Lab 3 Report

Robotics Integration Group Project I

Yuwei ZHAO (23020036096)
Group #31 2025-11-26

Abstract

Lab3 explores the fundamentals of rigid body transformations and the implementation of a geometric controller for a quadrotor UAV. It separated into the following two parts: **(1)** The individual part focuses on the practical manipulation of rotations using `tf2`, `geometry_msgs`, and `Eigen` libraries within the ROS ecosystem, alongside a theoretical analysis of quadrotor dynamics and drag compensation. **(2)** The collaborative component involves implementing a non-linear geometric controller on SE(3) to track dynamic trajectories. The system is integrated and validated using the TESSE Unity simulator, demonstrating the UAV's ability to follow a reference path effectively.

See Resources on github.com/RamessesN/Robotics_MIT.

1 Introduction

Precise trajectory tracking is a fundamental challenge in UAV robotics due to the system's fast and underactuated dynamics. This laboratory aims to bridge the gap between theoretical control derivation and practical software implementation within the ROS1.

The objectives of this report are threefold: **(1)** First, we practice handling rigid body transformations using `tf2` and `Eigen` libraries to ensure accurate state estimation. **(2)** Second, we analyze quadrotor dynamics, specifically examining the mixing matrix and drag force compensation. **(3)** Finally, the report details the implementation of a geometric tracking controller on the Special Euclidean group $\mathrm{SE}(3)$, based on the work of Lee et al. We integrate this controller into the MIT-TESSE simulation environment to validate its performance. The procedure demonstrates the complete workflow from mathematical modelling to C++ implementation, culminating in the successful tracking of a complex trajectory.

2 Procedure

2.1 Individual Work

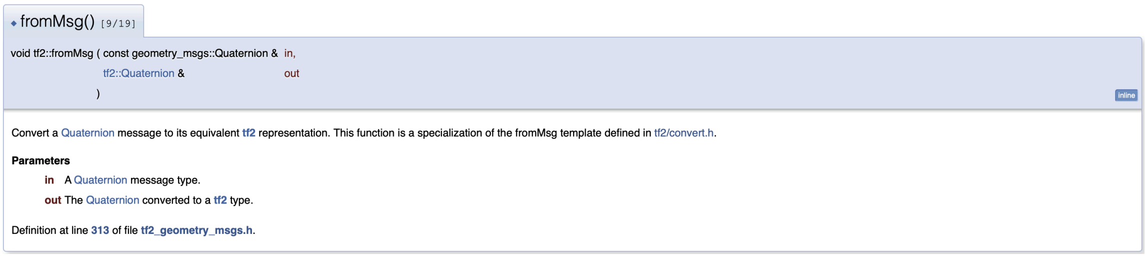
2.1.1 Transformations in Practice

1. **MESSAGE VS. TF**

- **Assume we have an incoming `geometry_msgs::Quaternion quat_msg` that holds the pose of our robot. We need to save it in an already defined `tf2::Quaternion quat_tf` for further calculations. Write one line of C++ code to accomplish this task.**

```cpp
tf2::fromMsg(quat_msg, quat_tf);
```

More specifically, we can find the official documentation of `fromMsg()` at this page:



◆ fromMsg() [9/19]

void tf2::fromMsg (const geometry_msgs::Quaternion & in,
tf2::Quaternion & out
)

inline

Convert a Quaternion message to its equivalent tf2 representation. This function is a specialization of the fromMsg template defined in tf2/convert.h.

Parameters
in A Quaternion message type.
out The Quaternion converted to a tf2 type.

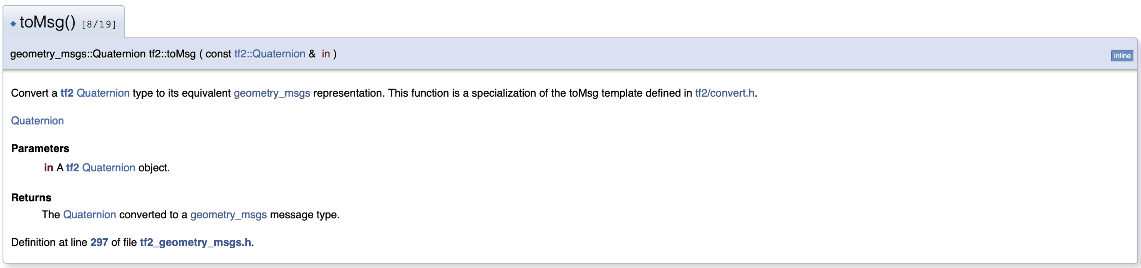
Definition at line 313 of file tf2_geometry_msgs.h.

