Lab experience 7: Filtering with Arduino

Objectives

- Gain experience with designing a single-lead ECG
- Set a precise sampling frequency for your Arduino DAQ
- Get hands-on experience with digital filtering

Lab Safety

Let the instructors know if you have a history of cardiac disease. You are not obligated to obtain the ECG signal from your own body. You are very welcome to use the ECG data from another student. If you feel any discomfort while obtaining the ECG signal, immediately disconnect the electrode and let the instructors know. You have agreed with the above safety rules by attending the lab session.

Hardware

1. Sparkfun AD8232 Single-Lead ECG Signal Conditioning IC

Check out the datasheet:

https://cdn.sparkfun.com/datasheets/Sensors/Biometric/AD8232.pdf

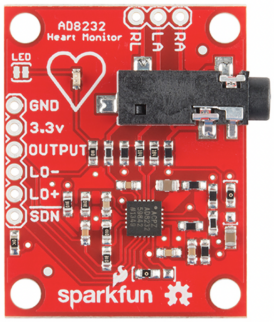

Figure 1. AD8232 Signal Conditioning IC. The LED inside the heart icon will fade in/out with the detected heartbeat.

Pre-Lab Exercise:

In this pre-lab, you will create two Arduino sketches: one for FIR filtering, the other for IIR filtering. We decided to record a tutorial (https://youtu.be/zMi8j64h4V4) that will guide you through the pre-lab programming because C programming is not a prerequisite of the course. Also, not having access to an Arduino board at home means that you will not be able to test if your code behaves correctly. Nevertheless, please feel free to do as much as the programming yourself before watching the tutorial. Keep in mind that there are often many ways of implementing each step, and there is no right or wrong implementation. The following steps are just our recommendations, and you are welcome to try out to code your own filters.

Exercise A. Setting a precise sampling frequency on Arduino

0. [If you have not downloaded the Arduino IDE] Head to the official Arduino website: https://www.arduino.cc/en/software. Download and install Arduino IDE. The IDE is free, but please feel free to make a few dollars of contribution.

1. Open an Arduino IDE. This should open a new .ino file with only the skeleton code containing two functions: `setup()` and `loop()`.

2. [Establishing serial communication] In the `setup()` function, initialize serial communication with a baud rate of 115200 bits per second.

3. From now on, you should click the checkmark on the top-left ("verify") every step of the way to confirm that your program does compile.

4. [Sampling ECG signal from AD8232] In the `loop()` function, do an `analogRead()` from channel A0, and save the result in an integer variable named `analogValue`. But in order to apply the digital filters, we must set a constant sampling rate. How do we sample in Arduino?

In LabVIEW, if we ask the DAQ to acquire one sample per loop, we could easily set the sampling rate using the Wait (ms) function. On the other hand, for Arduino, we have only introduced the `delay()` function, which is not the best way to set the sampling frequency. Consider the following code:

```
void loop() {
  data = analogRead(A0);
  delay(1);  // delay 1 ms
  filter();
  Serial.println(data);
}
```

By setting the delay time to 1 ms, do we really get a sampling frequency of 1 kHz (=1/0.001)? According to the Arduino Reference, calling `analogRead()` takes about 100 μs, and calling `Serial.print()` takes about 400-600 μs. We don't even know how long it takes to call the `filter()` function. Apparently, we are underestimating the loop period using the `delay()` function. This is why we are introducing you to the other two timing functions today: `millis()` and `micros()`.

The `millis()` and `micros()` function keep track of how many milliseconds and microseconds have passed since the start of the program, respectively. They are analogous to the High Resolution Relative Seconds function in LabVIEW. We will use `micros()` today since we want to set a sampling frequency at 500 Hz, as `millis()` might be too crude for that task. To set the sampling frequency, we typically use code structure is as follows:

```
const int FS = 500; //Hz
const int INTERVAL = 1000000L / FS; // in us
uint32_t lastMicros = 0;
```

```
void loop() {
    if (micros() - lastMicros > INTERVAL) {
        lastMicros = micros();
        int data = analogRead(A0);
    }
}
```

When the difference between the current `micros()` and the last `micros()` exceeds the specified interval, we acquire one data point and update the `lastMicros` variable.

