Engineering 100-Quadcopter Lab 3: Introduction to Sensor Signal Filtering

Two weeks are permitted for this assignment.

In this lab you will observe how noise and filtering can affect the flight of a quadcopter, by examining various filters that are used to process sensor readings. More specifically, you will

- observe how on-board filters can contribute to a stable and smooth flight and
- work with horizontal proximity sensors to enable your quadcopter to avoid obstacles, and tune filters to get a smooth response when avoiding obstacles.

1 Background Material

At this point you should have a good understanding of the sensors used in the quadcopter. You are learning about noise and filters in lecture. In this lab you will work with digital filters. Digital filters are used to process digital signals, which are either generated from sources that are digital to begin with (e.g., texts, pictures taken by a digital camera, measurements taken by instruments with a finite precision, etc.) or digitized from analog signals as we have seen in class, by following processes such as sampling (digitizing or discretization in time) and quantization (digitizing or discretization in value).

1.1 Noise and autonomous flight

For stable and responsive flight control, the quadcopter needs to know its orientation (i.e., roll, pitch, and yaw), speed, acceleration, and altitude at any time during flight. There are many types of sensors that can be mounted to facilitate a reliable controller, including the following.

- *Accelerometer*: used to measure acceleration, and integrated to estimate speed.
- *Gyroscope*: used to measure angular velocity, and integrated to obtain roll, pitch, and yaw.
- *Magnetometer*: used to measure orientation relative to the Earth's magnetic field; not reliable indoors due to large metallic structures.
- *Barometer*: used to measure altitude based on air pressure; not reliable indoors or with unstable weather conditions.
- *Ultrasonic range finder*: used to measure altitude using ultrasound sensors; more reliable than a barometer indoors, but can only be used at low (e.g., less than 7 m) altitudes.
- *GPS*: used to measure global location/speed; can only be used outdoors.

All of these sensors are prone to noise and interference. For instance, vibrations of the frame during flight can introduce noise into accelerometer and gyroscope readings, and thermal noise in analog signals can affect sensors with analog outputs. Luckily, one does not expect rapid changes in measurements like the altitude or orientation, and low-pass filters can be used to eliminate high frequency noise components, while retaining the desired low-frequency signals. Note, however, that cutting off signals at very low frequencies also prevents the controller from reacting to rapid changes in orientation/position, e.g., a sudden gust of wind pushing the quadcopter off course. Therefore, there is a general trade-off between noise reduction in the signal and controller responsiveness.

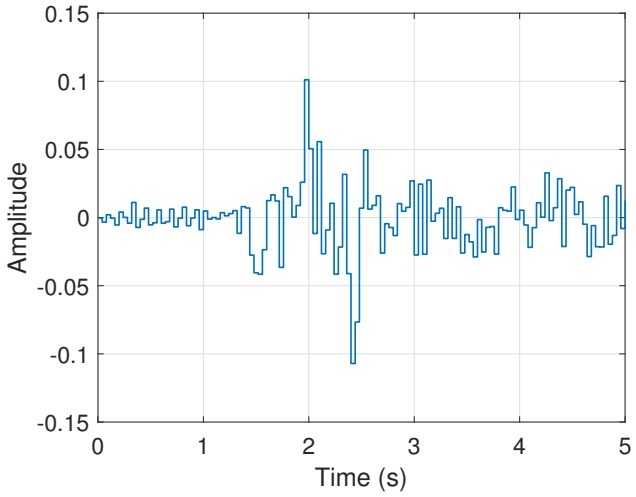
1.2 Representation of a digital signal

A digital signal can be represented by a sequence of numbers, denoted as $x[n]$, where n is used to index samples; typically we have $n = 0, 1, 2, \ldots$. The expression $x[1] = 2\,\mathrm{V}$ means that the signal has a magnitude of two volts at time step $n = 1$, but this could mean $t = 1\,\mathrm{s}$, or $t = 1\,\mathrm{ms}$, or any time unit of your choice. What is hidden here is the sampling rate that we used to obtain this digital signal (if we generated it from another, analog signal, or the actual rate of generation of the original digital signal). For this reason, when expressing a digital signal $x[n]$, we also often specify the sampling rate. For example, you may be given the following signal:

$$x[n] = 2\sin\left(\frac{2\pi n}{1000}\right), \quad n = 0, 1, 2, \ldots, \text{ with a sampling rate of } 1\,\mathrm{kHz}.$$

This would mean that we are taking 1,000 samples per second, or one sample per millisecond, implying that $x[1]$ is taken at time $t = 1\,\mathrm{ms}$, $x[2]$ is taken at $t = 2\,\mathrm{ms}$, and so on.

