

ENGR 40M Project 3c: Responding to music

For due dates, see the overview handout

1 Introduction

This week, you will build on the previous two labs and program the Arduino to respond to an input from the environment or users. There are a number of things you can do. This handout covers one of those options.

Specifically, in this lab, you will program your Arduino to make the LED cube respond to music. To do this, you'll connect the audio output from your computer/phone to the **analog-to-digital converter** on the Arduino, so it can “read” the value of the sound. If you want it to display frequency bands, you can use a third-party library that computes the Fourier coefficients of the waveform so you can build an audio spectrum visualizer, the display you often see in graphic equalizers.

You'll make good use of the code we wrote for the last lab, and you'll see it all come together into something mesmerizing.

By completing this lab you will:

- Learn how to use the analog-to-digital conversion facility of the Arduino.
- Learn how to use a signal generator and oscilloscope.
- Understand how the frequency domain relates to audio signals.
- See Fourier analysis in action before your very eyes.

We omit deeper details from this handout: we've included only what is necessary to complete the lab and understand what is happening. There will be a number of supplementary handouts on the website for those who are curious about the FFT or wish to add better speaker amplifiers to their breadboards.

2 Prelab and new equipment

2.1 Requirements

This handout covers two ways in which you can interpret an audio signal:

- (a) the raw signal itself
- (b) the *frequency representation* of the signal, *i.e.*, its Fourier coefficients

You can choose to complete this entire handout, which walks you through a third-party library that returns a signal's Fourier coefficients, or you can choose to implement your own idea. Some ideas:

- (a) You could track a “recent history”, and make one of the axes of the cube a time axis. For example, you might move each vertical plane over by one every tenth of a second, updating just the frontmost plane with new information.
- (b) You could improve the “raw signal” idea by tracking the last local maximum of the signal, sometimes known as the *envelope* of the signal.
- (c) You could improve some aspect of your LED display by adding a pushbutton or potentiometer input, *e.g.* to adjust the dynamic range.

You should start by thinking about what you'd like your LEDs to do. We don't require you to write code for this prelab, but you must have an idea of what effect you want to generate. Remember that your cube is three-dimensional, so you have three axes to play with! If you implemented the analysis question of lab 3b, you can also vary the brightness of your LEDs.

Some simpler proposals:

- (a) Each row corresponds to a particular amplitude or frequency, and lights up when it is loud/there is a strong treble component (meets no complexity requirement)
- (b) As above, but the number of LEDs in a plane or the intensity of a plane is proportional to the amplitude (meets modest complexity requirement)

P1: Think about how you want your LEDs to respond to sound. Describe in detail what you'd like to do, in enough detail that your CA can give you feedback on the complexity and feasibility of your idea. It may help to read the rest of the prelab first, if you'd like to make you thinking a bit more concrete—prototypes of the `setLEDs()` functions are below (one for raw signal, one for frequency spectrum).

2.2 Analog-to-digital conversion

So far, we've been using `digitalRead()` to read inputs as either HIGH or LOW. The analog input pins (A0 through A5) can also read an analog voltage. We can access this using `analogRead()`, which returns an integer that represents the voltage. The facility in the Arduino that converts the input voltage to a digital value (an integer) is called the **analog-to-digital converter** (ADC).

The number of bits in the ADC determines how precise the value will be, or equivalently, how many different digital values are possible. This is known as the *resolution* of the ADC. If the input voltage is V_{in} , the reference voltage is V_{ref} and the ADC has n bits, the digital value given by an ADC will be

$$\text{ADC value} = \frac{V_{\text{in}}}{V_{\text{ref}}} 2^n, \quad 0 \leq V_{\text{in}} < V_{\text{ref}},$$

rounded down to the nearest integer. The Metro Mini has six 10-bit analog input channels. You read them using the `analogRead()` method on any of the analog pins (A0-A5):

```
const byte AUDIO_INPUT_PIN = A5; // or any other analog pin

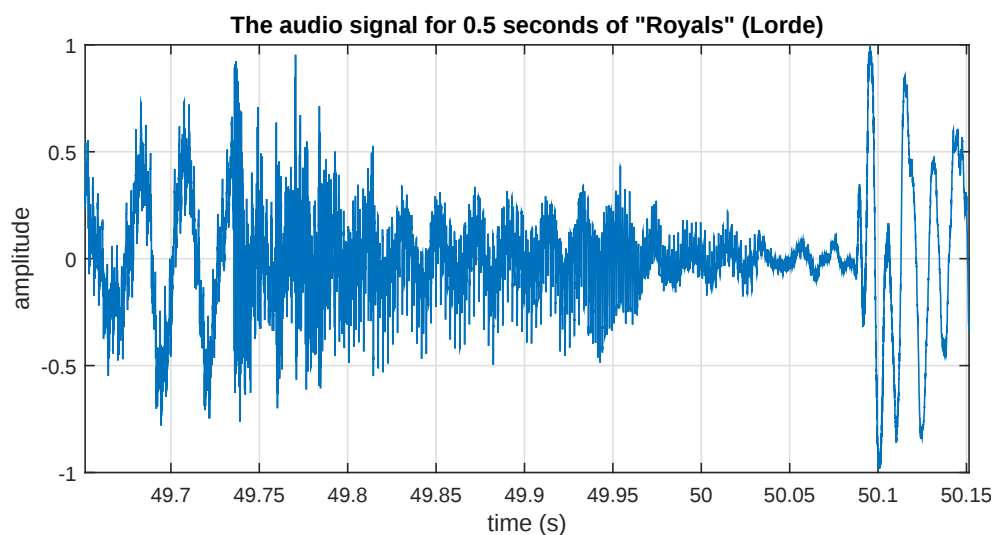
void loop() {
    int value = analogRead(AUDIO_INPUT_PIN);
    Serial.println(value);
}
```

