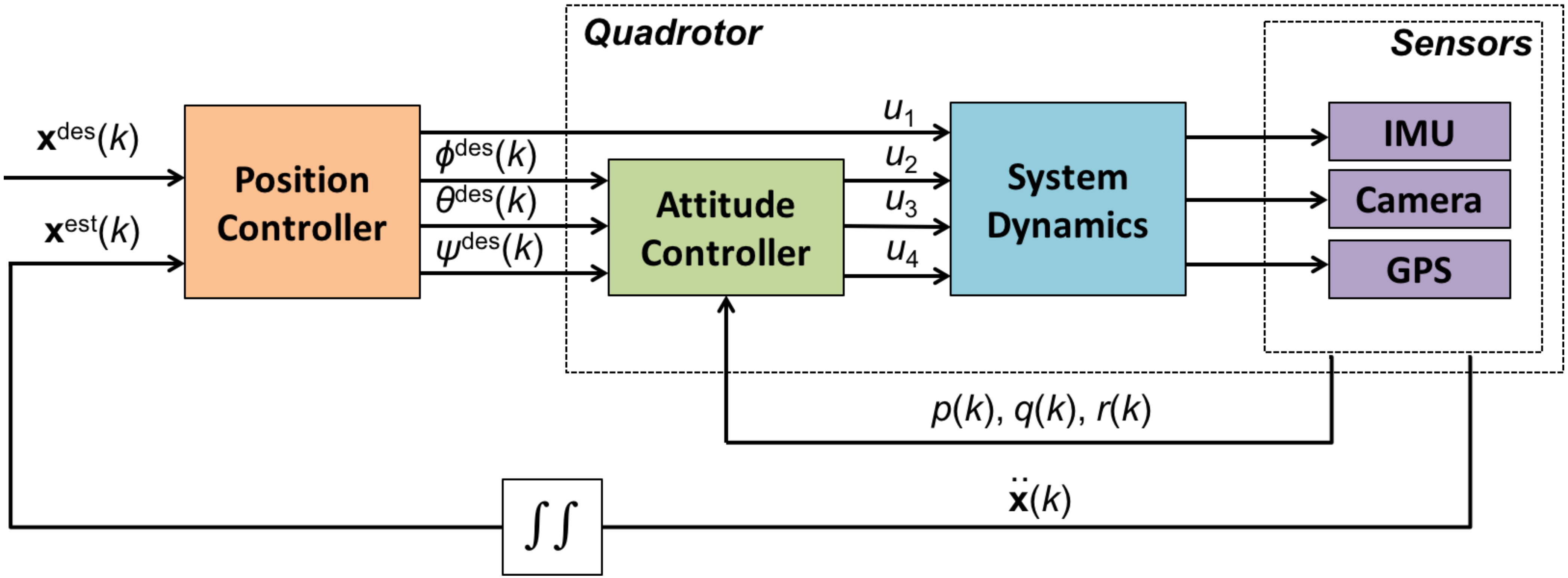
Quadrotor Control Architecture



Crazyflie 2.0

- Flight time
  - ~7min
- Max payload:
  - 15g
- Microcontroller
  - STM32F405 main application MCU (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash)
- Sensors
  - 3 axis gyro
  - 3 axis accelerometer
  - 3 axis magnetometer
  - High precision pressure sensor
- Communication
  - 2.4GHZ via custom USB dongle
  - Address Format:
    - radio://0/80/2M
    - Flie 0 on channel 80 at 2Mbit/s
  - Range: about 1 mile

OPEN-SOURCE

Topics

- /crazyflie/imu
  - sensor_msgs/Imu
  - updates every 10ms
- /crazyflie/cmd_vel
  - geometry_msgs/Twist
  - linear.y: roll {-30 to 30 degrees}
  - linear.x: pitch {-30 to 30 degrees}
  - angular.z: yawrate {-200 to 200 degrees/s}
  - linear.z: thrust 10000 to 60000

Flie Addresses

Team1	radio://0/80/2M
Team2	radio://0/81/2M
Team3	radio://0/82/2M
Team4	radio://0/83/2M
Team5	radio://0/84/2M
Team6	radio://0/85/2M
Team7	radio://0/86/2M
Team8	radio://0/87/2M
Team10	radio://0/88/2M

Connect to your flie for the first time

- Plug in the CrazyRadio dongle
- Turn on your flie
- Open a terminal
  - ***source ~/catkin_ws/devel/setup.bash***
  - ***rosrun crazyflie_tools scan*** *{Not necessary, just to make sure Crazyradio is working properly}*
  - ***roslaunch crazyflie_demo teleop_xbox360.launch uri:=radio://0/X/2M*** *{where **X** is your flie's channel number}*

