B(E)3M33MRS - Task 01 Controller

Tomáš Báča

October 24, 2023

v1.03

The first task focuses on control and state estimation of multirotor Unmanned Aerial Vehicles (UAVs).

1 Requirements

- Implement and tune a feedback controller to stabilize and control the UAV according to a given feasible reference.
  - Update the `calculateControlSignal()` method in `task_01_controller:packages/controller/src/controller.cpp`.
- Implement a Linear Kalman Filter (LKF) to estimate the position of the UAV.
  - Devise a linearised model of a quadrotor UAV translational dynamics around the "hover" operation point.
  - Fill the `lkfPredict()` function in `task_01_controller:packages/controller/src/lkf.cpp`.
  - Fill the `lkfCorrect()` function in `task_01_controller:packages/controller/src/lkf.cpp`.

The red-labeled circle tasks are verified automatically using the Brute upload system.

2 Preliminaries

Aerial multi-robot research first requires the ability to control a single aerial robot. This assignment focuses on state estimation and feedback control of a single underactuated multirotor UAV. The assignment is divided into the following sections.

2.1 and 2.2 introduce the translational dynamic motion model of a generic underactuated multirotor UAV. The model will help us define and understand the available control inputs.

2.3 shows the programming interface of the simulation environment.

3.–5. define the assignment sub-tasks, including the evaluation criteria.

2.1 Translational motion model of a multirotor UAV

Let $\mathbf{r}^{\mathcal{W}} \in \mathbb{R}^3$ be the 3D position of the UAV in the world coordinate frame $\mathcal{W}$, $\mathbf{R} \in \mathbb{R}^{3\times3}$ be the rotation matrix from the UAV's body frame $\mathcal{B}$ to the world frame $\mathcal{W}$, $m \in \mathbb{R}$ be the mass of the UAV, and let us assume that the only forces acting on the UAV are

- the gravity force causing the gravitational acceleration $\mathbf{g}^{\mathcal{W}} = \begin{bmatrix}0, 0, -g\end{bmatrix}^\intercal$, $g \approx 9.81\,\mathrm{m\,s^{-2}}$,
- and the total collective thrust force $\mathbf{F}^{\mathcal{B}} = \begin{bmatrix}0, 0, F\end{bmatrix}^\intercal$ created by all the propellers, acting perpendicular to the body plane of the quadrotor.

Then, we can use the 2nd Newton's law to define the translational dynamics of the UAV as

$$\ddot{\mathbf{r}}^{\mathcal{W}} = \frac{1}{m}\mathbf{R}\mathbf{F}^{\mathcal{B}} + \mathbf{g}^{\mathcal{W}}. \tag{1}$$

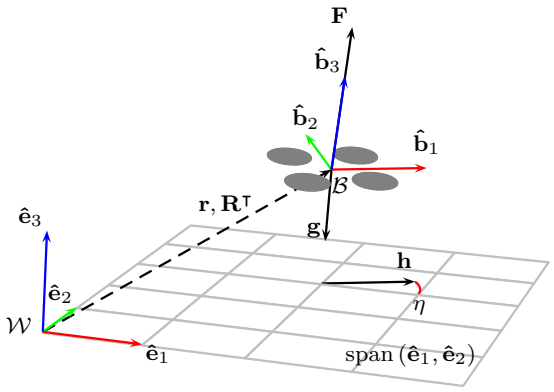

Figure 1: The image depicts the world frame $\mathcal{W} = \{\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3\}$ in which the 3D position and the orientation of the UAV body is expressed. The body frame $\mathcal{B} = \{\hat{\mathbf{b}}_1, \hat{\mathbf{b}}_2, \hat{\mathbf{b}}_3\}$ relates to $\mathcal{W}$ by the translation $\mathbf{r} = [x, y, z]^\intercal$ and by rotation $\mathbf{R}^\intercal$. The UAV heading vector $\mathbf{h}$, which is a projection of $\hat{\mathbf{b}}_1$ to the plane defined by span $(\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2)$, forms the heading angle $\eta = \text{atan2}\left(\hat{\mathbf{b}}_1^\intercal \hat{\mathbf{e}}_2, \hat{\mathbf{b}}_1^\intercal \hat{\mathbf{e}}_1\right) = \text{atan2}\left(\mathbf{h}_{(2)}, \mathbf{h}_{(1)}\right)$.

Therefore, we can control the total acceleration of the UAV by controlling the orientation $\mathbf{R}$ and the force F. Figure 1 depicts the UAV and the used coordinate frames.

