Problem 1 (Extended Kalman Filter, 25 points). In this problem, we will see how to use filtering to estimate an unknown system parameter. Consider a dynamical system given by

$$\begin{aligned} x_{k+1} &= a x_k + \epsilon_k \\ y_k &= \sqrt{x_k^2 + 1} + \nu_k \end{aligned} \tag{1}$$

where $x_k, y_k \in \mathbb{R}$ are scalars, $\epsilon_k \sim N(0, 1)$ and $\nu_k \sim N(0, 1/2)$ are zero-mean scalar Gaussian noise uncorrelated across time k. The constant a is unknown and we would like to estimate its value. If we know that our initial state has mean 1 and variance 2

$$x_0 \sim N(1, 2),$$

develop the equations for an Extended Kalman Filter (EKF) to estimate the unknown constant a.

(a) **(5 points)** You should first simulate (1) with $a = -1$. This is the ground-truth value of a that we would like to estimate. Provide details of how you simulated the system, in particular how did you sample the noise ϵ_k, ν_k. The observations $D = \{y_k : k = 1, \ldots, \}$ are the "dataset" that we thus collect from the system. Run the simulation for about 100 observations.

(b) **(15 points)** You should now develop the EKF equations that will use the collected dataset D to estimate the constant a. Discuss your approach in detail. Your goal is to compute two quantities

$$\begin{aligned} \mu_k &= \mathrm{E}\left[a_k \mid y_1, \ldots, y_k\right] \\ \sigma_k^2 &= \mathrm{var}\left(a_k \mid y_1, \ldots, y_k\right). \end{aligned}$$

for all times k.

(c) **(5 points)** Plot the true value $a = -1$, and the estimated values $\mu_k \pm \sigma_k$ as a function of time k. Discuss your result. In particular, do your estimated values $\mu_k \pm \sigma_k$ match the ground-truth value $a = -1$? Does the error reduce as your incorporate more and more observations?

Problem 2 (Unscented Kalman Filter, 100 points). In this problem, you will implement an Unscented Kalman Filter (UKF) to track the orientation of a robot in three-dimensions. We have given you observations from an inertial measurement unit (IMU) that consists of gyroscopes and accelerometers and corresponding data from a motion-capture system (called "Vicon", see https://www.youtube.com/watch?v=qgS1pwsHQIA for example). We will develop the UKF for the IMU data and the vicon data for calibration and tuning of the filter, this is typical of real applications where the robot uses an IMU but the filter running on the robot will be calibrating before test-time in the lab using an expensive and accurate sensor like a Vicon.

(a) **Loading and understanding the data**: First, load the data given on Canvas (file "hw2_p2_data.zip") using code of the form.

```python
from scipy import io

data_num = 1
imu = io.loadmat('imu/imuRaw'+str(data_num)+'.mat')
accel = imu['vals'][0:3,:]
gyro = imu['vals'][3:6,:]
T = np.shape(imu['ts'])[1]
```

Ignore other fields inside the .mat file, we will not use them.

You can use the following code to load the vicon data

```python
vicon = io.loadmat('vicon/viconRot'+str(data_num)+'.mat')
```

while calibrating and debugging. But do not include this line in the autograder submission because we do not store the vicon data on the server.

(b) **(15 points) Calibrating the sensors**. Check the arrays accel and gyro. The former gives the observations received by the accelerometer inside the IMU and the latter gives observations from gyroscope. The variable T denotes the total number of time-steps in our dataset. You will have to read the IMU reference manual (file "imu_reference.pdf" on Canvas) to understand the quantities stored in these arrays. Pay careful attention to the following things. First, the accel/gyro readings are integers and not metric quantities, this is because there is usually an analog-to-digital conversion (ADC) that happens in these sensors and one reads off the ADC value as the actual observation. Because of the way these MEMS sensors are constructed, they will have biases and sensitivity with respect to the working voltage. In order to convert from raw values to physical units, the equation for both accel and gyro is typically

$$\text{value} = (\text{raw} - \beta)\,\frac{3300 \text{ mV}}{1023\ \alpha}$$

where β called the bias, mV stands for milli-volt (most onboard electronics operators at 3300 mV) and α is the sensitivity of the sensor. For the accelerometer, α has units of mV/g where g refers to the gravitational acceleration 9.81 m/s^2. In other words, if $\alpha = 100$ mV/g and bias β is zero, and if the raw accelerometer reading is 10, the actual value of the acceleration along that axis is

$$\text{value} = 10 \times \frac{3300}{1023 \times 100} \times 9.81 = 3.16 \text{ m/s}^2$$

Similarly, the sensitivity of a gyroscope has units mV/degrees/sec. You will have to convert the sensitivity into mV/radians/sec to make everything into consistent units.

