Sound Generator

July 2, 2019

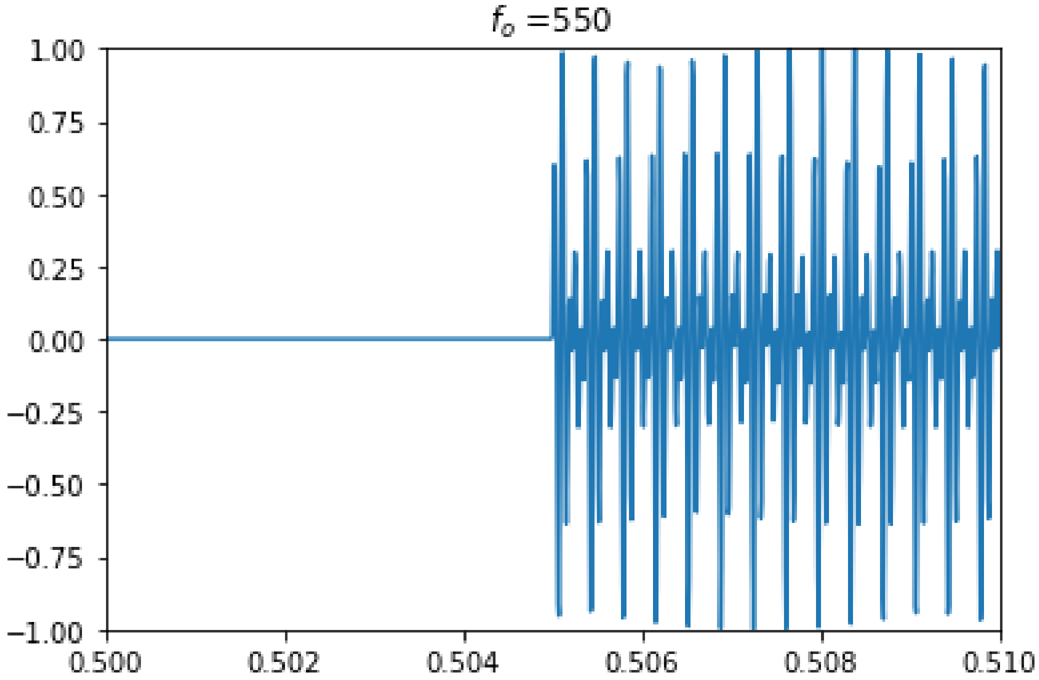
[2]:

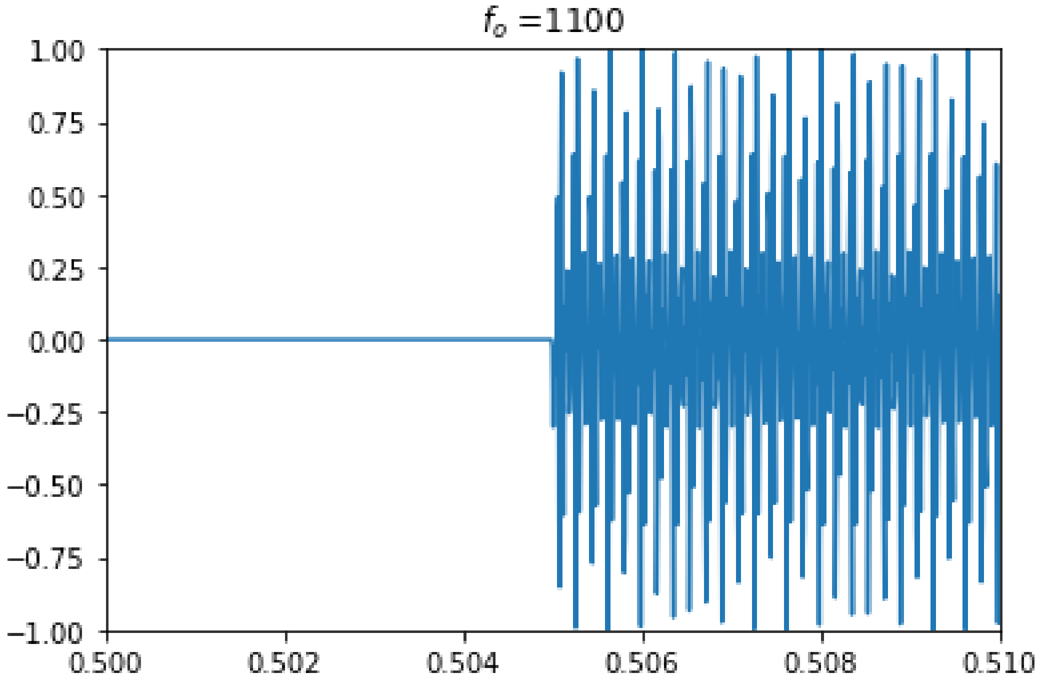
```
import numpy as np
import scipy
import IPython as ip
import matplotlib.pyplot as plt
import wave
from sympy.utilities.iterables import multiset_permutations
```