Note: the `lastMicros` variable is initialized as a 32-bit unsigned integer because we expect this number to get quite large. But don't worry. A 32-bit unsigned integer can last for $2^{32} - 1 = 4,294,967,295$ ms (about 70 minutes) without overflowing.

5. Save the sketch and make a copy of it. Name the original file "`lab7_fir`" and the second file "`lab7_iir`" (or some other file names that make sense).

Exercise B. Implementing a FIR filter

If you have the ECG hardware with you right now, you can upload the program to the Arduino. But you might encounter something like the following:

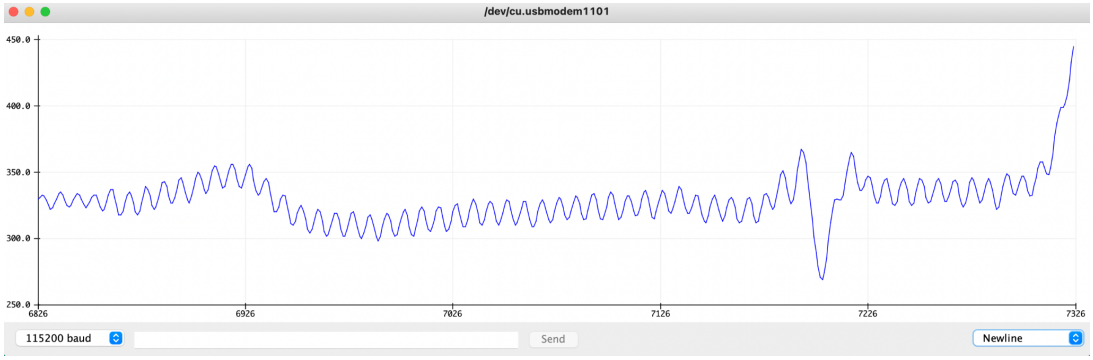

Figure 2. ECG signal polluted by mains hum.

In addition to the ECG signal, there is a small yet consistent oscillation throughout the time series. This is due to the electric hum at 60 Hz (because I am charging my laptop). It turns out the AD8232 chip does not come with an analog filter for this noise. Let's implement a digital notch filter to remove this 60 Hz noise.

Although Arduino contains several filter libraries, they often suffer from low performance and low customizability. Therefore, it is standard to compute the filter coefficients beforehand and hard-code them in the embedded program. We will use MATLAB to generate the coefficients of three different FIR filters (Table 1), and then implement the FIR filter in Arduino by ourselves.

Note: We understand that some students may not have access to an active MATLAB

license, so we have uploaded all the filter coefficients you need for this lab onto the assignment page. Note that Scipy's signal processing module is a free alternative for filter coefficient generations. We have also uploaded the Python code onto the assignment page.

Table 1. FIR filter parameters.

FIR Filter #	Sampling rate (Hz)	Cutoff frequencies (Hz)	Deviations allowed in the pass/stopbands	Filter type	Filter length
1	500	[55 59 61 64]	[0.5 0.1 0.5]	bandstop	141
2	500	[55 59 61 64]	[0.5 0.25 0.5]	bandstop	48
3	350	[55 59 61 64]	[0.5 0.1 0.5]	bandstop	99

1. [Generate FIR filter coefficients] The following MATLAB commands generate FIR Filter #1 in Table 1. Copy the coefficients into the Arduino sketch "`lab7_fir`".

```
fs = 500;  % sampling freq in Hz
fcomb = [55 59 61 64];  % cutoff freq in [pass1 stop1 stop2 pass2] format
mags = [1 0 1];
dev = [[0.5 0.1 0.5]];
[n,Wn,beta,ftype] = kaiserord(fcomb,mags,dev,fs);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
% print out coefficients in C array format
disp(['const float h[] = {', sprintf('%0.8f,', hh), '};']);
```