ROS Parameter Server

- A parameter server is a shared, multi-variate dictionary that is accessible via network APIs.
- Nodes use this server to store and retrieve parameters at runtime.
- Supported Parameter Types
  - *32-bit integers*
  - *booleans*
  - *strings*
  - *doubles*
  - *iso8601 dates*
  - *lists*
  - *base64-encoded binary data*

- Set/Get Paramaters with roscpp
  - *nh.setParam("pid_rate/pitch_kp",kp)*
  - *nh.getParam("pid_rate/pitch_kp",kp)*

- Set/Get Paramaters with rospy
  - *rospy.set_param('pid_rate/pitch_kp',kp)*
  - *kp= rospy.get_param('pid_rate/pitch_kp')*

ROS Parameter Server(Cont’d)

- Crazyflie pid controller parameter names (Pitch and Roll)
  - *pid_rate/pitch_kp*
  - *pid_rate/pitch_kd*
  - *pid_rate/pitch_ki*
  - *pid_rate/roll_kp*
  - *pid_rate/roll_kd*
  - *pid_rate/roll_ki*

Crazyflie Minimal Launch

```
<?xml version="1.0"?>
<launch>
  <arg name="uri" default="radio://0/80/2M" />
 <include file="$(find crazyflie_driver)/launch/crazyflie_server.launch">
  </include>
  <group ns="crazyflie">
    <include file="$(find crazyflie_driver)/launch/crazyflie_add.launch">
      <arg name="uri" value="$(arg uri)" />
      <arg name="tf_prefix" value="crazyflie" />
      <arg name="enable_logging" value="True" />
    </include>
  </group>
 <node pkg="rviz" type="rviz" name="rviz" args="-d $(find crazyflie_demo)/launch/crazyflie.rviz" />
 <!--Add your nodes here-->
</launch>
```

Fly Node(C++)

```cpp
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
geometry_msgs::Twist vel;

int main(int argc, char **argv)
{
  ros::init(argc, argv, "fly");
  ros::NodeHandle nh;
  ros::Publisher vel_pub;
  vel_pub = nh.advertise<geometry_msgs::Twist>("/crazyflie/cmd_vel", 1, true);
  ros::Rate loop_rate(200);
  vel.linear.z = 50000;//Set Thrust to fly vertically, Max=60000
```

```cpp
 while (ros::ok())
  {
   vel_pub.publish(vel);
   ros::spinOnce();
   loop_rate.sleep();
  }
  return 0;
}
```

Fly Node(Python)

```
#!/usr/bin/env python


import rospy
import time
from geometry_msgs.msg import Twist
class Fly():
    def __init__(self):
        rospy.init_node('fly', anonymous=False)
        rospy.on_shutdown(self.shutdown)
        self.cmd_vel = rospy.Publisher('/crazyflie/cmd_vel',
Twist,    queue_size=10)
        r = rospy.Rate(200)
        fly_cmd = Twist()
        fly_cmd.linear.z = 50000
```

```
while (not rospy.is_shutdown()):
            self.cmd_vel.publish(fly_cmd)
            r.sleep()


    def shutdown(self):
        rospy.loginfo("Stopping the Crazyflie...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)


if __name__ == '__main__':
    try:
        Fly()
    except:
        rospy.loginfo("fly node terminated.")
```

PID Tuner Starter Code(C++)

```
#include "ros/ros.h"
#include <sensor_msgs/Imu.h>
void imuCallback(const sensor_msgs::Imu::ConstPtr& imuData)
{
  //read & save Imu data
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "pid_tuner");
  ros::NodeHandle nh;
  ros::Publisher vel_pub;
  ros::Subscriber sub = nh.subscribe("/crazyflie/imu", 10, imuCallback);
  ros::Rate loop_rate(200);
  double kp =0;
  double kd =0;
```

```
 while (ros::ok())
 {
   nh.setParam("pid_rate/pitch_kp",kp);//setting kp gain for pitch controller
   nh.setParam("pid_rate/pitch_kd",kd);//setting kd gain for pitch controller


   //Set same tuned gains for roll controller after the tuning process has
finished
   nh.setParam("pid_rate/roll_kp",kp);//setting kp gain for roll controller
   nh.setParam("pid_rate/roll_kd",kd);//setting kd gain for roll controller
   ros::spinOnce();
   loop_rate.sleep();
 }
 return 0;
}
```

