Theoretical Introduction

June 14, 2018

John Marshall
j21marshall@gmail.com
June 2018

I'm currently developing a program in Python 3 for composing and saving WAV and MIDI files in non-standard Western (i.e. microtonal/xenharmonic) tuning systems called Vihuela. This document demonstrates the theory and some of the basic functionality behind my project so that someone less familiar with the relevant musicology, acoustics, and discrete signal processing can understand.

I have written this document with the hope that a person with a background in Python can replicate the results and understand them, and that a person without a programming background (but some experience with math) can still understand what is being communicated without following the code.

The unsurprising first step is our library imports, some of which you may need to install if you want to follow along with the code:

```
In [1]: import numpy as np
        import scipy.fftpack as fft
        import wave
        import pyaudio
        import matplotlib.pyplot as plt
```

Our signals will be saved as NumPy arrays because they are useful for array operations and Fourier transforms from SciPy's fftpack. PyAudio and wave are needed to render these signals as audio. MatPlotLib is used in this demo for visualizations (and is useful for debugging), but will not be used in actual Vihuela versions.

1 Basic Theory of Discrete Audio Signals

In the real world, sound is a continuous longitudinal wave. Within the audible range (the average person is sensitive to between 20 and 20,000 Hz), we can hear a tone of any frequency, as well as any combination of frequencies. However, digital audio is a discrete transverse wave. Converting an analog signal to digital is like taking a measurement of the height of a buoy every second to characterize the waves in the ocean. Significant information will be lost, and that which is collected can be misleading. Therefore, it is important to follow certain rules when producing digital audio.

1.1 Sampling and generating a tone

In Western tuning, it is conventional to define the note A4 as having frequency 440 Hz. We'll start by generating a sinusoidal waveform with this frequency for .03 seconds, i.e.

$$x(t) = \sin(2\pi 440t), t \in [0, .03)$$

To avoid aliasing when processing a discrete-time signal, we follow the Nyquist-Shannon sampling theorem: when sampling a continuous-time signal, the sampling rate must be twice the frequency of the highest frequency component in the signal. Mathematically this can be written

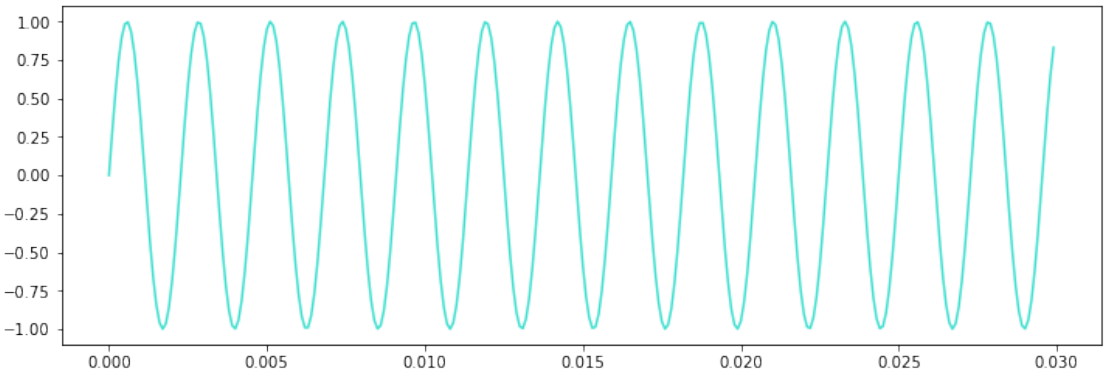
$$s(t) = \sum_{k=1}^{n} a_k \sin(Tkt + c_k)$$

$$f_{sampling} \geq 2Tn$$

for a continuous periodic signal s with period T. Therefore, for our x, the sampling rate must be at least $2 \times 440\text{Hz} = 880\text{Hz}$. This yields a maximum sampling interval of $\frac{1}{2\pi 880} = .0001809$ seconds. Let's choose ours to be .0001 seconds. The result is shown below.