P2: The Arduino's ADC produces 10-bit (unsigned) values. What is the range of possible values? If you don't know, write some code and measure it with your Arduino – just put Vdd and Gnd on the analog pin.

P3: We will use the default reference voltage of $V_{\text{ref}} = 5 \text{ V}$ for our maximum value. If the input voltage is 2.5 V , what digital value would you expect to read?

P4: Say that your audio signal, as seen by the Arduino's ADC, is always between 2 and 3 volts. (We'll get to how we achieve this later.) With the same reference voltage of $V_{\text{ref}} = 5 \text{ V}$, what will your digital value range be?

How does the audio signal fit in? The audio signal from your computer (or phone) is just an analog voltage varying with time. So when you call `analogRead(AUDIO_PIN)`, it returns the converted digital value for the analog voltage at the time. We call `analogRead()` repeatedly to collect lots of samples of the audio signal.



In the starter code, there is a function called `setLEDs()`, which accepts an ADC sample and uses it to generate an LED pattern.

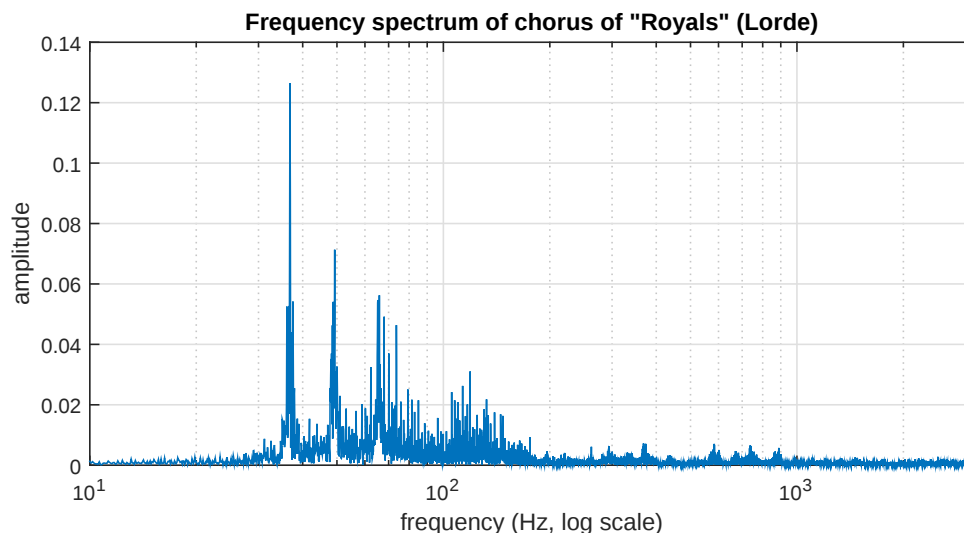
```
/* Function: setLEDs
 * -----
 * Sets the LEDs array based on the given sample. The sample is assumed to
 * be returned by getSample() (and have the consequent properties).
 */
void setLEDs(byte ledOn[4][4][4], int sample) {
    Serial.println(sample);
    // TODO: fill in this gap (see prelab question P1)
}
```

The fork in the road: If you only want to have your cube react to the raw audio signal, skip to section 2.4. If you want your cube to react to the frequency spectrum of the audio signal, read on.

2.3 The frequency domain (optional)

This section is optional. You are not required to make your cube respond to frequency output.

As we discussed in lecture, every signal can be presented as a sum of sine waves. We refer to the coefficients of this representation as the *frequency domain*, in contrast to the time domain. The frequency-domain representation of a signal is known as its “frequency spectrum”. For example, if we examine the frequency spectrum of a typical audio signal, we’ll notice that most of it is in between 10 Hz and 5 kHz. Bass frequencies are lower, and treble are higher. The highest pitch you can hear is around 16–17 kHz.



The Fourier series we’ve studied in class aren’t ideal for representing audio: they only represent periodic (repeating) signals, and most music isn’t periodic. There is a variant of the Fourier series, called the *discrete Fourier transform* (DFT) for finite-length signals.