PID Tuner Starter Code (Python)

```
#!/usr/bin/env python
import rospy
import numpy as np
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Imu
class PIDTuner():
    def imuSubscriber(self,imu):
        #Read & Save imu data
```

```
def __init__(self):
        rospy.init_node('pid_tuner', anonymous=False)
        rospy.on_shutdown(self.shutdown)
        rospy.Subscriber('/crazyflie/imu', Imu, self.imuSubscriber)
        r = rospy.Rate(100)
        while (not rospy.is_shutdown()):
            rospy.set_param('pid_rate/pitch_kp',self.kp)
            rospy.set_param('pid_rate/pitch_kd',self.kd)
            r.sleep()
    def shutdown(self):
        rospy.loginfo("Stopping the pid_tuner...")
        rospy.sleep(1)
if __name__ == '__main__':
    try:
        PIDTuner()
    except:
        rospy.loginfo("pid_tuner node terminated.")
```

Reminder: Create a package for crazyflie in ROS

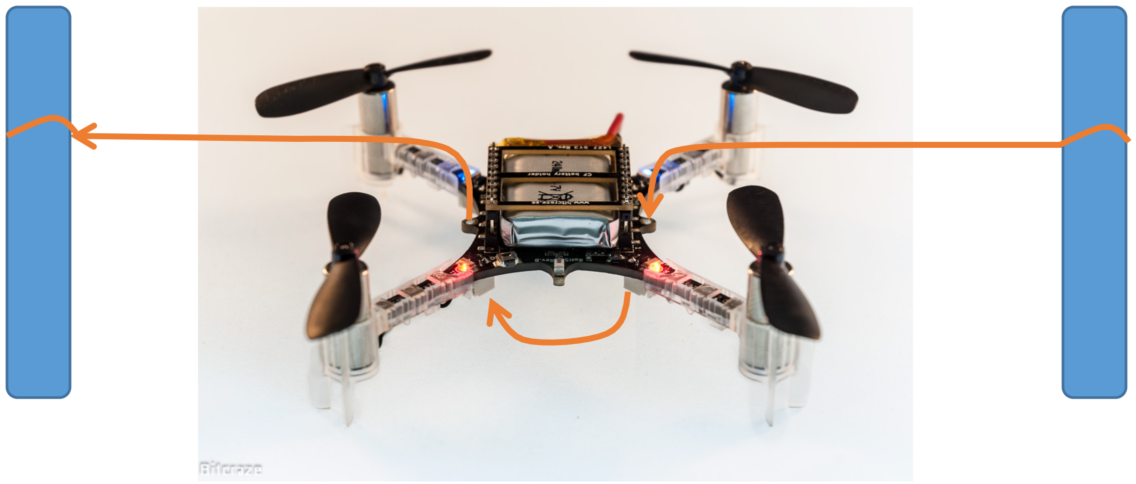
- Step 1: Create package
  - Open a terminal and cd into ~/catkin_ws/src
  - Create a package by typing in:
    - ***catkin_create_pkg your_pkg_name roscpp rospy***
- Step 2: Edit CmakeLists.txt if you intend to use the starter C++ code
  - Open the CMakeLists.txt file that is in the root of **your package**
    - Add the following lines somewhere
    - ***add_executable(fly src/fly.cpp)***
    - ***add_executable(pid_tuner src/pid_tuner.cpp)***
    - ***target_link_libraries(fly ${catkin_LIBRARIES})***
    - ***target_link_libraries(pid_tuner ${catkin_LIBRARIES})***
- Step 3: Copy fly.cpp and pid_tuner in the src folder inside **your package(not the workspace)**
- *Note: If you're using the python files there's no need to edit CMakeLists, just copy them in the src folder and make sure the files are executable.*

Use the minimal launch

- If you want to use the minimal.launch, make a directory called “launch” in your package and copy the file
- Make sure to change the default uri in the minimal launch file to the appropriate address that points to your own crazyflie
- You can either add your nodes to the minimal launch file or make sure to launch it before trying to run the nodes

AMR Lab #4 – Due 11/17/2016 during Lab hours

- Use IMU data on board the quadrotor to automatically tune the pitch PD controller such that the tethered quadrotor is hovering with no oscillations.
  - Specifically implement a **gradient descent method** that increases Kp until oscillations starts to appear. Your algorithm has to detect oscillations, stop increasing Kp, and start increasing Kd until the oscillations disappears.
  - The PD gains are initialized to 0