Figure 1: tf2 Quaternion doc

- **Assume we have just estimated our robot's newest rotation and it's saved in a variable called `quat_tf` of type `tf2::Quaternion`. Write one line of C++ code to convert it to a `geometry_msgs::Quaternion` type. Use `quat_msg` as the name of the new variable.**

```cpp
geometry_msgs::Quaternion quat_msg = tf2::toMsg(quat_tf);
```

More specifically, we can find the official documentation of `toMsg()` in the same link as `fromMsg()`:



◆ toMsg() [8/19]

geometry_msgs::Quaternion tf2::toMsg (const tf2::Quaternion & in)

inline

Convert a tf2 Quaternion type to its equivalent geometry_msgs representation. This function is a specialization of the toMsg template defined in tf2/convert.h.

Quaternion

Parameters
in A tf2 Quaternion object.

Returns
The Quaternion converted to a geometry_msgs message type.

Definition at line 297 of file tf2_geometry_msgs.h.

Figure 2: geometry_msgs Quaternion doc

- **If you just want to know the scalar value of a `tf2::Quaternion`, what member function will you use?**

```cpp
double scalar = quat_tf.getW();
```

More specifically, we find the official documentation of `getW()` here:

◆ getW()

const TF2SIMD_FORCE_INLINE tf2Scalar& tf2::Quaternion::getW ( ) const inline

Definition at line 348 of file Quaternion.h.

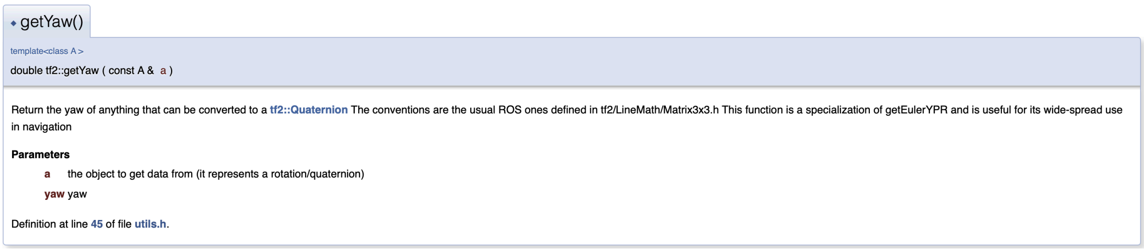
Figure 3: Quaternion get_w doc

2. CONVERSION

- **Assume you have a `tf2::Quaternion quat_t`. How to extract the yaw component of the rotation with just one function call?**

```cpp
double yaw = tf2::getYaw(quat_t);
```

More specifically, the doc of `getYaw()` is shown at this page:



◆ getYaw()

template<class A >

double tf2::getYaw (const A & a)

Return the yaw of anything that can be converted to a tf2::Quaternion The conventions are the usual ROS ones defined in tf2/LineMath/Matrix3x3.h This function is a specialization of getEulerYPR and is useful for its wide-spread use in navigation

Parameters

a the object to get data from (it represents a rotation/quaternion)

yaw yaw

Definition at line 45 of file utils.h.

Figure 4: Quaternion get_yaw doc

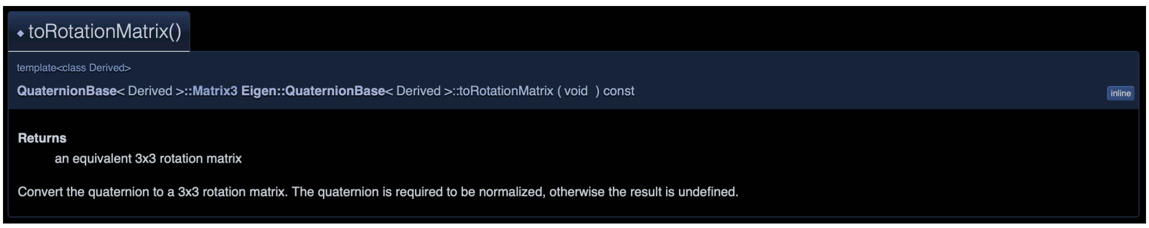
- **Assume you have a `geometry_msgs::Quaternion quat_msg`. How to you convert it to an Eigen 3-by-3 matrix? Refer to this for possible functions. You probably need two function calls for this.**

```cpp
#include <tf2_eigen/tf2_eigen.h>

Eigen::Quaterniond eigen_quat;

// The first function to call
tf2::fromMsg(quat_msg, eigen_quat);
// The second function to call
Eigen::Matrix3d eigen_mat3 = eigen_quat.toRotationMatrix();
```