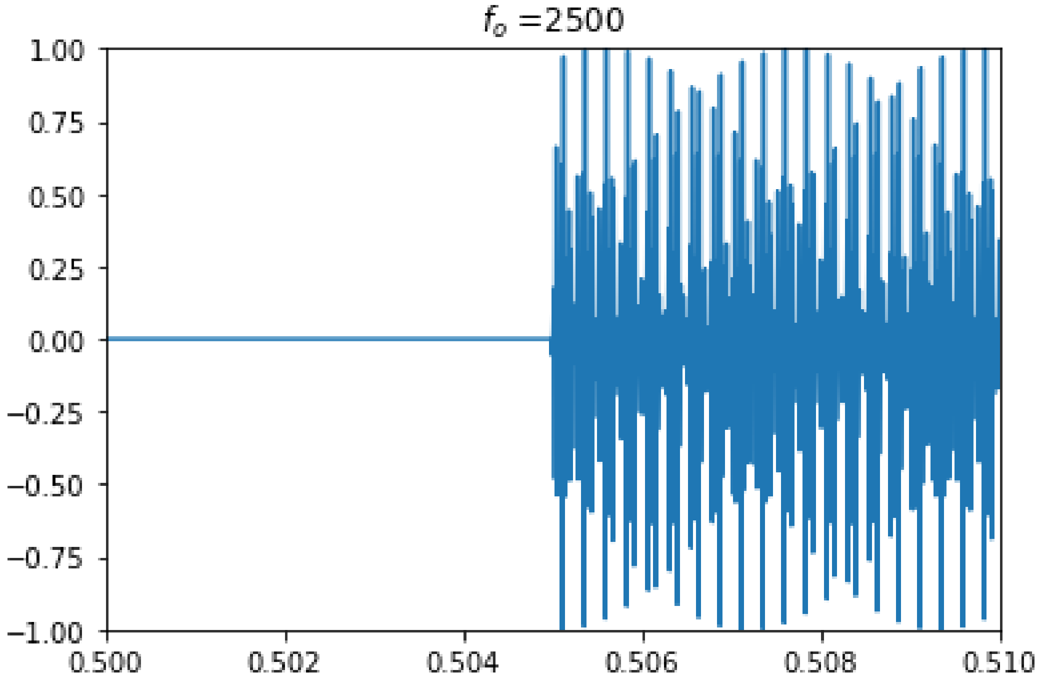
[3]:

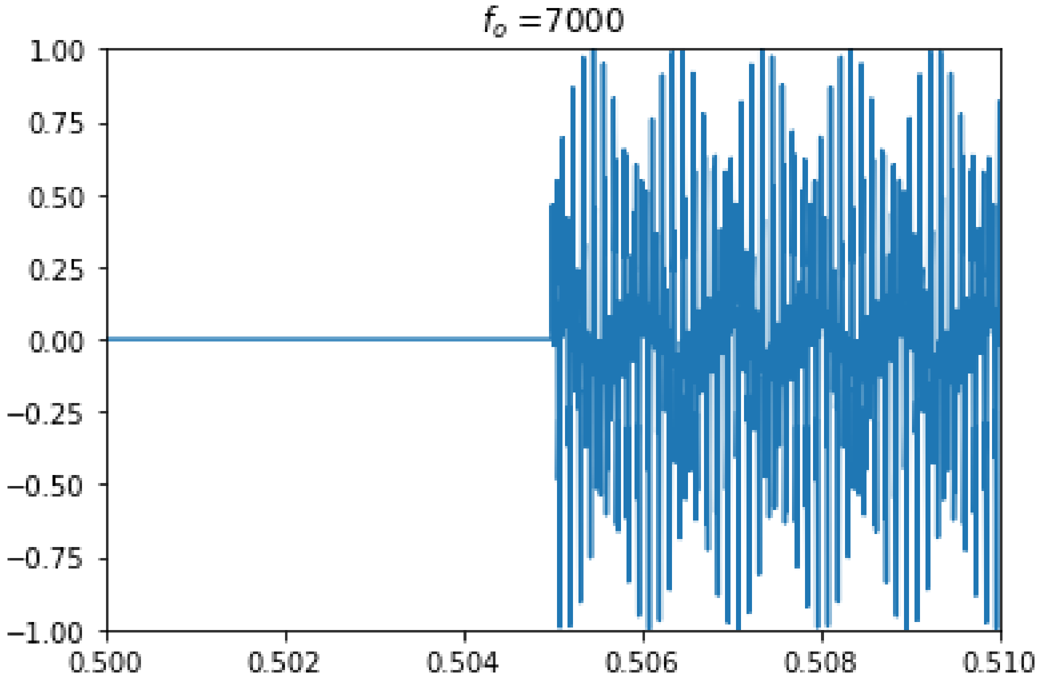
```
#Sample rate and duration of the sound
rate = 96000
duration = 0.5
t = np.linspace(0, duration, int(rate * duration))

def norm(v):
    return 2 * (v - np.mean(v)) / np.ptp(v)



class ToneBlock:

    #Takes in silence duration, tone duration, the fundamental freq, and number␣
 ↪of overtones
    def __init__(self, freq, n_freq, sil=0.4, dur=0.1,):
        self.sil = sil
        self.dur = dur
        self.freq = freq
        self.nfreq = n_freq

    #Generates a single tone with a silence after
    def generate_tone(self, freq):
        tone = np.sin(freq * 2.0 * np.pi * np.linspace(0,duration,int(rate *␣
 ↪self.dur)))
        ramplen = int(0.005 * rate)
        window = np.ones(int(self.dur * rate))
        hann = np.hanning(2 * rate)
        window[:ramplen] = hann[:ramplen]
        window[-ramplen:] = hann[-ramplen:]
        #print(window*tone)
        return np.concatenate([window * tone, np.zeros(int(rate*self.sil))])
```