A common algorithm for computing the DFT is called the *fast Fourier transform* (FFT). The FFT is one of the commonly-used analytical techniques in electrical engineering. Many tools, including Matlab and most oscilloscopes, have built-in FFT capabilities. The Arduino doesn’t (it’s a microcontroller, not an analytics package), but happily for us, someone has written a third-party library that does this on the Arduino. It’s called `ArduinoFFT`, you’ll need to install it on your computer before the lab.

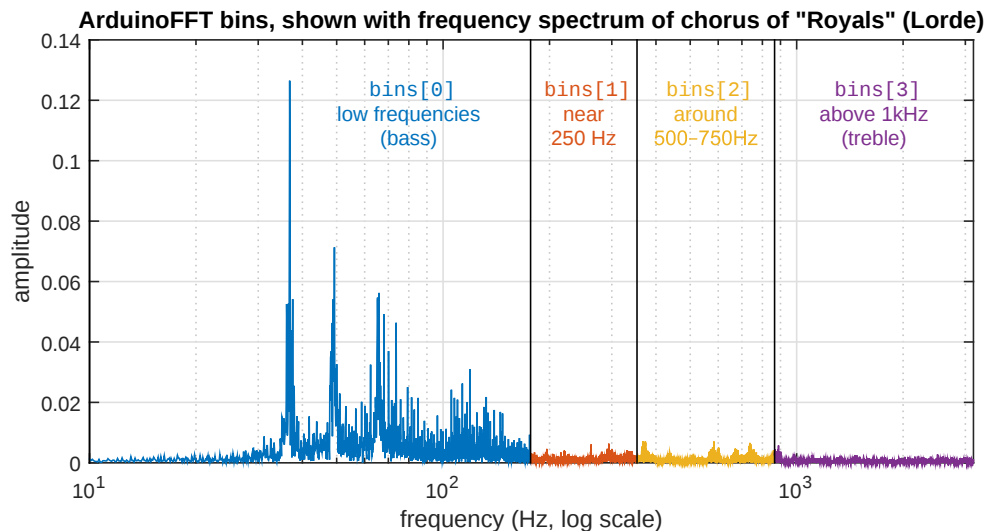
P5: Install the ArduinoFFT library at <http://wiki.openmusiclabs.com/wiki/ArduinoFFT>. To do this, you'll need to go to the website, click on the link to "Arduino FFT library (V3.0)", download and unzip **ArduinoFFT3.zip** and move the "FFT" folder to your Arduino libraries folder.

Note: Be sure to download version 3.0. Older versions won't work.

The intricacies of the FFT are beyond the scope of ENGR 40M¹, and we will provide the framework for using ArduinoFFT. If you're interested, we discuss this in more detail in supplementary notes.

How does this translate to our project? The starter code will provide four numbers to you every time it runs the FFT. These will be provided in an array called `bins[4]`. Roughly speaking, the four bins divide frequencies as follows shown in the image below. You can think of the four numbers going from bass to treble, so when a bass note sounds, the number representing the bass "bin" will be large.

The values stored in bins represent the sums of the amplitudes of all the frequency signals in the given bin range. For example, if we had a 0.5V, 10Hz sine wave and a 1V, 50Hz sine wave, `bins[0]` would return $0.5V + 1V = 1.5V$, or whatever the representation of 1.5V is in your ADC. You don't have to worry about the exact values - you just need to know that a larger value of `bins[0]` means that you have more/louder bass frequencies.



In the starter code, there is a function called `setLEDs()`, which accepts the `bins[4]` array and uses it to generate an LED pattern.

```

/* Function: setLEDs
 * -----
 * Sets the LEDs array based on the given frequency bin values. The bins are
 * assumed to be of the form generated by fft_mag_octave() of the ArduinoFFT
 * library.
 */
void setLEDs(byte ledOn[4][4][4], byte bins[4]) {
    static char message[50];
    sprintf(message, "%d_%d_%d_%d\n", bins[0], bins[1], bins[2], bins[3]);
    Serial.print(message);
    // TODO: fill in this gap (see prelab question P1)
}

