Lab 3 Report: EXERCISES

Yilin Zhang, 23020036094, Group 31
School of Computer Science
Ocean University of China
Qingdao, China
zyl8820@stu.ouc.edu.cn

I. Individual

A. Deliverable - Transformations in Practice

A. MESSAGE VS. TF

It says: "Based on the documentation for tf2 and its package listing page, answer the following questions in a text file called `deliverable_1.txt`." [1] Well, I will put the questions to Appendix A.1, and post my answers below.

1 Assume we have the premise code below:

```
#include <tf2_geometry_msgs/
tf2_geometry_msgs.h>
tf2::Quaternion quat_tf;
geometry_msgs::Quaternion quat_msg
= ...;
```

I fond three methods on Quaternion Basics:

```
tf2::convert(quat_msg , quat_tf);
```

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

or for the other conversion direction, `tf2::toMsg`.

```
quat_msg = tf2::toMsg(quat_tf);
```

2 Assume we have the premise code below:

```
#include "tf2_geometry_msgs/
tf2_geometry_msgs.h"
tf2::Quaternion quat_tf;
quat_tf.setRPY(1.0, 0.0, 0.0);
geometry_msgs::Quaternion quat_msg;
```

We may use this method.[1]

```
tf2::convert(quat_tf, quat_msg);
```

3 I would use `getW()` in Quaternion class.[2]

```
TF2SIMD_FORCE_INLINE const tf2Scalar&
getW() const { return m_floats[3]; }
```

[1]According to Converting Datatypes, `tf2/transform_datatypes.h` doesn't have the quaternion helper functions (you should use `tf2::Quaternion` methods in conjunction with `tf2::convert()`)

[2]See member function documentation of `tf2::Quaternion Class` or the 346 line in file `Quaternion.h`.

B. CONVERSION

The following questions will prepare me for converting between different rotation representations in C++ and ROS. [1]

1 This single function call is discribed in tf2: Humble documentation. Here `a` is the object to get data from (it represents a rotation/quaternion).

```
template<class A>
double tf2::getYaw(const A &a)
```

2 We need two function calls to implement this: `tf2::fromMsg`[3] and `toRotationMatrix()`.

```
#include <tf2_eigen/tf2_eigen.h>
#include <Eigen/Geometry>
#include <geometry_msgs/Quaternion.h>

// Assume we have an Eigen Quaternion
container
Eigen::Quaterniond eigen_quat;

// We use
tf2::fromMsg(quat_msg, eigen_quat);
Eigen::Matrix3d rotation_matrix =
eigen_quat.toRotationMatrix();
```

B. Deliverable - Modelling and control of UAVs

A. STRUCTURE OF QUADROTORS

Set that we have the crossing x-axis, and y-axis, shown in Fig. 1. The length from each wing to x-axis or y-axis are set to L. Assume all the wings *1* to *4* has the same lift force, and at origen, the drone is horizontal.

We have the same yaw orientation Ψ for quadrotor *(a)*. To make it simple, the value are all set to 1.

$$\Psi = [1 \ 1 \ 1 \ 1]. \tag{1}$$

For the x-axis, due to the amperes right handed screw rule, wing *1* and *2* are positive, while wing *3* and *4* are negative. For example, $\tau_{\{x1\}} = F \times d = 1 \times L = L$. Therefore,

$$x = [L \ L \ -L \ -L]. \tag{2}$$

Similarly, for the y-axis, we have

$$y = [L \ -L \ -L \ L]. \tag{3}$$

Since wing *1* and *3* spin counter clockwise, they generate a torque with a upper direction. While wing *2* and *4* spin