NOTE: It is traditional to use the notation $s[n]$ instead of $s(t)$ for a discrete signal so that it is clearly communicated that the signal is not continuous and n is countable. I have chosen to use $s(t)$ for the purpose of this tutorial because it is more intuitive for someone unfamiliar with sampling to continue to think in terms of time.

```
In [2]: t = np.arange(0,.03,.0001)
        x = np.sin(2*np.pi*440*t)
        plt.figure(figsize=[12,4])
        plt.plot(t, x, color='turquoise')
        plt.show()
```



1.2 Harmonics and Fourier analysis

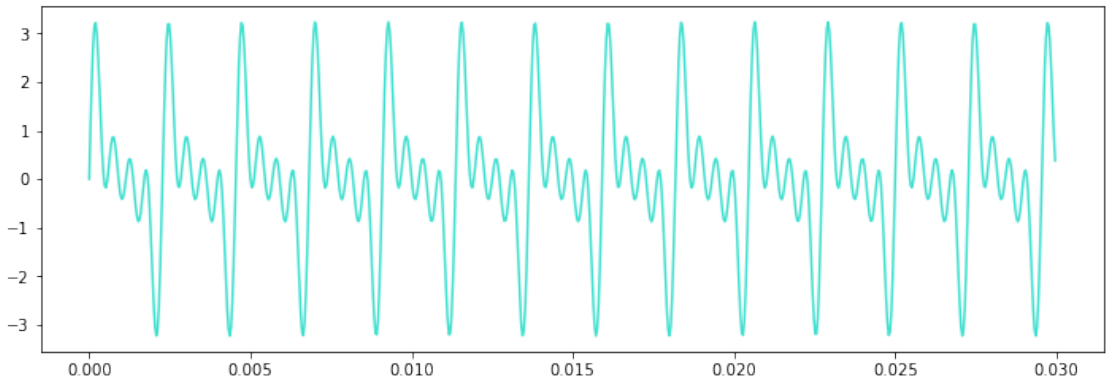
When you pluck A4 on a guitar, the note it produces consists not only of the 440 Hz waveform but also those of its harmonic series, e.g. 880 Hz, 1320 Hz, etc., which occur at integer multiples of the base frequency. The relative amplitudes of the contributions of each of these harmonics determines timbre, which is what gives each real-world instrument its unique sound.

Let's generate another 440 Hz tone, but this time with its first three harmonics, 880 Hz, 1320 Hz, and 1760 Hz. We'll set the amplitude of each of these harmonics to be 1/3 that of the base, so the total signal is

$$x(t) = \sin(2\pi 440t) + \sum_{k=2}^{4} \frac{1}{3} \sin(2\pi 440kt)$$

Note that according to the Nyquist-Shannon sampling theorem, our sampling rate must now be $f_{sampling} = 2 \times 4 \times 440 = 3520$Hz to preserve behavior, so the new maximum interval is $\frac{1}{2\pi 3520} = .00004521$s. We'll use .00004 s.

```
In [3]: t = np.arange(0,.03,.00004)
        x = np.sin(2*np.pi*440*t)
        for k in range(2,5):
            x += np.sin(2*np.pi*440*k*t)
        plt.figure(figsize=[12,4])
        plt.plot(t, x, color='turquoise')
        plt.show()
```