```

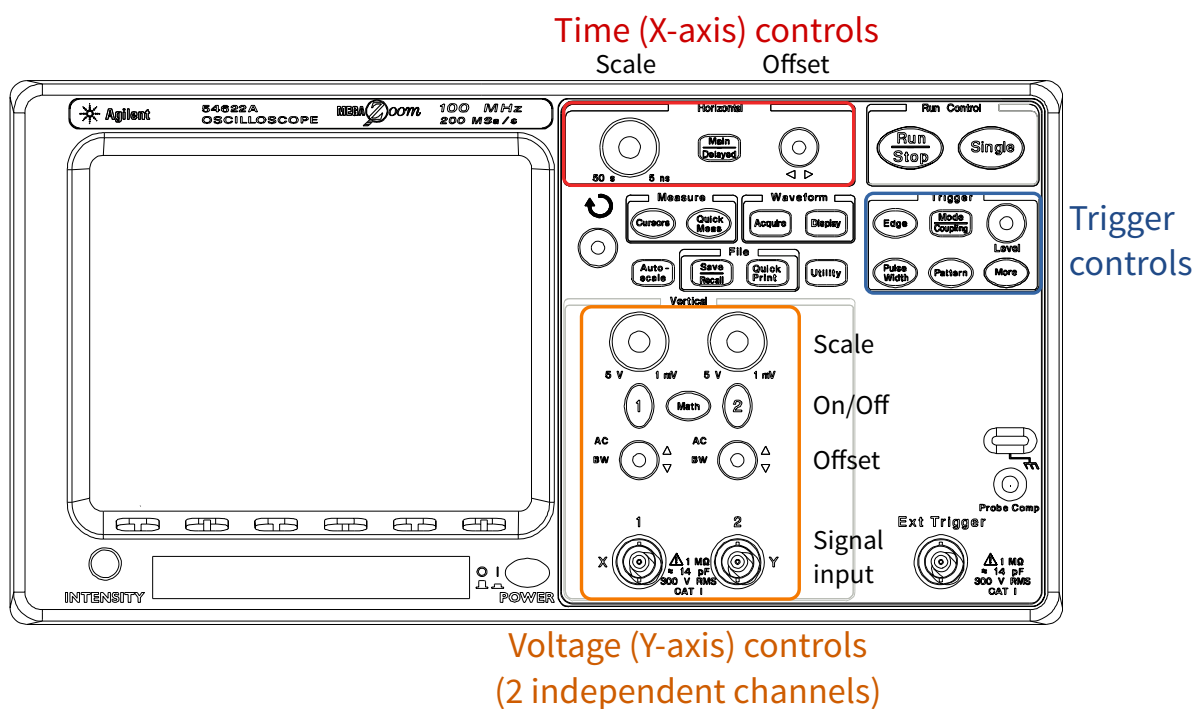
¹You'll learn all about it when you take EE 102A.

2.4 Oscilloscope

An oscilloscope is one of the most useful tools for electrical engineers in debugging electronic circuits. At its core, an oscilloscope makes plots of voltage over time. This is very useful for trying to understand circuits that have rapidly changing voltages.

While an oscilloscope has an intimidating number of buttons and knobs, its basic operation is straightforward. The diagram below shows the front panel of the Agilent 54621A (which is very similar to the DSO6012 that we have in the lab), but all oscilloscopes have a similar set of controls.

By the way: You might notice a button labelled “Auto-scale”. **Don’t use the Auto-scale button!** While this can be a convenient shortcut, it often fails and gives you something that you didn’t want. More importantly, it short-circuits the process of thinking about what you expect to see. Too many students press “auto-scale” and assume that whatever they see must be the right answer. Once you get the hang of it, adjusting the scope manually will be nearly as fast, and far more likely to get you the right answer.



- Time (X-axis, Horizontal) controls:
 - The **large knob** controls the **scale** of the X-axis. The time covered in one vertical grid division is shown on the display (e.g, 10 ns, 50 μ s).
 - The **small knob** shifts the signal left and right. The **offset** from the center is shown on the display.
- Voltage (Y-axis, Vertical) controls:
 - The BNC connector at the bottom is where you connect your probe.
 - The **On/Off** button (the one with the number on it) toggles whether the channel is displayed. The button lights up when the channel is active.
 - The **large knob** controls the scale. The display indicates the voltage per grid division.
 - The **small knob** shifts the waveform up and down. The **offset** from the middle is shown on the display.

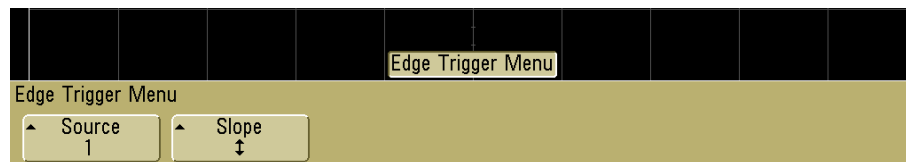
The **trigger** tells the oscilloscope when it should start its plot of voltage versus time. You set a **trigger event**, and it will center that event on the display. It's easiest to describe these by example:

- “Trigger when the voltage goes from below 3 V to above 3 V.”
- “Trigger when the voltage crosses 0 V in either direction.”
- “Trigger when the voltage stays below -1 V for at least 5 ms.”

To set a trigger:

- The little knob controls the **trigger level**, the voltage threshold the oscilloscope uses to judge an “event”.
- The buttons **Edge**, **Pulse Width** and so on are different **types** of triggers. In our work, we will always want to use the **edge** trigger. An edge trigger is when the voltage goes from below the trigger level to above it (“*rising edge*”), or from above the trigger level to below it (“*falling edge*”).
- These buttons all open menus which allow you to adjust other details about the trigger.

For example, if you press **Edge**, you'll pull up the “Edge Trigger” menu, which allows you to select which channel to monitor and whether to trigger on a rising edge, falling edge or either.