For this assignment, **students will generate control outputs** for the UAV in the form of:

- the desired orientation matrix $\mathbf{R}_d$,
- the desired scalar thrust force F_d,

based on the **control reference** consisting of $\mathbf{r}_{\text{ref}}, \ddot{\mathbf{r}}_{\text{ref}}, \eta_{\text{ref}}$. The **control reference** is smooth and feasible and it needs **no filtration**. See Fig. 2 more visual representation of the control pipeline.

Remark 1. *The orientation matrix $\boldsymbol{R}$ can be used to transform vectors from the body frame to the world frame. For example, acceleration measured by the Inertial Measurement Unit (IMU) in the body frame, is often transformed to the world frame:*

$$\boldsymbol{a}^{\mathcal{W}} = \boldsymbol{R}\boldsymbol{a}^{\mathcal{B}}.$$

Remark 2. *The orientation matrix $\boldsymbol{R}$ contains the orthonormal basis of the body frame expressed in the world frame in its columns:*

$$\boldsymbol{R} = \left[\hat{\boldsymbol{b}}_1^{\mathcal{W}}\ \hat{\boldsymbol{b}}_2^{\mathcal{W}}\ \hat{\boldsymbol{b}}_3^{\mathcal{W}}\right].$$

Remark 3. *The orientation matrix $\boldsymbol{R}$ is orthonormal, i.e.,*

$$\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{I}, \qquad \boldsymbol{R}^T = \boldsymbol{R}^{-1}.$$

2.2 Augmenting the control inputs

Working with the rotational matrix and the thrust directly can be troublesome at first. We suggest augmenting these inputs into more intuitive ones: tilt in the world's xz-plane, yz-plane, and acceleration along the world's z-axis. To do that, the total acceleration $\ddot{\mathbf{r}}^{\mathcal{W}}$ can be decoupled into three components along the world's three axes (x, y and z) using the following parametrization:

$$\begin{aligned}
\ddot{r}_x^{\mathcal{W}} &= \frac{\sin\alpha}{m}\|\mathbf{F}_{xz}^{\mathcal{W}}\|, \\
\ddot{r}_y^{\mathcal{W}} &= \frac{\sin\beta}{m}\|\mathbf{F}_{yz}^{\mathcal{W}}\|, \\
\ddot{r}_z^{\mathcal{W}} &= \frac{1}{m}F_z^{\mathcal{W}} - g,
\end{aligned} \tag{2}$$