[3]199-201 lines from `tf2_eigen.h`

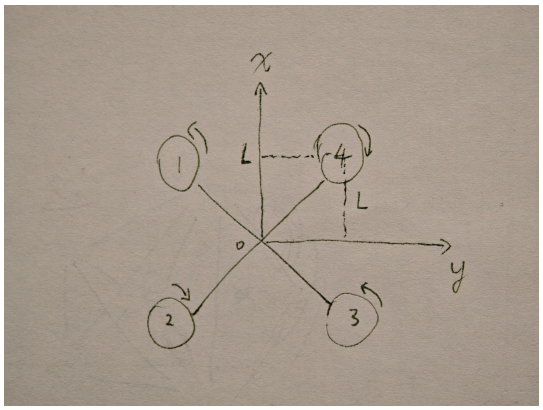


Fig. 1. My hand draw (qaq), with length L and coordinate system depicted.

clockwise, they are just the opposite. We cannot calculate the exact torque generated by each wing, so here we use c s.t. $\tau_o = c \cdot \tau_q$, where τ_q refers to the torque generated by any single wing, and τ_o is the influence of the quadrotor.

$$z = [-c \;\; c \;\; -c \;\; c]. \tag{4}$$

Since we got yaw orientation Ψ and position $[x, y, z]$, we can write down the F matrix of quadrotor *(a)*:

$$F_a = \begin{bmatrix} 1 & 1 & 1 & 1 \\ L & L & -L & -L \\ L & -L & -L & L \\ -c & c & -c & c \end{bmatrix}. \tag{5}$$

For quadrotor *(b)*, the rotation direction of wing *3* and *4* are opposite to the ones of quadrotor *(a)*. Similarly, we can write down the F matrix of quadrotor *(b)*:

$$F_b = \begin{bmatrix} 1 & 1 & 1 & 1 \\ L & L & -L & -L \\ L & -L & -L & L \\ -c & c & c & -c \end{bmatrix}. \tag{6}$$

You can see that only position z is different. Through calculation in (10) and (11), we get rank of both matrices. [2]

$$\begin{cases} \text{rank\_a} = 4 \\ \text{rank\_b} = 3 \end{cases} \tag{7}$$

B. CONTROL OF QUADROTORS

The standard position controller computes a desired net external force F_{des} (expressed in the inertial frame) to achieve the desired acceleration. However, the **true dynamics** of the system are:

$$m\ddot{p} = \underbrace{fRe_3}_{\text{thrust}} - mge_3 + \underbrace{F^I_{\text{drag}}\}}_{\text{drag (inertial frame)}} \quad . \tag{8}$$

To ensure the actual acceleration $\ddot{p}$ matches the controller's desired acceleration, the thrust fRe_3 must not only supply F_{des} but also counteract the drag force. Therefore, F_{des} must be adjusted accordingly.

From onboard sensors we get all v_x, v_y, v_z, thus calculate $\left(v^b\right)^2 = \begin{bmatrix} -v^b_x|v^b_x| \\ -v^b_y|v^b_y| \\ -v^b_z|v^b_z| \end{bmatrix}$.

$$F_{\text{adj}} = F_{\text{des}} - F^I_{\text{drag}}. \tag{9}$$

II. Team

Follow appendix Appendix B.1 and Appendix B.2 for the set up.

A. Deliverable - Geometric controller for the UAV

The implementation is strongly based on the publication [3]. A lot of functions can be find as equations here. The whole job is divided into six parts, with strongly recommended suggestions in each part. Yet the introduction also says "but not mandatory if you have better alternatives in mind:)". Awsome… Basicly, we are implementing a controller node called `controllerNode`, which is a class. The class has its field, a constructor, mothods `onDesiredState()`, `onCurrentState()`, and `controlLoop()`.

Part 1 was put in the declaration field. Here, we were asked to declare two subscribers, one publisher and a timer. This part had nothing special. I would put all my code in the appendices.

In *part 2*, we focused on the constructor, and initialized our handlers from *part 1*. Specifically bind `controllerNode::onDesiredState()` to the topic "desired_state", bind `controllerNode::onCurrentState()` to the topic "current_state", and bind `controllerNode::controlLoop()` to the created timer, at frequency given by the "hz" variable. Noticed a `ROS::nodeHandle nh` variable was already available as a class member, `nh.subscribe()` was the method chosen for the subscribers.

Part 3 was all about method `onDesiredState()`. We filled xd, vd, ad and yawd using "v << vx, vy, vz;" to fill a vector with Eigen.

