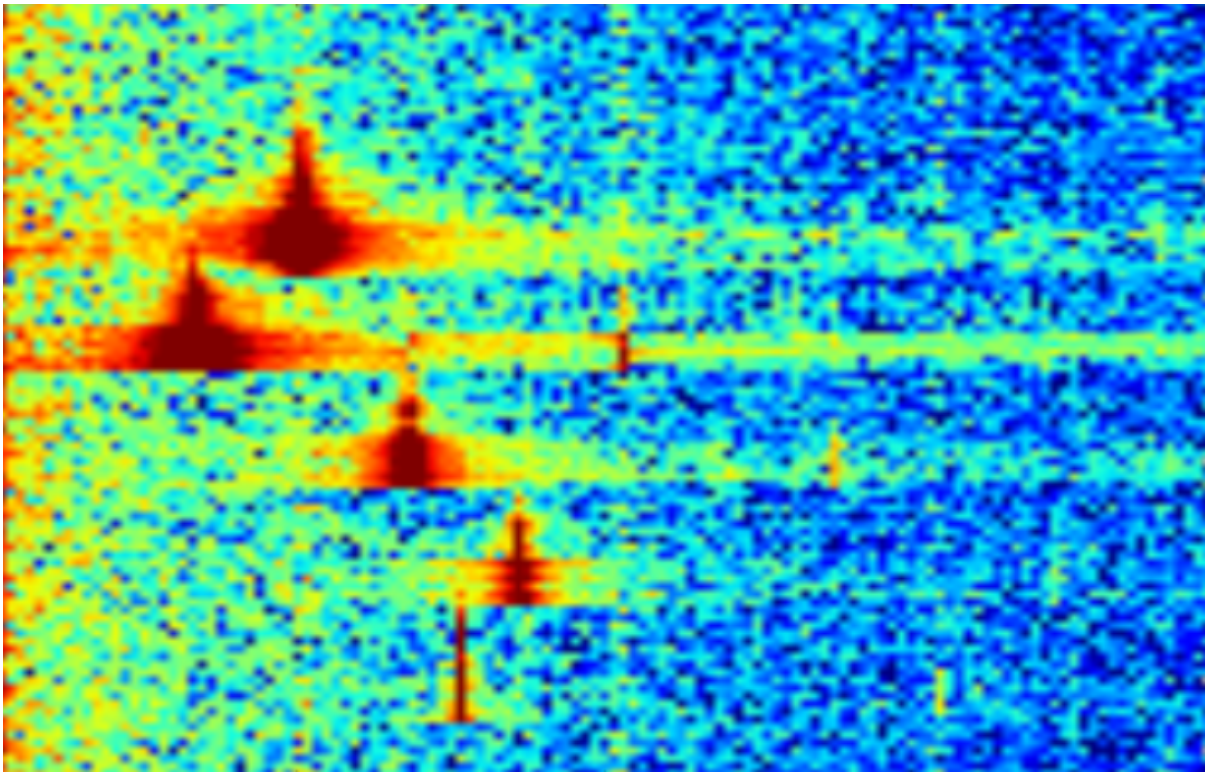
FFT: Fun with Fourier Transforms

Created by Tony DiCola

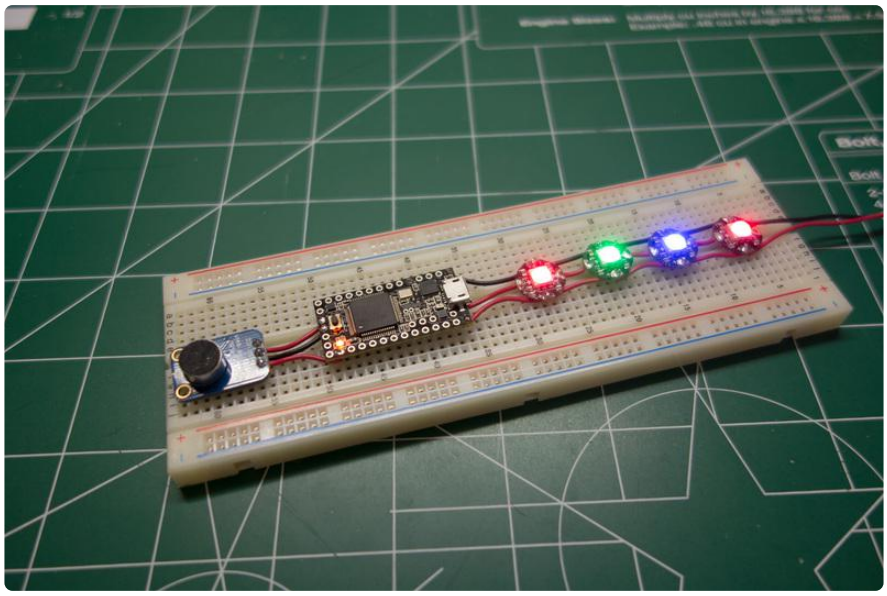


https://learn.adafruit.com/fft-fun-with-fourier-transforms

Last updated on 2023-08-29 02:23:53 PM EDT

Table of Contents

Overview



Have you ever wanted to build devices that react to audio, but have been unsure about or even intimidated by analyzing signals? Don't worry! This guide is an overview of applying the Fourier transform, a fundamental tool for signal processing, to analyze signals like audio. I'll show you how I built an audio spectrum analyzer, detected a sequence of tones, and even attempted to detect a cat purr--all with a simple microcontroller, microphone, and some knowledge of the Fourier transform.

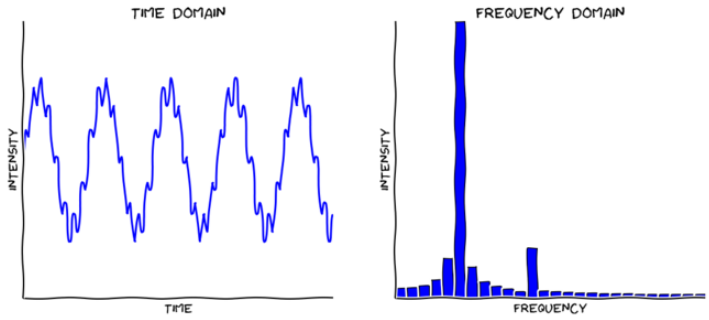
Continue on to learn some background information about the Fourier transform.

Background

What is the Fourier transform?

At a high level the Fourier transform is a mathematical function which transforms a signal from the time domain to the frequency domain. This is a very powerful transformation which gives us the ability to understand the frequencies inside a signal. For example you can take an audio signal and detect sounds or tones inside it using the Fourier transform.

As an example of what the Fourier transform does, look at the two graphs below:



Awesome XKCD-style graph generated by http://matplotlib.org/users/whats_new.html#xkcd-style-sketch-plotting ()

The graph on the left represents a complex signal in the time domain, like what a microphone might produce. This signal is actually the sum of two sine waves. You can see a low frequency sine wave with high intensity gives the signal its overall up and down shape. However a higher frequency sine wave with a lower intensity is added to the signal so it has small rough edges that protrude out as it rises and falls.