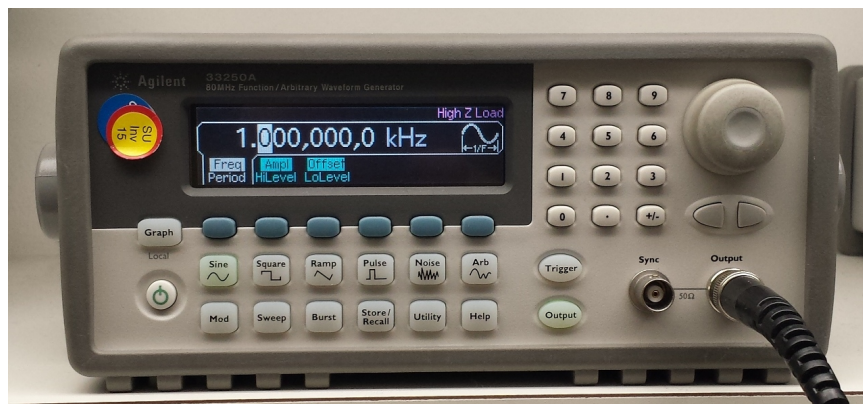


The oscilloscope will horizontally align the plot so that the **trigger event** happens at $t = 0$. You adjust where $t = 0$ is on the display using the **time offset** knob (the small knob in the “Horizontal” section described above).

Don't worry if triggering doesn't make sense now; it'll make more sense when you see it in action.

2.5 Signal generator

The **signal generator** (also known as a **waveform generator**) is the piece of equipment shown below. It can generate many types of signals and frequencies.



This piece of equipment is a bit more intuitive, so we omit a detailed description here. The main things to know are:

- The knob adjusts whatever number is on the display.

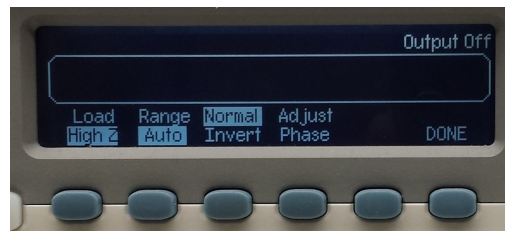
- For our work, you must enable **High Z Load** (as shown in the image), and this is **not** enabled by default. Instructions for this are below.
- The output won't turn on until you press the **Output** button so that it is lit up.

To enable high Z load:

1. Press the **Utility** button. This should bring up the menu below.



2. Go to **Output Setup**. This should bring up the menu below.



3. Change the **Load** to “**High Z**” (if it isn't already there).

3 Getting familiar with the ADC

First, we'll familiarize ourselves with the ADC. We'll use a power supply unit, then a signal generator, as inputs to the ADC and show that we can control the LEDs using an analog voltage signal.

To do this part, download the file `rawsignal.cube.ino` or `rawsignal.plane.ino` from the website. Have a read of the `loop()` function, to get an idea of what it does. Basically, it retrieves a sample, then sets the `ledOn[][][]` array to reflect the sample, then displays the array on the LEDs, then repeats.

In our starter code, we have set up the Serial Monitor to run at a baud rate of 115200. You might find this useful. To use it, open the Serial Monitor by clicking the button in the top-right corner of the Arduino window, and set the baud rate to 115200.

1. Copy the functions `getValue()` (if you did a cube) and `display()` from your code for Lab 3b. You shouldn't need to modify these; we're building on top of this code.
2. Complete `getSample()` by using `analogRead()` to return the raw result of an ADC conversion on the pin you specified in `audioPin`. (There's nothing tricky about this, it's a simple call of `analogRead()`).
3. Write `setLEDs()` to populate `ledOn[4][4][4]` (or `ledOn[6][6]`) based on the value of `sample`. Your code should react to samples anywhere in the entire range of the ADC.

This is your first go at implementing the pattern you had in mind in prelab question P4. The simplest thing to do is just to set the number of LEDs that are on, *i.e.* turn 1 on if the value is at least 16, 2 if it's at least 32, and so on, so that they're all on if the value is at least $1024 - 16 = 1008$. Alternatively, you might want your LEDs to “build” from the bottom like a bar graph, to “pulse” out from the center. It's up to you, but don't get carried away just yet!

