EE4308 Project 2

Simplified Extended Kalman Filter on a Drone

AY 23/24 Semester 2
March 16, 2024

Lai Yan Kai lai.yankai@u.nus.edu
Xue Junyuan xue.junyuan@u.nus.edu
Wang Fei elewf@nus.edu.sg
A/P Prahlad Vadakkepat prahlad@nus.edu.sg

This document is scaled for easier reading on phones.
Please consider a 2-page layout when printing.

Contents

1 Introduction

Implement a simplified, extended Kalman Filter (EKF) to estimate the pose and speed (twist) of a flying drone. Use the drone to travel between different waypoints while a turtle navigates an obstacle course.

The drone will first take off. It will then repeat the following three steps until the turtlebot reaches the turtlebot's final waypoint – (i) move to the moving turtlebot, (ii) move to the turtlebot's waypoint, (iii) and back to the initial position above the take off point. Once the turtlebot reaches the final waypoint, the cycle continues until (iii), when the drone begins to land. The drone maintains a cruising height of 3m, and has to yaw at a constant rate of 0.3 rad/s while flying. The maximum horizontal (x and y) and vertical (z) speed have to be 1 m/s and 0.5 m/s respectively.

This project **is entirely simulated**, and roll and pitch on the drone can be ignored. This year, the behavior node has been programmed for you, and it will guide the drone through the different waypoints. Teams are expected to code the estimator, controller and smoother nodes. The estimator node contain the EKF implementation, and should be handled by two people. The controller node can be handled by one person, and the smoother node is trivial. **The turtlebot code should not be touched**.

All teams are required to do all tasks, except for the sonar and barometer EKF corrections for 3-man teams. 3-man teams are strongly encouraged to implement the sonar EKF correction due to its accuracy, simplicity, and extra points.

1.1 First-time Setup

1.1.1 SJTU Drone

1. Use an existing terminal or open a terminal with `Ctrl+Alt+T`.
2. You need an internet connection to clone a GitHub repository. Run the following to clone and build the repository.

```
mkdir -p ~/sjtu_drone/src
cd ~/sjtu_drone/src
git clone https://github.com/NovoG93/sjtu_drone.git
cd ~/sjtu_drone
colcon build --symlink-install
cd ~
```

Some of you prefer to put the `sjtu_drone` folder elsewhere. If so, make sure that the `params.sh` file in the `ee4308` workspace folder (next section) is adjusted to point to the correct location of the `sjtu_drone` folder.

1.1.2 EE4308 Files

1. Backup your project 1 or lab files.
2. Download `proj2.zip` from the `Project 2` folder on Canvas.
3. Extract the `ee4308` workspace folder into the Home (`~`) folder.
4. Use an existing terminal or open a terminal with `Ctrl+Alt+T`, and build the workspace.

```
cd ~/ee4308
chmod +x *.sh
./bd.sh
```

Wait for about 2 minutes.

5. You may run the program with

```
./run.sh
```

   The default implementation uses ground truth to move the robot around. Ground truth is used if `use_gt` is set to `true` in `proj2.yaml`.

6. Send a `Ctrl+C` to interrupt or stop the execution.

1.2 Location of Important Files

1. The `.hpp` files can be found in `src/ee4308_drone/include/ee4308_drone/`.
2. The `proj2.yaml` parameter file for the project can be found in `src/ee4308_bringup/params/`.

1.3 Drone Teleoperation

To run teleoperation,

1. Terminate any running `./run.sh`.
2. In `proj2.yaml`, map the `drone:controller2:topic:cmd_vel` parameter to a name that is *not* `cmd_vel`.
3. Run `./run.sh`
4. In another terminal, run

```
./teleop_drone.sh
```

5. `Ctrl+C` when done.

To switch back to the controller, map the parameter back to `cmd_vel`.

1.4 Using Ground Truth

1. Terminate any running `./run.sh`.
2. In `proj2.yaml`, map the `drone:estimator:use_gt` parameter to `true`.
3. Run `./run.sh`

For the estimator node to output the values from the designed EKF, set the parameter to `false`. **Ground truth cannot be used for the demonstration**, but can be used for testing the controller.

2 Estimator Task (EKF)

The inertial navigation system (INS) combines multiple readings to estimate the motion states of a robot, and this can be done using the Extended Kalman Filter (EKF). In this project you are required to use EKF to estimate the drone's position by incorporating several sensor measurements.

2.1 Simplified Motion Model

To simplify the calculations, we will:

1. **Ignore roll** ϕ (rotation about x-axis) **and pitch** θ (rotation about y-axis). So, the robot's frame z-axis is always pointing in the same direction as the world frame's z-axis, with the yaw ψ (rotation about z-axis) existing in the system.
2. Estimate x, y, z and ψ and their velocities **separately**.

The states in the state vector $\hat{\mathbf{X}}$ are modelled as:

$$\hat{\mathbf{X}}_{k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{k-1|k-1}, \mathbf{U}_k) \tag{1}$$

where $\mathbf{f}$ is a vector of functions, usually non-linear as real-world dynamics are (for example, containing sin and cos), depending on the current input $\mathbf{U}_k$ and the previous filtered state $\hat{\mathbf{X}}_{k-1|k-1}$.