The graph on the right is the result of running a Fourier transform on the signal at the left. You can see the Fourier transform output as a histogram, or bar graph, of the intensity of each frequency. It's immediately apparent that two frequencies, the two spikes in the graph, have much stronger intensities than the others. These frequencies actually represent the frequencies of the two sine waves which generated the signal. The output of the Fourier transform is nothing more than a frequency domain view of the original time domain signal.

For more information and background on the Fourier transform, take a look at this link (). This is a great resource because it doesn't dwell on the mathematics and instead focuses on building an intuition of the Fourier transform.

Why should I care about the Fourier transform?

Complex signals made from the sum of sine waves are all around you! In fact, all signa ls in the real world can be represented as the sum of sine waves. The Fourier transform gives us insight into what sine wave frequencies make up a signal.

You can apply knowledge of the frequency domain from the Fourier transform in very useful ways, such as:

- Audio processing, detecting specific tones or frequencies and even altering them to produce a new signal.
- Compression (), how representing a signal in the frequency domain can lead to more compact representations in memory.

- Radar (), detecting how a shift in frequency of reflected electromagnetic signals relates to the distance traveled and speed of an object.
- And many more applications!

How do I use the Fourier transform?

Libraries exist today to make running a Fourier transform on a modern microcontroller relatively simple. In practice you will see applications use the Fast Fourier Transform () or FFT--the FFT is an algorithm that implements a quick Fourier transform of discrete, or real world, data. This guide will use the Teensy 3.0 and its built in library of DSP functions, including the FFT, to apply the Fourier transform to audio signals.

You can find more information on the FFT functions used in the reference here (), but at a high level the FFT takes as input a number of samples from a signal (the time domain representation) and produces as output the intensity at corresponding frequencies (the frequency domain representation).