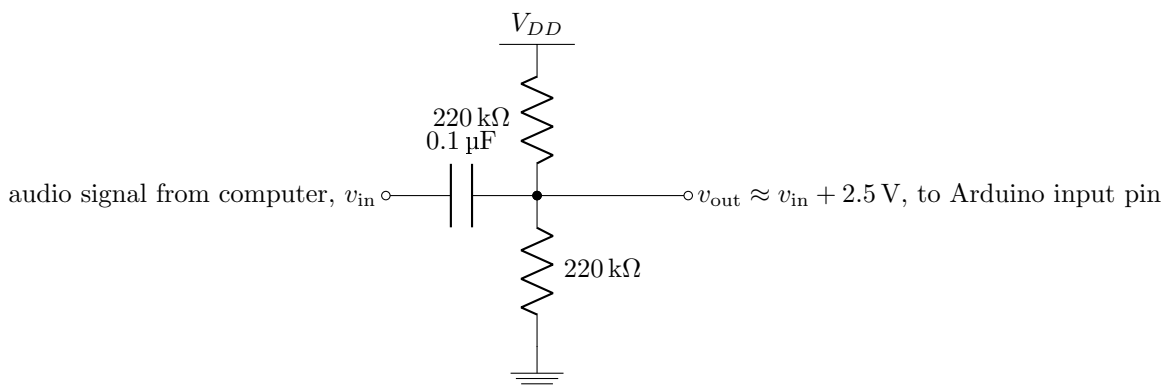
4. Connect a power supply unit directly to your audio input pin. The negative lead should connect to ground, and the positive lead should connect to your input pin. (Despite the name, we are not using the power supply to supply power: we're just using it to provide a voltage so we can test the ADC.) Adjust the voltage on the power supply between 0 and 5 V. Verify that your code reacts to that entire range.
5. Set up your signal generator as follows:
 - Set it to a “high Z” load. Instructions for this were in Section 2.5. Ask your TA if you're not sure how to do this (after rereading that section).
 - Set the waveform to “sine”.
 - Set the frequency to 1 Hz.
 - Set the offset to 2.5 V. This will center your sine wave at 2.5 V instead of 0 V.
 - Set the amplitude to 5 Vpp.²
6. Connect your signal generator to your oscilloscope to check that the generator's working. The oscilloscope should show a signal that has the properties you specified to the generator, with a frequency of 1 Hz, a maximum value of 5 V, and a minimum value of 0 V. You might have to adjust the horizontal and vertical scales and offsets to get it to show the whole waveform.
7. Disconnect your power supply, then connect your signal generator and marvel as your LED pattern varies with the signal. Play around a little with the frequency, offset and amplitude, *but be careful not to allow your signal output to exceed 5 V at any time*. Do your LEDs do what you expect?

4 Preparing for an audio signal

4.1 Connecting a biasing circuit

In the last section, we showed how the ADC works with a range from 0 to 5 V. In this section, we'll connect an audio signal to our ADC input pin, and see what the signal looks like.

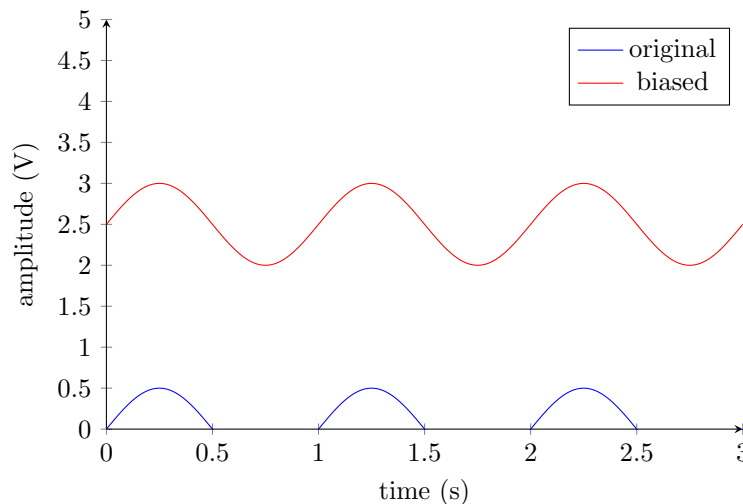
We can't connect the audio signal straight to the ADC input. Audio signals are centred around zero, so are negative around half the time, and the ADC can't detect voltages below 0 V. So we need a circuit that will “lift” the entire signal by 2.5 V, so that it is centered in the middle of the ADC expected input voltage range. We call this the *DC bias*, and a circuit that adds a constant voltage to a signal is called a *biasing circuit*. The circuit we will use looks like this:



We will analyze and understand this circuit in this week's homework. For now, here's the idea: the capacitor “blocks” the DC component (average value), so that the DC bias of the output is determined solely by

²The “pp” means “peak-to-peak”, and means the difference between the highest point and the lowest point.

the voltage divider comprising the two 220 k Ω resistors. At the same time, the capacitor “allows” the AC component (the audio signal) to pass through. The plot below shows what the Arduino’s ADC will see from a 1 V_{pp}, 1 Hz sine wave before (blue) and after (red) biasing.



1. Use an audio cable to connect your phone’s or computer’s audio output to the oscilloscope. Set your phone’s or computer’s volume to a moderate-to-high volume (say, 75% or so). (Be sure to do this, the volume matters and we want the range to be as big as it can be!)

On the audio connector, the middle pin is ground, and the other two are the left and right (stereo) signals respectively. The ground pin should connect to ground. It doesn’t matter which of the other two you use, but when you insert it into your breadboard, make sure the left and right pins aren’t connected together in the same row.

L1: What is the voltage range that it covers, roughly?

