Software Task Round

Autonomous Ground Vehicle

Congratulations on making it this far into the selections of the AGV Software Team!

For this final phase of selection, **five (5)** tasks are given, the first one being **mandatory**. Attempt as many tasks as you can. For each task, you first need to understand the problem statement, read concepts from the internet (some resources are provided in this doc itself), and then implement (code) it.

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

1. Make your programs Object Oriented if possible, although this is not a requirement.
2. The resources given in this document are not sufficient. You are expected to find more on your own.
3. You may discuss among yourselves and help each other, but sharing your code is a strict no. The use of plagiarized code will result in complete and immediate disqualification for both candidates, from whom the code/idea has been copied and who have copied it.
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 visualize 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 have to attempt the first task mandatorily, but you are encouraged to do more (Typically the number of tasks we have attempted for our selections are two to three:)) - 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.

Reading and Implementation Task

This task is **mandatory** for all participants, necessitating a comprehensive understanding of the research paper and its underlying concepts. Participants are expected to immerse themselves in the paper's content, grasping the fundamental ideas and methodologies presented. Moreover, each participant is required to make a sincere attempt at **implementing** the search algorithm outlined in the paper. Although the achievement of a fully functional implementation may vary in difficulty for each participant, the exertion to comprehend and apply the concepts is indispensable. This task provides a valuable opportunity for participants to delve deeply into advanced algorithms in multi-agent pathfinding, fostering their overall learning and skill development in autonomous systems research.

Research Paper Details

The assignment involves reading and implementing a portion of the research paper titled "Conflict-based search for optimal multi-agent pathfinding" by Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. The paper introduces an innovative framework for determining optimal paths for a given set of agents within a multi-agent system, accounting for constraints such as collisions, memory limitations, and communication delays.

While collaborative discussions with peers are encouraged, participants are strictly advised against plagiarizing code, as it may lead to disqualification.

Detailed task details can be found here.

VizDoom

Introduction

While the task that follows below does not directly relate to autonomous vehicles, a lot of the ideas and concepts that you will use to solve it will directly carry over to autonomous vehicles and automation in general. This includes but is not limited to planning in continuous spaces, designing controllers to carry out your planned trajectory, and using simulators and integration so that you can test out your algorithms before deploying them in the real world.

For this task, you will be using ViZDoom, which is an AI research platform/simulator based on the game which arguably birthed the FPS genre - Doom. You can visit their GitHub repo here. Follow the instructions there to set it up on your machine (which is just a simple pip install for the most part) and go through the documentation and examples on how to use it. The overall goal of this task is to navigate a maze and reach a checkpoint.

Level 1

Load this custom .wad file into the simulator. A WAD file is a game data file used by Doom and Doom II, as well as other first-person shooter games that use the original Doom engine. On correctly loading the .wad file, you will see something like this:

Global Planning

You can either use the automap from the simulator or use the map directly from here. The white pixel is the initial position of your ego and blue is where you will find the blue skull needed to finish the level. You can use any planning algorithm you like to plan a trajectory but we prefer RRT*. You will need to keep an eye out for dynamic and kinetic constraints of your ego since any path connecting the two points might not be feasible for your controller to follow in the next level.

Trajectory Following

You will need to now translate your global trajectory to actions in space and direct your ego to the blue skull. You might need a closed loop controller that corrects the errors that can build up in following the trajectory.

Level 2

You will no longer have access to the automap or a predefined map. You are only allowed to use the data from the buffers. The objective remains the same. You will most likely need to implement some sort of DFS search in the space while keeping approximate track of where you are, controls for backtracking and behaviors to search for open passages; however, you are free to take any approach you like. An example depth buffer is given below:

Submission

To make your submission for this task, use this modified version of the ViZDoom repository. The examples folder contains two python files, for Level 1 and Level 2 of this task. You can add your code in these files and submit them. We have already set up the .wad file of the custom map in this repository and made changes in the configuration files accordingly.

Feel free to run and change the existing scripts in the examples folder to get a better understanding of the environment.

In addition to the mentioned codes, submit an image of the final global path that your designed algorithm suggested for the given map image. Submit a screen recording of the next parts of the task.