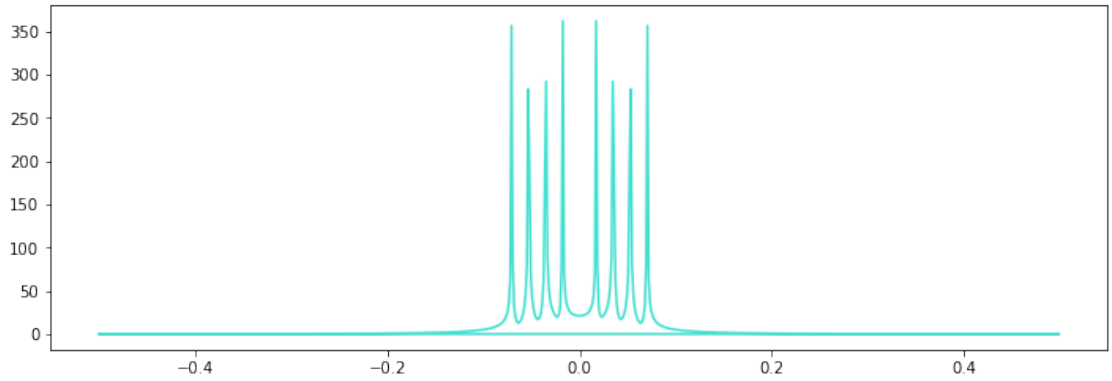


If you want to identify all the frequencies making up a signal, the best tool to use is the discrete Fourier transform (DFT). The Fourier transform is a complex-valued function of frequency whose magnitude is the amplitude of the sinusoid at that frequency within the signal (additionally, its complex phase is proportional to the offset of the sinusoid; this will be covered later). This is why the Fourier transform is said to be in the frequency domain while the actual audio signal is in the time domain. The mathematical definition of the DFT S of function s is

$$S(\omega) = \sum_{t=0}^{T-1} s(t) e^{-2\pi i \omega t/T}$$

Clearly, the math is very tedious. Luckily, functions in the SciPy fftpack do the work for us. This is what the magnitude of the Fourier transform of the above signal looks like:

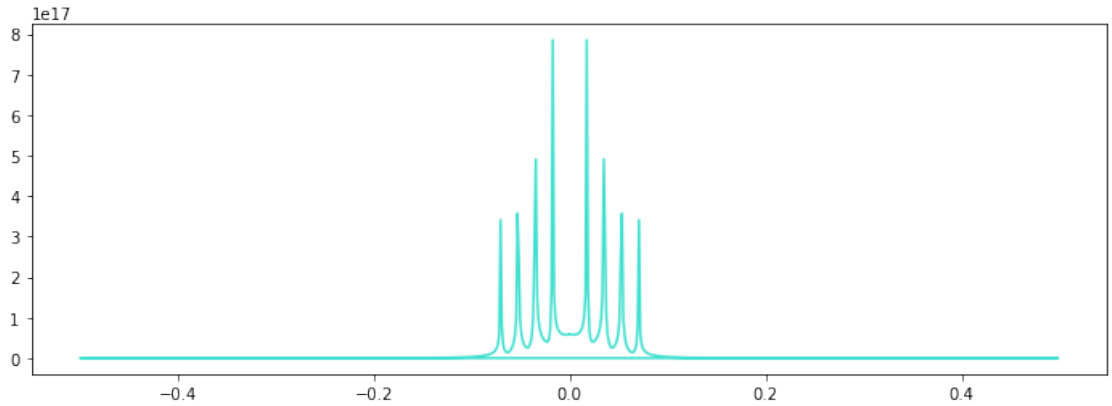
```
In [4]: xfft = fft.fft(x)
        tfft = fft.fftfreq(len(x))
        plt.figure(figsize=[12,4])
        plt.plot(tfft, abs(xfft), color='turquoise')
        plt.show()
```



The symmetry about $\omega = 0$ is unimportant for us; in fact, half of the transform can be ignored (assume it's a mirror image of the other side). This leaves us with four equally-spaced peaks, which correspond to each of the four frequencies we generated (each of which are separated by 440 Hz).