2. [Optional, visualize frequency response of the FIR filter] We can visualize the frequency response of the filter with the following commands:

```
figure
freqz(hh, 1, 2^20, fs)
set(subplot(2,1,1), 'XLim', [0 100])
set(subplot(2,1,2), 'XLim', [0 100])
```

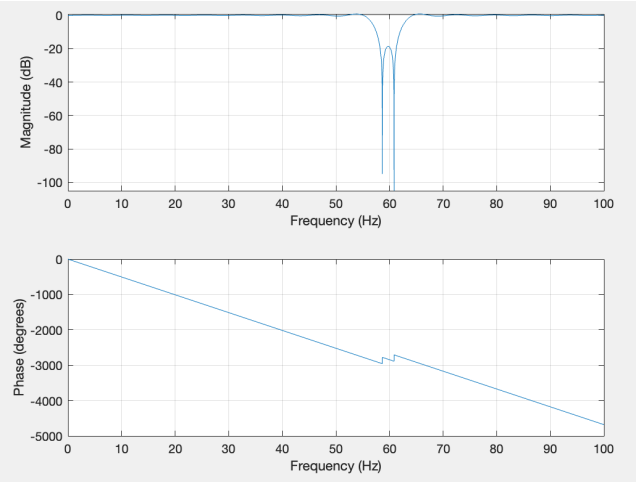

Figure 3. Frequency response of FIR Filter #1.

3. [Calculating filter length] Calculate how many coefficients are there with the `sizeof()` function.

4. [Implement an input buffer] Initialize an integer array named "`x`" that has the same length as the filter coefficient array. Implement it as a circular buffer.

5. [Implement a FIR filter function] Make a function named `filter()` that will compute and return the filtered value of the most recent input. Recall that FIR filter is applied by convolving the filter coefficients with the latest input signal:

$$y[n] = \sum_{k=0}^{M-1} h[k]\, x[n-k]$$

where *M* is the filter length, *y[n]* is the current input, and h is the filter coefficients.

Note: Since we are using a circular buffer, the index $n - k$ can be negative. You should write code to prevent negative indexing.

Exercise C. Implementing an IIR filter

Table 2. IIR filter parameters.

IIR Filter #	Sampling rate (Hz)	Cutoff frequencies (Hz)	Filter type	Filter length
1	500	[59 61]	bandstop	5
2	700	[59 61]	bandstop	5
3	500	[59 61]	bandstop	9

1. [Generate FIR filter coefficients] The following MATLAB commands generate IIR Filter #1 in Table 2. Copy the coefficients into the Arduino sketch "`lab7_iir`".

```
fc = [59 61];   % cutoff freq in Hz
fs = 500;       % sampling freq in Hz
n = 2;          % n is actually only 1/2 order for a bandstop filter
[B,A] = butter(n, fc./(fs/2), 'stop');
% print out coefficients in C array format
disp(['const float A[] = {', sprintf('%0.8f,', A), '};']);
disp(['const float B[] = {', sprintf('%0.8f,', B), '};']);
```

2. [Optional, visualize frequency response of the FIR filter] We can visualize the frequency response of the filter with the following commands:

```
fvtool(B,A);
```

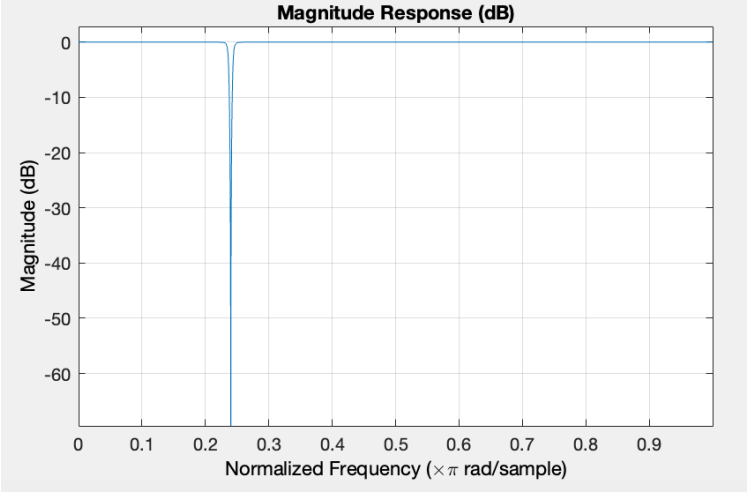

Figure 4. Frequency response of IIR Filter #1.