More specifically, the doc of `toRotationMatrix()` can be found here:



◆ toRotationMatrix()

template<class Derived>

QuaternionBase< Derived >::Matrix3 Eigen::QuaternionBase< Derived >::toRotationMatrix (void) const inline

Returns

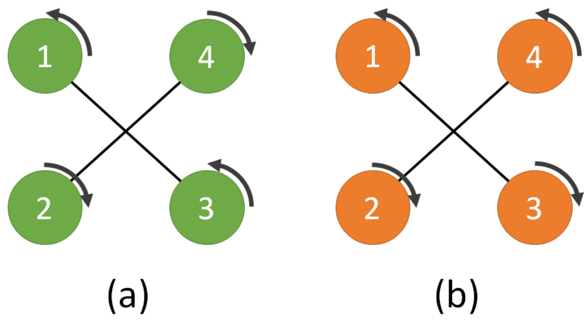
an equivalent 3x3 rotation matrix

Convert the quaternion to a 3x3 rotation matrix. The quaternion is required to be normalized, otherwise the result is undefined.

Figure 5: Eigen toRotationMatrix doc

2.1.2 Modelling and control of UAVs

1. **STRUCTURE OF QUADROTORS**



The figure above depicts two quadrotors (a) and (b). Quadrotor (a) is a fully functional UAV, while for Quadrotor (b) someone changed propellers 3 and 4 and reversed their respective rotation directions.

Show mathematically that quadrotor (b) is not able to track a trajectory defined in position $[x, y, z]$ and yaw orientation Ψ.

In order to proof quadroter (b) is not able to track a trajectory, we have to judge whether the rank of the matrix $\boldsymbol{F}$ is full.

The quadrotor has four inputs - the thrust of 4 motors: f_1, f_2, f_3, f_4, and four outputs free degrees (total thrust T and three-axis torque τ_{roll}, τ_{pitch}, and τ_{yaw}). The linear equation is:

$$u = \mathrm{F}f \tag{1}$$

which $u = [T,\tau_{\text{roll}},\tau_{\text{pitch}},\tau_{\text{yaw}}]^T$ is the output vector, $f = [f_1,f_2,f_3,f_4]^T$ is the input vector.

	Quadrotor (a)	**Quadrotor (b)**
Thrust $\boldsymbol{T}$	$[1,1,1,1]$	$[1,1,1,1]$
Roll $\boldsymbol{\tau}_{\textbf{roll}}$	$[-d,-d,d,d]$	$[-d,-d,d,d]$
Pitch $\boldsymbol{\tau}_{\textbf{pitch}}$	$[d,-d,-d,d]$	$[d,-d,-d,d]$
Yaw $\boldsymbol{\tau}_{\textbf{yaw}}$	$[-c,c,-c,c]$	$[-c,c,c,-c]$
Matrix $\boldsymbol{F}$	$F_a = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -d & -d & d & d \\ d & -d & -d & d \\ -c & c & -c & c \end{bmatrix}$	$F_b = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -d & -d & d & d \\ d & -d & -d & d \\ -c & c & c & -c \end{bmatrix}$
Rank	4 (full)	3 (not full)

Table 1: Comparison of Quadrotors

$\because \text{rank}_b = 3 < 4 \therefore$ the matrix F_b isn't full rank, which means that the output space of the system has only 3 dimensions so that it is not able to track a trajectory defined with $[x, y, z, \Psi]$.

2. **CONTROL OF QUADROTORS**
Assume that empirical data suggest you can approximate the drag force (in the body frame) of a quadrotor body as:

$$\boldsymbol{F^b} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \left(v^b\right)^2 \tag{2}$$

With $\left(v^b\right)^2 = \left[-v_x^b \cdot |v_x^b|, -v_y^b \cdot |v_y^b|, -v_z^b \cdot |v_z^b|\right]^T$**, and** v_x, v_y, v_z **being the quadrotor velocities along the axes of the body frame.**

With the controller discussed in class (see referenced paper [1]), describe how you could use the information above to improve the tracking performance.

