Localizing Legged Robots Using Sensor Fusion

Guzzi Rémi, Waeles-Devaux Adrien
ROB4, Polytech Nice-Sophia

03/02/25 - 08/04/25

Contents

1 Project Overview

1.1 Objectives

The goal of this project is to create a lab session for future robotics students to introduce them to the importance of sensor fusion in mobile robotics. Through the use of the GO2 EDU, a quadruped robot from Unitree Robotics to:

- Get the sensor data (IMU, foot forces, etc.).
- Apply filters (complementary, Kalman, etc.) to estimate position/orientation/...
- Use these data to control the robot
- Optional: SLAM

1.2 Purpose

This lab has an educational purpose: allow the students to concretely manipulate the concepts of sensor fusion and control in autonomous robotics, in a real environment (with noise, delays and uncertainties)

2 Interfacing with the Unitree Go2 Robot

2.1 Overview

We can interface with the GO2 robot programmatically by using the `Unitree_sdk2`, which is a dedicated SDK (Software Development Kit) provided by Unitree Robotics. It offers the ability to design real-time control and feedback.

2.2 SDK Architecture

The `Unitree_sdk2` is built around a publish-subscribe communication model by using DDS (Data Distribution Service), which is a standard middleware for real-time distributed systems.
Used DDS Libraries :

- `ddsc`
- `ddscxx`

2.3 Communication Layer

The robot and the host computer communicate via Ethernet using DDS, ensuring reliable data transmission and timing guarantees, which is critical for robotics.

2.4 Communication Protocols

Built around two key data structures:

- **Low Commands**: For sending motor-level commands (torque, position, etc.)
- **Low State**: For receiving feedback (from joints, IMU, etc.)

These are sent at high frequency.

2.5 Build System and Integration

The SDK uses `cmake` as its build system, with:

- C++ (but also compatible with Python)
- Libraries
- Example compilations

This design allows easy integration into existing C++ projects.

2.6 Hardware Used

- GO2 EDU from Unitree Robotics
- Computer with Ubuntu
- Phone with Unitree Go (app)
- Lidar
- `Unitree_sdk2` (SDK directly provided by Unitree Robotics)

2.7 Network Configuration

- Disable `sport_mode` in the Unitree Go app

2.8 Software Setup

```
sudo apt update
sudo apt upgrade
sudo apt install cmake g++ ...
unminimize
```

2.9 Getting the SDK

```
git clone https://github.com/unitreerobotics/unitree_sdk2
cd unitree_sdk2
mkdir build
cd build
cmake ..
make
```

3 Complementary Filter: Principles and Application

The complementary filter combines data from an IMU's accelerometer and gyroscope to estimate orientation (pitch, roll, yaw). Position estimation isn't feasible with an IMU alone due to drift and requires additional sensors.

3.1 Sensor Characteristics

- **Accelerometer**: Measures linear acceleration (motion) and gravitational acceleration along three axes (x, y, z). When the robot is static or moving at constant velocity, it measures only gravity, enabling roll (ϕ) and pitch (θ) estimation via the direction of the gravity vector. It cannot measure yaw (rotation around the vertical axis) and is affected by high-frequency noise (vibrations, ...).
- **Gyroscope**: Measures angular velocity ($\omega_x, \omega_y, \omega_z$) around three axes, which can be integrated to estimate orientation changes (pitch, roll, yaw). It's highly accurate over short periods but suffers from drift over time due to integration errors and bias, leading to low-frequency errors.

3.2 Purpose of the Complementary Filter

The filter takes advantage of the strengths of both sensors to improve orientation accuracy:

- The accelerometer provides long-term stability for roll and pitch using gravity to correct drift.
- The gyroscope offers short-term accuracy for all angles.
- Accelerometer noise (high frequency) and gyroscope drift (low frequency) are complementary, making the filter effective.

3.3 Accelerometer-Based Orientation Estimation

When the robot isn't accelerating (only gravity acts), the accelerometer's measurements are:

$$
\begin{aligned}
a_x &= g \cdot \sin(\theta) \\
a_y &= -g \cdot \cos(\theta) \cdot \sin(\phi) \\
a_z &= -g \cdot \cos(\theta) \cdot \cos(\phi) \\
\phi_{acc} &= \tan^{-1}\left(\frac{a_y}{a_z}\right) \\
\theta_{acc} &= \sin^{-1}\left(\frac{a_x}{g}\right)
\end{aligned}
$$

These equations isolate roll and pitch from the gravity vector but fail during dynamic motion (when linear acceleration is present). A low-pass filter is also applied to reduce high-frequency noise in accelerometer data.