3. [Calculating filter length] Again, calculate how many coefficients are there with the `sizeof()` function.

4. [Implement two circular buffers] Recall that IIR filter is a recursive filter, meaning that the current output value depends on both the previous inputs and the previous outputs. Therefore, we need two arrays of the same sizes as the filter coefficients: one to keep track of the previous inputs, one to keep track of the previous outputs. Let's name the input array "x" and the output array "y".

Note: Let's think about what the data types the buffers should have: since the input is always an integer between 0 and 1023, using int16_t will suffice. On the other hand, the output array "y" should be a float array so that we do not lose precision.

5. [Implement a IIR filter function] Make a void function named `filter()` that will compute the filtered value of the most recent input and save it in the corresponding index location in the output array "y". Recall that IIR filters are almost the same as FIR filters, but IIR filters have the feedback terms (the second summation):

$$y[n] = \sum_{k=0}^{M-1} b[k]\, x[n-k] - \sum_{k=1}^{M-1} a[k]\, y[n-k]$$
$$= b[0]\, x[n] + \sum_{k=1}^{M-1} b[k]\, x[n-k] - a[k]\, y[n-k]$$

where *M* is the filter length, *y[n]* is the current input, ***b*** are the feedforward coefficients, and ***a*** are the feedback coefficients.

In-Lab Exercise:

In this lab this week's lab we will explore the characteristics of FIR and IIR filters. We will then implement their fixed-point versions to observe the differences in performance.

Exercise D. Deploying the filters

1. Following the hookup guide to connect the AD8232 board to Arduino Uno.

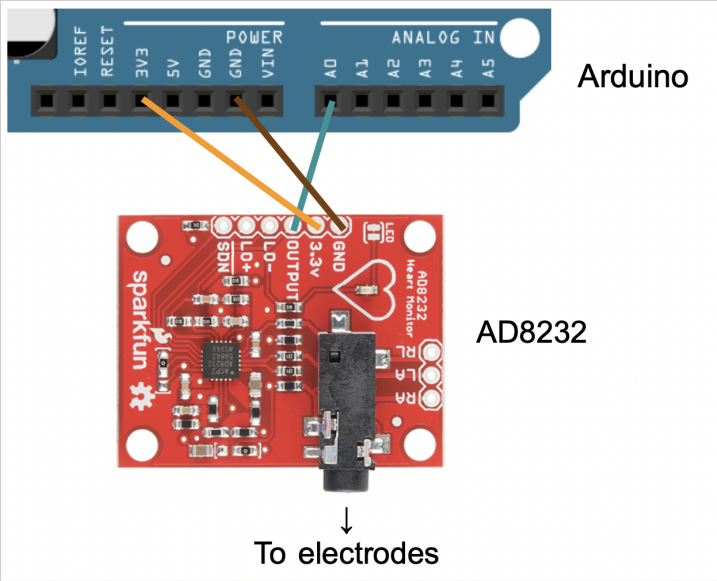

Figure 5. You should connect AD8232 and Arduino like this (you decide which analog input pin to use).