There are two important parameters to keep in mind with the FFT:

- Sample rate, i.e. the amount of time between each value in the input. Sample rate has an impact on the frequencies which can be measured by the FFT. Nyquist's sampling theorem () dictates that for a given sample rate only frequencies up to half the sample rate can be accurately measured. Keep this in mind as sample rate will directly impact what frequencies you can measure with the FFT.
- FFT size, the number of output frequency bins of the FFT. The FFT size dictates both how many input samples are necessary to run the FFT, and the number of frequency bins which are returned by running the FFT. In practice I found an FFT size of 256 was most usable on the Teensy 3.0. You can go higher to 1024, but a significant amount of the Teensy's memory is consumed to hold the input and output of the FFT.

One other important thing to keep in mind when applying the FFT is that the input and output is typically in the complex number plane (). You might need to transform input and output data between real and complex numbers to use the FFT. For our purposes we're only dealing with real data so the complex coefficients in the input and output are zero.

Finally, the output of the FFT on real data has a few interesting properties:

- The very first bin (bin zero) of the FFT output represents the average power of the signal. Be careful not to try interpreting this bin as an actual frequency value!

- Only the first half of the output bins represent usable frequency values. This means the range of the output frequencies detected by the FFT is half of the sample rate. Don't try to interpret bins beyond the first half in the FFT output as they won't represent real frequency values!

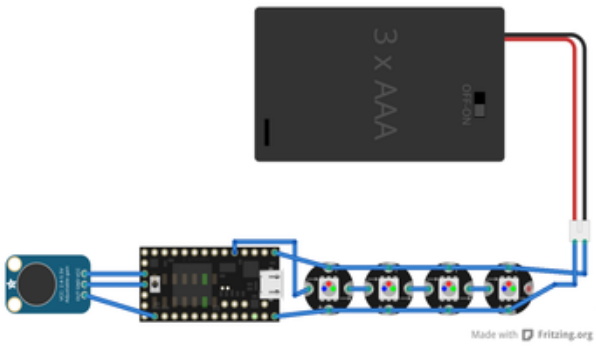
Continue on to learn about the hardware setup for this guide.

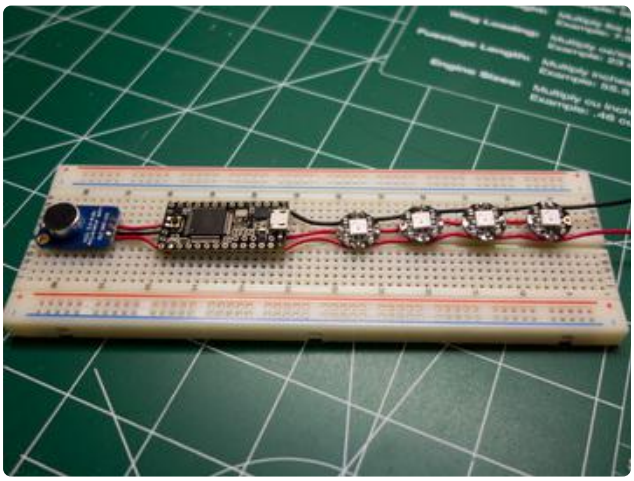
Hardware

You'll need the following hardware for this guide:

- Teensy 3.0 microcontroller (http://adafru.it/1044)
- Microphone with built-in amplifier (http://adafru.it/1063)
- Flora RGB neo pixels (http://adafru.it/1260)

The microphone will be hooked up to an analog input of the Teensy 3.0 which will sample audio and use the neo pixels as a display.





To setup the hardware you'll want to make the following connections:
Connect the microphone output to pin 14 (analog input) on the Teensy.
Connect pin 3 (digital output) on the Teensy to the input pin on a Flora RGB neo pixel.
Connect the output of the neo pixel to the input of another neo pixel. Continue chaining the neo pixel outputs to inputs for all the pixels.
Connect all power and grounds.
Connect a 5 volt power source (such as 3x alkaline AAA batteries) to VIN and ground on the Teensy.
Look at the diagram and photo on the left for an example of how I setup my hardware in a compact single row (good for turning into a wearable device).

Don't worry about the exact pin connections to the Teensy. If necessary you can adjust in the code the analog input pin for the microphone, and the digital output pin for the neo pixels.

Finally, turn up the gain on the microphone amplifier to its maximum (about 1V peak to peak output) by turning the small trim potentiometer on the back all the way to the left.

Why Teensy 3.0?