From the referenced paper, we have:

$$\begin{cases} m\dot{v} & = mge_3 - fRe_3 \\ \vec{b}_{3d} & = -\frac{-k_x e_x - k_v e_v - mge_3 + m\ddot{x}_d}{\| -k_x e_x - k_v e_v - mge_3 + m\ddot{x}_d \|} \\ f & = -(-k_x e_x - k_v e_v - mge_3 + m\ddot{x}_d) \cdot Re_3 \end{cases} \tag{3}$$

$\therefore$ Expected resultant force vector: $u_{\text{nominal}} = -k_x e_x - k_v e_v - mge_3 + m\ddot{x}_d$

$\because$ Drag Force: $D_{\text{inertial}} = R \cdot F^b_{\text{drag}} = R \cdot \left(\begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} (v_b)^2_{\text{signed}} \right)$

$\therefore u_{\text{new}} = u_{\text{nominal}} - D_{\text{inerital}} = -k_x e_x - k_v e_v - mge_3 + m\ddot{x}_d - R \cdot F^b_{\text{drag}}$

$\therefore$ we have $\begin{cases} \vec{b}_{3d} = -\frac{u_{\text{new}}}{\|u_{\text{new}}\|} \\ f = -u_{\text{new}} \cdot Re^3 \end{cases}$.

2.2 Team Work

2.2.1 Trajectory tracking for UAVs

This section is to set up the environment, launch the simulator with ROS bridge, and implement the geometric controller for trajectory tracking.

First, we need to set up the workspace and install the dependencies.

```shell
sudo apt install ros-noetic-ackermann-msgs
```

Then, change to the `labs` directory, pull the latest code, and copy the lab3 files to the ROS workspace.

```shell
cd ~/labs
git pull

cp -r ~/labs/lab3/. ~/vnav_ws/src
cd ~/vnav_ws
```

Install the `tesse-interface` package and clone the `mav_comm` repository.

```shell
cd ~/vnav_ws/src/tesse-ros-bridge/tesse-interface
pip install -r requirements.txt # Dependencies
pip install .

cd ~/vnav_ws/src && git clone https://github.com/ethz-asl/mav_comm.git
```

Build the workspace and make the simulator executable.

```shell
catkin build
source devel/setup.bash

cd ~/vnav-builds/lab3/
chmod +x lab3.x86_64
```

2.2.2 Launching the TESSE simulator with ROS bridge

This section is to launch the TESSE simulator with ROS bridge and visualize the UAV in RViz. First, we need to run the simulator and the ROS bridge.

```shell
cd ~/vnav-builds/lab3/
./lab3.x86_64

source devel/setup.zsh
roslaunch tesse_ros_bridge tesse_quadrotor_bridge.launch
```

Then, we can open RViz with the provided configuration file to visualize the UAV and its trajectory.

```shell
cd ~/vnav_ws/src/controller_pkg
rviz -d rviz/lab3.rviz
```

2.2.3 Implement the controller

This section describes the implementation of the geometric controller for the UAV in C++. The controller subscribes to the desired and current state topics and publishes the rotor speed commands.

Topic Name	Message Type	Structure & Description
`/desired_state`	`trajectory_msgs/MultiDOFJointTrajectoryPoint`	**`transforms`** (`geometry_msgs/Transform[]`): • `translation`: 3D vector, desired position (World frame). • `rotation`: Quaternion, desired orientation (Yaw only). **`velocities`** (`geometry_msgs/Twist[]`): • `linear`: 3D vector, desired velocity (World frame). • `angular`: *Ignored*. **`accelerations`** (`geometry_msgs/Twist[]`): • `linear`: 3D vector, desired acceleration (World frame). • `angular`: *Ignored*.
`/current_state`	`nav_msgs/Odometry`	**`pose.pose`** (`geometry_msgs/Pose`): • `position`: 3D vector, current position (World frame). • `orientation`: Quaternion, current UAV orientation. **`twist.twist`** (`geometry_msgs/Twist`): • `linear`: 3D vector, current linear velocity (World frame). • `angular`: 3D vector, current angular velocity (**World frame**).
`/rotor_speed_cmds`	`mav_msgs/Actuators`	**`angular_velocities`** (`float64[]`): • Array containing the desired speeds of the propellers.

2.2.4 Simulator conventions

There are three key differences between the mathematical model presented in the reference paper and the conventions used in the TESSE simulator (and standard ROS frames). We have adapted the controller implementation to account for these differences:

1. **Reference Frame (Z-axis direction)**:
   - The paper assumes a Z-down coordinate system where the gravity vector is positive along the z-axis (mge_3).