2. Place the electrodes according to the figure found below (either is fine):

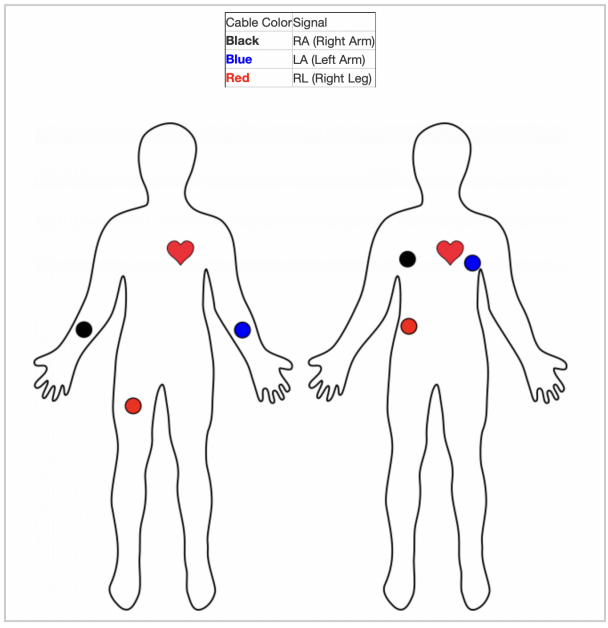

Figure 6. Typical sensor placement. Figure source: Sparkfun AD8232 Hookup Guide.

3. Now print out both the raw and filtered ECG signal to observe the difference. You probably also want to add an offset between the two, so you may better visualize the difference. This is what I have:

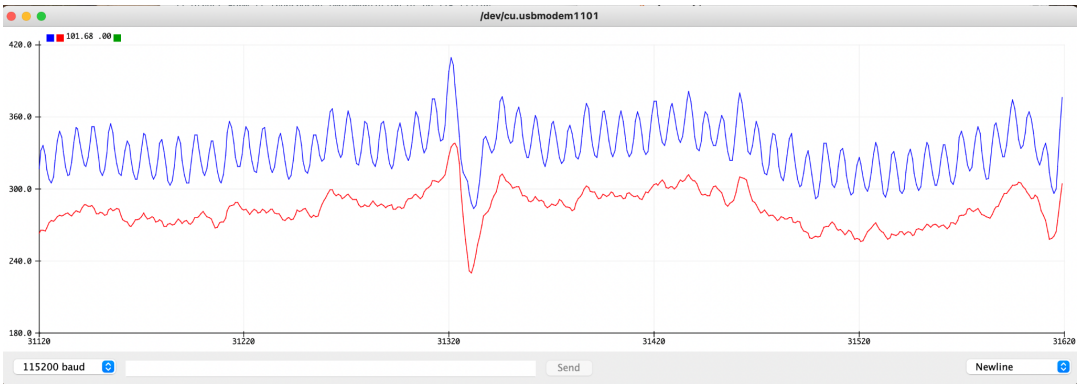

Figure 7. Filtered (red) vs. raw (blue) ECG signal. The filter used is IIR Filter #1 (floating-point).

As you can see, the filtered signal in red is much cleaner than the raw signal in blue. Feel free to try out different filter types and parameters.

Tip: you can add a small constant offset between the filtered and the raw signals to separate them vertically on the serial plotter.

4. Record your observations in Tables 3 and 4. Note that there are no "correct" answers, only honest observations. You will be graded entirely on completeness. To find the actual sampling rate, you can print out the inverse of the loop time, for example:

```
const int FS = 500; //Hz
const int INTERVAL = 1000000L / FS; // in us
uint32_t lastMicros = 0;
void loop() {
  int32_t timeDiff = micros() - lastMicros;
  if (timeDiff > INTERVAL) {
        lastMicros = micros();

        // Other sampling code not shown.

        Serial.print(float(1e6) / timeDiff); // loop rate in Hz
        Serial.print(',');
        Serial.print(rawData);
        Serial.print(',');
        Serial.println(filteredData);
  }
}
```

As highlighted in the code above, the loop rate can be calculated by dividing 1e6 by the time difference (in milliseconds). Notice that 1e6 is first converted into a float variable so that we are not doing integer division.

The global variable dynamic memory usage can be found at the bottom of the Arduino IDE everytime you finish compiling the sketch. Figure 8 shows examples of an unstable filter and a filter with a noticeable delay (i.e., the filtered signal is lagging behind the raw signal).