3.4 Gyroscope-Based Orientation Estimation

The gyroscope measures angular velocities $(\omega_x, \omega_y, \omega_z)$ in the body frame. These are integrated to compute angle changes:

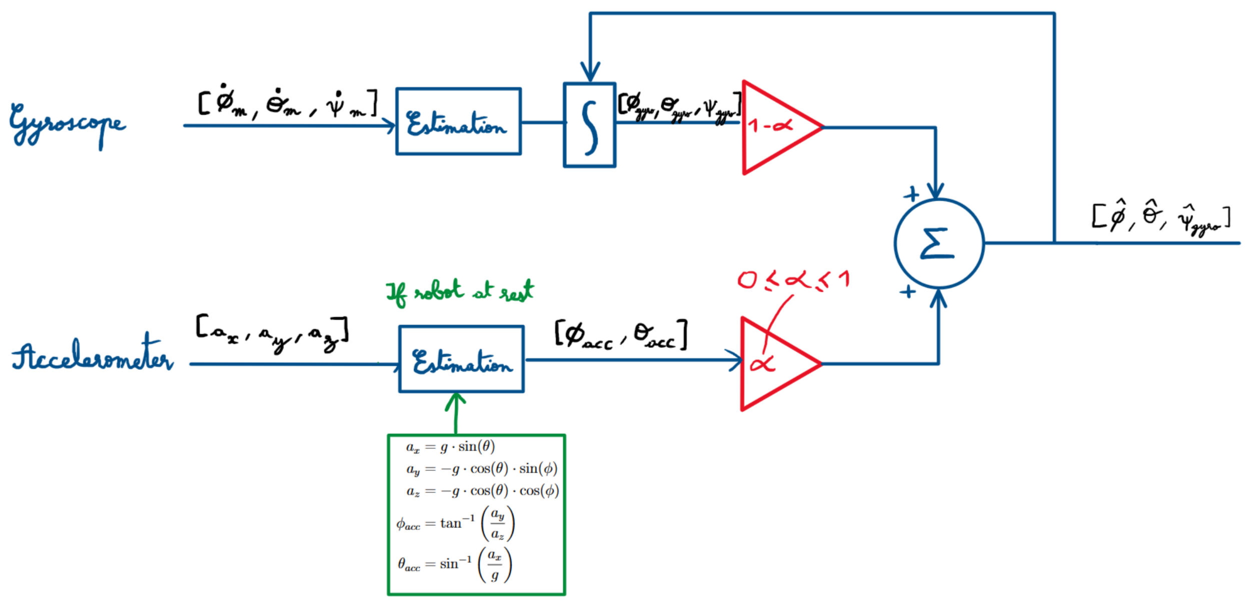
$$\Delta\theta = \omega_y \cdot \Delta t, \quad \Delta\phi = \omega_x \cdot \Delta t, \quad \Delta\psi = \omega_z \cdot \Delta t$$

These changes are added to the previous orientation estimate.

3.5 The Complementary Filter

The filter combines both estimates using a weighting factor α (0 to 1):

$$
\begin{aligned}
\theta &= (1-\alpha) \cdot (\theta_{prev} + \omega_y \cdot \Delta t) + \alpha \cdot \theta_{acc} \\
\phi &= (1-\alpha) \cdot (\phi_{prev} + \omega_x \cdot \Delta t) + \alpha \cdot \phi_{acc}
\end{aligned}
$$


Gyroscope
$[\dot{\phi}_m, \dot{\theta}_m, \dot{\psi}_m]$
Estimation
$\int$
$[\phi_{gyro}, \theta_{gyro}, \psi_{gyro}]$
1-α
+
Σ
$[\hat{\phi}, \hat{\theta}, \hat{\psi}_{gyro}]$
If robot at rest
$0 \leq \alpha \leq 1$
Accelerometer
$[a_x, a_y, a_z]$
Estimation
$[\phi_{acc}, \theta_{acc}]$
α
+
$a_x = g \cdot \sin(\theta)$
$a_y = -g \cdot \cos(\theta) \cdot \sin(\phi)$
$a_z = -g \cdot \cos(\theta) \cdot \cos(\phi)$
$\phi_{acc} = \tan^{-1}\left(\frac{a_y}{a_z}\right)$
$\theta_{acc} = \sin^{-1}\left(\frac{a_x}{g}\right)$

- Generally, $\alpha \approx 0$ to prioritize the gyroscope's short-term accuracy, as it is typically more reliable.
- The accelerometer is however useful to correct long-term drift in roll and pitch.
- Yaw (ψ) relies solely on the gyroscope, as the accelerometer can't measure it. A magnetometer could be added to correct yaw drift.