Part 4 continued with the method `onCurrentState()`. The job here was to fill the current state variables x, v, R, and omega from the incoming odometry message. For position and linear velocity, it was straightforward to copy from `cur_state.pose.pose.position` and `cur_state.twist.twist.linear`. For the orientation matrix R, we had to convert the quaternion in the message to a rotation matrix. I used `tf2::Matrix3x3` to do that. The tricky part was the angular velocity omega. The message gave it in the world frame, but the paper and our code needed it in the body frame. So we transformed it by multiplying with the transpose of R.

Part 5 was the biggest part, where we actually implemented the geometric controller in the `controlLoop()` method. It followed the paper's equations step by step. First, we computed the position and velocity errors ex and ev. Then we computed the desired force vector F_des, and from that, we got the desired body z-axis b3d. Using the desired yaw, we constructed a desired body x-axis b1d, and then used cross products to get a right-handed coordinate system for the desired orientation matrix Rd. After that, we computed the orientation error er and the angular velocity error eomega. The next step was to compute the total thrust force f and the control moment M. Finally, we needed to convert this desired wrench (force and moment) into actual propeller speeds. This

involved solving a linear system using the F2W matrix we built in the constructor, and then taking the signed square root of the result. The last bit was to pack these four rotor speeds into a ROS message and publish it.

Part 7 was not listed in the original instructions, but it was in the code. It was about building the F2W matrix. This matrix maps the four rotor speed squares to the total thrust and three body moments. Its coefficients came from the geometry of the quadrotor (arm length d) and the aerodynamic coefficients cf and cd. Basically, it summed up how much force and torque each propeller contributed. We computed a `d_by_sqrt2` term for convenience and then filled the 4x4 matrix. Without initializing this matrix, the solver in part 5 would have nothing to work with and crash.

That sums up the implementation. The rest was just the main function to start the node and let ROS spin.

Then we comes to *Part 6*, which was about tuning the controller gains. The gains kx, kv, kr, komega were loaded from the ROS parameter server in the constructor. This way, we could change them easily using a launch file without recompiling the code. The constructor printed them out with `ROS_INFO` so we could check. I tried "4, 4, 8, 4" as the initial values, surprisingly, it worked very well. Therefore, I just need to fine-tune them a bit. "kx" means how much the drone wants to go towards the desired position. "kv" means how much the drone wants to match the desired velocity. "kr" means how much the drone wants to go correct its posture errors. "komega" means how much the drone wants to match the desired angular velocity. In this lab's situation, the desired drone was simple. There is no need to bahave presicely, "kx" and "komega" could be lower. And on the opposite, "kv" and "kr" coule be a little bit higher. After a 6 hours of tuning, I got "3.75, 4.5, 10.0, 3.75".

I got a video showing the quadrotor completing one round of the circular trajectory in `rviz`, and put the link on my Google Drive here.

III. Some Words

This lab was terrible. I spent 3 weeks just to encourage myself, I have no time to waste and I need to start somewhere. Thanks for the delay of submission, I was able to finish it in time. Facing the deadline staring at me next Tuesday, I finally start to do the team part this Wednesday. All I got was a bunch of code that I had to understand and modify and a publicated paper. I hate reading papers, I cannot understand a word! I's horrible! Then my teemmate appeared. They shout "What is going on!", "We are all waiting for your part! Come on! What are you waiting for?", something like that. Dear brother, I have my own stuff, okay? I also got a lot of homework to do, I had to deal with my individual tasks first, I also need to prepare this Saturdat's show, so that my team member there are not going to blame me.

Anyway, also in this lab report, I learned a new paper writing language `typst`. And this report was my first homework submitted using `typst`. This language is definitely new and fresh, with much easier grammer than `latex`. It has some little problems, such as no `#thanks` option, no `#appendix` nor `#appendices`. I managed to deal with them, searching for alternatives and finding a way to work around them. And I got this, as you can see. It is hard to tell if it worth to replace `typst` from `latex`, but `typst` definately worth a try.