Pitch PID Tuner Logic

- Collect IMU data in 2 sec intervals and calculate the following over the collected samples:
  - Tracking error for linear acceleration, which is the mean distance from the target -> norm(Accl-Target_Accl)
  - Variance of angular velocity readings.
  - Note that you need to find thresholds for detecting stabilized and oscillating states of the system according to your measures for tracking and variance.
  - Combine the tracking and variance error into one measure of error which will be used as a factor in updating the gains
- Start from a small value for proportional gain of the pitch attitude controller, kp=0
- At each iteration, update the gain and check if the system is stabilized according to your thresholds, keep track of the best variance which will later be used for detecting oscillation
- When the system is stabilized keep increasing kp until the system is oscillating
- When oscillation is detected, keep the same kp and start increasing the derivative gain until the system is stabilized again

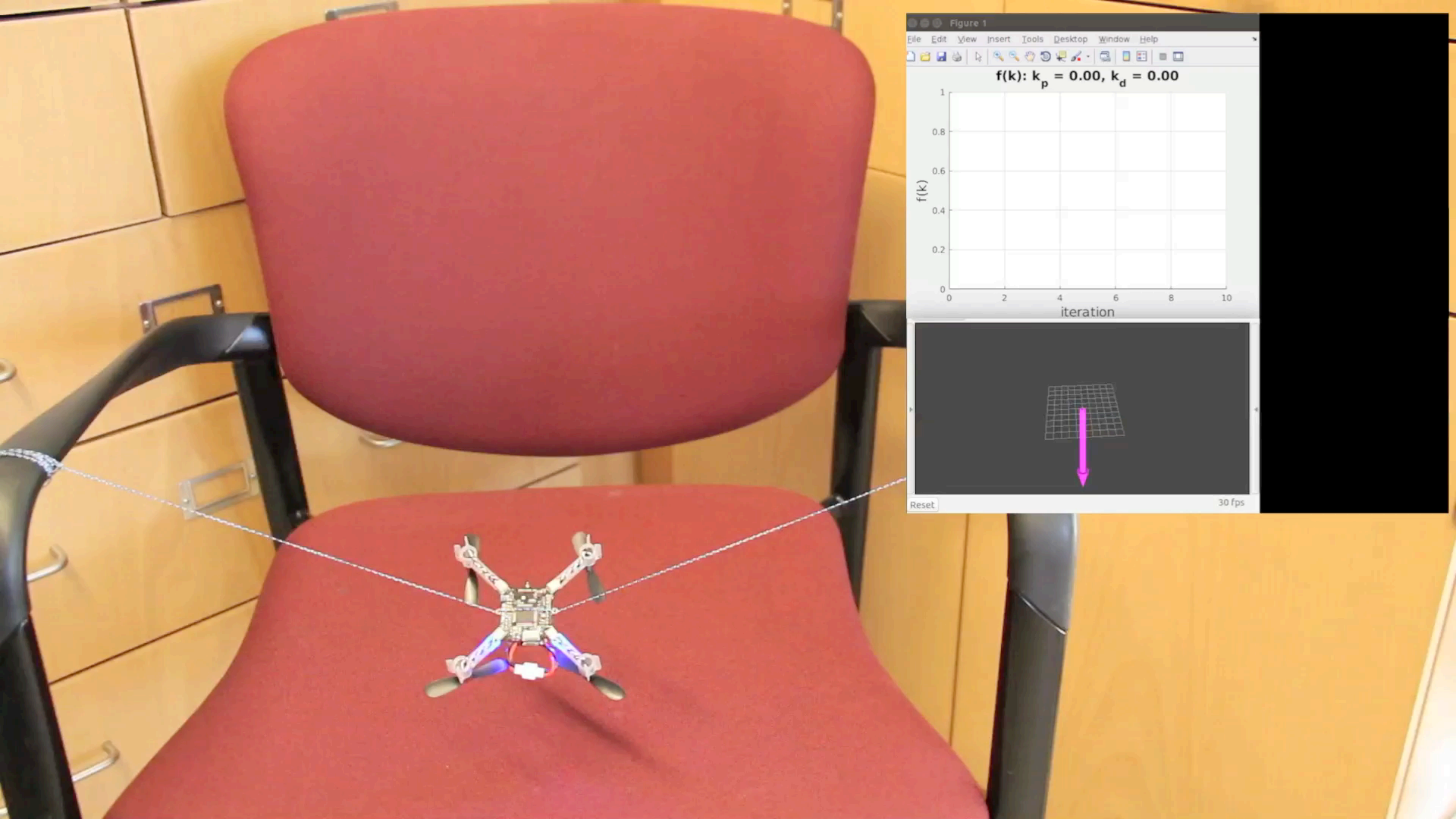


Figure 1
File Edit View Insert Tools Desktop Window Help
f(k): k_p = 0.00, k_d = 0.00
1
0.8
0.6
0.4
0.2
0
f(k)
0 2 4 6 8 10
iteration
Reset
30 fps