This guide uses the Teensy 3.0 microcontroller for a couple reasons. Teensy 3.0 is a very powerful device that runs a full 32-bit ARM Cortex-M4 processor at 48 mhz. With such a powerful processor it's easy to sample audio and run an FFT in real time without resorting to low-level commands outside the Arduino/Teensyduino programming library. Furthermore the ARM Cortex-M4 core on the Teensy has native support for running Fourier transforms and other signal processing functions with the CMSIS DSP math library ().

However you can still apply the principles and code from this guide to other microcontrollers like Arduino. Look for existing FFT libraries to give you the code you need for running a Fourier transform, and be aware of how quickly you can sample audio with the microcontroller. T ()his tiny music visualizer guide () is a great example of running an FFT and analyzing audio in real time on an Arduino.

Continue on to get the software necessary for this guide.

Software

Code

To follow this guide you'll want to download the following code and unzip it somewhere convenient:

Download Code

The code includes:

- spectrum, a folder with a Teensyduino sketch for the spectrum analyzer.
- toneinput, a folder with a Teensyduino sketch for tone detection.
- Spectrogram.py, a python script to display a real-time spectrogram from the hardware.
- SpectrogramUI.py, the user interface code used by Spectrogram.py.
- SpectrogramDevice.py, an abstract class for extending the spectrogram to other devices in the future.
- SerialPortDevice.py, an implementation of SpectrogramDevice.py to interface with the hardware over a serial port.

Dependencies

To run the sketches you'll want to make sure you have Teensyduino installed. Follow these instructions () to download and install Teensyduino.

To run the Spectrogram python script you'll need python 2.7 () and a few libraries installed:

- matplotlib (), a library for plotting data.
- NumPy (), a library for numeric computing.
- PySide (), a python binding to the Qt user interface library.
- pySerial (), a library for serial code IO.

You can install python and these dependencies manually, however be warned the installation on Windows and Mac OSX is not easy. As an alternative you can download a pre-built distribution of python and the necessary scientific computing libraries. The Anaconda distribution () by Continuum Analytics is what I recommend--it's free, includes the necessary dependencies, and can install side by side with existing python installations easily. Canopy () by Enthought is another popular python scientific computing distribution too.

Assuming you installed a distribution such as Anaconda, you'll want to install the pySerial library (which is not included in the distribution) by executing the following command (be sure the anaconda /bin directory is in your path before executing):

pip install pyserial

Continue on to learn about the applying the Fourier transform to build an audio spectrum analyzer.

Spectrum Analyzer

Let's look at the first application of Fourier transforms by creating an audio spectrum analyzer. A spectrum analyzer () is used to view the frequencies which make up a signal, like audio sampled from a microphone. Let's make the hardware visualize audio frequencies by changing the intensity of LEDs based on the intensity of audio at certain frequencies.

To get started, load the 'spectrum' sketch from the code download in Teensyduino. If necessary, adjust the AUDIO_INPUT_PIN and NEO_PIXEL_PIN variables at the top of the program to the values for your hardware. Compile and load the sketch onto your hardware.

If the sketch loaded successfully you should see the neo pixel lights start to pulse and flash based on the intensity of audio read from the microphone. Each pixel represents a different window of audio frequencies, with the first pixel representing the lowest frequencies and the last pixel representing the highest frequencies. The sample rate of the audio will determine the total range of frequencies--remember because of Nyquist's thereom () only frequencies up to half the sample rate can be analyzed.

By default the spectrum program runs with a sample rate of 9000 hz and an FFT size of 256 bins. This means audio from 0 to 4500 hz can be analyzed. Each FFT result bin will represent about 35 hz of frequencies (calculated by taking sample rate divided by FFT size). The spectrum analyzer program works by assigning a range of frequencies to each LED, calculating the average intensity of the signal over those frequency

ranges (by averaging the value of the FFT output bins associated with the frequency range), and lighting the LED's appropriately.

You can adjust some of the parameters to the spectrum analyzer by changing the variables and reloading the sketch, or by sending commands over the serial port to the device. For example try changing the sample rate by opening the serial monitor in Teensyduino and sending this command:

SET SAMPLE_RATE_HZ 4000;

You can also read the value of a variable by sending a get command, such as:

GET SAMPLE_RATE_HZ;

Be sure to type the command exactly, including the semicolon at the end. With a sample rate of 4000 hz you should see the LEDs pulse a little slower (a slower sample rate will take longer to fill the 256 samples necessary for running the FFT), and higher frequencies more easily light up from talking or other sounds.