Typically, in a real application, we do not know the bias and sensitivity of either sensor. Your goal is to use the rotation matrices in the Vicon data as the ground-truth orientation (see section on quaternions below) to estimate the bias and sensitivity of *both* the accelerometer and the gyroscope. You should be careful on two counts.

(1) The orientation of the IMU need not be the same as the orientation of the Vicon coordinate frame. Plot all quantities in the arrays accel, gyro and vicon rotation matrices to make sure you get this right. Do not proceed to implementing the filter if you are not convinced your solution for this part is correct.

(2) The acceleration a_x and a_y is flipped in sign due to device design. A positive acceleration in body-frame will result in a negative number reported by the IMU. See the IMU manual for more insight.

Hint: To find the sensitivity for the accelerometer, we can assume the only force acting is the gravitational force. Then the magnitude of your 3-dimensional accelerometer readings should be as close to 9.81 as possible. Plot the roll, pitch, and yaw values from the Vicon data; you can extract these from the Vicon rotation matrix. You should compare the Vicon plots with some simple plots obtained only from the accelerometer, and separately only the gyroscope values, to predict orientation. From the accelerometer, you can directly compute roll and pitch for each timestep and compare these with ground truth (Vicon) data to ensure your sensitivity and axes are correct. For the gyroscope, you can use the initial orientation from the accelerometer and then integrate the angular velocity values from the gyroscope for the rest of the time series. Note that the orientation estimates you get from this method will have significant drift, but you should be able to get a sense of the scale and check your sensitivity values. The purpose here is to ensure you are converting the raw digital values in the dataset into meaningful physical units before we begin the filtering.

(c) **(0 points) Quaternions for orientation** We have given you a file named quaternion.py that implements a Python class for handling quaternions. Read this code carefully. In particular, you should study the function euler_angle which returns the Euler angles corresponding to a quaternion, from_rotm which takes in a 3×3 rotation matrix and assigns the quaternion and the function __mul__ which multiplies two quaternions together. Try a few test cases for converting to-and-fro from a rotation matrix/Euler angles to a quaternion to solidify your understanding here.

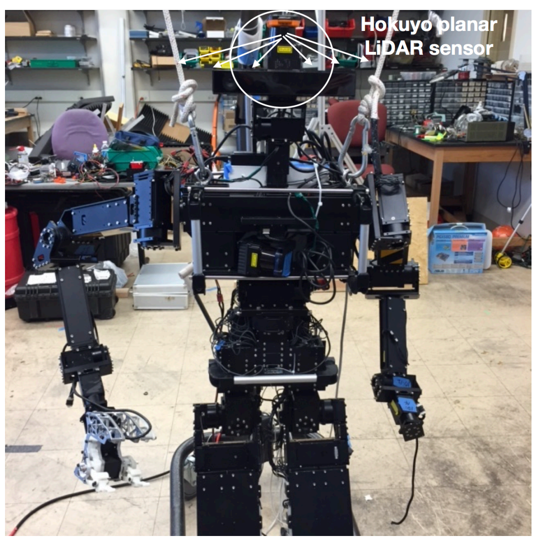
(d) **(0 points) Implementing the UKF** Given this setup, you should now read the PDF on Canvas titled "hw2_p2_ukf_writeup" to implement an Unscented Kalman Filter for tracking the orientation using these observations. The state of your filter will be

$$x = \begin{bmatrix} q \\ \omega \end{bmatrix} \in \mathbb{R}^7$$

where q is the quaternion that indicates the orientation and ω is the angular velocity. The observations are of course the readings of the accelerometer and the gyroscope that we discussed above; recall that gyroscopes measure the angular velocity ω itself which simplifies their observation model. The paper also has a magnetometer as a sensor (which measures the orientation with respect to the magnetic north pole) but we will not use it here. You should implement quaternion averaging as described in Section 3.4 of the paper; this is essential for the UKF to work. **You will have to choose yourself the values of the initial covariance of the state, dynamics noise and measurement noise.** You can discuss your steps and choices in your solution PDF if you want us to follow through your implementation.