- The simulator uses the standard ROS ENU (East-North-Up) frame where the z-axis points upward. Consequently, the gravity compensation term in the desired force calculation must be inverted.
- **Modification**: In Equation (12) of the paper, the term $-mge_3$ is replaced by $+mge_3$ to counteract gravity in the Z-up frame.

2. **Motor Configuration**:
   - The paper assumes a + configuration where the body X-axis is aligned with one of the rotors. This results in a mixing matrix where pitch and roll moments are decoupled.
   - The simulator uses an x configuration where the body axes are offset by 45° from the rotor arms. This means all four rotors contribute to both roll and pitch moments.
   - **Modification**: The mixing matrix (mapping from rotor forces to body wrench) is adapted. Let $L = \frac{d}{\sqrt{2}}$ be the projected arm length, the mapping becomes:

$$\begin{bmatrix} f \\ M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -L & L & L & -L \\ -L & -L & L & L \\ -\kappa & \kappa & -\kappa & \kappa \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}$$

3. **Aerodynamic Coefficients**:
   - The paper defines a single ratio coefficient $c_{\tau f}$ relating torque to thrust.
   - The simulator specifies separate lift coefficient c_f and drag coefficient c_d.
   - **Modification**: We define $\kappa = \frac{c_d}{c_f}$ to match the term $c_{\tau f}$ used in the paper's derivation.

2.2.5 Geometric controller for the UAV

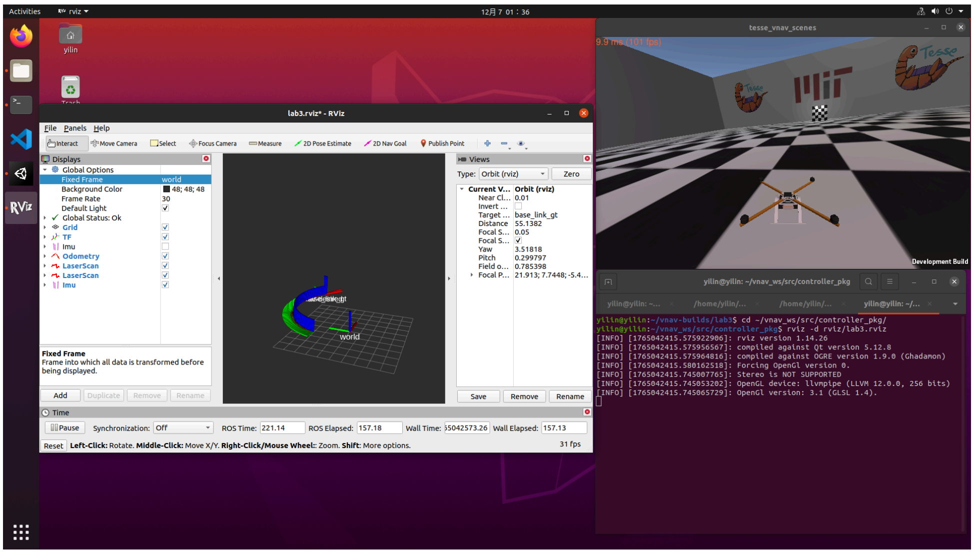


Figure 7: UAV trajectory tracking in TESSE simulator

The figure above demonstrated the successful implementation of the geometric controller for trajectory tracking in the TESSE simulator and the detailed implementation is described below.

Geometric controller part is encapsulated within the `controlLoop` function of the `controllerNode` class. The control logic follows the tracking control on the Special Euclidean Group SE(3) proposed by Lee et al., adapted for the ENU (East-North-Up) reference frame used in the TESSE simulator.

The control process is divided into four main steps:

1. **Translational Dynamics Control**
   First, we compute the position and velocity errors:

   $$e_x = x - x_d, \quad e_v = v - v_d$$

   The desired force vector F_{des} is computed to stabilize the translational error. Note that unlike the reference paper which assumes a Z-down frame, our simulation uses a Z-up frame. Therefore, the gravity compensation term is positive ($+mge_3$) to provide an upward force:

   $$F_{\text{des}} = -k_x e_x - k_v e_v + mge_3 + ma_d$$

   This vector defines the desired direction of the body z-axis (b_{3d}), which represents the thrust direction:

   $$b_{3d} = \frac{F_{\text{des}}}{\|F_{\text{des}}\|}$$

2. **Attitude Generation**
   We construct the desired rotation matrix $R_d = [b_{1d}, b_{2d}, b_{3d}]$ to align the thrust vector while maintaining the desired yaw angle ψ_d.

