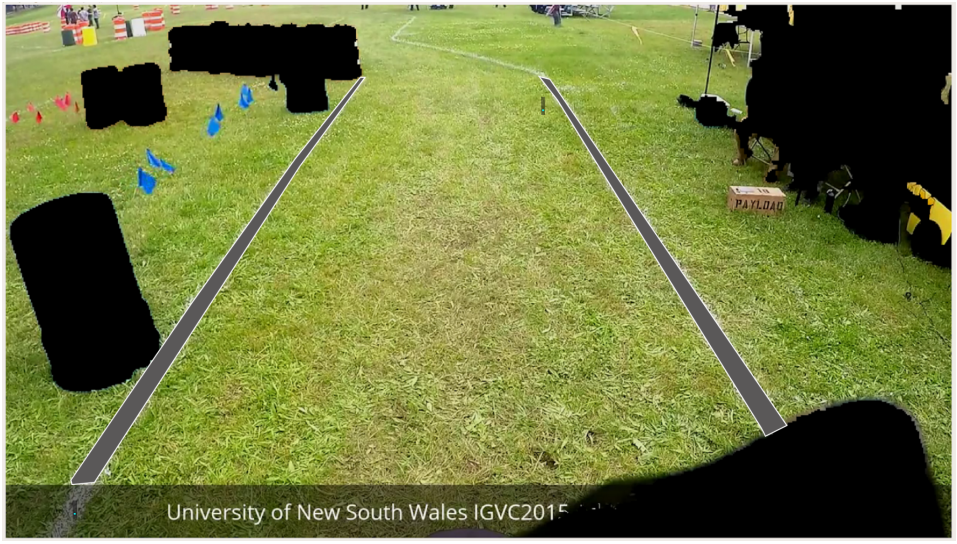
In order to drive autonomously, the vehicle needs to segment drivable regions and identify lane markers as well as process this information for planning its trajectory. The problem statement is to develop a robust lane marker detection algorithm that can detect both straight and curved lanes. Many algorithms are proposed for this task, and innovative solutions will be highly appreciated. Sample output is shown above.

Be careful as many of the obstacles contain white stripes so only taking out the white colour won't work well. Similar to the above task, you will have to Google and read research papers, articles and any other resources you may find.

3. Bonus:

Lane Detection or Obstacle Segmentation alone cannot serve the purpose of navigating an autonomous vehicle. In the real world, you have to move in the lane while avoiding the obstacles simultaneously. Hence, the final task is to merge the above two methods and make a semantic map containing the segmented obstacles as well as the detected lane.

See the sample output shown below, we have labelled the obstacles with black and the detected lanes as grey. You have to reproduce similar results on the entire video.



Task 5

In computer vision and robotics, a typical task is to identify specific objects in an image and to determine each object's position and orientation relative to some coordinate system. This information can then be used, for example, to allow a robot to manipulate an object or to avoid moving into the object. This is called Pose Estimation.

Pose estimation is of great importance in many computer vision applications: robot navigation, augmented reality, etc. This process is based on finding correspondences between points in the real environment and their 2d image projection. This is usually a difficult step, and thus it is common to use synthetic or fiducial markers to make it easier.

One of the most popular approaches is the use of binary square fiducial markers called Aruco Tags. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them especially robust, allowing the possibility of applying error detection and correction techniques.

In this task, you will perform Pose Estimation using Aruco Tags. We have divided this task into two parts.

1. Reading Task
2. Pose Estimation using Aruco Tags

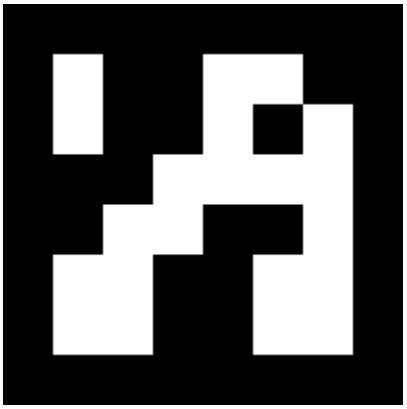
Reading Task:

1. Read about camera calibration
2. Read about the camera's intrinsic and extrinsic parameters.

Resources:

This is a very standard task in computer vision and you can search the net yourself to completely understand these concepts. However, you can refer to the Udacity course for computer vision (by Georgia Tech https://classroom.udacity.com/courses/ud81). Don't watch the whole course. Just see the relevant content. Official OpenCV documentation and Stackoverflow will help too (:P)

Pose Estimation using Aruco Tags:



Let's now move on to the actual coding part. Here's how an Aruco Tag looks like. First, get this Aruco tag printed on an A-4 size sheet. Then use your camera webcam to capture the Aruco tag and use OpenCV to get the relative position and orientation of the webcam with respect to the Aruco tag. You need to find the height of the camera, the distance of the camera from the Aruco tag, and the orientation of the camera in terms of roll, pitch, yaw.

Optional Task: Generate a top view of your surroundings by generating a homography matrix using the parameters obtained from the Aruco Marker.

Feel free to contact any one of us regarding any problem you face. Here is the contact information of all the 2nd years in the Software Team.

Contact Information

1.	Ishan Goel	7076210444	ishangoel04062001@gmail.com
2.	Vinit Raj	8927766234	vinitraj412@gmail.com
3.	Sakshi Dwivedi	7522095441	dwivedi.sakshi809@gmail.com
4.	Rahul Gorai	9547773307	rahul.gorai.3141@gmail.com
5.	Parv Maheshwari	8958008880	parvmaheshwari2002@gmail.com
6.	Animesh Jha	9717911679	jha.animesh01@gmail.com
7.	Karan Uppal	8527107806	karan.uppal3@gmail.com
8.	Shrinivas Khiste	9561112577	shrinivaskhistesk@gmail.com
9.	Aryan Mehta	9909364324	victor2001vini@gmail.com
10.	Nishant Goyal	9826185111	nishantgoyal918@gmail.com
11.	Bhushan Malani	7479215552	bhushanmalani3@gmail.com