As an example, here is a plot of the output from one of the quadcopter's on-board gyroscopes. The amplitude is measured in deg/sec. Again note that when you only present a sequence of numbers, the time unit has to be supplied separately to fully define the signal; e.g., the sampling rate for this signal is 25 Hz (25 readings per second).



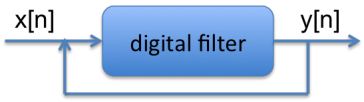
1.3 Digital filters

The goal of a digital filter is very similar to that of an analog filter; each is designed with some objective in mind: low-pass, high-pass, band-pass, and band-reject. The representation of a digital

filter can look a lot simpler, as essentially algebraic operations over successive signal samples, i.e., the input samples $x[n]$ step through the filter and generate output samples $y[n]$.

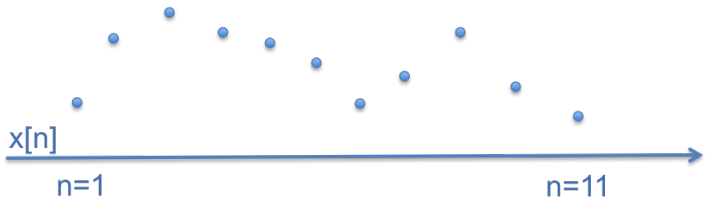
This could be done in two ways. For Finite Impulse Response (FIR) filters, the output samples $y[n]$ are only a function of the current and past input samples $x[n]$, as shown below.

Alternatively, the output samples $y[n]$ are functions of both the current/past input samples $x[n]$ and past outputs $y[n-1], y[n-2], \ldots$. This is an Infinite Impulse Response (IIR) filter.

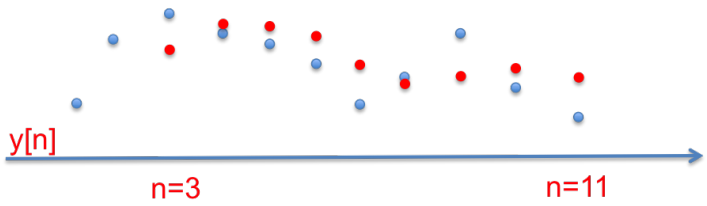


The most significant difference between the two is the presence of a (sometimes positive) feedback loop in the IIR filter. Conceptually, what this suggests is that an FIR filter only has a finite amount of memory (i.e., the output of the filter does not depend on inputs too far back in the past), whereas an IIR filter has infinite memory (i.e., every output can potentially depend on all previous inputs). Where do these filters get their names from, respectively? We will get to that and the implications of their differences shortly below.

The moving average filter We will begin with one of the simplest FIR filters, the moving average filter. The following figure shows a digital signal with 11 samples. You can observe ups and downs in this signal; one way to smooth out some of the ups and downs, especially if you believe some of it is due to noise, is to simply take the average of a few samples and use this average to replace the actual sample value, repeating this over and over. Averaging is very effective in smoothing out high frequency noises.



The next figure shows the result of averaging every three samples, and writing these averages to the output signal $y[n]$. This is called the *moving average* filter because the time window containing the three samples is moving as you slide along the x-axis, one sample at a time. More precisely: $y[3] = (x[1]+x[2]+x[3])/3$, $y[4] = (x[2]+x[3]+x[4])/3$, and so on. That is, each output y[n] is the average of the current input $x[n]$, plus two previous input values $x[n-1]$ and $x[n-2]$. In general, we have:

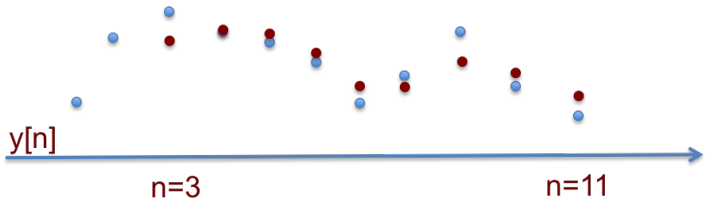


$$y[n] = \frac{1}{W}\left(x[n] + x[n-1] + \cdots + x[n-W+2] + x[n-W+1]\right),$$

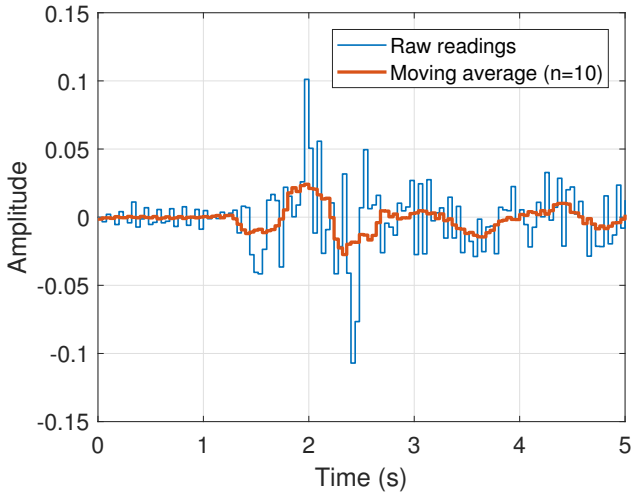
where W is called the *window size* of this moving average filter. This is a type of low-pass filters.

The larger the window size, the more the filter dampens the input samples, and the lower the cutoff frequency of the filter.

One can also place more emphasis on the more recent/current samples when performing this type of averaging: why treat input samples long in the past the same way as we treat more recent samples? The next figure shows a simple example of such, again with a window size of $W = 3$, but placing more weight on the current sample: $y[n] = 0.6x[n] + 0.2x[n-1] + 0.2x[n-2]$. As expected, compared to the previous filter, this one generates output that better tracks the input signal while smoothing out some of the extreme values.



The figure below illustrates the gyro sensor readings after going through a moving average filter with $W = 10$.



The general formula for FIRs and IIRs We can now extend the moving average filter to more general FIRs, where the filter output is simply a weighted linear combination of the current and past input samples.

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \cdots + b_Mx[n-M].$$

Similarly, the IIR filter output is a weighted linear combination of current and pass inputs as well as past outputs.

$$\begin{aligned} y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \cdots + b_Mx[n-M] \\ + a_1y[n-1] + a_2y[n-2] + \cdots + a_Ly[n-L]. \end{aligned}$$

Now we are ready to discuss nomenclature. Consider stepping through the signal samples $x[n] = 1, 0, 0, 0, 0, \ldots$ into the filters (this is also called an impulse function/signal, with an initial unit value followed by an indefinite sequence of zeros). Suppose you filter this signal using an FIR filter with arbitrary coefficients and some finite value M. After some initial non-zero outputs, the output will become zero and remain so, i.e., when the input signal dies down, eventually the output also dies down. The impulse response is bounded in time, or finite.

Now suppose you do this through an IIR filter, with $b_0 = a_1 = 1$ and with all other coefficients set to zero. What is going to happen? Even though there is only one non-zero sample in the input ($x[0] = 1$), this non-zero value shows up in the output sample $y[0] = b_0 x[0] = 1$, and is fed back into the filter to produce $y[1] = b_0 x[1] + a_1 y[0] = 1$, which is fed back again, producing $y[2] = a_1 y[1] = 1$, and on and on in perpetuity. The impulse response is not bounded in time, i.e., there is not some finite amount of time that it effects the output.

You might ask why do we need the feedback? The answer is that by using feedback we can often obtain far superior filter performance without explicitly needing a large memory (which in implementation translates to fewer operations and components). However, feedback in the signal loop can also be dangerous, especially if very large samples are possible.

2 Pre-Lab (30 Points)

You will have two weeks to complete this lab. Note the due dates on Canvas; the pre-lab will be due by 11:59pm the day before the lab on the first week, and you will have one week following the second week to complete the in-lab and post-lab.

Question 2.1. (10 points) ArduPilot uses a simple digital filter for accelerometer and gyroscope readings. The output of this filter is given by $y[n] = \alpha x[n] + (1 - \alpha) y[n - 1]$, where $0 \leq \alpha \leq 1$. Is this filter an IIR or an FIR filter? Why?

Question 2.2. (10 points) Assume that we provide the input $x[n] = 1$ for $n \geq 0$ to the filter described in Question 2.1. Further assume that $y[n] = 0$ for $n < 0$. It can be show using induction that $y[n] = 1 - (1 - \alpha)^{n+1}$ for $n \geq 0$. Using this result, what is the steady-state output of the filter, i.e., the output as $n \to \infty$? How long does it take for the output to reach 95% of the steady-state output? Your answer may be in terms of α.