(e) **(10 points) Analysis and debugging** Plot the quaternion q (mean and diagonal of covariance), the angular velocity ω (mean and diagonal of covariance), the gyroscope readings in rad/sec and the quaternion corresponding to the vicon orientation as a function of time in your solution PDF. Do not plot on the server, it may crash out. You should show the results for one dataset and discuss whether your filter is working well. You should also use these plots to debug your performance on the other datasets; plotting everything carefully is the fastest way to debugging the UKF.

Problem 3 (Simultaneous Localization and Mapping (SLAM) with a particle filter, 80 points. Do not use Google Colab to do this homework). In this problem, we will implement mapping and localization in an indoor environment using information from an IMU and a LiDAR sensor. We have provided you data collected from a humanoid named THOR that was built at Penn and UCLA (https://archive.darpa.mil/roboticschallenge/finalist/thor.html). You can read more about the hardware in this paper (https://ieeexplore.ieee.org/document/7057369.)



Hardware setup of Thor The humanoid has a Hokuyo LiDAR sensor (https://hokuyo-usa.com/products/lidar-obstacle-detection on its head (the final version of the robot had it in its chest but this is a different version); details of this are in the code (which will be explained shortly). This LiDAR is a planar LiDAR sensor and returns 1080 readings at each instant, each reading being the distance of some physical object along a ray that shoots off at an angle between (-135, 135) degrees with discretization of 0.25 degrees in an horizontal plane (shown as white rays in the picture).

We will use the position and orientation of the head of the robot to calculate the orientation of the LiDAR in the body frame.

The second kind of observations we will use pertain to the location of the robot. However, in contrast to the previous homework were we used the raw accelerometer and gyroscope readings to get the orientation, we will directly use the (x, y, θ) pose of the robot in the world coordinates (θ denotes yaw). These poses were created presumably on the robot by running a filter on the IMU data (such estimates are called odometry estimates), and just as you saw some tracking errors in the previous homework, these poses will not be extremely accurate. However, we will treat them conceptually the same way as we treated Vicon in the previous homework, namely as a much more precise estimate of the pose of the robot that is used to check how well SLAM is working.

Coordinate frames The body frame is at the top of the head (X axis pointing forwards, Y axis pointing left and Z axis pointing upwards), the top of the head is at a height of 1.263m from the ground. The transformation from the body frame to the LiDAR frame depends upon the angle of the head (pitch) and the angle of the neck (yaw) and the height of the LiDAR above the head (which is 0.15m). The world coordinate frame where we want to build the map has its origin on the ground plane, i.e., the origin of the body frame is at a height of 1.263m with respect to the world frame at location (x, y, θ).

Data and code

(a) **(0 points)** We have provided you 4 datasets corresponding to 4 different trajectories of the robot in Towne Building at Penn. For example, dataset 0 consists of two files data/train/train_lidar0.mat and data/train/train_joint0.mat which contain the LiDAR readings and joint angles respectively. The functions load_lidar_data and load_joint_data inside load_data.py read the data. You can run the function show_lidar to see the LiDAR data. Each of the data reading functions returns a data-structure where t refers to the time-stamp (in seconds) of the data, xyth refers to (x, y, θ) *pose of the LiDAR* and rpy refers to Euler angles (roll, pitch, yaw). The joint data contains a number of fields, but we are only interested in the angle of the head and the neck at a particular time-stamp. You should read these functions carefully and check the values returned by them. The dicts joint_names and joint_names_to_index can be used to read off the data of a specific joint (we only need the head and the neck).

(b) **(0 points)** Next look at the slam.py file provided to you. Read the code for the class map_t and slam_t and the comments provided in the code very carefully. You are in charge of filling in the missing pieces marked as TODO: XXXXXX. A suggested order for studying this code is as follows: slam_t.read_data, slam_t.init_sensor_model, slam_t.init_particles, slam_t.rays2world, map_t.__init__, map_t.grid_cell_from_xy. Next, the file utils.py contains a few standard rigid-body transformations that you will need. You should pay attention to the functions smart_plus_2d and smart_minus_2d that will be used to code up the dynamics propagation step of the particle filter.