Other variables you can adjust include:

- SPECTRUM_MIN_DB - This is the minimum intensity in decibels (from 0 to 100) which corresponds to the lowest LED light intensity. Values around 20 to 30 decibels are best for audio in a quiet room.
- SPECTRUM_MAX_DB - This is the maximum intensity in decibels which correpends to the highest LED light intensity. Values around 60-80 decibels are best, unless you're in a very loud environment.

The video below shows the spectrum display listening to powerful orchestral music with a sample rate of 4000 hz, 40 minimum decibels and 60 maximum decibels. You can see the effects of instruments playing at different frequencies and how the LEDs representing those frequencies respond.

Continue on to learn about how to better visualize the audio spectrum with your computer.

Spectrogram Tool

Flashing LEDs is a simple but limited view of the audio frequency spectrum. To get a full view of the frequencies let's hook the hardware up to our computer and build a better spectrogram. A spectrogram () is a visual representation of the frequencies in a signal--in this case the audio frequencies being output by the FFT running on the

hardware.

With the spectrum program from the last page still loaded on your hardware, make sure the hardware is connected to your computer's USB port so you have a serial connection to the device. You will also want to make sure you have all the python software dependencies installed (through a distribution such as Anaconda)--go back to the software page to learn more about how to install these dependencies if necessary.

Also if you're using a distribution like Anaconda, make sure the anaconda/bin folder is in your system path before continuing. You can check this by running the command 'p ython --version' (without quotes) at the command prompt. You should see something like 'Python 2.7.5 :: Anaconda 1.7.0 (64-bit)' in the response.

Now run the Spectrogram.py file included in the code download. Do this by running the following command:

python Spectrogram.py

Note for Mac OSX: On Mac OSX you might need to do the following first to work around a matplotlib bug ():
1. First set the QT_API variable in your terminal session to the value 'pyside' by executing:
export QT_API=pyside
2. Next start the Spectrogram.py program by executing (notice the python.app instead of python command):
python.app Spectrogram.py

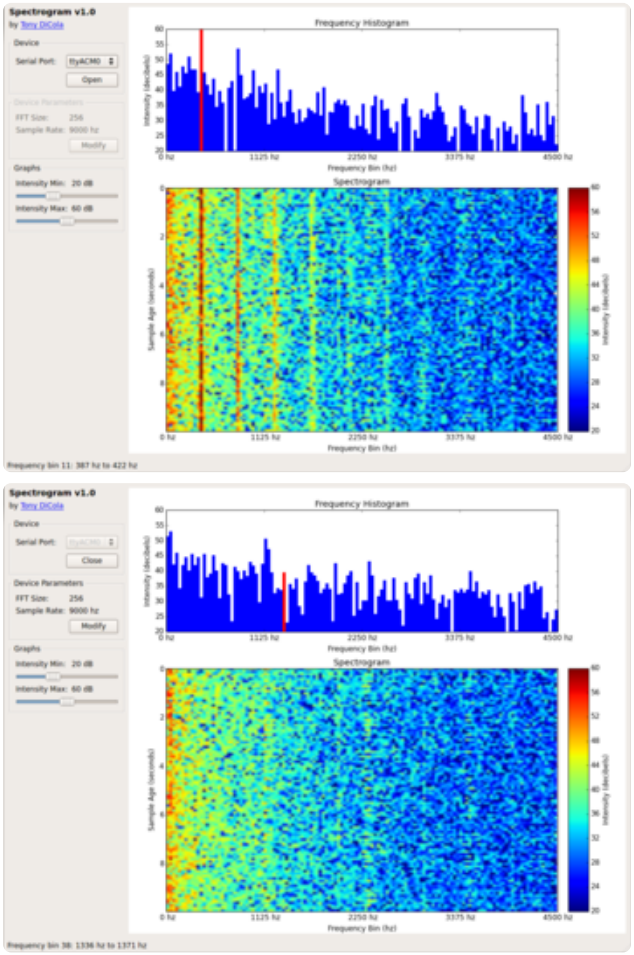
You should see a window load with empty graphs on the right and controls/ parameters on the left. Click the Serial Port combo box on the left to select the serial port which your hardware is connected to, and click the Open button to establish communication with the device.

Once the device is connected you should see the two graphs on the right start to display data. The top graph is a histogram of the latest frequency intensities. This is similar to what the LEDs are displaying, but showing you the full spectrum of audio frequencies. The height of the bars in the chart represent the intensity in decibels () of the audio at a frequency. Rolling your mouse over the chart allows you to highlight specific bars which represent bins in the FFT output. Look in the status bar at the bottom left of the program to see the exact frequency of the selected bin.