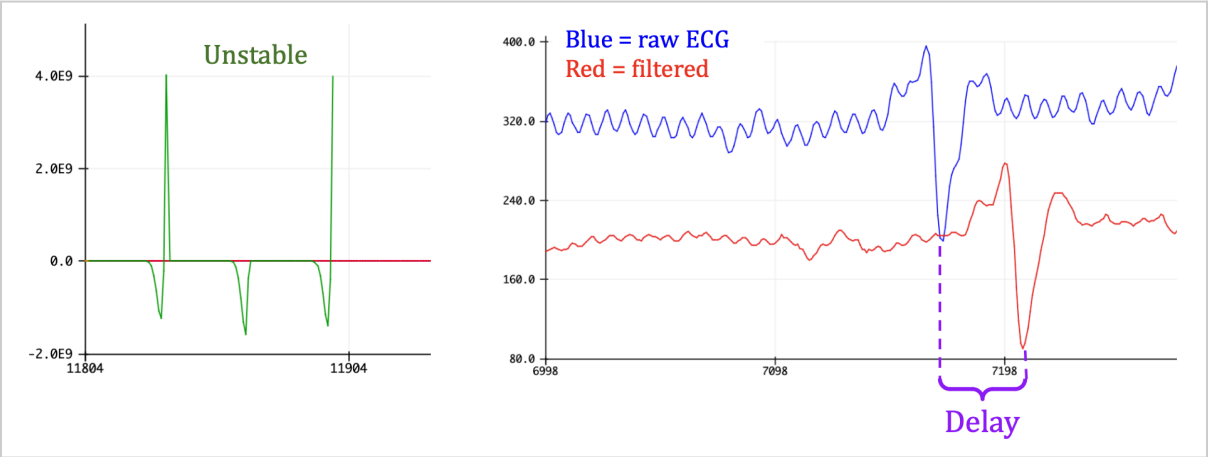

Figure 8. Examples of an unstable filter (left) and a filter with a noticeable delay (right).

Table 3. Floating-point FIR filter observations.

FIR Filter #	Actual sampling rate (Hz), given as a range	Global variable dynamic memory usage (%)	Stable?	Any noticeable delay?	*Is the 60Hz noise removed?
1					No
2					No
3					No

**Because the 60 Hz noise is not always present in the lab, it may be difficult to determine if it can be removed by the filters. For your convenience, we have included our results in the column "Is the 60 Hz noise removed".*

Table 4. Floating-point IIR filter observations.

IIR Filter #	Actual sampling rate (Hz), given as a range	Global variable dynamic memory usage (%)	Stable?	Any noticeable delay?	*Is the 60Hz noise removed?
1					Yes
2					No
3					No

Exercise E. Making fixed-point FIR

You will observe that some of the filters listed above require so many mathematical operations that we cannot achieve the intended sampling rates. In this case, people usually resort to faster hardware or integer math. Integer math, or more accurately referred to as *fixed-point arithmetics*, is fast because microcontrollers like Arduino do not come with floating-point units, which is a kind of hardware that is specialized in accelerating floating-point arithmetics. Instead, floating-point math is emulated with integer operations, which take up more clock cycles per operation.

Actually, you might already know fixed-point arithmetics quite well. The idea is simple: we take a floating-point number and multiply it by some scaling factor. We then round this number into an integer. Yes, we do lose some precision due to rounding, but in many cases it does not matter if the scaling factor is big enough. We then use this fixed-point representation to do the calculation. When we need the number to be in its floating-point representation again, we can just divide it by the same scaling factor.

What scaling factor should we use? It is recommended to use a power of 2 as a scaling factor because the compiler is smart enough to substitute integer multiplication/ division with the left shift (<<)/right shift (>>) operators, which can be done in one single instruction. In addition, to preserve as much precision as possible, we want to use as big a scaling factor as possible. But at the same time, we need to test the operation (filtering in this case) to make sure that it does not overflow. In this lab, we will convert floating-point numbers into Q15.16 presentations. That is, we will multiply them by 2^16 and store them in int32_t variables. A Q15.16 variable can represent numbers from -32768.999969482421875 to 32767.999969482421875.

1. Make a copy of the Arduino sketch "`lab7_fir`" and name it "`lab7_fir_fixed`". Just like the other sketches, this new .ino file must be placed inside a new folder that has the same name.