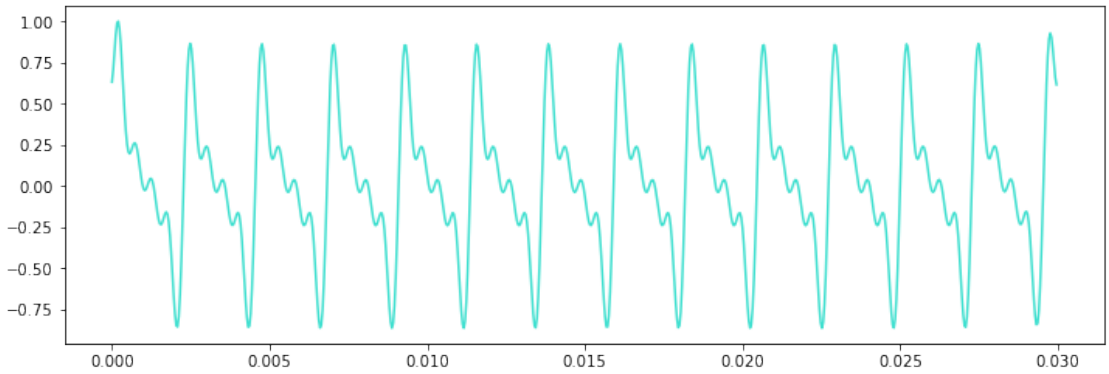
By changing the Fourier transform, we can change the timbre of the sound. Let's see what effect changing the amplitudes has on the time-domain shape of the signal.

```
In [5]: for k in range(len(xfft)):
            xfft[k] = abs(k-len(xfft)/2)**7/(len(xfft)/2)*xfft[k]
        plt.figure(figsize=[12,4])
        plt.plot(tfft, abs(xfft), color='turquoise')
        plt.show()
```



The function ifft performs the inverse discrete Fourier transform, returning to the time domain.

```
In [27]: x = np.real(fft.ifft(xfft))
         x = x/np.amax(x)
         plt.figure(figsize=[12,4])
         plt.plot(t, x, color='turquoise')
         plt.show()
```



Comparing this to the original tone with harmonics, damping the amplitudes at higher frequencies appears to make the signal smoother.

Modulating the relative amplitudes in the Fourier transform is an important part of tuning the sound of a digital instrument.

2 Basic Theory of Tonality

Music is said to be the universal language, however this theory doesn't hold true under scrutiny. A song that sounds happy to a person of one culture can be perceived as sad by a person from another. A chord in a Western scale might sound clean to us would sound dissonant and unsettling to a villager from Sumatra.

Instead, different musical traditions are like different languages: they share ideas of syntax and semantics, but are still very different procedurally from place to place. The theory of tonality underlies the melodic differences between cultures and a thorough understanding of it is useful in making syncretic music.

2.1 Cents and tones

If A4 is 440 Hz, it's intuitive that A3 (the next A down) is half this, 220 Hz, and A5 (the next A up) is twice this, 880 Hz. We are used to calling the interval between an octave, and there are always twelve notes (called semitones) between them (A#/B, B, C, C#/D, ...). We're also used to the idea that these notes are evenly spaced, such that if you play a melody E-E-F-G, you can transpose it up to B-B-C-D and the melody will still be identifiable but in a different key.

However, we also must realize that this equal spacing of notes (called equal temperament) does not imply equal spacing of tone frequencies. Think about how A3 is 220 Hz, A4 is 440 Hz, and A5 is 880 Hz. That means the spacing between A3 and A4 is 220 Hz, while between A4 and A5 it's 440 Hz, even though in both cases they are separated by one octave (twelve semitones).

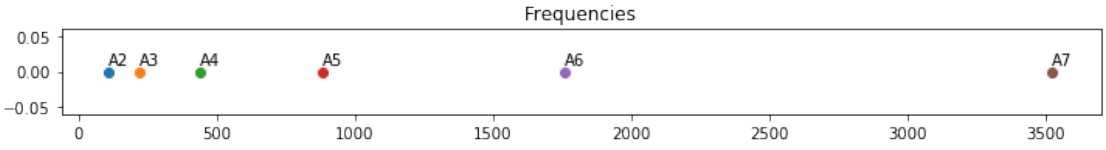
This is why a different unit of measure is needed to identify the interval between tones: the cent. Like voltage, which is defined as an electric potential difference between two points, cents are a relative unit of measure: they can describe the perceived spacing between two tones, but cannot describe a tone exclusively. I could say that the difference between C# and C is that C# is 100 cents higher, but not that C# is 100 cents because I haven't given a basis. The definition of cents between two notes of frequency f_1 and f_2 is