3.6 Limitations

- Fails during dynamic motion, as accelerometer data includes linear acceleration.
- Yaw estimation drifts over time without a magnetometer.
- Assumes constant α, which may not adapt to varying sensor noise.
- Cannot estimate position due to drift in double integration of acceleration.

4 Kalman Filter: Functionalities and Limitations

The Kalman filter is a method to estimate the state of a system, such as a legged robot, by combining noisy sensor data (from IMU, encoders, ...) with a predictive model of the system's behavior.

4.1 How It Works

1. **Predict Step**: The filter predicts the robot's next state (in our case, its position and orientation) using a model of the robot's dynamics. It also estimates the uncertainty of this prediction due to model inaccuracies or external disturbances.

2. **Update Step**: The filter incorporates new sensor measurements and compares them to the prediction. Since sensors are noisy, it evaluates the reliability of the sensor data versus the prediction, fusing them to refine state estimate and reduce uncertainty.

4.2 Limitations with Legged Robots

The standard Kalman filter assumes linear dynamics and Gaussian noise, but this version is less effective for legged robots due to:

- **Non-Linear Movements**: Legged robots exhibit non-linear motion (jumping, turning). The standard Kalman filter struggles with non-linear systems, leading to inaccurate predictions.

- **Dependent States**: The robot's states (position, orientation, and velocity) are highly interdependent, which the standard Kalman filter does not handle well.

- **Non-Gaussian Noise**: Vibrations and terrain variations cause noise that isn't always Gaussian, violating the standard Kalman filter's assumptions.

4.3 EKF vs UKF

For legged robots, an Extended Kalman Filter (EKF) or Unscented Kalman Filter (UKF) is typically used to handle non-linear dynamics. We chose the UKF over the EKF, despite the EKF being more computationally efficient, because the UKF offers the following advantages, which are critical for legged robots:

- It better handles strong non-linear motions, common with legged robots.

- It provides more accurate uncertainty estimates by preserving non-linear distributions (further explained in the next chapter).

- It avoids complex Jacobian calculations.

- It's more robust to discontinuities (foot impacts).

5 Unscented Kalman Filter: An Enhanced Approach

To address non-linear challenges, the Unscented Kalman Filter introduces a step that generates a sample of points (called sigma points) from the Gaussian distribution, which represent the mean and covariance of the state. These sigma points are then passed directly through the non-linear system equations to predict the next state. This approach avoids the need to linearize

the equations, which would be necessary if working directly with the Gaussian distribution in a standard Kalman Filter.
Its steps are the following :

1. **Sampling States with Sigma Points**
   Generates a set of sigma points from the current state estimate, representing possible states in a Gaussian distribution. These points are passed through the robot's non-linear dynamics to predict the next state.
2. **Prediction Using IMU Measurements**
   Uses IMU data (acceleration and angular velocity, with gravity removed) to estimate changes in the robot's velocity, orientation, and position over time through integration.
3. **Correction with Leg Odometry Feedback**
   Calculates the robot's foot positions using forward kinematics, applied when feet contact the ground. Adjusts the predicted state by comparing IMU-based predictions with leg odometry measurements.

6 Challenges Encountered and Implemented Solutions

Throughout the development of this project, several technical and conceptual challenges were encountered. These difficulties were valuable learning opportunities and allowed us to better understand both the practical limitations of robotics systems and the complexity of working with real-world sensor data. On top of that, we also had to work within a tight two-month deadline, which made time management and prioritization essential.