EKF and Particle Filters

The key goal of this task is to get an understanding of the properties of Kalman filters and Particle filters for state estimation. You will be implementing an Extended Kalman Filter (EKF) and a Particle Filter (PF) for landmark based localization. You will also analyze their performance under various conditions. The zip file containing the code for this task can be found here: Download task

Useful reading: Probabilistic Robotics

Motion Model Jacobian

Figure 1 describes a simple motion model. The state of the robot is its 2D position and orientation: $s = [x, y, \theta]$. The control to the robot is $u = [\delta_{\text{rot 1}}, \delta_{\text{trans}}, \delta_{\text{rot 2}}]$, i.e. the robot rotates by $\delta_{\text{rot 1}}$, drives straight forward δ_{trans}, then rotates again by $\delta_{\text{rot 2}}$.

The equations for the motion model g are as follows:

$$
\begin{aligned}
x_{t+1} &= x_t + \delta_{\text{trans}} * \cos(\theta_t + \delta_{\text{rot 1}}) \\
y_{t+1} &= y_t + \delta_{\text{trans}} * \sin(\theta_t + \delta_{\text{rot 1}}) \\
\theta_{t+1} &= \theta_t + \delta_{\text{rot 1}} + \delta_{\text{rot 2}}
\end{aligned}
$$

where $s_{t+1} = [x_{t+1}, y_{t+1}, \theta_{t+1}]$ is the prediction of the motion model. Derive the Jacobians of g with respect to the state $G = \frac{\partial g}{\partial s}$ and control $V = \frac{\partial g}{\partial u}$

Description of Task

In this task, you will implement an Extended Kalman Filter (EKF) and Particle Filter (PF) for localizing a robot based on landmarks.

We will use the odometry-based motion model. We assume that there are landmarks present in the robot's environment. The robot receives the bearings (angles) to the landmarks and the ID of the landmarks as observations: (bearing, landmark ID).

We assume a noise model for the odometry motion model with parameters α and a separate noise model for the bearing observations with parameter β . The land-

mark ID observation is noise-free. See the provided starter code for implementation details.

At each timestep, the robot starts from the current state and moves according to the control input. The robot then receives a landmark observation from the world. You will use this information to localize the robot over the whole time sequence with an EKF and PF.

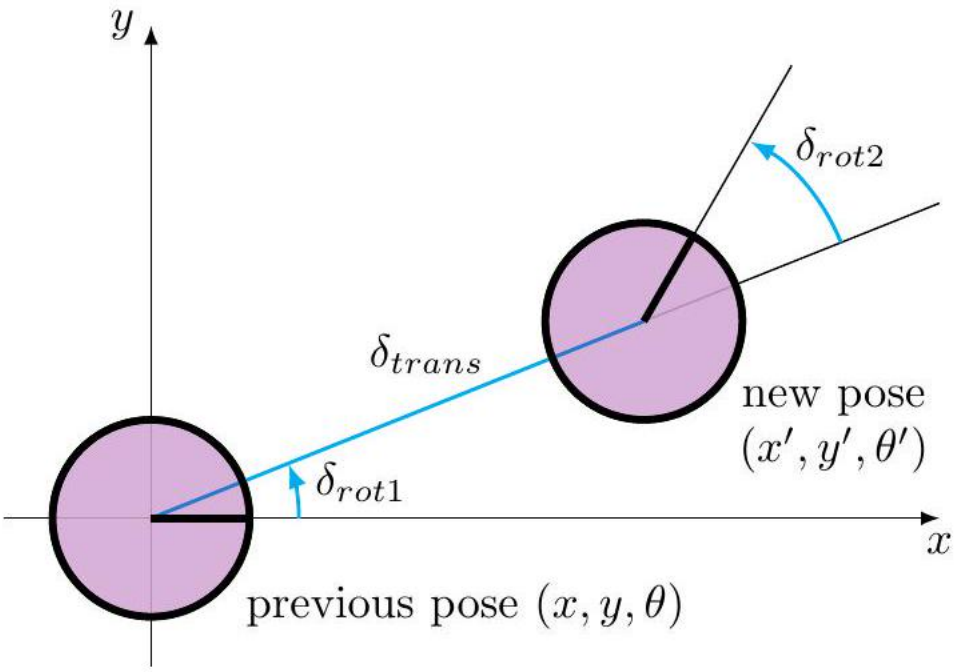

Figure 1: Odometry-based motion model

Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib. This section gives a brief overview of each file. Feel free to make changes to the skeleton code given as well.