2.1.1 States Along x Degree-of-freedom

Let us see an example of $\mathbf{f}$ for the scalar case. In this simplified INS model, we assume that x, the x-coordinate of the drone in the world frame, has the following relation:

$$\begin{aligned} x_{k|k-1} &= x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}a_{x,k}(\Delta t)^2 && (2)\\ &= f(x_{k-1|k-1}, \dot{x}_{k-1|k-1}, a_{x,k}) && (3) \end{aligned}$$

where $x_{k|k-1}$ is the predicted state, $x_{k-1|k-1}$ and $\dot{x}_{k-1|k-1}$ are the previous filtered position and velocity respectively in the world frame (the prior), and $a_{x,k}$ is the acceleration input in the world frame at time step k. You can quickly verify from high school kinematics that the equation is correct if we assume the acceleration to be constant across Δt.

Usually, we need to estimate more state variables to do more things. This is the reason we go from scalar (f) to vector ($\mathbf{f}$). In this project, we want to estimate the velocity $\dot{x}$ as well:

$$\begin{aligned} \hat{\mathbf{X}}_{x,k|k-1} &= \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) && (4)\\ \begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} &= \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}a_{x,k}(\Delta t)^2 \\ \dot{x}_{k-1|k-1} + a_{x,k}\Delta t \end{bmatrix} && (5) \end{aligned}$$

Here, $a_{x,k}$ refers to the acceleration input in the *world* frame. In the INS model, we treat the IMU accelerations as if they were inputs to the process. To use them, we must first transform the IMU frame into the robot frame and then from the robot frame to the world frame. For this project, **we**

ignore roll and pitch, so the transformation of the accelerations between the robot frame and world frame is simply:

$$\begin{bmatrix} a_{x,k} \\ a_{y,k} \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{6}$$

where $u_{x,k}$ and $u_{y,k}$ are the IMU accelerations in the robot frame, **and ψ is shorthand for** $\psi_{k-1|k-1}$. So, we have:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{7}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}(\Delta t)^2(u_{x,k}\cos\psi - u_{y,k}\sin\psi) \\ \dot{x}_{k-1|k-1} + \Delta t(u_{x,k}\cos\psi - u_{y,k}\sin\psi) \end{bmatrix} \tag{8}$$

Of course, we should consider other states like y and $\dot{y}$ in $\hat{\mathbf{X}}$. However, in this simplified INS, they can be estimated **separately** from each other, so $\hat{\mathbf{X}}$ is not a big vector combining all of these states.

2.1.2 States Along other Degrees-of-freedom

For linear kinematics on the y-axis:

$$\hat{\mathbf{X}}_{y,k|k-1} = \begin{bmatrix} y_{k|k-1} \\ \dot{y}_{k|k-1} \end{bmatrix} = \begin{bmatrix} y_{k-1|k-1} + \dot{y}_{k-1|k-1}\Delta t - \frac{1}{2}(\Delta t)^2(u_{x,k}\sin\psi + u_{y,k}\cos\psi) \\ \dot{y}_{k-1|k-1} - \Delta t(u_{x,k}\sin\psi + u_{y,k}\cos\psi) \end{bmatrix} \tag{9}$$

For linear kinematics on the z-axis, where $u_{z,k}$ is the measured z acceleration in the robot frame and $G = 9.8$:

$$\hat{\mathbf{X}}_{z,k|k-1} = \begin{bmatrix} z_{k|k-1} \\ \dot{z}_{k|k-1} \end{bmatrix} = \begin{bmatrix} z_{k-1|k-1} + \dot{z}_{k-1|k-1}\Delta t + \frac{1}{2}(\Delta t)^2(u_{z,k} - G) \\ \dot{z}_{k-1|k-1} + \Delta t(u_{z,k} - G) \end{bmatrix} \tag{10}$$

For angular kinematics on the z-axis, where $u_{\psi,k}$ is the measured yaw angular velocity in the robot frame:

$$\hat{\mathbf{X}}_{\psi,k|k-1} = \begin{bmatrix} \psi_{k|k-1} \\ \dot{\psi}_{k|k-1} \end{bmatrix} = \begin{bmatrix} \psi_{k-1|k-1} + \Delta t u_{\psi,k} \\ u_{\psi,k} \end{bmatrix} \tag{11}$$

2.2 EKF Prediction (All Teams)

2.2.1 Predicting x

INS uses the Extended Kalman Filter (EKF) to estimate the states. Now that we know the simplified INS model, we can then proceed to put it in a form suitable for EKF prediction. The prediction step estimates the next state based on the INS model. Consider $\hat{\mathbf{X}}_{x,k|k-1}$:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{12}$$

$$\approx \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{13}$$

$$= \frac{\partial \mathbf{f}}{\partial \hat{\mathbf{X}}_{x,k-1|k-1}}\hat{\mathbf{X}}_{x,k-1|k-1} + \frac{\partial \mathbf{f}}{\partial \mathbf{U}_{x,k}}\mathbf{U}_{x,k} \tag{14}$$

where $\mathbf{F}$ and $\mathbf{W}$ are Jacobian matrices (first-order partial derivative matrix) with respect to the previous state and input respectively. For this project, we find that the Jacobians are:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{15}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} \approx \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{16}$$

$$= \begin{bmatrix} \frac{\partial x_{k|k-1}}{\partial x_{k-1|k-1}} & \frac{\partial x_{k|k-1}}{\partial \dot{x}_{k-1|k-1}} \\ \frac{\partial \dot{x}_{k|k-1}}{\partial x_{k-1|k-1}} & \frac{\partial \dot{x}_{k|k-1}}{\partial \dot{x}_{k-1|k-1}} \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} \frac{\partial x_{k|k-1}}{\partial u_{x,k}} & \frac{\partial x_{k|k-1}}{\partial u_{y,k}} \\ \frac{\partial \dot{x}_{k|k-1}}{\partial u_{x,k}} & \frac{\partial \dot{x}_{k|k-1}}{\partial u_{y,k}} \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{17}$$

$$= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \cos\psi & -\frac{1}{2}(\Delta t)^2 \sin\psi \\ \Delta t \cos\psi & -\Delta t \sin\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{18}$$

which is just conveniently substituting Eq. (8) into a matrix form, as all the prior variables are linear with respect to the predicted state.

EKF assumes the true state $\mathbf{X}$ is gaussian. Therefore, $\hat{\mathbf{X}}$ is the expectation of the state, also known as the *mean*. Estimating a gaussian variable requires that the *variance* is estimated as well, which indicates how certain the filter is of the estimated mean. Let $\mathbf{P}$ represent the covariance matrix (a general

form of the variance) of the state. We find that the predicted $\mathbf{P}$ is:

$$\mathbf{P}_{x,k|k-1} = \mathbf{F}_{x,k}\mathbf{P}_{x,k-1|k-1}\mathbf{F}_{x,k}^{\top} + \mathbf{W}_{x,k}\mathbf{Q}_x\mathbf{W}_{x,k}^{\top} \tag{19}$$

where $\mathbf{P}_{x,k-1|k-1}$ is the previous filtered covariance matrix and $\mathbf{Q}_x$ is the diagonal covariance matrix of the IMU noise in the IMU frame:

$$\mathbf{Q}_x = \begin{bmatrix} \sigma_{\text{imu},x}^2 & 0 \\ 0 & \sigma_{\text{imu},y}^2 \end{bmatrix} \tag{20}$$

2.2.2 Predicting other degrees of freedom

To summarise, we have converted the simplified INS model using Jacobians into a form suitable for EKF, so that we estimate the current state $\hat{\mathbf{X}}$ and its corresponding uncertainty $\mathbf{P}$:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{21}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}a_{x,k}(\Delta t)^2 \\ \dot{x}_{k-1|k-1} + a_{x,k}\Delta t \end{bmatrix} \tag{22}$$

$$= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2\cos\psi & -\frac{1}{2}(\Delta t)^2\sin\psi \\ \Delta t\cos\psi & -\Delta t\sin\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{23}$$

$$= \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{24}$$

$$\mathbf{P}_{x,k|k-1} = \mathbf{F}_{x,k}\mathbf{P}_{x,k-1|k-1}\mathbf{F}_{x,k}^{\top} + \mathbf{W}_{x,k}\mathbf{Q}_x\mathbf{W}_{x,k}^{\top} \tag{25}$$

Using this example, you should derive for the other three degrees-of-freedom y, z and ψ respectively:

1. For y, refer to Eq. (9). Find $\mathbf{F}_{y,k}$, $\mathbf{U}_{y,k}$ and $\mathbf{W}_{y,k}$.
2. For z, refer to Eq. (10). $\mathbf{Q}_z = \sigma_{\text{imu},z}^2$ and $\mathbf{W}_{z,k}$ is 2×1 in size. The IMU frame's z-axis is assumed to be aligned with the world frame's z-axis, since roll and pitch are ignored. Find $\mathbf{F}_{z,k}$, $\mathbf{U}_{z,k}$ and $\mathbf{W}_{z,k}$.
3. For ψ, refer to Eq. (11). $\mathbf{Q}_\psi = \sigma_{\text{imu},\psi}^2$ and $\mathbf{W}_{\psi,k}$ is 2×1 in size. Find $\mathbf{F}_{\psi,k}$, $\mathbf{U}_{\psi,k}$ and $\mathbf{W}_{\psi,k}$.

2.2.3 Implementation

Use the following variables, which were initialised for you. Code it in the `predict()` function in `estimator.hpp`. Remember to set `use_gt` to `false` in `proj2.yaml`.

The IMU noise variances can be tuned from `proj2.yaml`. To trust the IMU more in a degree of freedom, decrease the respective variance. Tuning should be done after the EKF corrections are implemented.

Symbol	Pvt. Class Var.	Description
$\hat{\mathbf{X}}_x$	`Xx_`	2×1 state vector for x in the world frame.
$\hat{\mathbf{X}}_y$	`Xy_`	2×1 state vector for y in the world frame.
$\hat{\mathbf{X}}_z$	`Xz_`	2×1 state vector for z. Similar in form as x. Is 3×1 if barometer correction is implemented. See Sec. 2.3.4.
$\hat{\mathbf{X}}_\psi$	`Xa_`	2×1 state vector for ψ. Similar in form as x.
$\mathbf{P}_x$	`Px_`	2×2 state covariance matrix for x.
$\mathbf{P}_y$	`Py_`	2×2 state covariance matrix for y. Similar to $\mathbf{P}_x$.
$\mathbf{P}_z$	`Pz_`	2×2 state covariance matrix for z. Is 3×3 if the barometer correction is implemented. See Sec. 2.3.4.
$\mathbf{P}_\psi$	`Pa_`	2×2 state covariance matrix for ψ.

Symbol	Msg. / Local Var.	Description
$u_{x,k}$	`msg.linear_acceleration.x`	Measured IMU x acceleration value in robot frame.
$u_{y,k}$	`msg.linear_acceleration.y`	Measured IMU y acceleration value in robot frame.
$u_{z,k}$	`msg.linear_acceleration.z`	Measured IMU z acceleration value in robot frame.
$u_{\psi,k}$	`msg.angular_velocity.z`	Measured IMU ψ velocity value in IMU frame.
Δt	`dt`	Time elapsed between predictions.

Symbol	YAML Param.	Description
G	`params_.G`	Acceleration due to gravity. Set to 9.8 as Gazebo uses 9.8.
$\sigma^2_{\text{imu},x}$	`params_.var_imu_x`	Variance of IMU x-axis acceleration noise in robot frame.
$\sigma^2_{\text{imu},y}$	`params_.var_imu_y`	Variance of IMU y-axis acceleration noise in robot frame.
$\sigma^2_{\text{imu},z}$	`params_.var_imu_z`	Variance of IMU z-axis acceleration noise in robot frame.
$\sigma^2_{\text{imu},\psi}$	`params_.var_imu_a`	Variance of IMU yaw velocity noise in robot frame

2.3 EKF Correction

The correction is done asynchronously depending on the availability of the measurements:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^\top + \mathbf{V}_k\mathbf{R}_k\mathbf{V}_k^\top)^{-1} \tag{26}$$

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} + \mathbf{K}_k\left[\mathbf{Y}_k - \mathbf{h}(\hat{\mathbf{X}}_{k|k-1})\right] \tag{27}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - \mathbf{K}_k\mathbf{H}_k\mathbf{P}_{k|k-1} \tag{28}$$

Where $\mathbf{h}$ is the forward sensor model – transforms the predicted state $\hat{\mathbf{X}}_{k|k-1}$ into what they would appear to be as sensor measurements; $\mathbf{H}$ is the Jacobian of the measurement with respect to the robot state; $\mathbf{V}$ is the Jacobian of the measurement with respect to raw sensor measurements; $\mathbf{Y}$ is the measurement, and $\mathbf{K}$ is the Kalman gain, which is basically an changing weighted average depending on the certainty of states and measurements.

$\mathbf{V}$ is useful if the measurement noise is not in the same frame as $\mathbf{Y}_k$. For this project, $\mathbf{V}$ is just one or an identity matrix, but it is useful to keep this in mind.

When there are no measurements between predictions, correction does not occur, and we simply treat:

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} \tag{29}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} \tag{30}$$

If there are multiple measurements from different sensors within a short time, you can still correct multiple times to get $\hat{\mathbf{X}}_{k|k}$ without any prediction step between corrections.

2.3.1 GPS (All Teams)

The GPS measurements λ (longitude), φ (latitude) and h (height) are converted in the Gazebo world frame before using the EKF correction to update the states. We first calculate the ECEF coordinates (x_e, y_e, z_e) from these measurements, by noting that the Earth can be modelled as an ellipsoid:

$$e^2 = 1 - \frac{b^2}{a^2} \tag{31}$$

$$N(\varphi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}} \tag{32}$$

$$\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} = \begin{bmatrix} (N(\varphi) + h) \cos\varphi \cos\lambda \\ (N(\varphi) + h) \cos\varphi \sin\lambda \\ \left(\frac{b^2}{a^2} N(\varphi) + h\right) \sin\varphi \end{bmatrix} \tag{33}$$

where $N(\varphi)$ is the prime vertical radius of curvature, a is the equatorial radius, b is the polar radius, and e^2 is the square of the first numerical eccentricity.

The equations above have to be run once in the **first callback** to calculate the initial ECEF coordinates ($x_{e,0}$, $y_{e,0}$ and $z_{e,0}$).

The local NED coordinates (n, e, d) can be subsequently found by using a rotation:

$$\mathbf{R}_{e/n} = \begin{bmatrix} -\sin\varphi\cos\lambda & -\sin\lambda & -\cos\varphi\cos\lambda \\ -\sin\varphi\sin\lambda & \cos\lambda & -\cos\varphi\sin\lambda \\ \cos\varphi & 0 & -\sin\varphi \end{bmatrix} \tag{34}$$

$$\begin{bmatrix} n \\ e \\ d \end{bmatrix} = \mathbf{R}_{e/n}^{\top} \left(\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} - \begin{bmatrix} x_{e,0} \\ y_{e,0} \\ z_{e,0} \end{bmatrix} \right) \tag{35}$$

From the WGS84 convention, the x-axis points to the East, the y-axis points to the North, and the z-axis points up. To transform from the NED frame

to the world frame, use:

$$\mathbf{R}_{m/n} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{36}$$

$$\begin{bmatrix} x_{\text{gps}} \\ y_{\text{gps}} \\ z_{\text{gps}} \end{bmatrix} = \mathbf{R}_{m/n} \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \tag{37}$$

where (x_0, y_0, z_0) are the initial coordinates in the world frame.

The EKF correction is as follows:

$$\mathbf{Y}_{\text{gps},x,k} = x_{\text{gps}} \tag{38}$$

$$\mathbf{h}(\hat{\mathbf{X}}_{x,k|k-1}) = x_{k|k-1} \tag{39}$$

$$\mathbf{H}_{\text{gps},x,k} = \frac{\partial \mathbf{Y}_{\text{gps},x,k}}{\partial \hat{\mathbf{X}}_{x,k|k-1}} = \begin{bmatrix} \frac{\partial x_{\text{gps}}}{\partial x_{k|k-1}} & \frac{\partial x_{\text{gps}}}{\partial \dot{x}_{k|k-1}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{40}$$

$$\mathbf{V}_{\text{gps},x,k} = 1 \tag{41}$$

$$\mathbf{R}_{\text{gps},x,k} = \sigma^2_{\text{gps},x} \tag{42}$$

where $\sigma^2_{\text{gps},x}$ is the variance of x_{gps}. The y and z measurements are of a similar form, with the numerical form of $\mathbf{H}$ and $\mathbf{V}$ being identical. Then, apply Eq. (26, 27, 28) to find the updated x , y and z states. The GPS measurements do not correct ψ.

The covariances σ^2_{gps} can be obtained by examining the `/drone/gps` topic on another terminal and while `./run.sh` is running with

```
ros2 topic echo /drone/gps
```

Googling the message type contained in the topic will help. The message type can be found with

```
ros2 topic type /drone/gps
```

The covariances should be set in the `proj2.yaml` file (see tables below).

Implement the GPS correction in the function `getECEF()` and `correctFromGps()` functions in `estimator.hpp`.

Symbol	Pvt. Class. Var.	Description
$\begin{bmatrix} x_{\text{gps}} & y_{\text{gps}} & z_{\text{gps}} \end{bmatrix}^\top$	`Ygps_`	3×1 vector containing the GPS measurements. Write to it to display into terminal.
$\begin{bmatrix} x_e & y_e & z_e \end{bmatrix}^\top$	`initial_ECEF_`	3×1 vector of the initial ECEF coordinates.
$\begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^\top$	`initial_`	3×1 vector of the drone's initial position in the world frame. Automatically written and can be read directly.

Symbol	Msg. / Local Var.	Description
λ	`lon`	Longitude, in radians.
φ	`lat`	Latitude, in radians.
h	`alt`	Height or altitude, in radians.

Symbol	YAML Param.	Description
a	`params_.rad_equator`	Equatorial Radius.
b	`params_.rad_polar`	Polar Radius
$\sigma^2_{\text{gps},x}$	`params_.var_gps_x`	Variance of GPS x-axis noise in world frame.
$\sigma^2_{\text{gps},y}$	`params_.var_gps_y`	Variance of GPS y-axis noise in world frame.
$\sigma^2_{\text{gps},z}$	`params_.var_gps_z`	Variance of GPS z-axis noise in world frame.

2.3.2 Magnetometer (All Teams)

The magnetometer is a magnetic compass, measuring the force vector $(x_{\text{mgn}}, y_{\text{mgn}}, z_{\text{mgn}})$ of the magnetic north. We convert to ψ by ignoring the z component and using the x and y components:

$$\mathbf{Y}_{\text{mgn},\psi,k} = \psi_{\text{mgn}} = f_{\text{mgn}}(x_{\text{mgn}}, y_{\text{mgn}}) \tag{43}$$

$$\mathbf{h}(\hat{\mathbf{X}}_{\psi,k|k-1}) = \psi_{k|k-1} \tag{44}$$

$$\mathbf{H}_{\text{mgn},\psi,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{45}$$

$$\mathbf{V}_{\text{mgn},\psi,k} = 1 \tag{46}$$

$$\mathbf{R}_{\text{mgn},\psi,k} = \sigma^2_{\text{mgn},\psi} \tag{47}$$

where $\sigma^2_{\text{mgn},\psi}$ is the variance of ψ_{mgn}, which is calculated from x_{mgn} and y_{mgn} via f_{mgn} that is to be determined. Note that the force vector is measured **in the robot frame**, so keep in mind that f_{mgn} may be counter-intuitive – picture yourself rotating with a compass on your hand starting from magnetic north, then measure the angle of the needle, and try to determine your rotation angle in the world frame from the former.

To find $\sigma^2_{\text{mgn},\psi}$, the robot can be kept still at the start (remap the `cmd_vel` topic in the `proj2.yaml` file), and the variance of $\mathbf{Y}_{\text{mgn},\psi,k}$ can be captured from many samples.

A gazebo bug places the magnetic north near the x-axis direction when it should be close to the y-axis direction (WGS84). As the robot always start pointing in the direction of the world frame's positive x-axis, no rotations are required in f_{mgn} for Gazebo and this project. In reality, the transformation of the magnetic north to the world frame has to be found.

Program the implementation in the function `correctFromMagnetic()` in `estimator.hpp`.

Symbol	Pvt. Class. Var.	Description
$\mathbf{Y}_{\text{mgn},\psi,k}p$	`Ymagnet_`	a `double` containing the measured yaw angle.

Symbol	Msg. / Local Var.	Description
x_{mgn}	`msg.vector.x`	The x-component of the vector pointing to magnetic north.
y_{mgn}	`msg.vector.y`	The y-component of the vector pointing to magnetic north

Symbol	YAML Param.	Description
$\sigma^2_{\text{mgn},\psi}$	`params_.var_magnet`	Variance of the yaw measurement noise in the world frame.

2.3.3 Sonar (4-man team)

All Teams should implement this as the correction is simple and offers good accuracy. Let z_{snr} be the sonar measurement:

$$\mathbf{Y}_{\text{snr},z,k} = z_{\text{snr}} \tag{48}$$
$$\mathbf{h}(\hat{\mathbf{X}}_{z,k|k-1}) = z_{k|k-1} \tag{49}$$
$$\mathbf{H}_{\text{snr},z,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{50}$$
$$\mathbf{V}_{\text{snr},z,k} = 1 \tag{51}$$
$$\mathbf{R}_{\text{snr},z,k} = \sigma^2_{\text{snr},z} \tag{52}$$

where $\sigma^2_{\text{snr},z}$ is the variance of z_{snr}. $\sigma^2_{\text{snr},z}$ can be estimated by collecting samples of $\mathbf{Y}_{\text{snr},z,k}$.

As ground obstacles directly beneath the drone will affect the reading of z_{snr}, a low-pass filter (exponential forgetting) can be implemented.

Program the implementation in the function `correctFromSonar()` in `estimator.hpp`. The Gazebo sonar sensor plugin produces very large readings near the start of the run. The behavior is corrected for you in the code provided.

Symbol	Pvt. Class. Var.	Description
$\mathbf{Y}_{\text{snr},z,k}$	`Ysonar_`	a `double` containing the measured height. If a low-pass filter is used, assign the filtered value to this variable.

Symbol	Msg. / Local Var.	Description
z_{snr}	`msg.range`	The measured height.

Symbol	YAML Param.	Description
$\sigma^2_{\text{mgn},\psi}$	`params_.var_sonar`	Variance of the height measurement noise. As the project ignores roll and pitch, the height measurement is parallel to the z-axis.

2.3.4 Barometer / Altimeter (4-man team)

The altimeter uses pressure measurements to determine the height from sea level. z_{bar} is the height measured by the barometer:

$$\mathbf{Y}_{\text{bar},z,k} = z_{\text{bar}} \tag{53}$$
$$\mathbf{h}(\hat{\mathbf{X}}_{z,k|k-1}) = z_{k|k-1} \tag{54}$$
$$\mathbf{H}_{\text{bar},z,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{55}$$
$$\mathbf{V}_{\text{bar},z,k} = 1 \tag{56}$$
$$\mathbf{R}_{\text{bar},z,k} = \sigma^2_{\text{bar},z} \tag{57}$$

where $\sigma^2_{\text{bar},z}$ is the variance of z_{bar}, and can be estimated by collecting samples of $\mathbf{Y}_{\text{bar},z,k}$.

The equations above does not estimate the bias in the barometer measurements, which frequently occurs. The bias should be estimated as a new state variable by augmenting the original state variable with $b_{\text{bar},k|k}$:

$$\hat{\bar{\mathbf{X}}}_{z,k|k} = \begin{bmatrix} \hat{\mathbf{X}}_{z,k|k} \\ b_{\text{bar},k|k} \end{bmatrix} = \begin{bmatrix} z_{k|k} \\ \dot{z}_{k|k} \\ b_{\text{bar},k|k} \end{bmatrix} \tag{58}$$

so $\bar{\mathbf{P}}_{z,k|k}$ is now a 3×3 matrix, $\bar{\mathbf{F}}_{z,k}$ is 3×3, and $\bar{\mathbf{W}}_{z,k}$ is 3×1. The latter two are trivial to derive. The equations remain the same, except for the correction in Eq. (27) which has to be modified into:

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} + \mathbf{K}_k \left[\mathbf{Y}_k - \mathbf{h}(\hat{\mathbf{X}}_{k|k-1}) - b_{\text{bar},k|k} \right] \tag{59}$$

Determine the modified $\mathbf{H}_{\text{bar},z,k}$ from here.

Program the implementation in the function `correctFromBaro()` in `estimator.cpp`. In the function `verbose()`, **please output the barometer bias to the terminal by uncommenting the correct pieces of code**.

Symbol	Pvt. Class. Var.	Description
$\mathbf{Y}_{\text{bar},z,k}$	`Ybaro_`	a `double` containing the measured barometer value that includes the bias.

Symbol	Msg. / Local Var.	Description
z_{bar}	`msg.point.z`	The measured barometer value that includes the bias.

Symbol	YAML Param.	Description
$\sigma^2_{\text{bar},z}$	`params_.var_baro`	Variance of the barometer measurement noise.

3 Controller Task (All Teams)

3.1 Get Lookahead Point

In the function `getLookahead()` in `controller.hpp`, find the lookahead point from a path.

Contrary to project 1, the front of the path (index 0) is located at the drone's position. The back of the path is located at the drone's desired waypoint.

The path will only be updated when the waypoint changes. Unless the waypoint is on the moving turtle, the path will not be updated regularly. As such, the path may remain unchanged for most of the time, and the lookahead point should be found by first finding the closest point. To find the lookahead,

1. Find the closest point on the path.
2. From the closest point, search along the path toward the desired waypoint (back of path). Identify the first point that exceeds the lookahead distance. The identified point is the lookahead point.
3. If no points can be found in the previous step, the drone is close to the desired waypoint. In this case, the lookahead point is the desired waypoint (back of path).

Use the following variables in the code.

Local Var.	Description
`plan`	A vector containing the path. To access the x-coordinate of the point at index i, use `plan[i].pose.pose.position.x`. Same for y and z.
`drone_pose`	The drone pose. To access the drone's x-coordinate, use `drone_pose.position.x`. Same for y and z.
`lookahead_.point`	Write the lookahead point into this variable. If the lookahead point is at index i, assign `plan[i].pose.position` to this variable.

YAML Param.	Description
`params_.lookahead_distance`	The lookahead distance (m).

3.2 Moving the robot

The function `move()` sends command signals to the robot based on the lookahead point. The controller is based on pure pursuit. The desired velocities are first determined by a proportional (P) controller, which considers the distance to a lookahead point. The desired velocities are subsequently constrained. As the drone is holonomic, the drone can move in any linear direction simultaneously.

Let x', y', and z' be the coordinates of the lookahead point in the robot frame.

$$x' = (x_p - x_t)\cos\psi_t + (y_p - y_t)\sin\psi_t \tag{60}$$
$$y' = (y_p - y_t)\cos\psi_t - (x_p - x_t)\sin\psi_t z' \qquad = (z_p - z_t) \tag{61}$$

x_p, y_p, and z_p are the lookahead point's coordinates in the world frame, x_t, y_t, and z_t are the coordinates of the drone in the world frame, and ψ_t is yaw heading of the drone in the world frame.

Let $d_{h,t}$ and $d_{v,t}$ be the desired horizontal and vertical command linear velocities to be published into the `cmd_vel` topic. The desired velocities are determined by the proportional controller

$$d_{h,t} = K_h\sqrt{x'^2 + y'^2} \tag{62}$$
$$d_{v,t} = K_v z' \tag{63}$$

where K_h and K_v are the proportional gains to be tuned via `proj2.yaml`.

Let $c_{h,t-1}$, $c_{v,t-1}$ be the previous horizontal and vertical command linear velocities

$$c_{h,t-1} = \sqrt{c_{x,t-1}^2 + c_{y,t-1}^2} \tag{64}$$
$$c_{v,t-1} = c_{z,t-1} \tag{65}$$

where $c_{x,t-1}$, $c_{y,t-1}$, and $c_{z,t-1}$ are the velocities sent to `cmd_vel` topic in the previous time step.

Next, constrain the accelerations of the desired velocities $d_{h,t}$ and $d_{v,t}$ by comparing against the previous velocities $c_{h,t-1}$ and $c_{v,t-1}$ respectively. After

constraining the accelerations, constrain the velocities. The steps can be found in Lab 1 and in the code you implemented in Project 1. The functions sgn() and abs() can be used directly. It may be useful to note that the horizontal velocity is always positive.

Let the final, constrained velocities be $c_{h,t}$ and $c_{v,t}$. Send the velocity commands as follows:

$$c_{x,t} = c_{h,t}\frac{x'}{\sqrt{x'^2 + y'^2}} \tag{66}$$

$$c_{y,t} = c_{h,t}\frac{y'}{\sqrt{x'^2 + y'^2}} \tag{67}$$

$$c_{z,t} = c_{v,t} \tag{68}$$

$$c_{\psi,t} = \dot{\psi} \tag{69}$$

where $\dot{\psi}$ is the constant yaw velocity determined by `proj2.yaml`.

Use the following variables in the code.

Symbol	Msg. / Local Var.	Description
x_t	`drone_pose.position.x`	The drone's x-coordinate in the world frame.
y_t	`drone_pose.position.y`	The drone's y-coordinate in the world frame.
z_t	`drone_pose.position.z`	The drone's z-coordinate in the world frame.
ψ_t	`drone_yaw`	The drone's yaw heading in the world frame.
x_p	`lookahead_.point.x`	The lookahead point's x-coordinate in the world frame.
y_p	`lookahead_.point.y`	The lookahead point's y-coordinate in the world frame.
z_p	`lookahead_.point.z`	The lookahead point's z-coordinate in the world frame.
c_x	`cmd_vel_.linear.x`	The linear x-axis command velocity in the robot frame. Read from this to get the previous time step's value, and then write into it at the end of the function.
c_y	`cmd_vel_.linear.y`	The linear y-axis command velocity in the robot frame. Read and write is the same as c_x.
c_z	`cmd_vel_.linear.z`	The linear z-axis command velocity in the robot frame. Read and write is the same as c_x.
c_ψ	`cmd_vel_.angular.z`	The yaw command velocity in the robot frame. Is a constant.

Symbol	YAML Param.	Description
K_h	`params_.kp_horz`	The proportional gain for the horizontal velocity.
K_v	`params_.kp_vert`	The proportional gain for the vertical velocity.
$\dot{\psi}$	`params_.yaw_rate`	The yaw rate of the drone (rad/s). Set to 0.3.
—	`params_.max_horz_vel`	Maximum horizontal speed (m/s). Set to 1.0.
—	`params_.max_vert_vel`	Maximum vertical speed (m/s). Set to 0.5.
—	`params_.max_horz_acc`	Maximum absolute horizontal acceleration (m/s^2).
—	`params_.max_vert_acc`	Maximum absolute vertical acceleration (m/s^2).

4 Smoother Task (All Teams)

In the `smooth()` function in `smoother.hpp`, interpolate the straight line path between the drone and the desired waypoint by generating points at regular distance intervals.

The interpolated points are to be pushed into the `plan` variable. To create a point and push into the variable, use the following template

```
geometry_msgs::msg::PoseStamped ps; // new interpolated
    point
ps.pose.position.x = 1; // change the value
ps.pose.position.y = 1; // change the value
ps.pose.position.z = 1; // change the value
plan.poses.push_back(ps); // push into plan via .poses.
```

Use the following variables in the code.

Msg. / Local Var.	Description
`goal.pose.position.x`	The x-coordinate of the desired waypoint.
`goal.pose.position.y`	The y-coordinate of the desired waypoint.
`goal.pose.position.z`	The z-coordinate of the desired waypoint.
`start.pose.position.x`	The drone's x-coordinate.
`start.pose.position.y`	The drone's y-coordinate.
`start.pose.position.z`	The drone's z-coordinate.
`plan`	The returned path (see above).

YAML Param.	Description
`params_.interval`	The interval between interpolated points.

5 Demonstration Task

1. Implement the required code and tune the required parameters.
2. Then, refer to the demonstration recording on Canvas, in the `Videos/Panopto` tab. The layout should be similar, with the terminal output on the left side and the Rviz output on the right side.
3. The terminal output has to show the messages from the `verbose()` function in `estimator.hpp`. If the barometer is implemented, the barometer bias has to be shown (see Sec. 2.3.4).
4. Rotate / orbit the Rviz visualization if the drone trajectory looks too flat on the screen.