6.1 Network Configuration and Communication

Challenge: One of the initial challenges was establishing a reliable network connection between the robot and the host computer. The Go2 EDU requires a specific setup in which the robot is connected via Ethernet while the PC maintains its Wi-Fi connection for internet access. Additionally, interface names were inconsistent across different systems, requiring careful inspection with `ifconfig`.
Solution: We created a shell script to automate the configuration at boot time. Additionally, we disabled Sport Mode on the Unitree Go app to enable development functionalities.

6.2 Setting Up the Development Environment

Challenge: The robot's Ubuntu environment was originally in a minimized state, which blocked the installation of many essential packages.
Solution: We used `unminimize` after establishing an SSH connection (`ssh -X unitree@192.168.123.18`) to enable package management and graphical dependencies. Then, we performed a full update (`sudo apt update/upgrade`, . . .) to prepare the environment before compiling the SDK.

6.3 Understanding and Using the Unitree SDK2

Challenge: The `unitree_sdk2` had approachable documentation, but obtaining precise materials required navigating extensively through the SDK to understand its full working process and to access everything needed.
Solution: We analyzed the example codes provided in the SDK and explored the class definitions. We managed to retrieve IMU data, encoder values, foot forces, . . . by testing and printing the results as we found them.

6.4 Sensor Fusion Implementation

Challenge: Implementing sensor fusion in a real environment had limitations with the raw sensor data:

- The accelerometer was noisy, so the results were never fully accurate.
- The gyroscope drifted over time.

Combining them in a complementary filter required a tuning process for the alpha parameter and ensuring that the data were synchronized when fused.
Solution: We implemented a complementary filter, blending the gyroscope and accelerometer data. We also introduced an error calculation phase at the beginning of the robot's initialization during program execution. This involved taking several error samples from the data, calculating the average error, and subtracting it later.

6.5 Lidar Instability

Challenge: Although the lidar was essential for obstacle detection, it was sometimes unreliable. At times, it returned unexpected values or no signals at all, requiring frequent resets. Its instability made it difficult to rely on, especially for tasks dependent on it, like SLAM.
Solution: In the obstacle avoidance code, we set small values for the object distance range to mitigate the impact of erroneous data and prevent unnecessary robot movements. Another potential solution would have been to implement a counter to verify that the handled value was not a lidar error signal.

7 Lab Session Proposal: Synthesizing All Covered Concepts

This section outlines a proposed lab session designed for future students in robotics. It aims to bring together all the key aspects explored in this project (sensor data acquisition, filtering, state estimation, and control) and to demonstrate the practical impact of these techniques in a real-world robotic application.

7.1 Educational Objectives

Students participating in this lab session will:

- Understand the challenges of using raw sensor data (noise, drift, delay).
- Retrieve and analyze data from various sensors on the Unitree Go2 robot.
- Implement and compare sensor fusion algorithms (complementary, Kalman).
- Control the robot using filtered data and evaluate the system's performance.
- Understand the benefits and limitations of each approach.

7.2 Getting the Sensor Data

Students will start by collecting sensor data from the Go2 robot using the `unitree_sdk2`. The relevant sensors are:

- **IMU** (Inertial Measurement Unit): Accelerometer, gyroscope.
- **Foot Sensor**: Force detection (ground contact force measurement).
- **Encoders**: On each leg articulation (joint position).
- **Lidar**: For surrounding object distances.

Code to Get IMU Data

In the script file, include:

```
#include <unitree/idl/go2/SportModeState_.hpp>
#define TOPIC_HIGHSTATE "rt/sportmodestate"
```

Then, in a robot controller class:

- To get gyroscope values: `state.imu_state().gyroscope()[i]` (with $0 \leq i \leq 2$, in roll-pitch-yaw order).
- To get accelerometer values: `state.imu_state().accelerometer()[i]` (with $0 \leq i \leq 2$, in roll-pitch-yaw order).

7.3 Sensor Fusion and Filtering

Once they have the data, students will implement filtering techniques to improve data quality and estimate robot orientation/position.