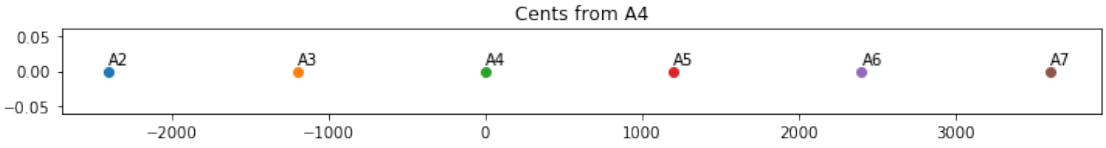
$$n = 1200 \log_2(\frac{f_2}{f_1})$$

The code below visualizes how an exponential scale of frequencies corresponds to a linear scale of cents.

```
In [7]: freqs = [110,220,440,880,1760,3520]
        plt.figure(figsize=[12,1])
        plt.title('Frequencies')
        for i, f in enumerate(freqs):
            plt.scatter(f,0)
            plt.text(f,.01,'A'+str(i+2))
        plt.show()

        plt.figure(figsize=[12,1])
        plt.title('Cents from A4')
        for i, f in enumerate(freqs):
            cents = 1200*np.log2(f/440)
            plt.scatter(cents,0)
            plt.text(cents,.01,'A'+str(i+2))
        plt.show()
```





Each octave now consists of 1200 cents, which means that each semitone is 100 cents.

2.2 Equal temperament

The easiest way to find the frequencies of the semitones that populate these octaves is to use the property

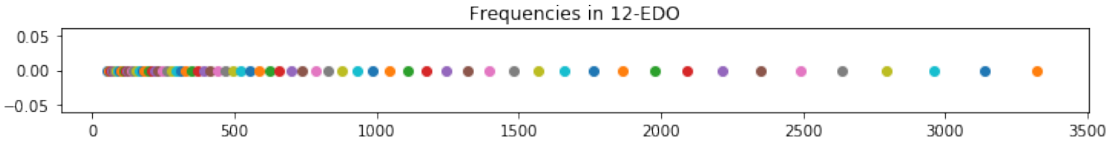
$$s(n) = 440 \times 2^{\frac{n}{12}}$$

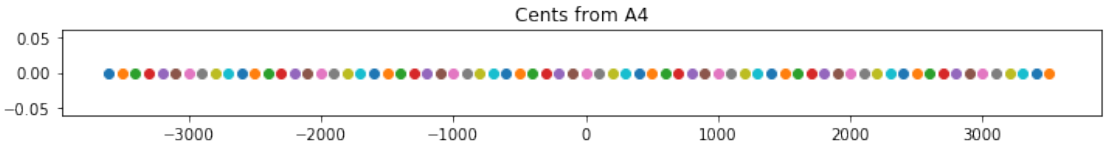
for a note n semitones (aka half steps) away from A4 (440 Hz) (this can be either positive or negative). For example, since C5 is 3 half steps above A4, C5 has a frequency in Hz of $440 \times 2^{\frac{3}{12}} = 523.2511$.

This is called twelve-tone equal temperament, or 12-TET. We will use the notation 12-EDO (equal distribution of the octave), which is more popular among microtonal musicians. Again, the frequencies and cents distributions are visualized below.

```
In [8]: plt.figure(figsize=[12,1])
        plt.title('Frequencies in 12-EDO')
        freqs = []
        for n in range(-12*3,12*3):
            freqs.append((2**(1/12))**n*440)
            plt.scatter((2**(1/12))**n*440,0)
        plt.show()

        plt.figure(figsize=[12,1])
        plt.title('Cents from A4')
        for f in freqs:
            cents = 1200*np.log2(f/440)
            plt.scatter(cents,0)
        plt.show()
```