2. Define a global `int` constant called "`FRAC_BITS`" and set it to 16. This means that the fixed-point representation uses 16 bits (out of 32 bits) to represent the fractional part of the floating-point number, leaving 15 bits for the integer part, and 1 bit for the sign.

```
const int FRAC_BITS = 16;
```

Define a global `uint32_t` constant named "`SCALE`". This is the scale that the floating-point numbers will be multiplied by. As mentioned above, this scale must be a power of 2 for efficiency. Here I use the left shift operator (<<) to define the scale:

```
const uint32_t SCALE = (1L << (FRAC_BITS));
```

which is equivalent to:

```
const uint32_t SCALE = pow(2, FRAC_BITS);
```

3. Declare a global, int32_t array of the same size as the filter coefficient array. This

array will be used to store the fixed-point version of the filter coefficients.

```
int32_t fixedH[FILTER_LENGTH];
```

4. Define a void function called `init_filter()` that converts the floating-point filter coefficients into fixed-point coefficients. In other words, this function multiplies the floating-point coefficients by "SCALE" and casts them into `int32_t`. Then, call this function inside `setup()` so that we can initialize the fixed-point filter before sampling. See **Appendix 1** for our implementation.

5. We need to make a few changes of the `filter()` function:

- Change the datatype of the accumulator variable "`acc`" to int32_t.
- Filter calculation should use the fixed-point filter coefficients "`fixedH`" instead of the original floating-point coefficients "h".
- Change the return statement: we need to re-scale the filtered result as follows:

```
return float(acc) / SCALE;
```

Table 5. Fixed-point FIR filter observations.

FIR Filter #	**Actual sampling rate (Hz), given as a range**	**Global variable dynamic memory usage (%)**	**Stable?**	**Any noticeable delay?**	***Is the 60Hz noise removed?**
1					No
2					Slightly
3					Yes

6. Summarize your observations in Table 6.

Table 6. Observation summary. Select either FIR or IIR in each row.

	FIR	**IIR**
On average, which is more compatible with a high sampling rate?		
Which uses more filter coefficients?		
Which uses more global variable dynamic memory?		
Which one is more prone to instability?		

Which one has a more noticeable delay?		
Which one is less prone to quantization error? (*Hint*: which one is more suitable for fixed-point calculation?)		
In your opinion, which one is easier to implement?		
In your opinion, which one is better for real-time signal processing in Arduino?		

(Optional) Exercise F. Making fixed-point IIR

Repeat the same steps in Exercise E to implement fixed-point IIR filters. Remember, if you are multiplying two numbers that have been scaled, you need to divide the product by the same scale to get back to the correct fixed-point format. Also, IIR filters are sensitive to quantization error. Make sure to use a larger scaling factor (e.g., 2^20).

Table 7. Fixed-point IIR filter observations.

IIR Filter #	**Actual sampling rate (Hz), given as a range**	**Global variable dynamic memory usage (%)**	**Stable?**	**Any noticeable delay?**	***Is the 60Hz noise removed?**
1					No
2					No
3					No

Deliverables

- Arduino sketches:
  - FIR
  - IIR
  - fixed-point FIR
  - fixed-point IIR (optional)
- A short report that includes:
  - Your name(s)
  - Tables 3 - 6. Table 7 is optional.

Note: You are graded entirely on completeness. All deliverables may be co-authored by up to three students.

Appendix 1. `init_filter()` for FIR

```
// Sampling constants
const int FS = 500;  // in Hz
const uint32_t SAMPLE_INTERVAL = float(1e6) / FS;  // in us

// Filter constants
const float h[] = {0.00397736,...};
const size_t FILTER_LENGTH = sizeof(h) / sizeof(float);

// Signal stuff
int16_t x[FILTER_LENGTH];
int writeIdx;

// Fixed point stuff
const int FRAC_BITS = 16;
const uint32_t SCALE = (1L << FRAC_BITS);
int32_t fixedH[FILTER_LENGTH];

void setup() {
  Serial.begin(115200);
  init_filter();
}

void loop() { // Sampling code not shown }

float filter() { // implementation not shown. }

void init_filter() {
  for (int i = 0; i < FILTER_LENGTH; i++) {
    fixedH[i] = (int32_t)(h[i] * SCALE);
  }
}
```