References

[1] Massachusetts Institute of Technology, "MIT16.485 - Visual Navigation for Autonomous Vehicles." Accessed: Nov. 30, 2025. [Online]. Available: https://vnav.mit.edu/labs_2023/lab3/exercises.html

[2] 李烈, 彻底搞懂矩阵的秩！. Accessed: Dec. 04, 2025. [Online Video]. Available: https://www.bilibili.com/video/BV1gp421R7VN

[3] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor UAV on SE(3)," *49th IEEE Conference on Decision and Control (CDC)*, pp. 5420–5425, 2010, doi: 10.1109/CDC.2010.5717652.

Appendix A
Individual

A. *Deliverable - Transformations in Practice*

A. MESSAGE VS. TF

`1` 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.

`2` 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.

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

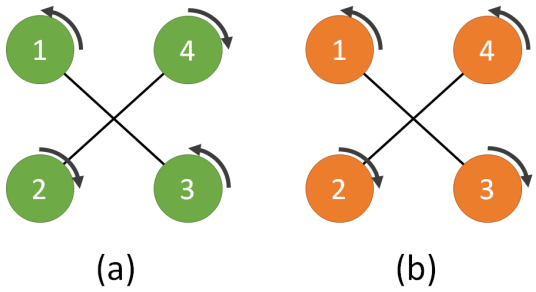
B. CONVERSION

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

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

B. *Deliverable - Modelling and control of UAVs*

A. 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.

$$
\begin{aligned}
F_a &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ L & L & -L & -L \\ L & -L & -L & L \\ -c & c & -c & c \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & -2 & -2 \\ 0 & -2 & -2 & 0 \\ 0 & 2 & 0 & 2 \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
\end{aligned} \tag{10}
$$

$$
\begin{aligned}
F_b &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ L & L & -L & -L \\ L & -L & -L & L \\ -c & c & c & -c \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & -2 & -2 \\ 0 & -2 & -2 & 0 \\ 0 & 2 & 2 & 0 \end{bmatrix} \\
&\Rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\end{aligned} \tag{11}
$$

B. CONTROL OF QUADROTORS

Assume that empirical data suggest you can approximate the drag force (in the body frame) of a quadrotor body as:

$$
F^b = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \left(v^b\right)^2 \tag{12}
$$

With $\left(v^b\right)^2 = \left[-v_x^b|v_x^b|, -v_y^b|v_y^b|, -v_z^b|v_z^b|\right]^\top$, 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 [3]), describe how you could use the information above to improve the tracking performance.

APPENDIX B
TEAM

A. *Trajectory tracking for UAVs*

`GETTING THE CODEBASE`

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

```
cd ~/labs
git pull
```

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

```
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
```

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

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

B. *Launching the TESSE simulator with ROS bridge*

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

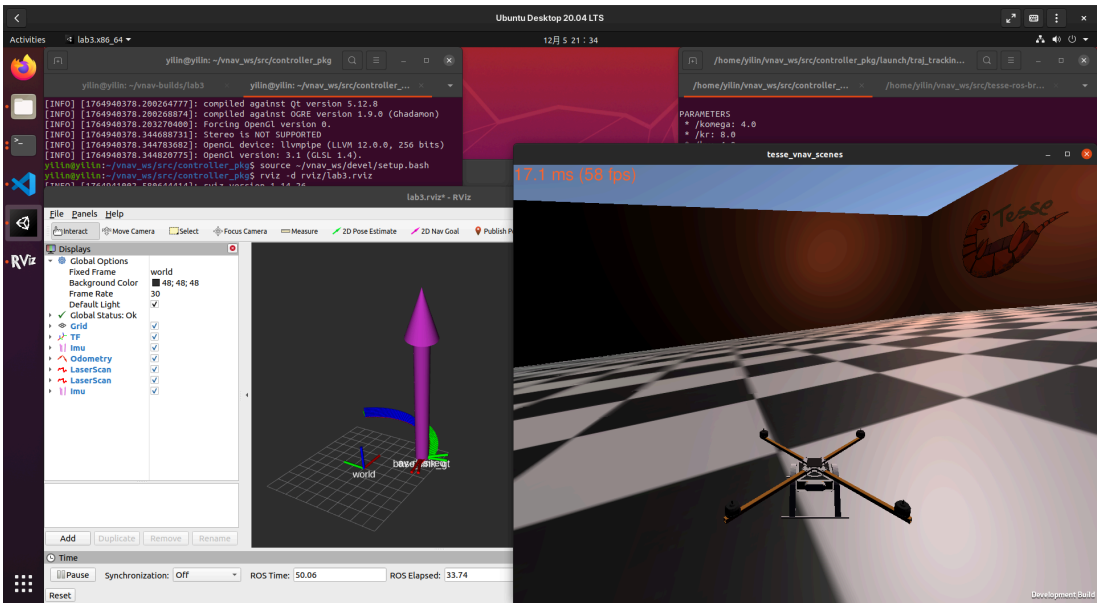
```
roslaunch tesse_ros_bridge
tesse_quadrotor_bridge.launch
```

