

### Task 4-1: Frequency Filter(30 points)

You are required to implement the functions in `task_4_1.py`.

Given a noisy signal, you are supposed to apply different filters accordingly. You are asked to use 5-order Butterworth filter. You are **NOT** allowed to directly use either band-pass filter or band-stop filter.

1. Apply a high-pass filter with cutoff frequency 20Hz.
2. Apply a low-pass filter with cutoff frequency 10Hz.
3. Apply a band-pass filter with cutoff frequency [10, 20]Hz. You should use low-pass and high-pass filter to implement this band-pass filter.

The sampling rate `self.fs` is 500 Hz. You should return the filtered signals.

Run `python check.py --task 1` to check your implementation. Passing the tests in `check.py` indicates that your filtering is effective.

### Task 4-2: Apply Filters (30 points)

You are required to implement the functions in `task_4_2.py`.

In this task, you are now given multiple noisy signals. Please use the filters that have been included in our tutorials to smooth these noisy data.

1. `task_4_2_1.pickle`: You are supposed to filter it in `apply_filter_1()`. The sampling rate is 10Hz.
2. `task_4_2_2.pickle`: You are supposed to filter it in `apply_filter_2()`. The sampling rate is 20Hz.

**Note:** You can pick the filters according to the shape of the signals. Smoothing is a quite subjective task, but as a programming task, we have some quantitative metrics. The data for Task 4-2 are synthetic, meaning we possess the original datasets without any added noise. This allows for a direct comparison between the filtered output and the true, noise-free signal to objectively evaluate the effectiveness of the filtering process. Therefore, we can evaluate your results using the following metrics:

- SNR (Signal-to-Noise Ratio): This measures the level of the desired signal relative to the background noise. A higher SNR means the signal is clearer compared to the noise.
- RMSE (Root Mean Square Error): RMSE quantifies the difference between values predicted by a model or filter and the values actually observed from the environment that is being modeled or about which predictions are being made. Lower RMSE values indicate a closer fit to the original data.

- Variations of the first derivative: This metric assesses the smoothness of a signal by measuring the changes in its rate of change. Smoother signals have less variation in their first derivative.

We have implemented the functions to calculate these metrics for you. Run `python check.py --task 2` to check your implementation. Passing the tests in `check.py` indicates that your filtering is effective.

### Task 4-3: Sensing System (40 points)

You are required to implement the functions in `task_4_3.py`.

Based on your excellent performances in task 3-3, you are promoted to the senior technical engineer of SenseAI. Your boss, Dr. C.S.Wu, decides to assign you some more challenging tasks. You will help develop some key features of the sensing system for smart homes.

Please use the knowledge you have learned in the course to realize the following tasks.

As established in lab 2, frequency is the most important feature of an IoT signal. In this task, you are required to extract the dominant frequency of the signal. Here dominant frequency refers to the frequency with the highest magnitude in the frequency spectrum. You should return the dominant two frequencies in ascending order. If there is only one dominant frequency, you should return the same frequency twice.

**Checkpoint 1 (5 points):** Get the dominant frequency of the signal in Hz. Implement `get_freq(s, fs)`.

In Task 1-1, you've learnt about the amplitude modulation, i.e.,

$$s(t) = A_c(1 + \mu \cdot m(t)) \cdot c(t).$$

$c(t) = \cos(2\pi f_c t)$  is the carrier signal, and  $m(t)$  is the message signal. The amplitude modulation is widely used in IoT devices. At the receiver's end, we need to demodulate the signal to extract the message signal. The common practice is to multiply the received signal by the carrier signal and then pass it through a low-pass filter to extract the message signal. Here is the basic principle.

$$\begin{aligned} r_-(t) &= s_{AM}(t) \cdot \cos(2\pi f_c t) \\ &= A_c(1 + \mu \cdot m(t)) \cdot c(t) \cdot \cos(2\pi f_c t) \\ &= \frac{A_c}{2}(1 + \mu \cdot m(t))(1 + \cos(2\pi \cdot 2f_c \cdot t)) \end{aligned}$$

After applying LPF, you will get

$$r_-(t) = \frac{A_c}{2}(1 + \mu \cdot m(t)).$$

Afterwards, we remove the DC component and get the message signal,

$$\tilde{m}(t) = r_-(t) - \frac{A_c}{2} = \frac{A_c}{2} \cdot \mu \cdot m(t).$$

We can normalize the message signal by dividing it by the maximum value of the message signal  $\tilde{m}(t)$ .

**Checkpoint 2 (15 points):** Implement `demodulate_signal(s, fs, fc)` to extract the message signal from the received signal. The sampling rate is 1000 Hz.

Due to the imperfections in the filters, you can never retrieve the original message signal perfectly. We use the average value and the frequency components to evaluate your implementations. The average value should be close to 0, and the frequency should be close to the original message signal frequency.

Another aspect of filtering is interpolation, which is used to fill in the missing data points.

Your company is developing algorithms that rely on Inertial Measurement Unit (IMU) sensors for tracking movement in smart devices. Sometimes, IMU sensors miss data points—like a scratch on a CD—marked as “NaN” (not a number), breaking the signal’s flow. Interpolation guesses missing values using nearby points, often making the signal look cleaner too. Here we introduce the simplest interpolation method, linear interpolation. Given a 1D signal, you can estimate the value at  $n$  by,

$$x[n] = x[n_1] + \frac{n - n_1}{n_2 - n_1} \cdot (x[n_2] - x[n_1])$$

where  $n_1$  and  $n_2$  are the nearest two points of  $n$ .

**Checkpoint 3 (10 points):** Implement `interpolate_signal(s)` to interpolate the missing data points (marked with NaN) in the signal.

You do not have to implement yourself. You can make use of `numpy.interp` to implement the interpolation. For facilitating the testing, you should use linear interpolation to fill the missing data points.

After filling gaps, the IMU’s acceleration data still has jitters from floor vibrations, like static on a TV, throwing off tracking. You are required to apply the filters (choose from low-pass, high-pass, or band-pass) to smooth the signal.

**Checkpoint 4 (10 points):** Implement `apply_filter(s, fs)` to apply the filter to the signal. The sampling rate is 1000 Hz.

Run `python check.py --task 3` to check your implementation. Passing the tests in `check.py` indicates that your implementation is accurate.

## How to submit

**Please run `python check.py --uid <YOUR_UID>` before submitting.** This script performs automated tests on the examples provided in the docstrings. Failing these tests indicates potential critical issues in your code. Strive to resolve these problems. After that, it will create a zip file named after your `uid`. Make sure you enter the right `uid`.

It's important to avoid changing the names of any files, including both the zip file and the program files contained within. Altering file names can lead to grading errors. Ensure that all file names remain as they are to facilitate accurate assessment of your work.

Your submission to **Moodle** should consist solely of the **generated \*.zip file**. It is your responsibility to double check whether your submitted zip file includes your latest work.