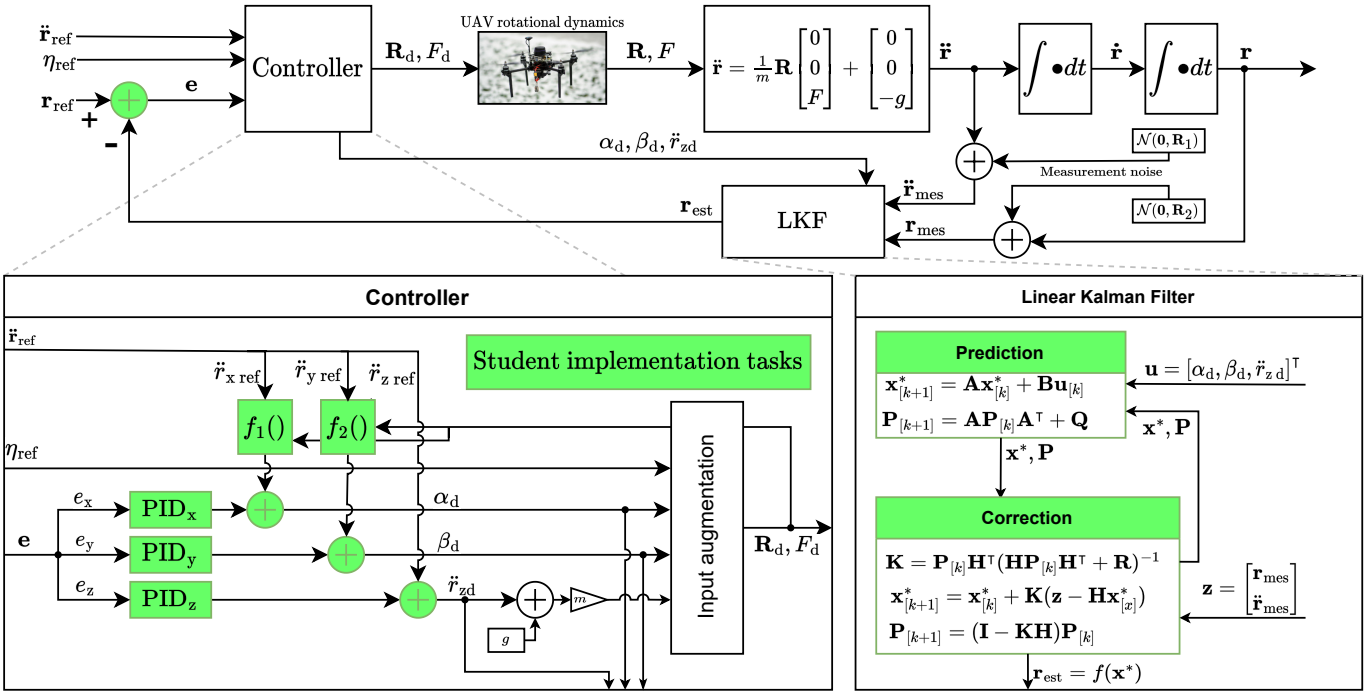

Figure 2: Control diagram of the UAV control pipeline. The green-colored blocks are to be implemented by the students.

where $\alpha, \beta \in \mathbb{R}$ are tilt angles of the UAV in the world's xz-plane and yz-plane respectively, $\mathbf{F}_{xz}^{\mathcal{W}}$ is the orthogonal projection of $\mathbf{F}^{\mathcal{W}}$ to the xz-plane, $\mathbf{F}_{yz}^{\mathcal{W}}$ is the orthogonal projection of $\mathbf{F}^{\mathcal{W}}$ to the yz-plane, and $F_z^{\mathcal{W}}$ is the orthogonal projection of $\mathbf{F}^{\mathcal{W}}$ to the z-axis.

With this decoupled model, we can define alternative control inputs that might be more intuitive and easier to work with at start:

- the desired xz tilt angle: α_d,
- the desired yz tilt angle: β_d,
- the desired force along the world's z-axis: $F_z^{\mathcal{W}}$,
- the desired heading angle $\eta = \text{atan2}\left(\hat{\mathbf{b}}_1^\intercal \hat{\mathbf{e}}_2, \hat{\mathbf{b}}_1^\intercal \hat{\mathbf{e}}_1\right)$.

The provided software **already contains the routine** for converting the augmented inputs $\left[\alpha, \beta, \eta, F_z^{\mathcal{W}}\right]$ to the raw inputs $[\mathbf{R}, F]$. Students can decide whichever control inputs they want to generate. Figure 2 shows a control diagram of the complete control pipeline.

2.3 Starting the simulation

1. Install the Singularity[1] container software on your system. The CTU university computers will have it installed already. If you are a Linux user, follow the directions provided for your particular Linux distribution. For Ubuntu, use our pre-configured install script in `task_01_controller:simulation/install/install_singularity.sh`. It is possible to set up Singularity for Windows. However, we provide no support for it. Follow the short manual here[2] for Windows installation instructions.
2. Download the pre-built Singularity image by executing: `./simulation/download.sh`. This will download approx. 4 GB of data. The resulting image will be placed in `./simulation/images`.
3. Compile the sources for the controller by running `./simulation/compile.sh`.
4. Start the simulation by running `./simulation/run_simulation.sh`.

[1] `https://sylabs.io/singularity/`
[2] `https://github.com/ctu-mrs/mrs_singularity/`