- `localization.py`

  - This is your main entry point for running experiments.

- `soccer_field.py`

  - This implements the dynamics and observation functions, as well as the noise models for both. Add your Jacobian implementations here!

- `utils.py`

  - This contains assorted plotting functions, as well as a useful function for normalizing an angle between $[-\pi, \pi]$.

- `policies.py`

  - This contains a simple policy, which you can safely ignore.

- `ekf.py`

  - Add your extended Kalman filter implementation here!

- pf .py - Add your particle filter implementation here!

Command-Line Interface

To visualize the robot in the soccer field environment, run:

```
$ python localization.py --plot none
```

The blue line traces out the robot's position, a result of noisy actions. The green line traces the robot's position assuming that actions weren't noisy. After implementing a filter, the filter's estimate of the robot's position will be drawn in red.

```
$ python localization.py --plot ekf
```

```
$ python localization.py --plot pf
```

To see other command-line flags available, run:

```
$ python localization.py -h
```

Data Format

- state: $[x, y, \theta]$
- control: $[\delta_{\text{rot } 1}, \delta_{\text{trans }}, \delta_{\text{rot } 2}]$
- observation: $\left[\theta_{\text{bearing }}\right]$

Hints

- Ensure to call `utils.minimized_angle` whenever an angle or angle difference could exceed $[-\pi, \pi]$.
- You can look up the low-variance systematic sampler. It provides a smoother particle distribution and requires only a single random number per resampling step.
- Consider turning off plotting for a significant speedup.

EKF Implementation

Implement the extended Kalman filter algorithm in `ekf.py`. You need to complete the `ExtendedKalmanFilter.update` method and the `Field` methods `G`, `V`, and `H`. Successful implementation results should be comparable to the following:

```
$ python localization.py ekf --seed 0
...
Mean position error: 8.9983675360847
Mean Mahalanobis error: 4.416418248584298
ANEES: 1.472139416194766
```

(a) Plot the real robot path and the filter path under the default parameters provided.

(b) Plot the mean position error as the α and β factors range over $r = [1/64, 1/16, 1/4, 4, 16, 64]$ and discuss any interesting observations. Run 10 trials per value of r.

```
$ python localization.py ekf --data-factor 4 --filter-factor 4
```

(c) Plot the mean position error and ANEES (average normalized estimation error squared) as the filter α, β factors vary over r (as above), while the data is generated with the default. Discuss any interesting observations.

PF Implementation

Implement the particle filter algorithm in `pf.py`. You need to complete the `ParticleFilter.update` and `ParticleFilter.resample` methods.

```
$ python localization.py pf --seed 0
...
Mean position error: 8.567264372950905
Mean Mahalanobis error: 14.742252771106532
ANEES: 4.914084257035511
```

(a) Plot the real robot path and the filter path under the default parameters.

(b) Plot the mean position error as the α, β factors range over r and discuss.

(c) Plot the mean position error and ANEES as the filter α, β factors vary over r while the data is generated with the default.

(d) Plot the mean position error and ANEES as the α, β factors range over r, and the number of particles varies over $[20, 50, 500]$.

Submission

Submit a zip file containing all files with your code, and a pdf file with the results and the plots mentioned in the implementation details above.

[01] Since the factors are multiplied with variances, this is between $1/8$ and 8 times the default noise values.

ImageMatchX

Introduction

The task revolves around a document classification method known as Bag of Words (BoW). This technique represents documents as vectors or histograms, where each word's count within the document is recorded. The objective is to identify documents of the same category by comparing their word distributions. By analyzing a new document's word frequencies and comparing them to existing class histograms, we can determine its likely classification. This method assumes that documents within the same class will share similar word distributions, enabling effective categorization based on word occurrence.

Provided Code

1. `createFilterBank`: This function will generate a set of image convolution filters. See Figure 6. There are 4 filters, and each one is made at 5 different scales, for a total of 20 filters. Filters are (Gaussian, Laplacian Gaussian, X Gradient of Gaussian, Y Gradient of Gaussian).
2. `utils.py`: Contains some useful functions to be used in tasks.
   - `imfilter()`: for image filtering
   - `rgb2Lab()`: convert the color space of I from RGB to Lab.
3. `batchToVisualWords.py`: Applies your implementation of `getVisualWords()` to every image in the training and testing set.
4. `computeDictionary.py`: A utility function that will do the legwork of loading the training image paths, processing them, building the dictionary, and saving the pickle file.
5. `extractFilterResponses.py`: Complete this function in **1.1**
6. `getDictionary.py`: Create a dictionary of words for an image.
7. `getHarrisPoints.py`: Complete this to get Harris points in an image.
8. `getImageFeatures.py`: Get a histogram of features of an image.