7.3.1 Complementary Filter

Combining gyroscope (fast but drifts) and accelerometer (stable but noisy) to estimate orientation/position.
Example of an orientation updating method, called in a handler:

```
void updateOrientation(double gr, double gp, double a_roll,
  double a_pitch) {
    double roll_gyro  = f_roll + gr * dt;
    double pitch_gyro = f_pitch + gp * dt;
    double roll_acc   = atan2(a_pitch, sqrt(a_roll * a_roll + g *
       g)) * 180.0 / M_PI;
    double pitch_acc  = atan2(-a_roll, g) * 180.0 / M_PI;
    f_roll  = alpha_filter * roll_gyro + (1 - alpha_filter) *
      roll_acc;
    f_pitch = alpha_filter * pitch_gyro + (1 - alpha_filter) *
      pitch_acc;
    // For yaw, rely directly on the IMU
    f_yaw = state.imu_state().rpy()[2];
}
```

This returns filtered values in the f_roll, f_pitch, and f_yaw attributes. Yaw cannot be found with the complementary filter. Values are calculated from the gyroscope and accelerometer and tuned with an alpha_filter factor chosen by the coder.
Example of a translation updating method, called in a handler:

```
void updateTranslation(double vx, double vy, double vz,
                      double ax, double ay, double az) {
    double x_vel = f_x + vx * dt;
    double y_vel = f_y + vy * dt;
    double z_vel = f_z + vz * dt;
    double x_acc = f_x + vx * dt + 0.5 * ax * dt * dt;
    double y_acc = f_y + vy * dt + 0.5 * ay * dt * dt;
    double z_acc = f_z + vz * dt + 0.5 * az * dt * dt;
    f_x = alpha_pos * x_vel + (1 - alpha_pos) * x_acc;
    f_y = alpha_pos * y_vel + (1 - alpha_pos) * y_acc;
    f_z = alpha_pos * z_vel + (1 - alpha_pos) * z_acc;
}
```

This returns filtered values in the f_x, f_y, and f_z attributes, calculated from velocity (derived from the accelerometer) and acceleration (direct accelerometer values), using a parameter alpha_pos that defines the weighting of each value for the final position states.

7.3.2 Unscented Kalman Filter

First, we need to collect the sensors data, that will be used by our UKF, from Unitree Go2 with unitree_sdk2:

- **IMU**: Accelerometer, gyroscope.
- **Foot Sensor**: Contact force.
- **Encoders**: Joint positions.

To access real-time sensor readings on the Unitree Go2 robot, you should include the necessary headers and define the topic for low-state data:

```
#include <unitree/idl/go2/LowState_.hpp>
#define TOPIC_LOWSTATE "rt/lowstate"
```

- Gyroscope: `state.imu_state().gyroscope()[i]` $(0 \leq i \leq 2)$.
- Acceleration: `state.imu_state().accelerometer()[i]` $(0 \leq i \leq 2)$.
- Joints: `state.motor_state()[i].q()` $(0 \leq i \leq 11)$.

Let's now fuse IMU (accelerometer and gyroscope) and leg odometry (from encoders) to estimate robot's state, handling non-linear dynamics.

Implementation of Predict and Update steps

Define `predict` and `update` methods in your UKF class to manage the state vector `[x, y, z, roll, pitch, yaw, vx, vy, vz]`.

The `predict` method uses IMU data (accelerometer and gyroscope) to estimate the next state. The position (`x, y, z`) is updated by integrating velocities (`vx, vy, vz`) over the time step `DT`, the orientation (`roll, pitch, yaw`) is updated using gyroscope data, and the velocities are updated using accelerometer data.

```
void predict(const Vec3& accel, const Vec3& gyro) {
    state[0] += state[6] * DT;
    state[1] += state[7] * DT;
    state[2] += state[8] * DT;
    state[3] += gyro.x * DT;
    state[4] += gyro.y * DT;
    state[5] += gyro.z * DT;
    state[6] += accel.x * DT;
    state[7] += accel.y * DT;
    state[8] += accel.z * DT;
}
```

The `update` method corrects this estimate using leg odometry and IMU orientation measurements. It creates a measurement vector `z` from odometry and orientation, computes the predicted measurement `h` from the state, calculates the innovation term `y` (difference between `z` and `h`), to apply a correction to the state.

```
void update(const Vec3& odometry, const Vec3& orientation) {
    std::vector<double> z = {odometry.x, odometry.y, odometry.z,
       orientation.x, orientation.y, orientation.z};
    std::vector<double> h = {state[6], state[7], state[8], state
       [3], state[4], state[5]};
    std::vector<double> y(6);
    for (int i = 0; i < 6; i++) y[i] = z[i] - h[i];
    for (int i = 0; i < 9; i++) state[i] += 0.1 * y[i % 6];
}
```

Our `state` is filtered by using IMU for prediction and leg odometry for correction. Tune noise covariances (`Q`, `R`) will be used to balance trust in the model versus sensor data.

Implementation of Legs Odometry

Let's now implement a function to compute the robot's velocity in the world frame using joint angles and forward kinematics.

For each leg, we need to calculate foot velocities from current and previous joint angles, detect stance phases (foot contact) to ensure odometry is computed only when the foot is on the ground, average the body velocity, and transform it to the world frame using the robot's orientation:

Here is the code to be completed by the students :

```
Vec3 compute_odometry(const std::vector<double>& joint_data,
   const std::vector<double>& prev_joint_data,
                        double dt, const Vec3& orientation) {
    if (prev_joint_data.empty()) return Vec3();

    Vec3 body_vel;
    int stance_count = 0;

    for (int leg = 0; leg < 4; leg++) {
        // Extract joint angles for the current leg from both
           time steps
        std::vector<double> current_leg_joints = ...;
        std::vector<double> previous_leg_joints = ...;

        // Compute foot velocity using forward kinematics and
           time delta
        Vec3 foot_vel = (... - ...) / dt;

        // Check if the foot is in stance phase (=low vertical
           velocity)
        if (foot_is_in_stance(foot_vel)) {
            // Accumulate velocity contribution
            body_vel = ...;
            stance_count++;
        }
    }

    // Average the velocity + Rotate from local to global frame
    return stance_count == 0 ? Vec3() : rotate_to_global_frame(
       body_vel / stance_count, orientation);
}
```

7.4 Experimental Scenario: Filter Comparison Through Position Recovery

To validate the filters, students will test a position recovery scenario:

1. **Obstacle Avoidance**: The robot starts at a certain position but moves away to avoid obstacles.
2. **Pause**: The robot stops and waits until the obstacle is cleared.
3. **Return**: The robot attempts to return to its initial position using estimated odometry data.

This test is performed multiple times:

- Once using raw, unfiltered data.
- Once using filtered data (complementary/Kalman filters).

The hypothesis is that filtering sensor data will allow the robot to better estimate its trajectory and return more accurately to its initial position.

7.4.1 Implementation Details

- **Obstacle Detection**: The robot uses its lidar to detect objects within a predefined distance.
- **Odometry**: Position is estimated using IMU data and/or encoder readings (depending on the filter). Filters reduce the impact of drift, noise, etc.
- **Position Control**: The robot saves its initial position and attempts to return when displaced.
- **Comparison Metric**: The final position error (distance between actual and expected position) is compared for filtered and unfiltered executions.

7.5 Robot Control with Filtered Data

Students will send control commands to the robot based on state estimation:

- Use filtered sensor data to estimate position.
- Implement a return-to-origin movement.

- Evaluate control accuracy and smoothness.

Example Control Logic: If the estimated position error exceeds a certain limit, command the robot to move back, rotate, etc.

7.6 Comparison and Analysis

The effectiveness of each approach is evaluated using:

- **Final Position Error**: Distance from expected vs. actual position.
- **Repeatability**: Running the scenario multiple times with each filter.
- **Quality Stability**: Robot's smoothness and response speed.

7.7 Suggested Workflow for the Lab

1. **Initialization**: Set up the robot, ensuring network and SDK are correctly configured.
2. **Data Acquisition**: Write basic code to retrieve IMU, encoder, and lidar data.
3. **Apply Filters**: Implement complementary filter, Kalman filter, etc.
4. **Execute Scenarios**: Run the obstacle avoidance and position recovery task.
5. **Compare Results**: Record final position error with and without filters.
6. **Analysis**: Discuss and submit findings in a lab report.

8 Future Improvements and Perspectives

8.1 SLAM

Using a cartographer to create a real-time map of the environment and enable absolute positioning.

8.2 Extensions

- Use filters for general robot operation, setting them as the robot's default state values.
- Implement trajectory following based on the planned path.
- Integrate all functionalities directly on the robot, eliminating the need for an external computer.

9 References

- Unitree developer guide website
- The provided sdk, for unitree robots
- Phil's Lab - Sensor Fusion : Part2 The Complementary Filter
- Sensor Fusion and Non-linear Filtering for Automotive Systems by ChalmersX University
- Robust State Estimation for Legged Robots with Dual Beta Kalman Filter