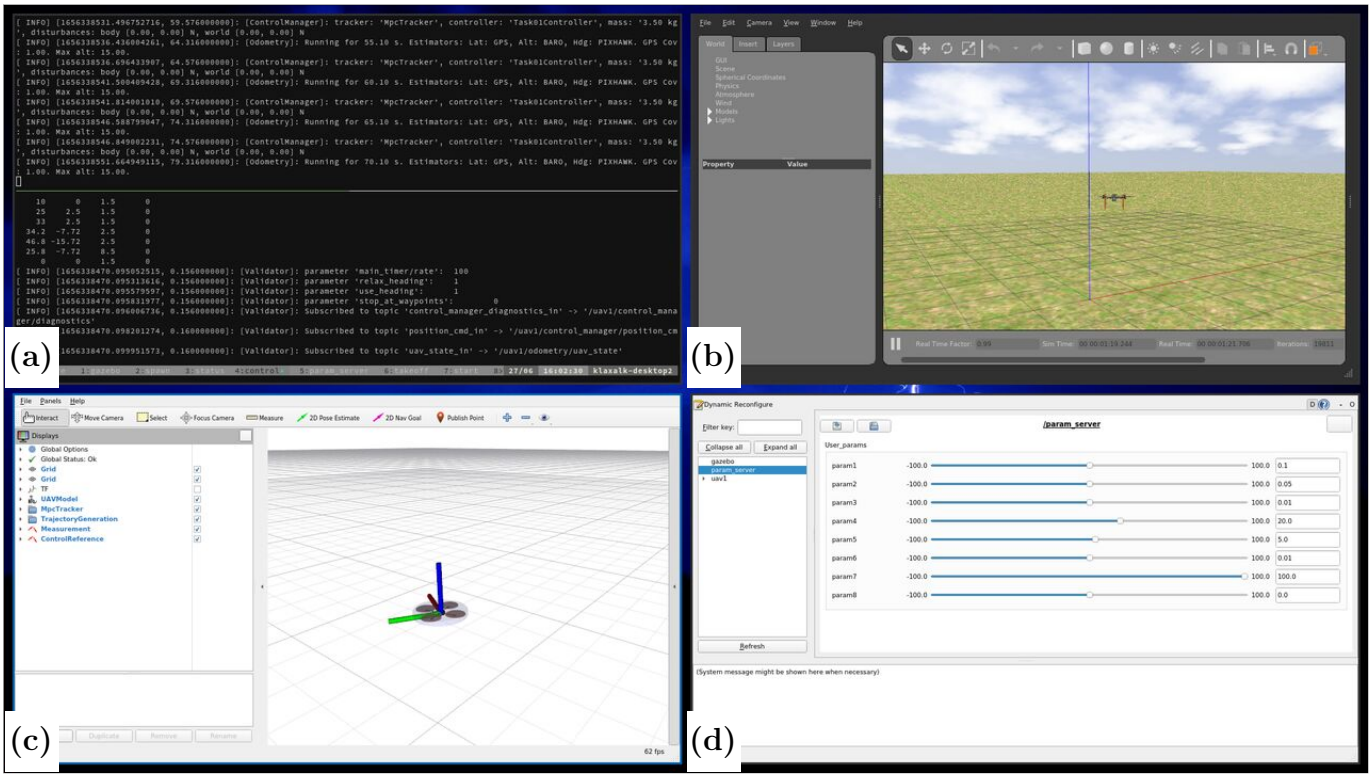

Figure 3: Windows shown during the simulation: a) the terminal window showing the controller's output and the output of the evaluation program, b) the Gazebo GUI shows what exactly is the UAV doing, c) the RViz GUI shows visualization of some internal data, d) the parameter server allows you to modify parameters of your program mid-flight.

- After the simulation starts, four windows will appear on your screen (see Fig. 3):
  (a) A terminal running all the simulation software. This window will show the terminal output of your controller, as well as the results of the evaluation,
  (b) A window of the Gazebo simulator.
  (c) A window of the RViz visualizer.
  (d) A window of the parameter server. You can modify user parameters mid-flight. These parameters will be passed to the controller during each iteration.

5. Initiate the evaluation of the controller by running `./simulation/test_controller.sh` when the UAV is airborne.
6. Initiate the evaluation of the Kalman Filter by running `./simulation/test_kalman.sh`.
7. Kill the simulation by running `./simulation/kill_simulation.sh`.
8. Start a code editor from within the container:
   - start the VSCode editor `./simulation/vscode.sh` or
   - start the Sublime Text editor `./simulation/sublimetext.sh`.

3 Tasks

The tasks consist of the two traditional problems of UAV control engineering:

- implementing a controller to control the position and heading of the UAV,
- implementing a Linear Kalman Filter to estimate and filter the UAV's position measurements, which should improve the control performance.

3.1 Implementing a PID controller with a feedforward term

The main task is implementation of a PID controller for each translational axis of the UAV. First, the positional control error is calculated as

$$\left[e_x, e_y, e_z\right]^\intercal \equiv \mathbf{e} = \mathbf{r}_{\text{ref}} - \mathbf{r}. \tag{3}$$

Then, the control law can be formulated for each axis as

$$\alpha_{\text{d}} = P_{\text{x}} e_x + D_{\text{x}} \frac{de_{\text{x}}}{dt} + I_{\text{x}} \int_0^t e_x d\tau, \tag{4}$$

$$\beta_{\text{d}} = P_{\text{y}} e_y + D_{\text{y}} \frac{de_{\text{y}}}{dt} + I_{\text{y}} \int_0^t e_y d\tau, \tag{5}$$

$$\ddot{r}_{\text{zd}} = P_{\text{z}} e_z + D_{\text{z}} \frac{de_{\text{z}}}{dt} + I_{\text{z}} \int_0^t e_z d\tau, \tag{6}$$

where P_{x}, P_{y}, P_{z} are the proportional control gains, D_{x}, D_{y}, D_{z} are the derivative control gains and I_{x}, I_{y}, I_{z} are the integral control gains.