   Let $b_{1d}^{\text{des}} = [\cos\psi_d, \sin\psi_d, 0]^T$ be the desired heading. The orthogonal body axes are computed via cross products:

   $$b_{2d} = \frac{b_{3d} \times b_{1d}^{\text{des}}}{\|b_{3d} \times b_{1d}^{\text{des}}\|}, \quad b_{1d} = b_{2d} \times b_{3d}$$

   This ensures $R_d \in \mathrm{SO}(3)$ is orthonormal and respects the desired yaw constraint projected onto the plane perpendicular to the thrust.

3. **Rotational Dynamics Control**
   We calculate the attitude error e_R and angular velocity error e_Ω in the body frame. The function `Vee()` implements the map from $so(3)$ to $\mathbb{R}^3$:

   $$e_R = \frac{1}{2}\left(R_d^T R - R^T R_d\right)^\vee$$

   $$e_\Omega = \Omega - R^T R_d \Omega_d \approx \Omega \quad (\text{Assuming } \Omega_d \approx 0 \text{ for stabilization})$$

   The control outputs are the scalar total thrust f and the moment vector M:

   $$f = F_{\text{des}} \cdot (Re_3)$$

$$M = -k_R e_R - k_\Omega e_\Omega + \Omega \times (J\Omega)$$

4. **Control Allocation**
   Finally, we map the computed wrench (f, M_x, M_y, M_z) to individual rotor speeds. The simulator simulates a quadrotor in an `X` configuration. The relationship between rotor speeds squared (ω_i^2) and the wrench is modeled by the mixing matrix $\mathcal{A}$:

$$\begin{bmatrix} f \\ M_x \\ M_y \\ M_z \end{bmatrix} = \mathcal{A} \cdot \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}$$

   In our implementation (`F2W` matrix), considering the arm length projection $L = \frac{d}{\sqrt{2}}$ and the aerodynamic coefficients c_f, c_d:

$$\mathcal{A} = \begin{bmatrix} c_f & c_f & c_f & c_f \\ Lc_f & Lc_f & -Lc_f & -Lc_f \\ -Lc_f & Lc_f & Lc_f & -Lc_f \\ c_d & -c_d & c_d & -c_d \end{bmatrix}$$

   We solve this linear system using `F2W.colPivHouseholderQr().solve(W)` to obtain the squared angular velocities, then compute the square root (preserving signs) to get the final commands sent to the `/rotor_speed_cmds` topic.

3 Reflection and Analysis

In this lab, we bridged the gap between theoretical nonlinear control and practical implementation on a robotic system. Several key insights and challenges emerged during the process:

1. **The Importance of Coordinate Consistency**
   One of the most critical challenges was reconciling the mathematical conventions in Lee et al.'s paper (NED frame, `+` configuration) with the simulation environment (ENU frame, `X` configuration).
   - We observed that a direct transcription of the paper's formulas led to immediate instability.
   - Specifically, the gravity compensation term required a sign inversion ($+mge_3$) to account for the Z-up frame.
   - Furthermore, the mixing matrix `F2W` had to be re-derived for the `X` configuration. Incorrect signs in the mixing matrix caused the drone to flip immediately upon takeoff, highlighting that geometric correctness is useless without correct physical mapping.

2. **Numerical Stability of Rotations**
   We encountered significant high-frequency vibrations during initial testing. Through analysis, we identified the cause as numerical errors in the quaternion representation.

- The `Eigen::Quaternion` constructor does not automatically normalize the input. When converting `nav_msgs::Odometry` (which contains slight sensor/integration noise) to a rotation matrix without normalization, the resulting matrix R was no longer strictly orthogonal ($R \notin \mathrm{SO}(3)$).
- This violated the geometric controller's assumption, causing the error term e_R to generate erroneous feedback. Adding an explicit `eigen_quat.normalize()` step completely eliminated the vibrations, proving that strict adherence to mathematical constraints is vital in geometric control.

3. **Cascaded Control Dynamics**
   The tuning process revealed the distinct time-scale separation required for the cascaded architecture.
   - The inner loop (attitude control, gains k_R, k_Ω) must be significantly faster than the outer loop (position control, gains k_x, k_v).
   - If k_x was too high relative to k_R, the drone would request aggressive attitude changes that it could not track, leading to overshoot. We found that a ratio where attitude dynamics are roughly 3-5 times faster than position dynamics yielded the most stable tracking performance.