2. Add the *biasing circuit* above to your breadboard, connecting it to the Arduino V_{DD} and ground where necessary, but *don’t connect it to the Arduino audio input pin* (yet). Have a look at the output voltage of the biasing circuit (which will become the input voltage to the ADC). Verify that it looks just like the input signal, except raised by 2.5 V.

L2: What is the voltage range that this signal covers?

3. Change your signal generator settings to emulate this voltage range. You should only need to change the amplitude to match your audio signal’s range. Leave the offset at 2.5 V, and leave the frequency at 1 Hz. This signal emulates your voltage range *after* the biasing circuit.

Leave the signal generator connected to your Arduino’s analog input. Your biasing circuit should not be connected to the Arduino.

4. Observe what your Arduino program does now. Do your LEDs run through the entire “range” you allowed for earlier? Have a look at the ADC values being reported in the Serial Monitor.

L3: Roughly, what is the range of numbers you see in the serial monitor? What is the average (center) value? (Ignore clear outliers, and calculate the center using the typical min/max values.)

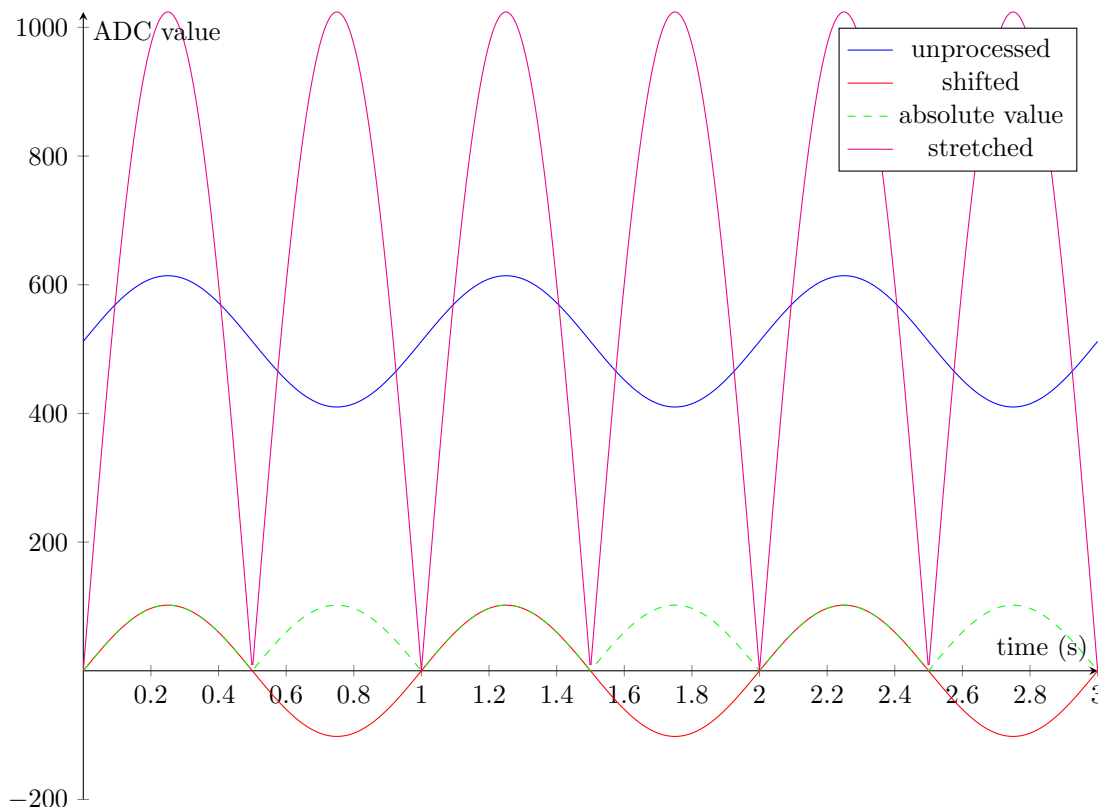
5. Modify your `setLEDs()` so that it only expects the ADC values to be within the range it's getting. Then, upload your modified program to the Arduino and verify that it does what you expect.

4.2 Unbiasing the ADC sample

As you've (probably) currently written the program, the LEDs will turn off (or get dimmer, or something) as the audio signal goes negative (and your biased audio signal gets closer to zero). This isn't quite our desired effect. Both large positive and large negative values correspond to high volumes, so to get a cube that responds to instantaneous amplitude, we'll want to take the absolute value of the signal.

1. Modify `getSample()` so that it returns an "unbiased" ADC value, *i.e.*, one that is centered around zero. You'll need to remove the bias we added in circuitry using the center value that you calculated in L3.
2. Modify `setLEDs()` so that it first takes the absolute value of the sample, and then sets the LEDs array based on the absolute value. Don't forget to adjust for the new range of possible signal values.

The plot below shows the evolution of your signal at different stages of your software, starting at "unprocessed" and progressing through to "stretched", which is your final signal. Test it with the signal generator, as it produces the same signal. (Your biasing circuit still isn't connected to the Arduino.)