Question 2.3. (10 points) Please provide a scenario where using the raw accelerometer and gyroscope readings without any filtering would cause the quadcopter to perform poorly. What kind of filter (i.e., low-pass or a high-pass) would you use to address this issue? Why?

3 In-Lab (40 Points)

This lab consists of two parts. In the first part you will examine how noise (e.g., that introduced by vibration) can affect IMU readings in a quadcopter, and how to eliminate noise using low-pass filters. Then you will observe the response of your quadcopter when using horizontal proximity sensors for obstacle avoidance and tune filters that can smoothen the response of the quadcopter.

For this lab, you may either keep your tuned PID values from the previous lab assignment or reset to the original values from the QC.params file. These were in-ideal for final performance because they were slower than needed. However, for this lab you will be investigating the effects of noise and filtering, and if the quadcopter takes off too quickly it may not have time to adjust properly, thus resulting in many unsuccessful flights.

3.1 The effects of vibration on flight control

Motors spinning at high speeds can cause high-frequency vibrations in the quadcopter. These vibrations are then registered by the on-board IMU, resulting in noisy accelerometer and gyroscope readings. The default flight controller eliminates most of this noise using simple low-pass filters. In this section you will disable these filters to examine their roles in maintaining a stable flight.

Note that ArduPilot implements filters at various stages in the software to eliminate noise. To fully observe the effects of noise, you must disable these filters that drive the generation of motor outputs based on noisy sensor measurements. In Mission Planner, set the `PSC_THR_CUTOFF` (the cutoff frequency of the filter applied to the target vertical acceleration) and `PSC_VEL_CUTOFF` (the cutoff frequency of the filter applied to the target vertical velocity) parameters to zero to disable these filters. Maker sure to write down the initial values for these parameters, so you can revert back to the default values at the end of the lab.

Now you will disable the filters that process readings directly out of the IMU. Conveniently, ArduPilot allows you to modify these filters directly from Mission Planner. The `INS_GYRO_FILTER` and `INS_ACCEL_FILTER` parameters specify the cutoff frequencies of filters applied to gyroscope and accelerometer, respectively. Set one of these parameters to zero at a time, and fly your quadcopter to observe the effect of disabling each filter on the flight controller. Specifically, look at whether the quadcopter can keep itself level (zero roll and pitch), and whether it can remain at a constant altitude.

Make sure to mod your board before each flight, and download the logs after the flight for inspection. After downloading the logs for both scenarios, look at the sonar readings and the target altitude (SALT and DSALT). In addition, when reducing the filtering on the `INS_ACCEL_FILTER`, evidence of response changes may be more easily observed in the throttle output variable (ThO) under the CTUN group than it is by visual observation. Likewise, when reducing the gyroscopic filtering, the individual motor PWM signals (Ch1, Ch2, Ch3, Ch4) in the RCOUT group, as well as the roll and pitch measurements (Roll, Pitch) in the ATT group are places where the effect of filtering changes may be seen more easily than what can be judged visually. Use your observations to comment on both cases below.

Checkpoint 1. (3 points) Show the logs that you collected to your instructor, and discuss your observations.

Question 3.1. (7 points) Compare the response of the quadcopter after disabling each of the filters described above. Use your observations during flight as well as the flight logs to describe the similarities and differences between the two cases.

Now set both `INS_GYRO_FILTER` and `INS_ACCEL_FILTER` to their initial values. Then start increasing each parameter one at a time until you start observing artifacts (e.g., vibrations, or an unstable altitude) when flying the quadcopter. Answer the following based on your findings.

Question 3.2. (6 points) Report the highest values for `INS_GYRO_FILTER` and `INS_ACCEL_FILTER` that result in a smooth flight.

3.2 Obstacle Detection Using LiDAR Sensors

This section consists of two parts. You will first learn how to setup your quadcopter's hardware and software for obstacle avoidance, using a sensor enclosure containing four horizontal (side-mounted) infrared (LiDAR) sensors[1] and a small Arduino microcontroller.[2] The microcontroller is then used to interface with the sensors, and send their measured distances to the BealgeBone. ArduPilot software will then use these readings to adjust the desired lean angles (roll and pitch) to avoid crashing into obstacles. You will then calibrate ArduPilot's obstacle avoidance parameters, along with filters for smoothening the sensor readings, the get a smooth response from the quadcopter when avoiding obstacles.