This idea can be generalized to tonal systems with more or fewer than 12 tones in an octave. For example, both 24-EDO and 17-EDO have been employed in music from the Middle East. In order to establish a list of frequencies in an equal temperament scale, we need both a base frequency (like A4 = 440 Hz above) and a number of divisions per octave (like 12 above). Once we have these, a generalized form of the above equation can be used:

$$s_m(n) = f_0 \times 2^{\frac{n}{m}}$$

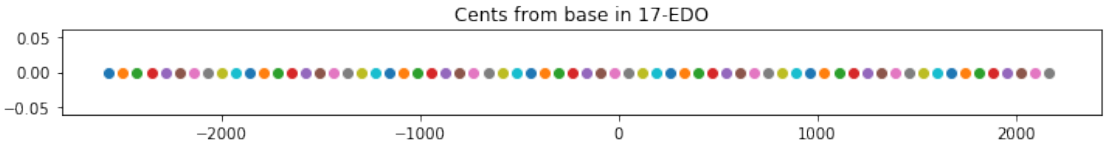
where f_0 is the base frequency, m is the number of divisions per octave, and n is the number of steps from the base note.

Below is an implementation of a function that generates an equal temperament scale from a number of divisions per octave and a base frequency, with an example usage for 17-EDO, base 400 Hz shown.

```
In [9]: def genedo(edo,base,rnge):
            nfreqs = []
            for n in range(-edo*rnge,edo*rnge):
                nfreqs.append((2**(1/edo))**n*base)
            return nfreqs

        freqs = genedo(17,400,2)

        plt.figure(figsize=[12,1])
        plt.title('Cents from base in 17-EDO')
        for f in freqs:
            cents = 1200*np.log2(f/440)
            plt.scatter(cents,0)
        plt.show()
```



2.3 Just intonation

Western music wasn't always 12-EDO; in fact, it wasn't until the Scientific Revolution of the 17th century that 12-EDO became the standard. In Antiquity, Greek theorists developed competing methods of tuning scales. One of the most influential of these was Pythagorean tuning. In this system, intervals between notes are exclusively built from frequency ratios of 3:2 and 2:1, called generators. This means the notes in a Pythagorean scale can be generated by

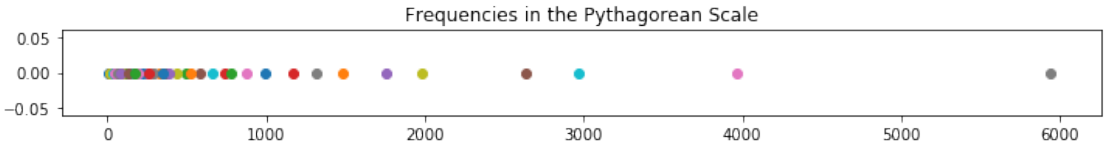
$$s(n,m) = (\frac{3}{2})^n \times 2^m f_0$$

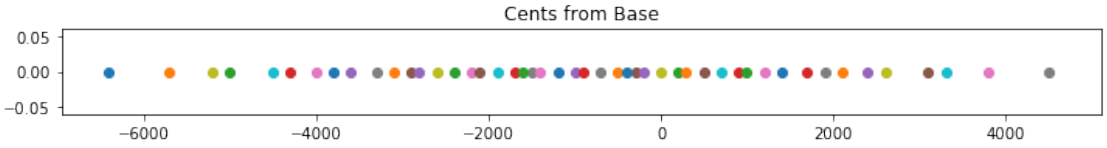
where n and m are integers. This is shown below.

```
In [24]: plt.figure(figsize=[12,1])
         plt.title('Frequencies in the Pythagorean Scale')
         freqs = []
         for m in range(-3,3):
             for n in range(-4,4):
```

```
            freqs.append((3/2)**n*2**m*440)
            plt.scatter((3/2)**n*2**m*440,0)
plt.show()

plt.figure(figsize=[12,1])
plt.title('Cents from Base')
for f in freqs:
    cents = 1200*np.log2(f/440)
    plt.scatter(cents,0)
plt.show()
```





It is clear from the graph of cents that this system is not equal temperament. The Pythagorean scale does not use all of the tones shown above; only certain (n, m) pairs are used, e.g. for the major second (2,-1), minor seventh (-2,2), minor second (-5,3), etc. One property of these is that n and m always have opposite signs.