The bottom graph is a waterfall display which shows the intensity of frequencies over

time. The oldest samples are at the bottom of the graph. You can see new samples come in at the top and roll down to the bottom over time. The frequency of the signal is on the X-axis so low frequencies are to the left and high frequencies to the right. The color of each point represents the intensity of the audio, and the bar at the far right shows the scale of colors to intensity values.

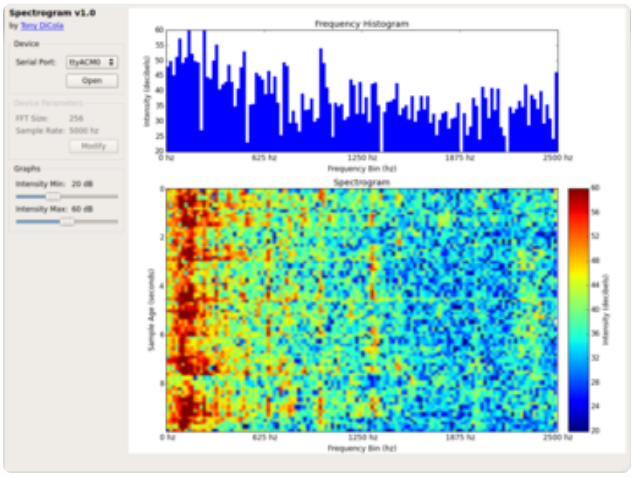
Finally on the left you can see some status and controls to manipulate the hardware. The FFT size and current sample rate should be displayed in the Device Parameters group. The Graphs group has a few sliders to change the scale of intensity values in the graphs--try dragging them up and down to see how the graphs change.



The pictures on the left show a spectrogram of audio in a quiet room. You might notice straight lines running down in the waterfall graph. This appears to be noise being picked up from the LEDs. You can disable the LEDs by sending the following command to the hardware from the serial monitor in Teensyduino (make sure to close the connection in the spectrogram application first):

SET LEDS_ENABLED 0;

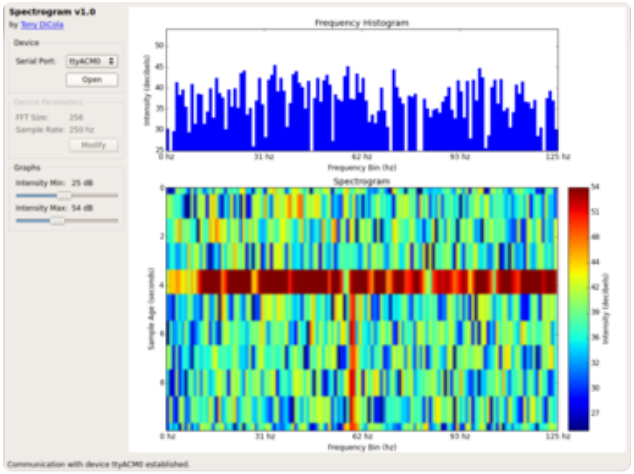
With the LEDs disabled you should see the bars disappear from the waterfall graph, like the second photo shows.



The spectrogram is a powerful tool we'll use in this guide to analyze audio. For now try playing some audio or making noise to see how it's represented on the graphs. For example the picture on the left is showing the spectrogram of audio from the opening of this orchestral piece ().

You can see low frequencies in the 50-300hz range are quite intense. The powerful brass instruments like the trombone, trumpet, and french horns in the music are generating a lot of audio at these frequencies.

You can also see many of the intensity peaks are at evenly spaced frequencies. Some of these are harmonics () generated by the instruments. For example a violin string vibrating to play a note at a specific frequency is also generating sound at integer multiples of the note's frequency. How an instrument generates harmonics contributes greatly to the timbre (), or character of sound, of the instrument.



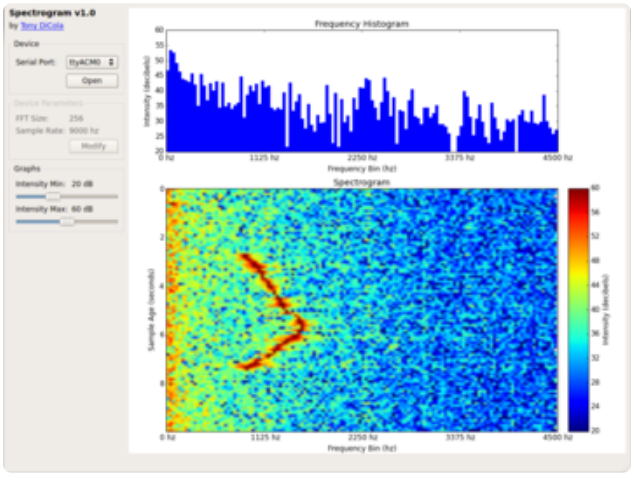
Finally, try changing the sample rate by clicking the Modify button in the Device Parameters group on the left. You can use values from ~150hz to ~9000hz. Notice as the sample rate decreases, both the range of frequencies decreases (at a rate of half the sample rate) and the amount of time to get a new sample increases (because it takes longer to fill the 256 samples for running the FFT).

At lower sample rates you can see each frequency bin is smaller and represents a tighter range of frequencies. The image on the left is audio captured at a sample rate of 250hz. This means each frequency bin from the FFT represents about 1 hz. It's interesting to see a strong line at 60hz at the bottom of the graph. This is from a 60hz vibration being picked up when the hardware is resting on my desk--I suspect the fans and hard drives in my computer are generating this noise. You can see when I picked up the device off the table there was a strong bump in intensity (red horizontal line in the middle) and then the 60hz vibration disappeared (newest samples at the top).

Continue on to learn about how to use the spectrogram to analyze and detect a sequence of tones.

Tone Detection

An interesting application of the Fourier transform to audio is detecting specific frequencies or tones. You might imagine building a device which uses a sequence of tones as a form of input. For example a door lock that only opens when you whistle the right tune. To demonstrate this, let's make the hardware built in this guide respond to specific sequence of tones.

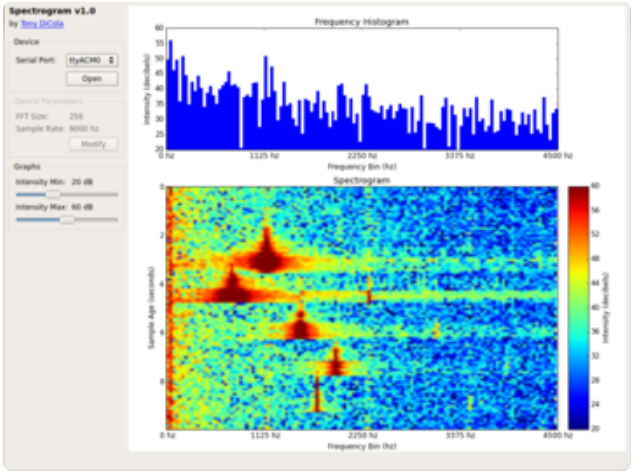


First we need to understand what frequencies make up the sequence of tones to detect. The spectrogram tool from the previous page is the perfect tool for this analysis.

For example the image on the left is a spectrogram of me whistling up and down slowly. Read the graph starting from the bottom and going up so you see the slow rise and fall in frequencies as I whistle.

If I wanted to detect this sequence I just need to look for a series of strong intensities from the FFT output at the rising and falling frequencies of the whistle.

Taking things a step further, the image on the left represents a spectrogram of 5 notes being played from an instrument. Again, read the graph from the bottom up. Low frequencies are to the left and high frequencies are to the right.



From analyzing the spectrogram I can see the notes I want to detect are highest in intensity between these frequencies:
Note 1: 1,723 to 1,758 hz
Note 2: 1,934 to 1,969 hz
Note 3: 1,512 to 1,546 hz
Note 4: 738 to 773 hz
Note 5: 1,125 to 1,160 hz
The toneinput sketch in the code is a program which looks for this sequence of notes and flashes the LEDs when it detects them being played in order. Look at the code's toneLoop() function to see how it uses an array of tone frequencies, count of current position in the tone frequency array, and threshold of intensity to detect the sequence.

Can you guess the tune being played from looking at the spectrogram above?

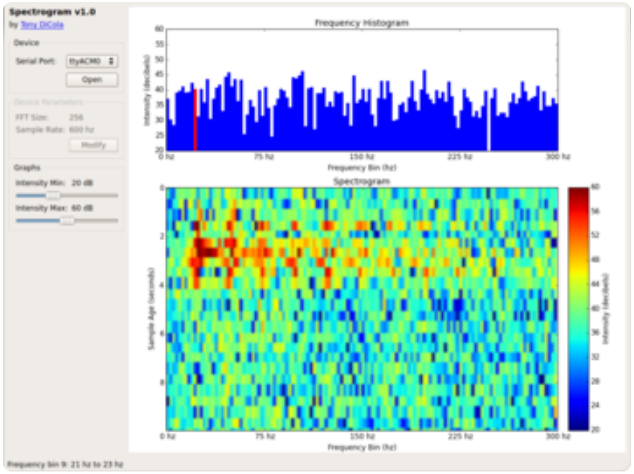
Here's a video below of the tone sequence detection with the answer. As you hear the tones played, look at the spectrogram to see how each note relates to output on the graph.

