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. You must complete at least one. 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 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

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.

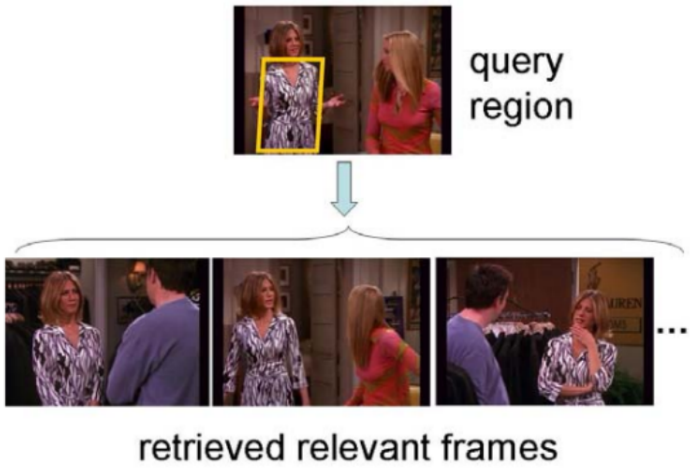
In addition to the mentioned codes, submit an image of the final global path that your designed algorithm suggested for the given map image. Submit a screen recording of the next parts of the task.

Video Search

Introduction

Image features, such as those extracted using the SIFT (Scale-Invariant Feature Transform) algorithm, are distinctive and locally-representative points or regions within an image. These features capture important visual information, such as edges, corners, or blobs, that are invariant to changes in scale, rotation, and illumination.

For this problem, you will implement a video search method to retrieve relevant frames from a video based on the features in a query region selected from some frame. We are providing image data and some starter code for this assignment.



Provided Data ($\sim$ 6GB)

- Pre–computed SIFT features
- Associated Images

Each .mat file in the provided SIFT data corresponds to a single image, and contains the following variables, where n is the number of detected SIFT features in that image:

descriptors	nx128	double // the SIFT vectors as rows
imname	1x57	char // name of the image file that goes with this data
numfeats	1x1	double // number of detected features
orients	nx1	double // the orientations of the patches
positions	nx2	double // the positions of the patch centers
scales	nx1	double // the scales of the patches

Provided Code

Link

- `loadDataExample.py` (ipynb): Run this first and make sure you understand the data format. It is a script that shows a loop of data files, and how to access each descriptor. It also shows how to use some of the other functions below. You can also run the code using the jupyter notebook.
- `displaySIFTPatches.py`: given SIFT descriptor info, it draws the patches on top of an image
- `getPatchFromSIFTParameters.py`: given SIFT descriptor info, it extracts the image patch itself and returns as a single image
- `selectRegion.py`: given an image and list of feature positions, it allows a user to draw a polygon showing a region of interest, and then returns the indices within the list of positions that fell within the polygon.
- `dist2.py`: a fast implementation of computing pairwise distances between two matrices for which each row is a data point

Problem Description

1. **Raw descriptor matching [15 pts]**: Allow a user to select a region of interest (see provided `selectRegion.py`) in one frame, and then match descriptors in that region to descriptors in the second image based on Euclidean distance in SIFT space. Display the selected region of interest in the first image (a polygon), and the matched features in the second image. Use the two images and associated features in the provided file `twoFrameData.mat` (in the gzip file) to demonstrate. Note, no visual vocabulary should be used for this one. Name your script `rawDescriptorMatches.py`.
2. **Visualizing the vocabulary [25 pts]**: Build a visual vocabulary. Display example image patches associated with two of the visual words. Choose two words that are distinct to illustrate what the different words are capturing, and display enough patch examples so the word content is evident (e.g., say 25 patches per word displayed). See provided helper function or `getPatchFromSIFTParameters.py`. Explain what you see. Name your script `vfillisualizeVocabulary.py`.

3. **Full frame queries [30 pts]**: After testing your code for bag-of-words visual search, choose 3 different frames from the entire video dataset to serve as queries. Display the M=5 most similar frames to each of these queries (in rank order) based on the normalized scalar product between their bag of words histograms. Explain the results. Name your script `fullFrameQueries.py`.

4. **Region queries [30 pts]**: Select your favorite query regions from within 4 frames (which may be different than those used above) to demonstrate the retrieved frames when only a portion of the SIFT descriptors are used to form a bag of words. Try to include one or more examples where the same object is found in the most similar M frames but amidst different objects or backgrounds, and also include a failure case. Explain the results, including possible reasons for the failure cases. Name your script `regionQueries.py`.