Although the PID controllers are more than capable of stabilizing the UAV around the current position setpoint, they are not well-fit to dynamically move the UAV through 3D space. This is mainly because the only input to the PID controller is the current position control error. Before the controller starts generating any acceleration, some non-zero control error must first appear, which limits the reaction time of the controller. To mitigate this, the **desired acceleration** (which is available) can be used as a **feedforward** term to improve the reaction time. To finish the controller, implement the feedforward term (functions $f_1()$, $f_2()$ in Fig. 2) for calculating feedforward tilt actions based on the desired acceleration (refer to Fig. 2):

$$\alpha_{\text{d,ff}} = f_1(\ddot{r}_{x,\text{ref}}, F_d, m), \tag{7}$$

$$\beta_{\text{d,ff}} = f_2(\ddot{r}_{y,\text{ref}}, F_d, m). \tag{8}$$

Remark 4. *The lateral control gains for the x- and y- axes can be set equally, thanks to the decoupled translational UAV dynamics.*

3.1.1 Controller's interface

The controller's interface is provided in the form of a C++ class in

- `task_01_controller:packages/controller/include/student_headers/controller.h`,
- `task_01_controller:packages/controller/src/controller.cpp`.

The controller will be first initialized via the method

```
/**
* @brief The controller initialization method. It is called ONLY ONCE in the lifetime of the ↩
    controller.
* Use this method to do any heavy pre-computations.
*
* @param mass UAV mass [kg]
* @param user_params user-controllable parameters
* @param g gravitational acceleration [m/s^2]
* @param action_handlers methods for the user
*/
void init(const double mass, const UserParams_t user_params, const double g, ActionHandlers_t &↩
    action_handlers);
```

where the user will obtain the nominal mass m of the UAV, the magnitude of the gravitational acceleration g, and the user-defined parameters. This method will be executed only once and is dedicated to the pre-flight initialization of the controller. Any longer pre-computations are meant to be executed here.

Moreover, the controller will be notified before its outputs will be used via the method

```
/**
* @brief This method is called to reset the internal state of the controller, e.g., just before
* the controller's activation. Use it to, e.g., reset the controller's integrators and estimators.
*/
void reset();
```

where any temporary values are supposed to be cleared. This method is called just before the activation or re-activation if the controller is de-activated in mid-air.

Finally, the main control loop will repeatedly call the following method to obtain fresh control inputs from the controller.

```
/**
* @brief the main routine, is called to obtain the control signals
*
* @param uav_state the measured UAV state, contains position and acceleration
* @param user_params user-controllable parameters
* @param control_reference the desired state of the UAV, position, velocity, acceleration, heading
* @param dt the time difference in seconds between now and the last time calculateControlSignal() got called
*
* @return the desired control signal: the total thrust force and the desired orientation
*/
std::pair<double, Matrix3d> calculateControlSignal(const UAVState_t uav_state, const UserParams_t user_params, const ControlReference_t control_reference, const double dt);
```

The user will receive

- current measurements of the UAV's state (noisy position, orientation and translational acceleration),
- the user-defined parameters (can be changed in mid-flight and used as, e.g., controller gains),
- a control reference (the desired position, the desired velocity, the desired acceleration, the desired heading),
- the time difference from the last update,

and the user is supposed to return

- the desired orientation $\mathbf{R}_d \in \mathbb{R}^{3\times 3}$ and
- the desired total thrust $F_d \in \mathbb{R}$.

Remark 5. *Tune the PID for controlling the altitude first, that will stop the UAV from crashing into the ground plane. Tune the lateral axes later. Both lateral axes will work with the exact same gains, since the model is the same.*

Remark 6. *You can save your last good value of the user parameters in the `task_01_controller: packages/controller/config/user_params.yaml`. However, do not attempt to rename the parameters!*

3.2 Adding the Linear Kalman Filter

With the PID controllers implemented, you may notice that the control performance is severely impaired by the noise in the measured data, which is even amplified by the derivative component of the controller. This problem can be addressed by introducing a state observer, which will filter the measured data and estimate the UAV's state closer to