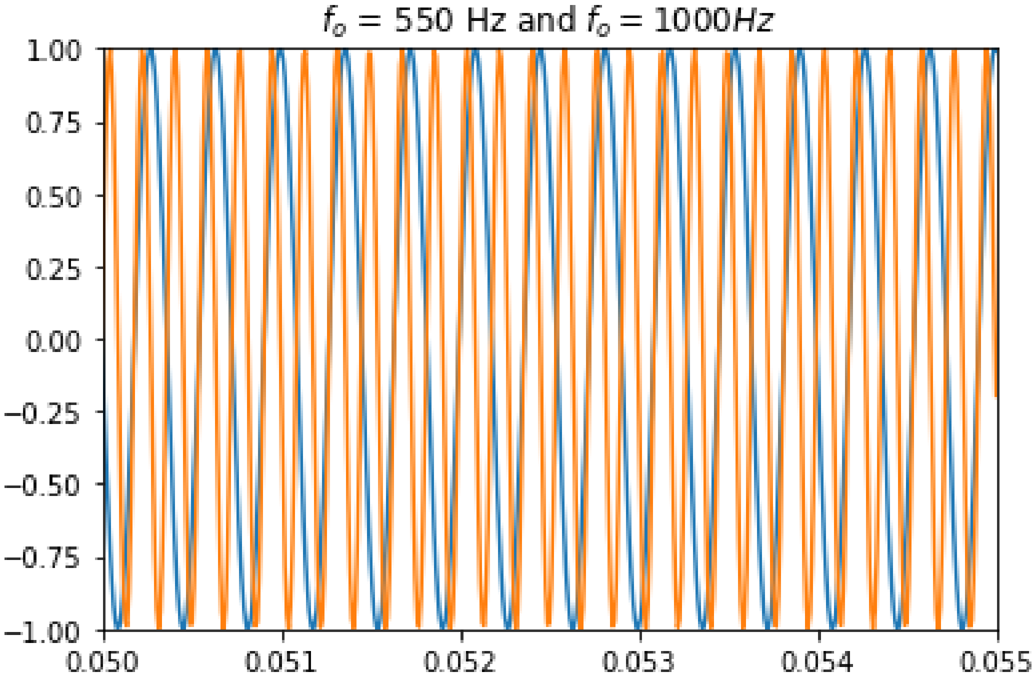
```python
    #Takes in boolean array of order of frequencies and returns a harmonic stack,
    #e.g. [0, 1, 1] returns stack of 3 tones with missing fundamental frequency.
    def generate_block(self, farray):
        block = np.zeros(len(farray))
        count = 0
        for i in farray:
            if i:
                block[count] = self.freq * (count + 1)
            count += 1
        #print(block)
        return block



    #Takes in boolean array and generates the tone using generate_block
    def generate_toneblock(self, boolarr):
        sum_series = np.zeros(len(boolarr), dtype = object)
        blok = self.generate_block(boolarr)
        #print(blok)
        count = 0
        for x in blok:
            sum_series[count] = self.generate_tone(x)
            count += 1
        return norm(np.sum(sum_series))

    #Creating random array of boolean arrays of desired length (currently configured to ignore missing midtones)
    def bool_gen(self):
        big_bool = []
        dim = self.nfreq
        array = np.zeros(dim)
        for i in range(dim):
            array[:i] = np.ones(i)
            for p in multiset_permutations(array):
                big_bool += [p]
        return big_bool

#Save the wav file with a 500 Hz left channel mark-track
def save_wav(filename, audio, stereo_on = True):
    nchannels = 1
    sampwidth = 2
    comptype = 'NONE'
    compname = 'not compressed'

    if stereo_on:
        nchannels = 2


    with wave.open(filename, 'w') as wav_file:
        wav_file.set_params((nchannels, sampwidth, fs, len(audio), comptype,␣
 ↪compname))

        for i in range(int(duration * rate)):

            wav_file.writeframes(struct.pack('<hh', np.sin(500 * 2.0 * np.pi *␣
 ↪np.linspace(0,duration,int(rate * 40))), audio))

        wav_file.writeframes('')
        wav_file.close()



#import wave, struct, math

#def save_wav(audio):
#    sampleRate = 44100.0 # hertz
#    duration = 1.0       # seconds

#    lFreq =  523.25  # C

 #   wavef = wave.open('sound.wav','w')
 #   wavef.setnchannels(2) # stereo
 #   wavef.setsampwidth(2)
  #  wavef.setframerate(sampleRate)

   # for i in range(int(duration * sampleRate)):
   #     l = int(32767.0*math.cos(lFreq*math.pi*float(i)/float(sampleRate)))
        #r = int(32767.0*math.cos(rFreq*math.pi*float(i)/float(sampleRate)))
  #  r = audio
  #  wavef.writeframesraw( struct.pack('<hh', l, r ) )

#wavef.writeframes('')
#wavef.close()




#Call bool_gen to create all possible stacks (without missing midtones).
#Then create a harmonic stack object for each fundamental frequency.
def main():
    j = 0
    tone0 = ToneBlock(550, 5)
    tone1 = ToneBlock(1100, 5)
    tone2 = ToneBlock(2500, 5)
    tone3 = ToneBlock(7000, 5)
    tone = [tone0, tone1, tone2, tone3]

    pure_tone = []
    boole =␣
↪[[1,1,1,1,1],[1,1,1,1,0],[1,1,1,0,0],[1,1,0,0,0],[1,0,0,0,0],[0,0,0,0,1],[0,0,0,1,1],[0,0,1
    #boole = tone0.bool_gen() #Calling for just tone0 since nfreq = 5 for all␣
↪tones
    np.random.shuffle(boole)

    toneblock0 = [tone0.generate_toneblock(i) for i in boole]
    toneblock1 = [tone1.generate_toneblock(i) for i in boole]
    toneblock2 = [tone2.generate_toneblock(i) for i in boole]
    toneblock3 = [tone3.generate_toneblock(i) for i in boole]
    toneblock = [toneblock0, toneblock1, toneblock2, toneblock3]
    #tonenorm = [norm(i) for i in toneblock]

    new_t = np.linspace(0, duration * len(toneblock0), int(rate * duration *␣
↪len(toneblock0)))
    tonecat = [np.concatenate(x) for x in toneblock]
    for cat in tonecat:

        plt.title('$f_{o}$ =' + str(tone[j].freq))
        plt.plot(new_t, cat)
        plt.axis([0.5, 0.51, -1, 1])
        plt.figure()
        j += 1

    plt.plot(new_t, tonecat[0])
    plt.plot(new_t, tonecat[1])
    plt.title('$f_{o}$ = 550 Hz and $f_{o} = 1000 Hz$')
    plt.axis([0.05, 0.055, -1, 1])
    #save_wav('wave_test.wav',tonecat)
#    return toneblock


main()
```


f_o =550
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.500
0.502
0.504
0.506
0.508
0.510


f_o =1100
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.500
0.502
0.504
0.506
0.508
0.510


f_o =2500
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.500
0.502
0.504
0.506
0.508
0.510


f_o =7000
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.500
0.502
0.504
0.506
0.508
0.510


f_o = 550 Hz and $f_o = 1000Hz$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.050
0.051
0.052
0.053
0.054
0.055

[ ]:

[ ]:

[ ]: