EE4308 Project 2

Using Extended Kalman Filter on the Hector

AY 21/22 Semester 2
March 10, 2022

Lai Yan Kai lai.yankai@u.nus.edu
Dr. 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 the hector drone. Use the drone to travel between different waypoints while a turtle navigates an obstacle course. These waypoints include the moving turtle, the hector's initial position, and the turtle's final goal. The hector must maintain a height of 2m and yaw at a constant rate of 0.5 rad/s while flying. Roll and pitch on the hector can be ignored for this project. This is project that **runs entirely on simulations**.

This is a team project – students are to form **teams of three or four**. Each team is expected to:

1. Record multiple videos and compile them into a maximum 7-min long video.
2. Submit a report detailing all requirements and analysis.
3. Submit the code.

The teams remain the same as project 1.

2 Submissions / Signups

2.1 Sign up for Presentation Slots in Week 13

You are required to be present while the compiled video is played back, and to answer questions after that. Go to `LumiNUS` → `Class and Groups` → `Class Groups` → `Project 2 Presentation` to sign up for the slots.

2.2 Submit Compiled Video, Report and Code

The compiled video, report and code must be submitted by midnight 10 Apr (W12 Sunday).

Name them in LOWERCASE as `team##.mp4`, `team##.pdf` and `team##.zip`, where `##` is the double digit team number: e.g. `team01.mp4`, `team11.zip`. Only these file types are allowed. Penalties will be given for wrong naming formats and file types.

2.2.1 Report (PDF)

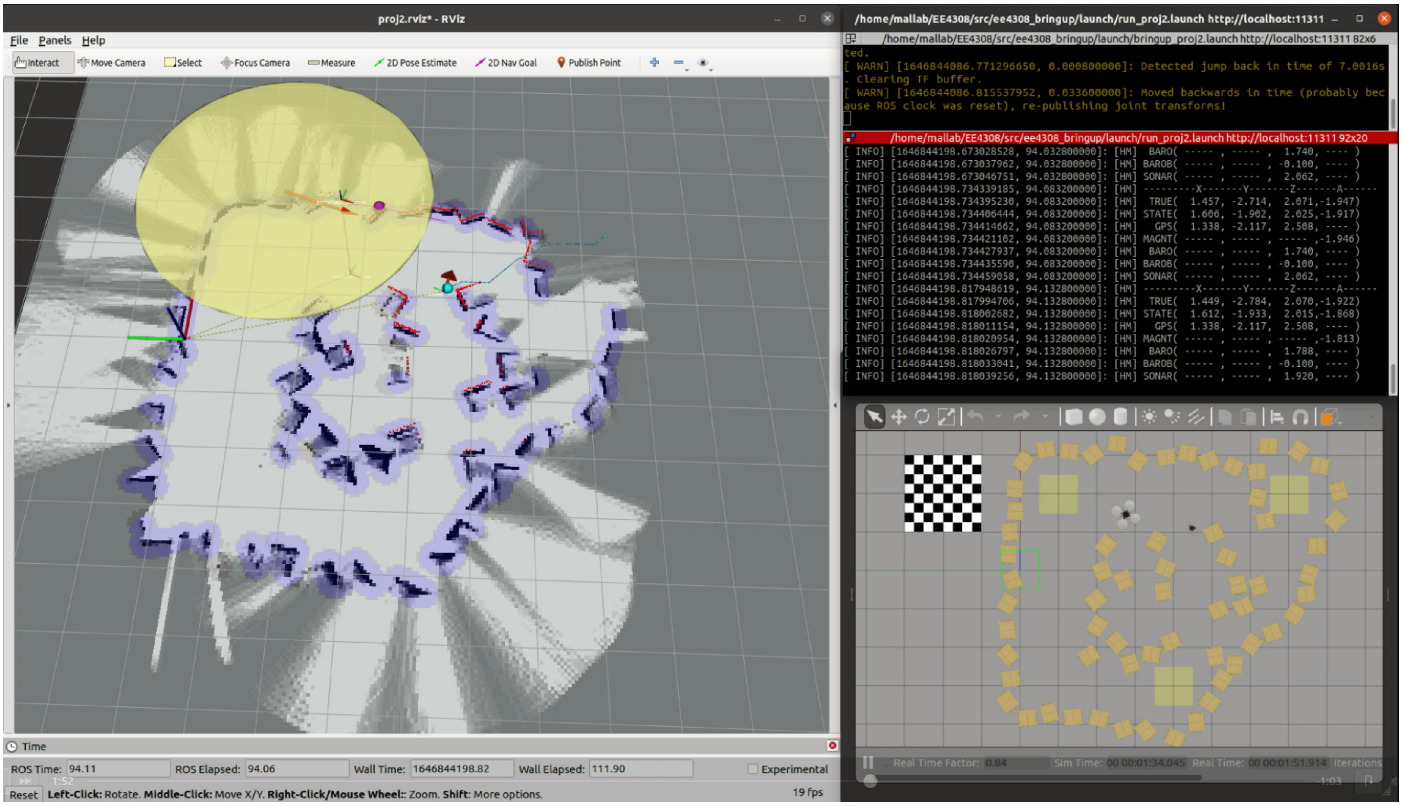
The report must include analysis on any problems identified and solved, all tasks done, and ample justification and data for every step taken. See the next section for the tasks. See Sec. 4 for more details. Submit the `team##.pdf` report to `LumiNUS` → `Files` → `Project 2` → `Report (PDF)`.

2.2.2 Compiled Video (MP4)

The compiled video must:

1. Contain a maximum 5-min presentation of the project, detailing the report succinctly.

2. Contain a maximum 2-min demonstration video of the robot moving in the physical area, and played back side-by-side a screen recording of the terminal outputs and RViz. Put these after the 5-min presentation. **The layout of the RViz, terminals and Gazebo** must follow that of the GA demonstration video on `LumiNUS` → `Multimedia`:



You may put the demonstration video and screen recording together in PowerPoint, and record the playback together with Zoom for a maximum 7-min long video. Submit the `team##.mp4` video to `LumiNUS` → `Files` → `Project 2` → `Compiled Video (MP4)`.

2.2.3 Code (ZIP)

Name your workspace as `team##` (see above) instead of `a456b`. Only one workspace is required for every team. Please make sure that you:

1. Zip workspace into `team##.zip`.
2. Open the zip and remove the `build` and `devel` folders.

Submit the `team##.zip` file to `LumiNUS` → `Files` → `Project 2` → `Code (ZIP)`.

3 Setup

3.1 Return Turtlebot3 Burger

No marks will be awarded for this project if the hardware is not returned, and any of the missing items are not replaced.

3.2 Reconfigure Environment Variables

All activities in this project are simulated and on the PC. Therefore, we need to undo the changes we made for project 1 in `.bashrc`.

1. In a terminal, open `.bashrc`:

```
nano ~/.bashrc
```

2. Scroll to the end (using `Ctrl+END` or `Ctrl+DOWNARROW`) where you put the `ROS_MASTER_URI` and `ROS_HOSTNAME`.

3. Comment out the lines using `#` like below:

```
# export ROS_MASTER_URI=http://192.168.0.1:11311
# export ROS_HOSTNAME=192.168.0.1
```

4. `Ctrl+S` and `Ctrl+X` to save and exit.

5. Source the bash script using:

```
source ~/.bashrc
```

3.3 Copy Files and Configure

1. Backup your project 1 files and rename / remove the `team##` workspace folder. Keep `ee4308_turtle/src/move.cpp` and `ee4308_turtle/config/turtle.yaml`.
2. Download files from `LumiNUS` → `Files` → `Project 2` → `proj2.zip`.
3. Extract everything in `proj2.zip`'s `EE4308` folder into `~` folder.
4. Rename the extracted `EE4308` folder to `team##`.
5. Copy over the code from project 1's `move.cpp` to project 2, as well as the corresponding PID gains in `turtle.yaml`. This will not be graded, but necessary for project 2 to run.
6. Set `use_internal_odom` to `true` in `turtle.yaml`, so the turtle uses the ground truth to move.
7. Set all `verbose` to `false` in `turtle.yaml`.
8. Make the workspace using:

```
cd ~/team##
chmod +x *.sh
./clean_make.sh
```

Obviously, replace `team##` with the your team name, and `team##` only needs to be run once for the life time of `team##`.

4 Project Task

In this project you are required to move a drone (hector) between different waypoints:

1. Take off to a height of 2m from the starting position and maintain the height thereafter.
2. Fly to the **moving** turtle.
3. After reaching the turtle, fly to the turtle's final goal.
4. After reaching the turtle's final goal, fly to the hector's starting position.
5. Repeat the **cycle** by flying to the turtle, start and goal in that order.
6. When the turtle reaches its final goal, the hector must **complete the current cycle**, and then land.

Additionally:

- Roll and pitch angles of the hector are assumed to be negligible. Therefore, the z-axis of the hector's robot frame is always aligned with the z-axis of the world frame.
- The estimated pose (not the true pose) of the hector must come within 0.2m of the waypoints before moving to the next state.
- The hector must yaw at a constant rate of 0.5 rad/s in either direction while moving between waypoints, excluding take off and landing.
- The hector's maximum linear speed along the x and y plane must be 2m/s (not individual axes). The hector can strafe (move along y) in addition to moving forward or backwards.
- The hector's maximum speed along z is 0.5 m/s.

4.1 All Tasks and Expectations

All teams are expected to design code in the `ee4308_hector` package:

1. Design a **finite state machine** and **generate trajectories for pure pursuit** in `ee4308_hector/src/main.cpp`.
   a) All teams are expected to use a look-ahead distance or duration to generate targets from trajectories.
   b) 3-man teams must use at least straight lines for trajectories.
   c) 4-man teams must use at least the cubic spline trajectories, and predict the turtle's motion when generating these trajectories.
   d) All parameters must be justified.
2. Design a PID motion controller in `ee4308_hector/src/move.cpp`
   a) All teams must satisfy all motion requirements above.
   b) All teams must justify the gains used. Do so from at least the data collected while using the ground truth for the hector's pose.
3. Program an extended Kalman filter (EKF) to estimate the pose of the hector in `ee4308_hector/src/motion.cpp`.
   a) All teams must justify and collect data for the noise values used.
   b) 3-man teams must implement at least the GPS and Magnetometer corrections. Implementing the sonar is recommended.
   c) 4-man teams must implement the GPS, magnetometer, barometer and sonar sensors.
4. Use most of the parameters in `ee4308_hector/config/hector.yaml`.

All teams are advised to use the ground truth to complete `main.cpp` and `move.cpp` before using the estimated pose from `motion.cpp`. The latter can be configured to use ground truth via `hector.yaml`.

4.1.1 Implementation in main.cpp

Take note of the following variables that can be used in `main.cpp`:

Variable	Description
`height`	The height (m) to maintain when flying between different way-points. Defined in `hector.yaml`, but should be set to 2.
`look_ahead`	The look-ahead distance (m) or duration (s) to generate the target from trajectories. Defined in `hector.yaml`. With this, only the point at the look-ahead should be generated.
`close_enough`	The threshold distance (m) to determine when the estimated pose is sufficiently close to the way-points to change the state of the finite machine. Defined in `hector.yaml`, but should be set as 0.2.
`average_speed`	The average speed (m/s) used to generate the trajectories.
`state`	The current state of the finite machine. Uses enums to make programming easier.

The following variables are provided and should only be read from:

Variable	Description
`x`	The x coordinate (m) of the hector.
`y`	The y coordinate (m) of the hector.
`z`	The z coordinate (m) of the hector.
`a`	The yaw angle (rad) of the hector.
`vx`	The x velocity (m/s) of the hector.
`vy`	The y velocity (m/s) of the hector.
`vz`	The z velocity (m/s) of the hector.
`va`	The yaw velocity (rad/s) of the hector.
`turtle_x`	The x coordinate (m) of the turtle.
`turtle_y`	The y coordinate (m) of the turtle.
`goal_x`	The x coordinate (m) of the turtle's final goal.
`goal_y`	The y coordinate (m) of the turtle's final goal.
`initial_x`	The initial x coordinate (m) of the hector.
`initial_y`	The initial y coordinate (m) of the hector.
`initial_y`	The initial z coordinate (m) of the hector.

Additionally:

1. Publish the targets into the `target` topic.
2. The `rotate` topic can be used to tell `move.cpp` to start rotating at the constant yaw rate.
3. While optional, the `trajectory` topic should be used to display the generated trajectory in RViz.
4. Code the finite state machine in the main while loop.

4.1.2 Variables in move.cpp

Variable	Description
`Kp_lin`	The K_p gain for the motion controller along x and y. Defined in `hector.yaml`.
`Ki_lin`	The K_i gain for the motion controller along x and y. Defined in `hector.yaml`.
`Kd_lin`	The K_d gain for the motion controller along x and y. Defined in `hector.yaml`.
`Kp_z`	The K_p gain for the motion controller along z. Defined in `hector.yaml`.
`Ki_z`	The K_i gain for the motion controller along z. Defined in `hector.yaml`.
`Kd_z`	The K_d gain for the motion controller along z. Defined in `hector.yaml`.
`yaw_rate`	The yaw rate (rad/s) when the hector is flying between way-points. Defined in `hector.yaml` but should be 0.5.
`max_lin_vel`	The maximum linear speed (m/s) along x and y. Defined in `hector.yaml`, but should be set as 2.
`maz_z_vel`	The maximum speed (m/s) along z. Defined in `hector.yaml`, but should be set as 0.5.
`cmd_lin_vel_x`	The x velocity (m/s) of the hector in the robot frame.
`cmd_lin_vel_y`	The y velocity (m/s) of the hector in the robot frame.
`cmd_lin_vel_z`	The z velocity (m/s) of the hector in the robot frame.
`cmd_lin_vel_a`	The yaw velocity (rad/s) of hector in the robot frame.
`dt`	The duration (s) between iterations.

You may design other kinds of PID controllers that use other gains. Additionally, the `rotate` topic can be used to tell `move.cpp` to start rotating at the constant yaw rate.

5 Motion Estimation in motion.cpp

Adapted from Dr. Wang Fei's notes. 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 hector's position in the Gazebo world frame by incorporating many sensor measurements.

5.1 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 ϕ (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 $\hat{\mathbf{X}}_{k-1|k-1}$.

5.1.1 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 hector in the world frame, has

the following relation:

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

$$= f(x_{k-1|k-1}, \dot{x}_{k-1|k-1}, a_{x,k}) \tag{3}$$

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:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{4}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}a_{x,k}(\Delta t)^2 \\ \dot{x}_{k-1|k-1} + a_{x,k}\Delta t \end{bmatrix} \tag{5}$$

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 and world is simply:

$$a_{x,k} = \bar{a}_{x,k}\cos\psi - \bar{a}_{y,k}\sin\psi \tag{6}$$

where 180° and $\bar{a}_{y,k}$ are the IMU accelerations in the robot frame. A quick inspection with tele-operation and topics reveals that the IMU's frame is rotated 180 about the robot's z-axis, so:

$$\bar{a}_{x,k} = -u_{x,k} \tag{7}$$

$$\bar{a}_{y,k} = -u_{y,k} \tag{8}$$

where $u_{x,k}$ and $u_{y,k}$ refers to the x and y-accelerations measured by the IMU. So, we have:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{9}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}(\Delta t)^2(u_{y,k}\sin\psi - u_{x,k}\cos\psi) \\ \dot{x}_{k-1|k-1} + \Delta t(u_{y,k}\sin\psi - u_{x,k}\cos\psi) \end{bmatrix} \tag{10}$$

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.

5.1.2 States Along other Degrees-of-freedom

For linear kinematics on the y-axis:

$$\hat{\mathbf{X}}_{y,k|k-1} = \begin{bmatrix} y_{k|k-1} \\ \dot{y}_{k|k-1} \end{bmatrix} = \begin{bmatrix} y_{k-1|k-1} + \dot{y}_{k-1|k-1}\Delta t - \frac{1}{2}(\Delta t)^2(u_{y,k}\cos\psi + u_{x,k}\sin\psi) \\ \dot{y}_{k-1|k-1} - \Delta t(u_{y,k}\cos\psi + u_{x,k}\sin\psi) \end{bmatrix} \tag{11}$$

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

$$\hat{\mathbf{X}}_{z,k|k-1} = \begin{bmatrix} z_{k|k-1} \\ \dot{z}_{k|k-1} \end{bmatrix} = \begin{bmatrix} z_{k-1|k-1} + \dot{z}_{k-1|k-1}\Delta t + \frac{1}{2}(\Delta t)^2(u_{z,k} - G) \\ \dot{z}_{k-1|k-1} + \Delta t(u_{z,k} - G) \end{bmatrix} \tag{12}$$

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

$$\hat{\mathbf{X}}_{\psi,k|k-1} = \begin{bmatrix} \psi_{k|k-1} \\ \dot{\psi}_{k|k-1} \end{bmatrix} = \begin{bmatrix} \psi_{k-1|k-1} + \Delta t u_{\psi,k} \\ u_{\psi,k} \end{bmatrix} \tag{13}$$

5.2 EKF Prediction (all teams)

5.2.1 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 $\hat{\mathbf{X}}_{x,k|k-1}$:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{14}$$

$$\approx \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{15}$$

$$= \frac{\partial \mathbf{f}}{\partial \hat{\mathbf{X}}_{x,k-1|k-1}}\hat{\mathbf{X}}_{x,k-1|k-1} + \frac{\partial \mathbf{f}}{\partial \mathbf{U}_{x,k}}\mathbf{U}_{x,k} \tag{16}$$

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:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{17}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} \approx \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{18}$$

$$= \begin{bmatrix} \frac{\partial x_{k|k-1}}{\partial x_{k-1|k-1}} & \frac{\partial x_{k|k-1}}{\partial \dot{x}_{k-1|k-1}} \\ \frac{\partial \dot{x}_{k|k-1}}{\partial x_{k-1|k-1}} & \frac{\partial \dot{x}_{k|k-1}}{\partial \dot{x}_{k-1|k-1}} \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} \frac{\partial x_{k|k-1}}{\partial u_{x,k}} & \frac{\partial x_{k|k-1}}{\partial u_{y,k}} \\ \frac{\partial \dot{x}_{k|k-1}}{\partial u_{x,k}} & \frac{\partial \dot{x}_{k|k-1}}{\partial u_{y,k}} \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{19}$$

$$= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} -\frac{1}{2}(\Delta t)^2 \cos\psi & \frac{1}{2}(\Delta t)^2 \sin\psi \\ -\Delta t \cos\psi & \Delta t \sin\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{20}$$

which is just conveniently substituting Eq. (10) 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:

$$\mathbf{P}_{x,k|k-1} = \mathbf{F}_{x,k}\mathbf{P}_{x,k-1|k-1}\mathbf{F}_{x,k}^{\top} + \mathbf{W}_{x,k}\mathbf{Q}_x\mathbf{W}_{x,k}^{\top} \tag{21}$$

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:

$$\mathbf{Q}_x = \begin{bmatrix} \sigma^2_{\text{imu},x} & 0 \\ 0 & \sigma^2_{\text{imu},y} \end{bmatrix} \tag{22}$$

5.2.2 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 $\mathbf{P}$:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{f}(\hat{\mathbf{X}}_{x,k-1|k-1}, \mathbf{U}_{x,k}) \tag{23}$$

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} + \dot{x}_{k-1|k-1}\Delta t + \frac{1}{2}a_{x,k}(\Delta t)^2 \\ \dot{x}_{k-1|k-1} + a_{x,k}\Delta t \end{bmatrix} \tag{24}$$

$$= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1|k-1} \\ \dot{x}_{k-1|k-1} \end{bmatrix} + \begin{bmatrix} -\frac{1}{2}(\Delta t)^2\cos\psi & \frac{1}{2}(\Delta t)^2\sin\psi \\ -\Delta t\cos\psi & \Delta t\sin\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} \tag{25}$$

$$= \mathbf{F}_{x,k}\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\mathbf{U}_{x,k} \tag{26}$$

$$\mathbf{P}_{x,k|k-1} = \mathbf{F}_{x,k}\mathbf{P}_{x,k-1|k-1}\mathbf{F}_{x,k}^{\top} + \mathbf{W}_{x,k}\mathbf{Q}_x\mathbf{W}_{x,k}^{\top} \tag{27}$$

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

1. For y, refer to Eq. (11). Find $\mathbf{F}_{y,k}$, $\mathbf{U}_{y,k}$ and $\mathbf{W}_{y,k}$.
2. For z, refer to Eq. (12). $\mathbf{Q}_z = \sigma^2_{\text{imu},z}$ 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. (13). $\mathbf{Q}_\psi = \sigma^2_{\text{imu},\psi}$ and $\mathbf{W}_{\psi,k}$ is 2×1 in size. Find $\mathbf{F}_{\psi,k}$, $\mathbf{U}_{\psi,k}$ and $\mathbf{W}_{\psi,k}$.

5.2.3 Implementation

Use the following variables, which were correctly initialised for you. Remember to set `use_ground_truth` to `false` in `hector.yaml`.

Symbol	Variable	Description
$\hat{\mathbf{X}}_x$	`X`	2×1 state vector for x. `X(0)` is the x position, `X(1)` is the x velocity in world frame.
$\hat{\mathbf{X}}_y$	`Y`	2×1 state vector for y. Similar in form as x.
$\hat{\mathbf{X}}_z$	`Z`	2×1 state vector for z. Similar in form as x. Resizing by re-declaration is necessary if barometer is used. See Sec. 5.3.3.
$\hat{\mathbf{X}}_\psi$	`A`	2×1 state vector for ψ. Similar in form as x.
$\mathbf{P}_x$	`P_x`	2×2 state covariance matrix for x. `P_x(0,0)` is the variance of x. `P_x(1,1)` is the variance of $\dot{x}$. It is a symmetrical, positive semi-definite matrix.
$\mathbf{P}_y$	`P_y`	2×2 state covariance matrix for y. Similar to $\mathbf{P}_x$.
$\mathbf{P}_z$	`P_z`	2×2 state covariance matrix for z. Similar to $\mathbf{P}_x$. May be 3×3 if the barometer is used. See Sec. 5.3.3.
$\mathbf{P}_\psi$	`P_a`	2×2 state covariance matrix for ψ. Similar to $\mathbf{P}_x$.
u_x	`Ux`	Measured IMU x acceleration value in IMU frame.
u_y	`Uy`	Measured IMU y acceleration value in IMU frame.
u_z	`Uz`	Measured IMU z acceleration value in IMU frame.
u_a	`Ua`	Measured IMU ψ velocity value in IMU frame.
G	`G`	Acceleration due to gravity. Value is 9.8, as the Gazebo world files use that value. Do not modify.
Δt	`imu_dt`	Time duration between predictions, tied to how fast the IMU messages are being published, which is 40Hz. Do not modify.

It is **not possible to collect the IMU data and find the variance** in the IMU measurements as Gazebo generates IMU signals with inconsistent noises. This is due to faulty collision calculations in Gazebo, which changes the noise depending on the current computational load. Just use reasonably large values for these noises in `hector.yaml` to allow your EKF to work. You should only decide these values after incorporating the EKF corrections in the next section.

Symbol	Variable	Description
$\sigma^2_{\text{imu},x}$	qx	Variance of x acceleration measurements in IMU frame. Define in `hector.yaml`.
$\sigma^2_{\text{imu},y}$	qy	Variance of y acceleration measurements in IMU frame. Define in `hector.yaml`.
$\sigma^2_{\text{imu},z}$	qz	Variance of z acceleration measurements in IMU frame. Define in `hector.yaml`.
$\sigma^2_{\text{imu},\psi}$	qa	Variance of ψ velocity measurements in IMU frame. Define in `hector.yaml`.

The IMU generates signals at 40Hz. Every measurement is processed in the callback function `cbImu` in `motion.cpp`. Therefore, the **entire prediction stage** has to be **implemented within** `cbImu` instead of the while loop in the function `main`.

5.2.4 Intermediate Task

1. Find the Jacobian matrices in Sec. 5.2.2 for y, z and ψ degrees-of-freedom.

2. Implement EKF predictions for x, y, z, ψ and their velocities in separate state vectors within the `cbImu` callback function in `motion.cpp`. Treat the IMU measurements as input.

5.3 EKF Correction (all teams)

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

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^\top + \mathbf{V}_k\mathbf{R}_k\mathbf{V}_k)^{-1} \tag{28}$$

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} + \mathbf{K}_k\left[\mathbf{Y}_k - \mathbf{h}(\hat{\mathbf{X}}_{k|k-1})\right] \tag{29}$$

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

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 known in a frame that is different from the measurement's frame, for example if the measurement is transformed from the raw sensor measurements. For this project, it is just one or an identity matrix, but useful to keep this in mind.

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

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} \tag{31}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} \tag{32}$$

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 them.

5.3.1 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:

$$e^2 = 1 - \frac{b^2}{a^2} \tag{33}$$

$$N(\varphi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}} \tag{34}$$

$$\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} = \begin{bmatrix} (N(\varphi) + h) \cos\varphi \cos\lambda \\ (N(\varphi) + h) \cos\varphi \sin\lambda \\ \left(\frac{b^2}{a^2} N(\varphi) + h\right) \cos\varphi \sin\varphi \end{bmatrix} \tag{35}$$

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.

Take good care to make sure that the λ and φ are in radians before using them. You should investigate https://docs.ros.org/en/api/sensor_msgs/html/msg/NavSatFix.html to see if they are in degrees or radians.

Note that, the above equations have to be run at least once to calculate the initial ECEF coordinates ($x_{e,0}$, $y_{e,0}$ and $z_{e,0}$). Then, we can find the local NED coordinates (x_n, y_n, z_n) by using a rotation:

$$\mathbf{R}_{e/n} = \begin{bmatrix} -\sin\varphi\cos\lambda & -\sin\lambda & -\cos\varphi\cos\lambda \\ -\sin\varphi\sin\lambda & -\cos\lambda & -\cos\varphi\sin\lambda \\ \cos\varphi & 0 & -\sin\lambda \end{bmatrix} \tag{36}$$

$$\begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = \mathbf{R}_{e/n} \left(\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} - \begin{bmatrix} x_{e,0} \\ y_{e,0} \\ z_{e,0} \end{bmatrix} \right) \tag{37}$$

After that, transform to the world frame in Gazebo using:

$$\mathbf{R}_{m/n} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{38}$$

$$\begin{bmatrix} x_{\text{gps}} \\ y_{\text{gps}} \\ z_{\text{gps}} \end{bmatrix} = \mathbf{R}_{m/n} \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \tag{39}$$

where (x_0, y_0, z_0) are the initial coordinates in the world frame. In Gazebo, the world frame's x-axis points north, y points west and z points up. You should verify that the rotation is correct.

To use the EKF correction, we make things easier, so:

$$\mathbf{Y}_{\text{gps},x,k} = x_{\text{gps}} \tag{40}$$

$$\mathbf{h}(\hat{\mathbf{X}}_{x,k|k-1}) = x_{k|k-1} \tag{41}$$

$$\mathbf{H}_{\text{gps},x,k} = \frac{\partial \mathbf{Y}_{\text{gps},x,k}}{\partial \hat{\mathbf{X}}_{x,k|k-1}} = \begin{bmatrix} \frac{\partial x_{\text{gps}}}{\partial x_{k|k-1}} & \frac{\partial x_{\text{gps}}}{\partial \dot{x}k|k-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{42}$$

$$\mathbf{V}_{\text{gps},x,k} = 1 \tag{43}$$

$$\mathbf{R}_{\text{gps},x,k} = \sigma^2_{\text{gps},x} \tag{44}$$

Where $\sigma^2_{\text{gps},x}$ is the variance of x_{gps}. Do note that the measurements contain drifts (biases), which may affect the calculation of your variance from the gathered data. You may refer to the web link above to get an idea of the variance, but you should collect data to justify. 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. (28, 29, 30) to find the updated x , y and z states. The GPS measurements do not correct ψ.

$\mathbf{H}$ can be calculated this way because the model is gaussian, so $x_{\text{gps}} = x_{k|k-1} + \mathcal{N}(0, \sigma^2_{\text{gps},x})$.

Implement the GPS correction in the function `cbGPS` of `motion.cpp`:

Symbol	Variable	Description
$\begin{bmatrix} x_{\text{gps}} & y_{\text{gps}} & z_{\text{gps}} \end{bmatrix}^\top$	`GPS`	3×1 vector containing the GPS measurements. Write to it to display into terminal.
$\begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^\top$	`initial_pos`	3×1 vector of the hector's initial position in the world frame. Automatically written. Read from it for your calculations.
$\pi/180$	`DEG2RAD`	Multiply with this value to convert degrees to radians.
a	`RAD_EQUATOR`	Equatorial Radius.
b	`RAD_POLAR`	Polar Radius
$\sigma^2_{\text{gps},x}$	`r_gps_x`	Variance of x GPS measurements in world frame. Define in `hector.yaml`.
$\sigma^2_{\text{gps},y}$	`r_gps_y`	Variance of y GPS measurements in world frame. Define in `hector.yaml`.
$\sigma^2_{\text{gps},z}$	`r_gps_z`	Variance of z GPS measurements in world frame. Define in `hector.yaml`.

Find the variances by reading the messages from the topic, and justifying why this could be done by using the web link above. The variances are constant for all simulations. Please define the values in `hector.yaml` and use the variables in the code.

5.3.2 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:

$$\mathbf{Y}_{\text{mgn},\psi,k} = \psi_{\text{mgn}} = f_{\text{mgn}}(x_{\text{mgn}}, y_{\text{mgn}}) \tag{45}$$

$$\mathbf{h}(\hat{\mathbf{X}}_{\psi,k|k-1}) = \psi_{k|k-1} \tag{46}$$

$$\mathbf{H}_{\text{mgn},\psi,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{47}$$

$$\mathbf{V}_{\text{mgn},\psi,k} = 1 \tag{48}$$

$$\mathbf{R}_{\text{mgn},\psi,k} = \sigma^2_{\text{mgn},\psi} \tag{49}$$

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. $\mathbf{H}$ can be calculated this way because $\psi_{\text{mgn}} = \psi_{k|k-1} + \mathcal{N}(0, \sigma^2_{\text{mgn},\psi})$.

Program the implementation in the function `cbMagnet` of `motion.cpp`:

Symbol	Variable	Description
ψ_{mgn}	`a_mgn`	The yaw angle measurement in the world frame, derived from the magnetic force vector in the robot frame. This is used for displaying to the terminal.
$\sigma^2_{\text{mgn},\psi}$	`r_mgn_a`	Variance of ψ_{mgn} calculations in world frame. Define in `hector.yaml`.

To determine $\sigma^2_{\text{mgn},\psi}$, calculate ψ_{mgn} and find the variance of these calculations by using **at least 100 calculations** of ψ_{mgn} **from at least 100 messages**.

5.3.3 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:

$$\mathbf{Y}_{\text{bar},z,k} = z_{\text{bar}} \tag{50}$$
$$\mathbf{h}(\hat{\mathbf{X}}_{z,k|k-1}) = z_{k|k-1} \tag{51}$$
$$\mathbf{H}_{\text{bar},z,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{52}$$
$$\mathbf{V}_{\text{bar},z,k} = 1 \tag{53}$$
$$\mathbf{R}_{\text{bar},z,k} = \sigma^2_{\text{bar},z} \tag{54}$$

where $\sigma^2_{\text{bar},z}$ is the variance of z_{bar}. To be able to use this correctly, one needs to account for the high bias of the measurement. We can do this by estimating the bias as a new state variable by augmenting the original state variable:

$$\hat{\bar{\mathbf{X}}}_{z,k|k} = \begin{bmatrix} \hat{\mathbf{X}}_{z,k|k} \\ b_{\text{bar},k|k} \end{bmatrix} = \begin{bmatrix} z_{k|k} \\ \dot{z}_{k|k} \\ b_{\text{bar},k|k} \end{bmatrix} \tag{55}$$
$$\tag{56}$$

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. We can modify the correction as:

$$\mathbf{Y}_{\text{bar},z,k} = z_{\text{bar}} - b_{\text{bar},k|k} \tag{57}$$
$$\mathbf{h}(\hat{\mathbf{X}}_{z,k|k-1}) = z_{k|k-1} \tag{58}$$
$$\mathbf{V}_{\text{bar},z,k} = 1 \tag{59}$$
$$\mathbf{R}_{\text{bar},z,k} = \sigma^2_{\text{bar},z} \tag{60}$$

where $\mathbf{H}_{\text{bar},z,k}$ is to be determined. Note that $z_{\text{bar}} = \cdots + \mathcal{N}(0, \sigma^2_{\text{bar},z})$.

Program the implementation in the function `cbBaro` of `motion.cpp`:

Symbol	Variable	Description
z_{bar}	`z_bar`	The z measurement of the barometer in the world frame. Used and displayed in terminal.
$\sigma^2_{\text{bar},z}$	`r_bar_z`	Variance of z_{bar} calculations. Define in `hector.yaml`.

To determine $\sigma^2_{\text{bar},z}$, calculate z_{bar} and find the variance of these calculations by using **at least 100 calculations** of z_{bar} **from at least 100 messages**.

5.3.4 Sonar (4-man team)

While marked for 4-man team, it is strongly encouraged for all teams to use this as it is pretty easy to implement and offers good accuracy in z. The sonar uses ultra-sound to find the distance to the ground. It has a maximum range of 3m and is affected by the heights of ground obstacles. Let z_{snr} be the sonar measurement:

$$\mathbf{Y}_{\text{snr},z,k} = z_{\text{snr}} \tag{61}$$

$$\mathbf{h}(\hat{\mathbf{X}}_{z,k|k-1}) = z_{k|k-1} \tag{62}$$

$$\mathbf{H}_{\text{snr},z,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{63}$$

$$\mathbf{V}_{\text{snr},z,k} = 1 \tag{64}$$

$$\mathbf{R}_{\text{snr},z,k} = \sigma^2_{\text{snr},z} \tag{65}$$

where $\sigma^2_{\text{snr},z}$ is the variance of z_{snr}. Again, keep in mind that ground obstacles directly beneath the hector will affect the reading of z_{snr}. You may wish to implement additional features to account for this problem.

Program the implementation in the function `cbSonar` of `motion.cpp`:

Symbol	Variable	Description
z_{snr}	`z_snr`	The z measurement of the sonar in the world frame. Used and displayed in terminal.
$\sigma^2_{\text{snr},z}$	`r_snr_z`	Variance of z_{snr} calculations. Define in `hector.yaml`.

To determine $\sigma^2_{\text{snr},z}$, calculate z_{snr} and find the variance of these calculations by using **at least 100 calculations** of z_{snr} **from at least 100 messages**.

5.3.5 Intermediate Task

1. 3-man teams: Implement EKF correction using at least the GPS and magnetometer. You are encouraged to use the sonar.
2. 4-man teams: Implement EKF correction using all sensors – GPS, magnetometer, barometer and sonar.
3. Identify the variances using the methods suggested above.

6 Programming Tips

6.1 cv::Matx

`cv::Matx` helps you to implement matrix arithmetic in the code, particularly for the EKF and splines. The documentation can be found here in https://docs.opencv.org/3.4/de/de1/classcv_1_1Matx.html. Below lists some tips that you may find helpful:

1. `cv::Matx21d M` or `cv::Matx<2,1,double> M` creates a 2×1 vector `M`.
2. Initialise it using `M = {8, 9}`, so it stores 8 on the first element, and 9 on the second.
3. Access the vector using `M(0)`, which accesses the element on the first row, etc.
4. `cv::Matx22d M` or `cv::Matx<2,2,double> M` creates a 2×2 matrix `M`.
5. Since Cpp ignores line breaks, you can initialise a matrix using:

```
M = {
      1, 2,
      3, 4
    };
```

   or

```
M = {1,2,3,4};
```

6. `M(0,1)` accesses the element on the first row and second column. `M(1,1)` access the element on the second row and column.

7. Matrix multiplications can be done using `*` like in MATLAB. Similarly for addition `+` and substraction `-`.
8. `M.t()` transposes the matrix.
9. `M.inv()` inverses the matrix, provided that it is square.
10. If `M` is 1×1, you cannot use `M.inv()`.
11. Suppose you want to multiply a matrix `N` with the inverse of a 1×1 `M`, then you need to use: `N / M(0)`, where `M(0)` is a normal `double` number.
12. You can print any matrix only using `ROS_INFO_STREAM` or `std::cout`.