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

C. *Deliverable - Geometric controller for the UAV (50 pts)*

After reading the reference [3] - *thoroughly* - get together with your teammates and open the source file `controller_node.cpp`. In this file, you will find detailed instructions to fill in the missing parts and make your quadrotor fly! To have full credits for this part of the lab, your team needs to complete the following:

- Implement all missing parts in `controller_node.cpp` and ensure everything compiles.
- Tune parameters so that the quadrotor achieves stable circular trajectory.
- A video showing the quadrotor completing one round of the circular trajectory in `rviz` (can either be a screen capture, or a video shot using your phone). Please upload the video onto either Google drive or Dropbox, generate



a publicly viewable link, and put the link in a text file called `rviz_drone_flight.txt` in your repo.

```
catkin build # from ~/vnav_ws
```

```
roslaunch controller_pkg traj_tracking.launch
```

a) *`controller_node.cpp`*:

```
// PART 1
ros::Subscriber des_state_sub, cur_state_sub;
ros::Publisher propeller_speeds_pub;
ros::Timer control_timer;
```

```
// PART 2
des_state_sub = nh.subscribe("desired_state",
1024, &controllerNode::onDesiredState, this);
cur_state_sub = nh.subscribe("current_state",
1024, &controllerNode::onCurrentState, this);
propeller_speeds_pub =
nh.advertise<mav_msgs::Actuators>("rotor_speed_cmds",
1024);
control_timer =
nh.createTimer(ros::Duration(1.0/hz),
&controllerNode::controlLoop, this);
```

```
// PART 3
// 3.1
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;
```

```
// PART 3
// 3.2
tf2::Quaternion quat;
tf2::fromMsg(des_state.transforms[0].rotation,
quat);
yawd = tf2::getYaw(quat);
```

```
// PART 4
// 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);

tf2::Matrix3x3 tfm(quat);
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    R(i,j) = tfm[i][j];
  }
}

// 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;
```

```
// PART 5
// 5.1
ex = x - xd;
ev = v - vd;
```

```
// PART 5
// 5.2
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;
```

```
// PART 5
// 5.3
er = Vee(0.5 * (Rd.transpose() * R -
R.transpose() * Rd));
eomega = omega;
```

```
// PART 5
// 5.4
double f = F_des.dot(R * e3);
Eigen::Vector3d M = -kr*er - komega*eomega +
omega.cross(J * omega);
```

```
// PART 5
// 5.5
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]);
}
```

```
// PART 5
// 5.6
mav_msgs::Actuators control_msg;
control_msg.angular_velocities.resize(4);
for (int i = 0; i < 4; i++) {
  control_msg.angular_velocities[i] =
rotor_speeds(i);
}
propeller_speeds_pub.publish(control_msg);
```

```
// PART 7
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;
```

```
<!-- Controller gains -->
<param name="kx" type="double" value="3.75">
<param name="kv" type="double" value="4.5">
<param name="kr" type="double" value="10.0">
<param name="komega" type="double"
value="3.75">
```