(c) **(20 points, dynamics step)** Next look at main.py which has two functions run_dynamics_step and run _observation_step which act as test functions to check if the particle filter and occupancy grid update has been updated correctly. The run_dynamics function plots the trajectory of the robot (as given by its IMU data in the LiDAR data-structure). It also initializes 3 particles and plots all particles at different time-steps while performing the dynamics step with a very small dynamics noise; this is a very neat way of checking if dynamics propagation in the particle filter is working correctly. This function will create two plots, one for the odometry trajectory and one more for the particle trajectories, both these trajectories should match after you code up the dynamics function slam_t.dynamics_step correctly. Include these plots for all datasets in your report. Briefly explain how you implemented the dynamics step.

(d) **(20 points, observation step)** The function run_observation_step is used to perform the observation step of the particle filter to get an estimate of the location of the robot and updates to the occupancy grid using observations from the LiDAR. First read the comments for the function slam_t.observation_step carefully.

We first discuss the particle filtering part.

(i) Compute the head and neck position for the time t. For each particle, assuming that that particle is indeed the true position of the robot, project the LiDAR scan slam_t.lidar[t]['scan'] into the world coordinates using the slam_t.ray2world function. The end points of each ray tell us which cells in the map are occupied, for each particle.

(ii) In order to compute the updated weights of the particle, we need to know the likelihood of LiDAR scans given the state (our current occupancy grid in the case of SLAM). We are going to use a simple model to do so

$$\log \mathrm{P}(\text{LiDAR scan as if the robot is at particle } p \mid m) = \sum_{ij \in O} m_{ij} \qquad (1)$$

where O is the set of occupied cells as detected by the LiDAR scan assuming the robot is at particle p and m_{ij} is our current estimate of the binarized map (more on this below). In simple words, if the occupied cells as given by our LiDAR match the occupied cells in the binarized map created from the past observations, then we say the log-probability of particle p is large.

(iii) You will next implement the function slam_t.update_weights that takes the log-probability of each particle p, its previous weights, calculates the updated weights of the particles.

(iv) Typically, resampling step (slam_t.stratified_resampling) is performed only if the effective number of particles (as computed in slam_t.resample_particles) falls below a certain threshold (30% in the code). Implement resampling as we discussed in the lecture notes.

We will now do the mapping part. We have a number of particles $p^i = (x^i, y^i, \theta^i)$ that together give an estimate of the distribution of the location of the robot. For this homework, you will only use the particle with the largest weight to update the map although typically we update the map using all particles. Our goal is simple: we want to increase map_t.log_odds array at cells that are recorded as obstacles by the LiDAR and decrease the values in all other cells. You should add slam_t.log_odds_occ to all occupied cells and add slam_t.log_odds_free from all cells in the map. It is also a good idea to clip the log_odds to like between [-slam_t.map.log_odds_max, slam_t.map.log_odds_max] to prevent increasingly large values in the log_odds array. The array slam_t.map.cells is a binarized version of the map (which is used above to calculate the observation likelihood).

Check the run_observation_step function after you have implemented the observation step. Since the map is initialized to zero at the beginning of SLAM which results in all observation log-likelihoods to be zero in (1), we need to do something special for the first step. We will use the first entry in slam_t.lidar[0]['xyth'] to get an accurate pose for the robot and use its corresponding LiDAR readings to initialize the occupancy grid. You can do this easily by initializing the particle filter to have just one particle and simply calling the slam_t.observation_step as shown in main.py.

Include in your report the output of the run_observation_step function for one time-step. Briefly explain how you implemented the observation step.

(e) **(40 points)** You will now run the full SLAM algorithm that performs one dynamics step and observation step at each iteration in the function run_slam in main.py. Make sure to start SLAM only after the time when you have both LiDAR scans and joint readings (the two arrays start at different times). For all 4 datasets, include in your report the plots of the final binarized version of the map, the (x, y) location of the particle in the particle filter with the largest weight at each time-step and the odometry trajectory (x, y) (in a different color); this counts for 10 points each.