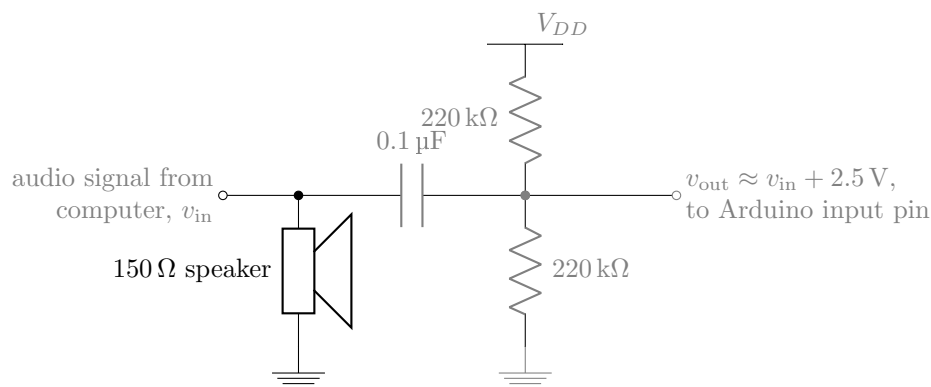
5 Connecting the audio signal

Now, we'll connect the audio signal and watch the LED cube respond to music.

1. Disconnect the signal generator, and connect the output of your biasing circuit to the ADC input pin. Does your cube work?

If it doesn't, check your biased signal on the oscilloscope, and check the ADC values reported on the Serial Monitor by the Arduino.

You might have anticipated, and now noticed, a small problem. Most devices mute the speakers when you plug in headphones, and we've just plugged in what appears to be headphones to your computer or phone. So we'll add a speaker to the circuit, so we can hear what's going on.



1. Solder two wires to a speaker.
2. Connect the speaker between the audio input (before the capacitor) and ground.
3. Does everything work? If so, congratulations! Take a breather, then move on to the next part.

The speaker's quite soft—your computer's audio output isn't really designed to drive anything than headphones. You might prefer to use a second audio connector, connected directly to the first, and plug in headphones or earphones instead. Alternatively, if you want to make the speaker louder, there are two optional amplifier circuits that are in a separate handout on the website.

6 A simple audio spectrum visualizer

In this section, we'll apply our knowledge of the frequency domain to program a basic audio spectrum visualizer. In it, we'll use the `ArduinoFFT` library you installed earlier.

To do this part, download the file `frequency_cube.ino` or `frequency_plane.ino` from the website. Have a read of the `loop()` function, to get an idea of what it does. Basically, it takes a number of samples of the audio signal, runs the FFT on them, and then sets the LEDs array based on the result.

1. Copy the functions `getValue()` and `display()` from your code for Lab 2b. You shouldn't need to modify these; we're building on top of this code.

2. Write `setLEDs()` to populate `ledOn[4][4][4]` based on the values in `bins`.

In the starter code, we've included a line that prints the values of `bins[LOG_N]` every time the function's called. (If you change `FFT_N`, and hence `LOG_N`, you'll need to modify this line to suit.) This will help you get an idea of what range of values to expect in `bins[LOG_N]`.

This is when you implement the pattern you had in mind in prelab question P6. You might choose to do it like the audio spectrum visualizer you see next to graphic equalizers: each column is one frequency bin, and bars go "up" or "down" depending on the value of that bin. Or, you might like to have the bass components (low frequency) occupy the center of the cube, and the treble components (high frequency) occupy the outside. It's up to you!

3. Disconnect your audio signal from the circuit, and reconnect the signal generator. Set the frequency to 250 Hz. (Leave the amplitude at 1 Vpp. It doesn't matter what the offset is.)

L4: Which frequency bin(s) do your LEDs (or Serial Monitor) indicate have significant components (*i.e.*, its value is not close to zero)?

4. Now vary the frequency of your input frequency.

L5: For each bin, at what frequency is it the *only* bin with significant components? (Note: Some other bins maybe still have nonzero components. We are just looking for the frequency at which those nonzero components are minimum.

5. Set the frequency to whatever made your *second* bin the only one with significant components. Now, change the waveform to "square", then "ramp".

L6: Which bins now have significant components?

6. Finally, disconnect the signal generator and connect your music back to the circuit, and marvel at your frequency-responsive LED cube!

7 Analysis

A1: You might have noticed that your lowest bin, `bins[0]`, corresponding to low frequencies, was nonzero even when your audio input is completely silent. Explain how this arises: why does the FFT always show a positive number for the lowest bin? *Hint: We learned earlier in the quarter that the resistors we're using have a 5% tolerance.*

8 Reflection

Individually, answer the questions below. Two to four sentences for each question is sufficient. Answers that demonstrate little thought or effort will not receive credit!

R1: What was the most valuable thing you learned, and why?

R2: What skills or concepts are you still struggling with? What will you do to learn or practice these concepts?

R3: If this lab took longer than the regular 3-hour allotment: what part of the lab took you the most time? What could you do in the future to improve this?