If you're curious the audio is being generated by Garage Band on an iPad, with the 'Fifties Sci Fi' lead keyboard playing A5 B5 G4 G3 D4.

Continue on to look at another application of the Fourier transform, attempting to detect cat purrs.

Cat Purr Detection

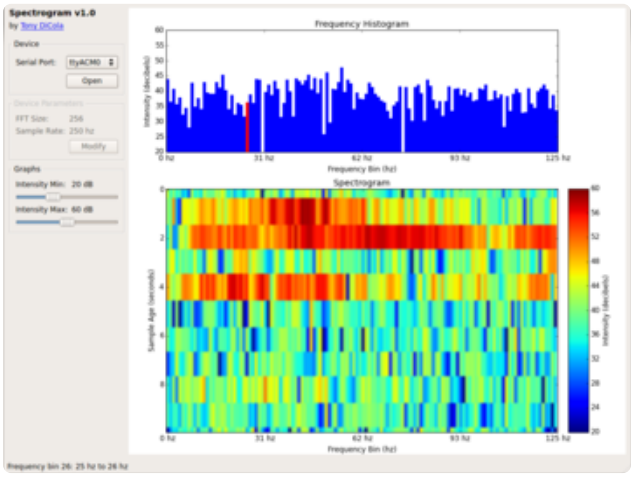
Armed with knowledge of the Fourier transform and tools to analyze audio let's investigate detecting another signal, a cat purr.



To the left is a spectrogram of my cat purring very clearly and loudly into the microphone. I've dropped the sample rate down to 600 hz so lower frequencies are more visible. You can see a very clear bump in intensity starting around 20-25hz and appearing again up at higher harmonic frequencies. This matches what I expect to see based on data which shows domestic cats have been measured to purr around 21-27hz ().

Based on this measurement it looks like detecting strong intensities around 21-23 hz would detect a cat purr.

You can also see in the second image the hardware can be put together into a simple wearable collar. I took two pieces of velcro and sandwiched the hardware in the center. Holes were cut out to show the LEDs, USB mini port, and microphone.



Unfortunately in practice I found trying to detect purrs reliably is quite difficult. The spectrogram to the left shows more typical data, where the movement of the cat against the microphone (or even biting the microphone!) generates a lot of noise which obscures the purr. The noise unfortunately leads to false positives which make the purr detection unreliable.

This investigation of cat purr detection highlights the challenge of dealing with noise when analyzing signals. It's relatively easy to get a signal and break it down into it's component frequencies with the Fourier transform. However with weak signals or noisy environments it can be quite challenging to make meaning of the output.

Reliable purr detection will likely require more work to analyze the audio and try to remove sources of noise. Some ideas to consider are:

- Run filters to remove higher frequency noise from the audio.
- Look into better ways to place the microphone so it's less susceptible to noise from movement or rubbing.
- Look into alternative ways of detecting purr vibrations. Could an accelerometer detect the ~20hz vibration without as much noise?

I'm open to feedback or ideas that might help better detect purrs. You can look at this github repository () for future info on my attempts at purr detection.

Continue on for a summary and some ideas for other applications of the Fourier transform.

Summary

In summary the Fourier transform is a very powerful function to transform from time domain to frequency domain representations of a signal. When a signal such as audio is in the frequency domain you can process the signal in interesting ways to create cool visualizations, or even detect specific tones.

This guide only scratches the surface of Fourier transform applications. You might consider more applications such as:

- Audio filtering, the CMSIS-DSP library actually has built in functions () for applying filters to remove and enhance audio frequencies.
- Building an instrument tuner which detects how far an audio signal is from a reference frequency. Add sound synthesis to even create an auto-tune device!
- Radar--the Fourier transform and frequency analysis is used extensively in radar systems. Check out this talk () on building a radar system at home!
- Analyzing the frequency of signals from other sensors such as accelerometers, light sensors, pressure sensors, temperature sensors, and more.

Can you think of more interesting applications?