3.2.1 Preparing the sensors

You first need to configure the Arduino IDE to recognize the microcontroller used for interfacing with the sensors. Log into the lab station computer and launch the Arduino IDE from the online CAEN software package. Open Google Chrome and the CAEN apps list should pop up. In the Arduino IDE open the `File` menu and select `Preferences`. Now find the `Additional Boards Manager URLs` text box and copy and paste the following URL into this box:

```
https://files.pololu.com/arduino/package_pololu_index.json
```

Under `Tools` → `Board`, select `Boards Manager...`, search for and select `Pololu A-Star Boards`, and click `Install`. Select the installed board by going to `Tools` → `Board` and selecting `Pololu A-Star 32U4`.

You also need to install the library for interfacing with the sensors. From the `Sketch` menu select `Include Library`, and then select `Manage Libraries...`. Search for and install `VL53L1X by Pololu`.

[1] `https://www.pololu.com/product/3415`
[2] `https://www.pololu.com/product/3101`

Now disconnect the USB cable connecting the sensor enclosure and the BeagleBone, and connect it to your computer using a USB extension cable. Go to Tools → Ports. Make sure the COM port with the Pololu board is selected. Open the `VL53L1X.ino` sketch in the files provided with this lab and compile and load it onto the microcontroller. After making changes, make sure to hit the 'SAVE' button on the top left section of your screen. Next, press the 'UPLOAD' button to upload the code to the Arduino board. Make note of the `BAUD_RATE` parameter in the code. This specifies the speed of transmissions from the Arduino microcontroller to the BeagleBone.

You can check the sensors by opening the serial monitor (top-right corner of the Arduino IDE) and observing the measured distances. Put your hand in front of each sensor and ensure that the program is displaying the correct distance.

Checkpoint 2. (3 points) Ask your instructor to double check your setup before continuing.

Note that the program collects measurement from the sensors one at a time in a circular pattern. In other words, we collect a distance measurement from one sensor, send it to ArduPilot, and then move on to the next sensor. A full cycle is then achieved after collecting measurements from all four sensors. As the readings are being collected, the software is also measuring the frequency of measurements (i.e., the number of full measurement cycles per second), and displaying them through the serial monitor. Take note of the frequency of measurements, you will later use this value to tune the cutoff frequency of filters that are applied to sensor readings. Note that the reported frequency of each channel can be different. They are read independently from each other in a loop that grabs the next available distance reading. For the following question and the cutoff frequency calculations, use the maximum frequency reading.

Question 3.3. (6 points) What is the approximate frequency of measurements? Based on this value, what is the time between readings.

3.2.2 Obstacle Avoidance with ArduPilot

Now you have to configure ArduPilot to read the values sent from the Arduino microcontroller. Connect Mission Planner to ArduPilot, and set the `PRX_TYPE` parameter to 2. Also set `SERIAL1_BAUD` to match the BAUD rate of the Arduino code. Restart ArduPilot and after reconnectiong Mission Planner, press `Ctrl-F` and push the `Proximity` button to display the horizontal sensor readings in Mission Planner. Ensure that all four sensors are working properly before proceeding.

Checkpoint 3. (3 points) Ask your instructor to double check that you can get distance readings for all four directions in Mission Planner.

ArduPilot already knows how to use horizontal distance measurements to avoid obstacles.[3] For non-GPS flight modes, this is done using the `AVOID_DIST_MAX` and `AVOID_ANGLE_MAX` parameters. The former specifies the minimum desired distance to an obstacle. If an object is detected closer

[3] `http://ardupilot.org/dev/docs/code-overview-object-avoidance.html`

than this value, then the software adjusts the roll and pitch angles (up to `AVOID_ANGLE_MAX`) so that the quadcopter flies away from the object.

The next step is to calibrate these hyper-parameters to allow the quadcopter to avoid crashes. Here, we have to mention an important work-around we have implemented. Ardupilot only accepts distance measurements with a resolution of 1cm. We would like resolution better than this, so we have multiplied the distance measurements from the arduino by a factor of 10 before sending them to ardupilot. This will then affect the `AVOID_DIST_MAX` parameter. The original units for this parameter are in meters, but since we multiplied the distances by a factor of 10, it is now, in effect, in units of decimeters. This is a good example of how to engineer a work-around solution for a system with an existing code base.

The maximum range of our LiDAR sensors is 1.36 meters and the minimum you should have is 0.2 meters (the distance from the sensors to the edge of the propellers' range). With our resolution modification, this gives a range for `AVOID_DIST_MAX` $\in [2, 13.6]$. You will have to think about what this parameter means relative to how you want it to react to obstacles around it, especially when you are implementing it for the final project. The ardupilot default unit for `AVOID_ANGLE_MAX` is centidegrees. It's maximum value able to be set is 10 degrees (1000 centidegrees) and it will take at least a 1 degree (100 centidegress) to see any reaction to an obstacle. Therefore, the range for `AVOID_ANGLE_MAX` $\in [100, 1000]$.