6 Challenging Optional Tasks

1. **Challenging tasks receive much less points than main tasks.** Make sure to fulfil the main tasks to the best of your ability before attempting the challenging tasks. For example, show the behavior of different sets of parameters and the educated guesses / inferences used to arrive at a good set of parameters.
2. The path is replanned only when the drone's desired waypoint changes. As such, a path is requested only once when the desired waypoint is not the moving turtlebot. If the path is replanned at regular intervals, the trajectory accounts for the drone's current position, resulting in less oscillatory behavior. As the current velocity of the drone can now be obtained, a cubic Hermite spline can be generated at the smoother node to provide a curved path suited for the drone's motion.
3. Other value-adding features or code.

7 Tips and Problems

7.1 Common Coding Problems

1. To move the workspace folder to another computer in a `.zip`, you should not zip the `build`, `log` and `install` folders. This applies to code submission.
2. Use `./clean_bd.sh` to remove the folders and build again, if the codes are not building correctly even after changing the files.
3. Running ./run.sh may occasionally cause the robots to spawn in the middle of the map as white (monochrome) 3D models, or some nodes to not spawn correctly, particularly the controller and smoother nodes. Simply rerun ./run.sh. If you have coded for an extended time, you may want to reboot the computer (or virtual machine).
4. VSCode Intellisense may fail abruptly. Try to reduce the number of errors first. Then, restart VSCode, or reboot the computer (or virtual machine) as a final resort to fix the problem. As long as `./bd.sh` builds without failure, and values are correctly returned in non-void functions, errors in Intellisense can be ignored.

7.2 Using Eigen

Eigen is a popular linear algebra library used in Computer Vision and engineering. It can be used in this project to implement the EKF.

1. An `Eigen Matrix<double, 2, 1>` is equivalent to a `Vector2d` type. A vector is a $1 \times n$ matrix.
2. `Matrix<double, 3, 1>` is equivalent to a `Vector3d` type.
3. `Matrix<double, 2, 2>` is equivalent to a `Matrix2d` type.
4. `Matrix<double, 3, 3>` is equivalent to a `Matrix3d` type.
5. When multiplying or dividing an `Eigen` matrix or vector with a scalar (`double` type), make sure that the scalar is on the right hand side.
6. A `Matrix` or `Vector m` can be transposed with `m.transpose()`.
7. Only a square `Matrix m` can be inversed with `m.inverse()`.
8. To do matrix multiplication, simply use the `*` operator.
9. Only matrices or vectors of the same size can be added or subtracted.
10. To access an element in a `Vector`, use the `()` operator and one index (e.g. `m(2)` to access the third element).
11. To access an element in a `Matrix`, use the `()` operator and the row and column indices (e.g. `m(0,1)` to access the element at the first row and second column.)
12. To initialize `m`, first declare it then use the `<<` operator.

```
Eigen::Matrix2d m, n;
m << 1, 2, // the new line is optional, but makes
    it clearer to see.
    3, 4;
n << 3, 4, 5, 6;
Eigen::Vector2d v;
v << 5, 6;
```

13. `Vector` types can be separately initialized with initializer lists (curly braces), but not `Matrix` types.

```
Eigen::Vector3d m = {2,3,4};
Eigen::Vector3d n;
n << 2, 3, 4; // equivalent to m.
```

14. The elements in the matrices and vectors are initialized with zeros. To initialize to another value, use the corresponding type's `Constant()` function.

```
Eigen::Matrix2d m = Eigen::Matrix2d::Constant(3);
    // all elements are 3.
Eigen::Matrix3d n =
    Eigen::Matrix3d::Constant(1e3); // all
    elements are 1000.
```

15. When printing the variables to terminal take care of the line breaks for matrices or vectors with multiple rows. You can use the transpose function to print a vector onto one line.

```
Eigen::Vector3d m = {1,2,3};
std::cout << m << std::endl; // prints in three
    lines.
std::cout << m.transpose() << std::endl; // prints
    in one line.
```

8 Signups / Submissions

8.1 Signup for Presentation Slots

1. Sign up for presentation slots before W12 Wed, 10 Apr, 23:59h.
2. Go to `Canvas` → `People` to sign up for the slots.
3. The time will be fixed at a later date, and will be on W12 Thu, 11 Apr, and W12 Fri, 12 Apr.

8.2 Report (PDF)

1. Deadline: W12 Wed, 10 Apr, 23:59h
2. `.pdf` type.
3. Name the report as `team##.pdf`, where `##` is the team number in two digits (e.g. Team 4 is `team04.pdf`).
4. Submit the report to `Canvas` → `Assignments` → `P2R`.

8.3 Compiled Video (MP4)

1. Deadline: W12 Wed, 10 Apr, 23:59h
2. The compiled video must:
   a) Be at most 7-min long.

b) Contain a 5-min presentation of the project, detailing the report succinctly. The length can exceed 5-min as long as the entire video is 7-min long.

c) Contain a 2-min maximum, screen recording of the terminal outputs and RViz of the project. Put the video after the 5-min presentation.

3. Name the video as `team##.mp4`.
4. Submit the video to `Canvas` → `Assignments` → `P2P`, via Panopto.
5. `.mp4` type.

8.4 Code (ZIP)

1. Deadline: W12 Wed, 10 Apr, 23:59h
2. Zip the `ee4308/src` folder.
3. Again, only the `src` folder, not the entire `ee4308` folder.
4. Name the zip file as `team##.zip`.
5. Submit the zip file to `Canvas` → `Assignments` → `P2C`.

9 Future Announcements and Updates

There may be errors in the code or the manual. Keep a lookout for these on Canvas Discussions.