   Note : Refer to the Readme.md file in the Codes directory to get more information about the basic framework requirements.

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

Localization

Introduction

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

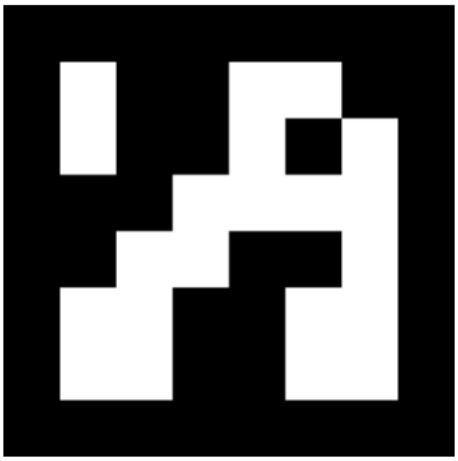
This is a very standard task in computer vision and you can search the net yourself to completely understand these concepts. However, you can refer to the Udacity course for computer vision (by Georgia Tech Link). Don't watch the whole course. Just see the relevant content. Official OpenCV documentation 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.

Sample Aruco Marker

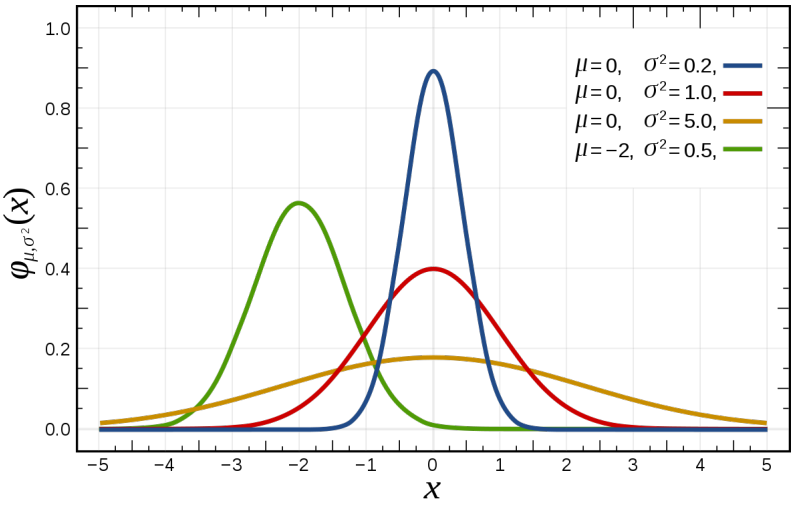


Kalman Filter

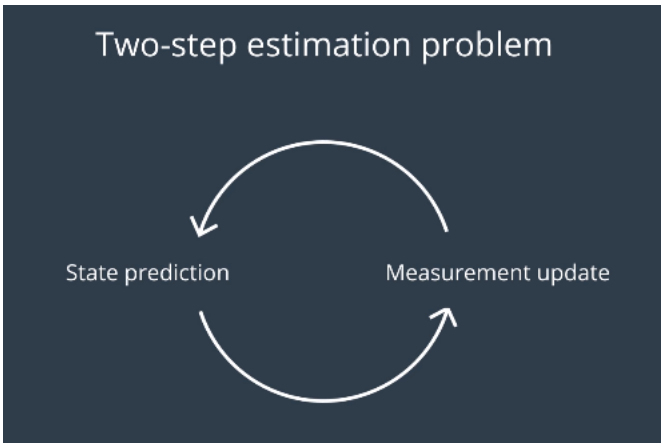
Introduction

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 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.



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 traveling 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. Visualize the processed points using OpenCV or matplotlib.
6. Read about the Extended Kalman Filter (EKF) and Unscented Kalman Filter. (Optional)

Resources: Probabilistic Robotics

Explanation of Input Data:

Get the input data from here
The format goes something like this:

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

Contact Information

Name	Phone Number	Email
Sabariswaran	8870252043	sabaris.offl@kgpian.iitkgp.ac.in
Yash Sirvi	8875988168	yashsirvi@kgpian.iitkgp.ac.in
Sreyas V	7397381091	vsreyas20@kgpian.iitkgp.ac.in
Bratin Mondal	9382416909	bratinmondal689@gmail.com
Apoorv Kumar	7428530275	apoorvapainal@gmail.com
Soumojit Bhattacharya	7045358969	soumojit048@gmail.com
Aatir Zaki	8603808837	aatirzaki@gmail.com
Om Sadhwani	9374609078	omsadhwani0603@gmail.com