For this experiment, you should start by setting `AVOID_DIST_MAX` to about 1 meter (10 decimeters) and `AVOID_ANGLE_MAX` to 5 degrees (500 centidegrees). Test your parameters by flying the quadcopter similar to previous labs, and then slowly move an appropriate obstacle (e.g., a cardboard box) toward the quadcopter to observe its response. Ideally you would like the quadcopter to gently back away from the obstacle. You can then test the aggressiveness of this response by approaching the quadcopter with an obstacle at different speeds. You will be able to tune it's reaction time in the next part by tuning the cutoff frequency of the low-pass filter applied to the LiDAR distances.

Checkpoint 4. (3 points) Show your instructor that you can achieve obstacle avoidance, by approaching the quadcopter with an appropriate obstacle and observing its avoidance reaction.

3.2.3 Low-Pass Filter Tuning

Your next task is to enable and tune a low-pass filter for smoothing the sensor readings, and thus the response of the quadcopter. This can be done by altering the `ALPHA` constant at the top of the Arduino code for the sensor box. The filter applied to the readings is similar to the one specfied in the pre-lab, where the output of the filter is given by:

$$y[n] = \alpha x[n] + (1 - \alpha) y[n - 1],$$

where α is a number between zero and one. $x[n]$ denotes the current raw distance reading, and $y[n]$ and $y[n-1]$ are the current and previous processed (filtered) distances, respectively. Note that $\alpha = 1$ (the default) means that the sensor readings are not filtered at all. Alternatively, a value of $0 < \alpha < 1$ smoothens the distance readings; a lower alpha results in more smoothing, or a lower cutoff frequency. The cutoff frequency of this filter is then given by:

$$f_c = -\frac{f_s \log(1 - \alpha)}{2\pi}.$$

Where f_s is the sampling frequency, i.e., $\frac{1}{T_s}$, where T_s is the time between the consecutive measurements $x[n-1]$ and $x[n]$. If we know our desired cutoff frequency, we can rearrange the previous equation and compute α as follows:

$$\alpha = 1 - e^{-2\pi \frac{f_c}{f_s}}.$$

Now use this equation to compute α based on the value you computed for f_s in Question 3.3, and an initial value for f_c. You can start with a fairly small f_c, e.g., 1 Hz. Now recompile and upload the Arduino code to the sensor box.

Checkpoint 5. (3 points) Show your computed value for α and the modified Arduino code to your instructor.

Fly your quadcopter with the filtered sensor readings. If you chose a fairly small cutoff frequency, then you might notice that the quadcopter response more smoothly to obstacles, though this also increases the sluggishness of the response, i.e., it takes time for the quadcopter to detect and back away from obstacles, especially when the obstacle is approaching the quadcopter at higher speeds.

Tune the cutoff frequency of your filter until you are satisfied with the responsiveness of the flight. With a tuned cutoff frequency, you should observe a smooth response, while minimizing the delay in reacting to obstacles. Note, the range of possible cutoff frequencies is $\in [0, f_s/2]$. This is due to something called the Nyquist sampling theorem.

Question 3.4. (6 points) Report your tuned values for `AVOID_DIST_MAX`, `AVOID_ANGLE_MAX`, the cutoff frequency for sensor readings, and the corresponding value for α.

4 Post-Lab (30 Points)

Question 4.1. (15 points) Submit an image of the throttle output (`THO`), and the target/actual altitude (`SALT`/`DSALT`) of the quadcopter when the accelerometer/gyroscope filters were turned off in Section 3.1. Compare the stability of the quadcopter's altitude in both cases. Do you observe more instability in either case? If so, why?

Question 4.2. (10 points) Assume that the quadcopter is flying over a over a non-flat surface; then small variations in altitude readings (when using a downward facing sonar) can make the quadcopter respond by rapidly moving up/down. Offer a solution for making the quadcopter fly smoothly, while roughly maintaining a constant altitude over this surface.

Question 4.3. (5 points) Consider a simple RC circuit with a resistance of 50 kΩ and a capacitance of 2 µF. At time $t = 0$, the switch is closed connecting it to a voltage source of V_o and allowing current to flow. At what time during charging would the voltage over the capacitor be half of V_o?