Any scale derived from generators that are ratios of integers is said to have just intonation.

A scale is considered equal tempered but not just intoned if its generator is irrational, as is the case for 12-EDO, whose generator is $\sqrt[12]{2} = 1.059...$. The only possible scale that is both just intoned and equal tempered is the one purely comprising octaves (in other words, with a sole generator of 2).

A general description of a just intonation scale with K generators $g_1 ... g_K$ (all of which are positive rational numbers, usually explicitly treated as ratios of integers) is

$$s_{(g_k)_{k=1}^K}((n_k)_{k=1}^K) = f_0 \prod_{k=1}^{K} g_k^{n_k}$$

where $(n_k)_{k=1}^K$ is the list of steps along each generator's path.

Because it is much more easily attainable by manipulating the geometry of an instrument, just intonation is far more common in traditional world music than equal temperament. For example, Chinese scales are just intoned with generators like $\frac{3}{2}$, $\frac{9}{8}$, and $\frac{81}{64}$. I will make a document later detailing the usage of different traditional just intonation systems.

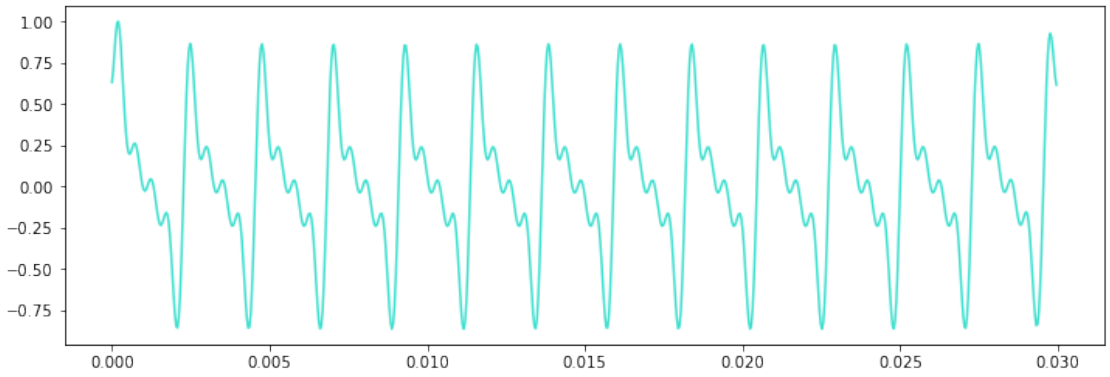
3 Using PyAudio and wave

Two libraries useful for converting arrays representing audio signals into actual audio are PyAudio, which allows us to stream audio within Python, and wave, which we can use to load and save WAV files.

3.1 Encoding a signal

Let's return to the signal from the end of part 1.

```
In [28]: plt.figure(figsize=[12,4])
         plt.plot(t, x, color='turquoise')
         plt.show()
```



Each value in this array must be converted to a hexadecimal number to be passed to the WAV file. The Python function chr() converts int arguments into Unicode characters corresponding to their hexadecimal form up to and including 255. This means that the array must first be normalized to (-1,1) (by dividing the array by its maximum value) and then converted to a value in [0,255]. Assuming x_{float} is a normalized float array (like above), the values in the integer array will then be

$$x_{int}(t) = 127x_{float}(t) + 128$$

```
In [38]: xint = (127*x+128).astype('int')
         plt.figure(figsize=[12,4])
         plt.plot(t, xint, color='turquoise')
         plt.show()
```