9. `getRandomPoints.py`: Get the features of an image by randomly sampling them.
10. `getVisualWords.py`: A function to map each pixel in the image to its closest word in the dictionary.

Problem Description

1. Build Visual Words Dictionary

1.1

Write a function to extract filter response, applying all of the n filters on each of the 3 color channels of the input image.

In your write-up: Show an image from the data set and 3 of its filter responses.

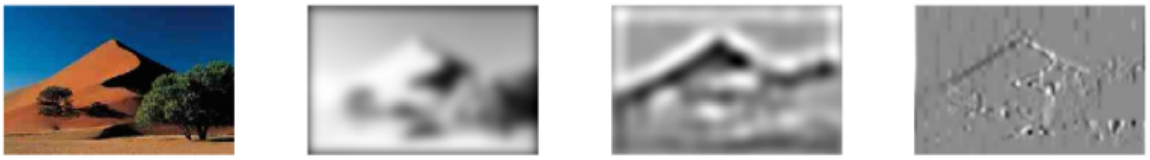

Figure 7: Sample desert image and a few filter responses (in CIE Lab color space)

Figure 1: Sample image and filter responses.

1.2

Write two functions that return a list of points in an image, that will then be used to generate visual words.

Next, write a function that uses the Harris corner detection algorithm to select key points from an input image.

In your write-up: Show results of your corner detector on 3 different images.

Try to make the implementation of Harris as fast as possible.

1.3

Compute Dictionary of Visual Words by using the above two functions.

For this question, you must produce two dictionaries.

Using both random points and Harris method.

2. Build a Visual Scene Recognition System

2.1

Write a function to map each pixel in the image to its closest word in the dictionary.

In your write-up: Show the word maps for 3 different images from two different classes (6 images total).

2.2

Create a function that extracts the histogram of visual words within the given image.

In your write-up: Show the word maps for 3 different images from two different classes (6 images total). Do this for each of the two dictionary types (random and Harris).

2.3

Build Recognition System - Nearest Neighbors.

Making use of nearest neighbor classification, write a script that saves `visionRandom.pkl` and `visionHarris.pkl` and in each pickle store a dictionary that contains:

1. `dictionary`: visual word dictionary, a matrix of size $K \times 3n$.
2. `filterBank`: array of image filters.
3. `trainFeatures`: $T \times K$ matrix containing all of the histograms of visual words of the T training images in the data set.
4. `trainLabels`: $T \times 1$ vector containing the labels of each training image.

You will need to load the train image names and train labels from `traintest.pkl`. Load the dictionary from `dictionaryRandom.pkl` and `dictionaryHarris.pkl` you saved in part **1.3**.

3. Evaluate the Visual Scene Recognition System

3.1

Image Feature Distance.

Write a function to search the most similar image from a set of images using a certain algorithm.

Try to run several algorithms and methods of matching images.

In your write-up: Show similar images to one and show similarity scores.

Links

- Data: Google Drive Link
- Code: Google Drive Link

3D Reconstruction

Introduction

In the realm of autonomous ground vehicles, the ability to perceive and understand the surrounding environment accurately is crucial for safe and efficient operation. One key component of perception is 3D reconstruction, which involves creating a detailed and comprehensive representation of the scene in three dimensions. This process plays a vital role in enabling autonomous vehicles to make informed decisions, accurately detect objects, navigate complex environments, and effectively plan their trajectories.

Task

This task requires you to construct 3D representation of surroundings from multiple view images. A detailed description of the concept behind the task and how you are expected to tackle it are given in the Link. All the required data is also given in the above link.

3D Reconstruction

Contact Information

Name	Phone Number	Email
Gayathri Anant	9481287032	anantgayathri@gmail.com
Swaminathan S K	7395977089	swamisathya2004@gmail.com
Ayush Kumar	9110037759	ayush1856kumar@gmail.com
Sachish Singla	8264097543	sachishs001@gmail.com
Pranaya Chowdary	96069 90317	pranayaschowdary@gmail.com