4 Conclusion

This lab successfully demonstrated the end-to-end workflow of an autonomous aerial robotics system, ranging from mathematical modeling to `C++` software implementation.

We first derived the solutions for Polynomial Trajectory Optimization, verifying that high-order polynomials are necessary to satisfy boundary constraints and ensure smoothness for aggressive maneuvers. We then implemented a Geometric Tracking Controller on $\mathrm{SE}(3)$, which avoids the singularities associated with Euler angles and provides almost global asymptotic stability.

By integrating these components into the ROS / TESSE ecosystem, we successfully navigated a quadrotor through a complex race course. The experiments highlighted that while mathematical derivation provides the foundation, robust robot autonomy relies equally on handling implementation details—such as coordinate transformations, numerical conditioning, and proper gain tuning. The resulting system is capable of precise dynamic tracking, validating the effectiveness of geometric control for underactuated mechanical systems.

5 Source Code

- ***controller_node.cpp***

```cpp
#include <ros/ros.h>

#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/utils.h>
#include <mav_msgs/Actuators.h>
#include <nav_msgs/Odometry.h>
#include <trajectory_msgs/MultiDOFJointTrajectoryPoint.h>
#include <cmath>

#define PI M_PI

#include <eigen3/Eigen/Dense>
#include <tf2_eigen/tf2_eigen.h>

class controllerNode{
  ros::NodeHandle nh;

  // PART 1: Declare ROS callback handlers
  ros::Subscriber des_state_sub, cur_state_sub;
  ros::Publisher propeller_speeds_pub;
  ros::Timer control_timer;

  // Controller parameters
  double kx, kv, kr, komega;

  // Physical constants (we will set them below)
  double m;              // mass of the UAV
  double g;              // gravity acceleration
  double d;              // distance from the center of propellers to the c.o.m.
  double cf,             // Propeller lift coefficient
         cd;             // Propeller drag coefficient

  Eigen::Matrix3d J;     // Inertia Matrix
  Eigen::Vector3d e3;    // [0,0,1]
  Eigen::MatrixXd F2W;   // Wrench-rotor speeds map

  // Controller internals
  // Current state
  Eigen::Vector3d x;     // current position of the UAV's c.o.m. in the world frame
```

```cpp
  Eigen::Vector3d v;     // current velocity of the UAV's c.o.m. in the world frame
  Eigen::Matrix3d R;     // current orientation of the UAV
  Eigen::Vector3d omega; // current angular velocity of the UAV's c.o.m. in the *body* frame

  // Desired state
  Eigen::Vector3d xd;    // desired position of the UAV's c.o.m. in the world frame
  Eigen::Vector3d vd;    // desired velocity of the UAV's c.o.m. in the world frame
  Eigen::Vector3d ad;    // desired acceleration of the UAV's c.o.m. in the world frame
  double yawd;           // desired yaw angle

  double hz;             // frequency of the main control loop

  static Eigen::Vector3d Vee(const Eigen::Matrix3d& in){
    Eigen::Vector3d out;
    out << in(2,1), in(0,2), in(1,0);
    return out;
  }

  static double signed_sqrt(double val){
    return val > 0 ? sqrt(val) : -sqrt(-val);
  }

public:
  controllerNode():e3(0,0,1),F2W(4,4),hz(1000.0){
      // PART 2: Initialize ROS callback handlers
      xd = Eigen::Vector3d::Zero();
      vd = Eigen::Vector3d::Zero();
      ad = Eigen::Vector3d::Zero();
      yawd = 0.0;
      kx, kv, kr, komega = 0, 0, 0, 0;

      des_state_sub = nh.subscribe("desired_state", 1, &controllerNode::onDesiredState, this);
      cur_state_sub = nh.subscribe("current_state", 1, &controllerNode::onCurrentState, this);
      propeller_speeds_pub = nh.advertise<mav_msgs::Actuators>("/rotor_speed_cmds", 1);
      control_timer = nh.createTimer(ros::Duration(1.0/hz), &controllerNode::controlLoop, this);

      // PART 6: Tune your gains!
      nh.getParam("kx", kx);
      nh.getParam("kv", kv);
```

```