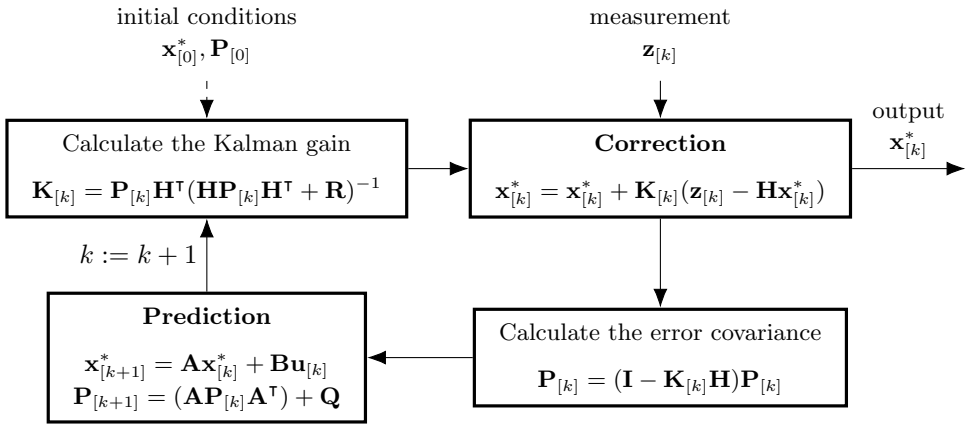

Figure 4: Diagram of the Linear Kalman Filter update loop.

the actual state than the raw measurements. Let us consider a stochastic Linear Time Invariant (LTI) system with state $\mathbf{x} \in \mathbb{R}^n$ and input $\mathbf{u} \in \mathbb{R}^m$

$$\mathbf{x}_{[k+1]} = \mathbf{A}\mathbf{x}_{[k]} + \mathbf{B}\mathbf{u}_{[k]} + \mathbf{w}_{[k]}, \tag{9}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is the main system matrix, $\mathbf{B} \in \mathbb{R}^{n \times m}$ is the input matrix and $\mathbf{w}_{[k]} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q} \in \mathbb{R}^{n \times n})$ is the process noise. We represent the estimated state of the system and its uncertainty by a tuple $\{\mathbf{x}^*_{[k]}, \mathbf{P}_{[k]}\}$ consisting of the state vector estimate $\mathbf{x}^*_{[k]}$ and the covariance matrix of the estimate $\mathbf{P}_{[k]} \in \mathbb{R}^{n \times n}$. We formulate the Linear Kalman Filter using two operations: **Prediction** and **Correction**, both of which update the current $\{\mathbf{x}^*_{[k]}, \mathbf{P}_{[k]}\}$. Figure 4 depicts the full cycle of the LKF algorithm.

3.2.1 Prediction

The prediction step iterates the model given the system's input (if applicable) as

$$\mathbf{x}^*_{[k+1]} = \mathbf{A}\mathbf{x}^*_{[k]} + \mathbf{B}\mathbf{u}_{[k]}, \tag{10}$$

$$\mathbf{P}_{[k+1]} = \mathbf{A}\mathbf{P}_{[k]}\mathbf{A}^\intercal + \mathbf{Q}, \tag{11}$$

where $\mathbf{Q}$ is the process noise covariance matrix. Then, $\{\mathbf{x}^*_{[k+1]}, \mathbf{P}_{[k+1]}\}$ is the **prediction** of the state estimate in the next time step and its corresponding uncertainty. The $\{\mathbf{x}^*_{[0]}, \mathbf{P}_{[0]}\}$ needs to be initialized before the very first step. We recommend setting $\mathbf{x}^*_{[0]}$ according to the very first measurement and $\mathbf{P}_{[0]} = \mathbf{I}$ (identity matrix).

3.2.2 Correction

The second step, often called the **correction**, updates $\{\mathbf{x}^*_{[k]}, \mathbf{P}_{[k]}\}$ given a measurement vector $\mathbf{z} \in \mathbb{R}^p$. It assumes a measurement model

$$\mathbf{z}_{[k]} = \mathbf{H}\mathbf{x}_{[k]} + \mathbf{v}_{[k]}, \tag{12}$$

where $\mathbf{H} \in \mathbb{R}^{p \times n}$ is the measurement matrix mapping the state space to the measurement space, and $\mathbf{v}_{[k]} \sim \mathcal{N}(\mathbf{0}, \mathbf{R} \in \mathbb{R}^{p \times p})$ is the measurement noise. The state estimate and its covariance matrix are updated using a ("corrected") measurement as

$$\mathbf{K}_{[k]} = \mathbf{P}_{[k]}\mathbf{H}^\intercal \left(\mathbf{H}\mathbf{P}_{[k]}\mathbf{H}^\intercal + \mathbf{R}\right)^{-1}, \tag{13}$$

$$\mathbf{x}^*_{[k]} := \mathbf{x}^*_{[k]} + \mathbf{K}_{[k]} \left(\mathbf{z}_{[k]} - \mathbf{H}\mathbf{x}^*_{[k]}\right), \tag{14}$$

$$\mathbf{P}_{[k]} := \left(\mathbf{I} - \mathbf{K}_{[k]}\mathbf{H}\right)\mathbf{P}_{[k]}, \tag{15}$$

where $\mathbf{K}_{[k]} \in \mathbb{R}^{n \times p}$ is called the *Kalman gain* and $\mathbf{I}$ is an identity matrix of the corresponding size.

3.2.3 Implementation

The task is to **implement** the LKF for estimating the **3D position** of the UAV given noisy measurements of the UAV **position and acceleration**. The LKF's interface is provided in the form of two C++ functions in

- `task_01_controller:packages/controller/src/lkf.cpp`.

The **Prediction** step is supposed to be implemented in the `lkfPredict()` function:

```
/**
 * @brief the prediction step of the LKF
 *
 * @param x current state vector: x = [pos_x, pos_y, pos_z, vel_x, vel_y, vel_z, acc_x, acc_y, ↩
     acc_z]^T
 * @param x_cov current state covariance: x_cov in R^{9x9}
 * @param input current control input: input = [tilt_xz, tilt_yz, acceleration_z]^T, tilt_xz = ↩
     desired tilt in the world's XZ [rad], tilt_yz = desired
 * tilt in the world's YZ plane [rad], acceleration_z = desired acceleration along world's z-axis ↩
     [m/s^2]
 * @param dt the time difference in seconds between now and the last iteration
 *
 * @return <new_state, new_covariance>
 */
std::tuple<Vector9d, Matrix9x9d> Controller::lkfPredict(const Vector9d &x, const Matrix9x9d &x_cov↩
    , const Vector3d &input, const double &dt) {
```

and the **Correction** step is to be implemented in the `lkfCorrect()` function:

```
/**
 * @brief LKF filter correction step
 *
 * @param x current state vector: x = [pos_x, pos_y, pos_z, vel_x, vel_y, vel_z, acc_x, acc_y, ↩
     acc_z]^T
 * @param x_cov current state covariance: x_cov in R^{9x9}
 * @param measurement measurement vector: measurement = [pos_x, pos_y, pos_z, acc_x, acc_y, acc_z↩
     ]^T
 * @param dt the time difference in seconds between now and the last iteration
 *
 * @return <new_state, new_covariance>
 */
std::tuple<Vector9d, Matrix9x9d> Controller::lkfCorrect(const Vector9d &x, const Matrix9x9d &x_cov↩
    , const Vector6d &measurement, const double& dt) {
```

Both functions should return the tuple of the updated state $\mathbf{x}^*$ and state covariance $\mathbf{P}$. **Both functions are later supposed to be called by the student** within the `calculateControlSignal()` function. Students are supposed to create the proper input to these functions and then handle the output.

This implementation task can be divided into the following fundamental steps:

- Devise the discrete LTI ($\Delta t = 0.01\,\mathrm{s}$) model ($\mathbf{A}$, $\mathbf{B}$) for the translational dynamics of the UAV, such that the state vector is
$$\mathbf{x} = \left[\mathbf{r}^\intercal, \dot{\mathbf{r}}^\intercal, \ddot{\mathbf{r}}^\intercal\right]^\intercal \tag{16}$$
and the input is
$$\mathbf{u} = \left[\alpha_\mathrm{d}, \beta_\mathrm{d}, \ddot{r}_\mathrm{zd}\right]^\intercal. \tag{17}$$
- Devise the measurement mapping matrix $\mathbf{H}$ for the measurement vector
$$\mathbf{z} = \left[\mathbf{r}_\mathrm{mes}^\intercal, \ddot{\mathbf{r}}_\mathrm{mes}^\intercal\right]^\intercal. \tag{18}$$
- Implement the equations (10–15).
- Tune the filter by adjusting the covariance matrices $\mathbf{Q}$ and $\mathbf{R}$ (it is recommended to start from identity matrices).

Make sure you comply with the definition of the state vector in (16) and of the measurement vector in (18). The automated test will fail if you do not. We recommend to use the `A.diagonal(i)` method of the *Eigen* library to build your system matrices.

Remark 7. *The filter is tuned mainly by adjusting the ratio between the elements on the diagonals of* $\mathbf{Q}$ *and* $\mathbf{R}$.

Remark 8. *The Kalman filter might work even without the system input. If you wish to use the input, note that the transient from the desired tilt to the actual tilt can be expressed as:*
$$\alpha_{[k+1]} = 0.95\alpha_{[k]} + 0.05\alpha_{d[k]} \tag{19}$$
and the transient from the desired vertical acceleration to the actual vertical acceleration can be expressed as
$$\ddot{r}_{z[k+1]} = 0.99\ddot{r}_{z[k]} + 0.01\ddot{r}_{zd[k]}. \tag{20}$$
Both apply only for $\Delta t = 0.01\,\mathrm{s}$, *which is the default control time step.*

4 Dos and Don'ts

4.1 Dos

- Study this document and orient yourself within the provided source codes first. This will help you be more efficient than if you start programming immediately.
- Write nicely-structured and commented code. Your teacher will be better able to help you if your code is readable.

- Use the provided parameter server to modify *tunable parameters* (such as the control gains) in your code. The default values (loaded after program start) are defined in `task_01_controller:packages/controller/config/user_params.yaml`.
- You can add new files to the project (e.g., to isolate your PID controller). However, these can only be header files placed in the `controller/include/student_headers` folder. Do NOT attempt to add new `*.cpp` files to the project, they will not be copied and any changes to the *CMakeLists.txt* will be discarded when evaluating your solution in BRUTE.
- You can use runtime debugging via GDB (The GNU Project Debugger). Enable debugging by changing `DEBUG:=false` to `DEBUG:=true` on line 4 of `task_01_controller:simulation/tmux/session.yaml`. Alternatively, you can run the `./debug.sh` script, which will attach itself to the control software (has to be already running). Learn to use the debugger by checking its official documentation page, or by checking the MRS Cheat Sheet[3].

4.2 Don'ts

- Do **NOT** modify any other files than the following:
  - `task_01_controller:packages/controller/src/controller.cpp`
  - `task_01_controller:packages/controller/include/student_headers/controller.h`
  - `task_01_controller:packages/controller/src/lkf.cpp`
  - `task_01_controller:packages/controller/config/user_params.yaml`
  - `task_01_controller:packages/simulation/tmux/session.yaml`
- Do **NOT** use any third-party library, besides Eigen[4], which is already provided.

5 Evaluation

5.1 On your local machine

5.1.1 Kalman filter

The performance of your filter can be checked by running the `./simulation/test_kalman.sh` script. This check is randomized. The criteria for passing are the Root Mean Square Errors (RMSEs) of the estimated position and velocity states from an original ground-truth signal. If both RMSEs are below 0.05, the test will pass.

5.1.2 Controller

Your controller's performance can be checked by a provided evaluation program in simulation. **With the simulation running**, execute the `./simulation/test_controller.sh` script. This will make the UAV fly along a pre-defined path while the RMS errors for the UAV position and tilt are calculated. If both RMSEs are below 0.5, the test will pass.

Remark 9. *A good controller cannot only minimize the control error but also provide a smooth and efficient control signal. The tilt RMSE is an efficient measure of the aggressivity of the controller. Significant tilt RMSE suggests that the controller produces a noisy control signal, which could quickly destabilize a real-world UAV.*

5.2 On the BRUTE server

The BRUTE server will run the same two tests that are available on your local machine. If the tests pass on your local machine, it is likely they will also pass on the server.

1. Create an archive (zip, tar, tag.gz) of the **controller** (located in the `packages` folder), e.g., by running the command:

```
zip -r controller.zip controller
```

[3] https://github.com/ctu-mrs/mrs_cheatsheet
[4] https://eigen.tuxfamily.org/dox/GettingStarted.html

2. Upload the archive to BRUTE (`https://cw.felk.cvut.cz/brute/`).
3. Wait for the evaluation.
4. If the compilation fails, you will be presented with the compilation log.
5. If the compilation succeeds, you will be presented with logs from the **compilation**, **simulation**, **control test** and **Kalman test**.

The evaluation takes approximately 3 minutes.

Remark 10. *The* Simulation log *from BRUTE (which corresponds to the main simulation output from your local machine) can contain large amounts of errors. These are probably not the reason for your solution failing the tests. However, these are caused by the limited computational resources and the claustrophobic Docker environment of the evaluation servers. We recommend to print your own messages but be warned, the final log page should not be larger than 3 MB. To limit the logs, throttle them to lower rate. That can be achieved by, e.g., printing once in every 100 iterations of the control loop.*

5.2.1 Scoring

Both the **Control test** and the **Kalman test** need to succeed to pass the task (10 points).

5.2.2 Late submission

Late submissions will be penalized by 2 points for each week after the deadline.