        nh.getParam("kr", kr);
        nh.getParam("komega", komega);
        ROS_INFO("Gain values:\nkx: %f \nkv: %f \nkr: %f \nkomega: %f\n", kx, kv, kr, komega);

        // Initialize constants
        m = 1.0;
        cd = 1e-5;
        cf = 1e-3;
        g = 9.81;
        d = 0.3;
        J << 1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0;

        double d_by_sqrt2 = d/std::sqrt(2.0);
        F2W <<
            cf,            cf,            cf,            cf,
            cf*d_by_sqrt2, cf*d_by_sqrt2,-cf*d_by_sqrt2,-cf*d_by_sqrt2,
           -cf*d_by_sqrt2, cf*d_by_sqrt2, cf*d_by_sqrt2,-cf*d_by_sqrt2,
            cd,           -cd,            cd,           -cd;
    }

    void onDesiredState(const trajectory_msgs::MultiDOFJointTrajectoryPoint& des_state){
        //  PART 3: Objective - fill in xd, vd, ad, yawd
        xd << des_state.transforms[0].translation.x,
              des_state.transforms[0].translation.y,
              des_state.transforms[0].translation.z;

        vd << des_state.velocities[0].linear.x,
              des_state.velocities[0].linear.y,
              des_state.velocities[0].linear.z;

        ad << des_state.accelerations[0].linear.x,
              des_state.accelerations[0].linear.y,
              des_state.accelerations[0].linear.z;

        tf2::Quaternion quat;
        tf2::fromMsg(des_state.transforms[0].rotation, quat);
        yawd = tf2::getYaw(quat);
    }

    void onCurrentState(const nav_msgs::Odometry& cur_state){
        // PART 4: Objective - fill in x, v, R and omega
        // Position
        x << cur_state.pose.pose.position.x,
            cur_state.pose.pose.position.y,
```

```
        cur_state.pose.pose.position.z;

    // Velocity
    v << cur_state.twist.twist.linear.x,
        cur_state.twist.twist.linear.y,
        cur_state.twist.twist.linear.z;

    // Orientation
    tf2::Quaternion quat;
    tf2::fromMsg(cur_state.pose.pose.orientation, quat);
    Eigen::Quaterniond eigen_quat(quat.w(), quat.x(), quat.y(), quat.z());
    eigen_quat.normalize();
    R = eigen_quat.toRotationMatrix();

    // Angular velocity
    Eigen::Vector3d omega_world;
    omega_world << cur_state.twist.twist.angular.x,
                   cur_state.twist.twist.angular.y,
                   cur_state.twist.twist.angular.z;

    omega = R.transpose() * omega_world;
}

void controlLoop(const ros::TimerEvent& t){
  Eigen::Vector3d ex, ev, er, eomega;
  // PART 5: Objective - Implement the controller!
  ex = x - xd; // position error
  ev = v - vd; // velocity error

  // Rd matrix
  Eigen::Vector3d F_des = -kx*ex - kv*ev + m*g*e3 + m*ad;
  Eigen::Vector3d b3d = F_des.normalized();
  Eigen::Vector3d b1d_desired(cos(yawd), sin(yawd), 0);

  Eigen::Vector3d b2d = (b3d.cross(b1d_desired)).normalized();
  Eigen::Vector3d b1d = (b2d.cross(b3d)).normalized();

  Eigen::Matrix3d Rd;
  Rd.col(0) = b1d;
  Rd.col(1) = b2d;
  Rd.col(2) = b3d;

  er = 0.5 * Vee(Rd.transpose() * R - R.transpose() * Rd); // Orientation error
  eomega = omega; // Rotation-rate error

```

```
    // Desired wrench
    double f = (-kx * ex + -kv * ev + m * g * e3 + m * ad).dot(R * e3);
    Eigen::Vector3d M = -kr * er - komega * eomega + omega.cross(J * omega);

    // Recover the rotor speeds from the wrench
    Eigen::Vector4d W;
    W << f, M.x(), M.y(), M.z();
    Eigen::Vector4d omega_sq = F2W.colPivHouseholderQr().solve(W);

    Eigen::Vector4d rotor_speeds;
    for (int i = 0; i < 4; i++) {
        rotor_speeds(i) = signed_sqrt(omega_sq[i]);
    }

    // Populate and publish the control message
    mav_msgs::Actuators control_msg;
    control_msg.angular_velocities.clear();
    for (int i = 0; i < 4; i++) {
        control_msg.angular_velocities.push_back(rotor_speeds(i));
    }
    propeller_speeds_pub.publish(control_msg);
  }
};

int main(int argc, char** argv){
  ros::init(argc, argv, "controller_node");
  controllerNode n;
  ros::spin();
}
```