Code in the MASS framework

Renato Fabbri (USP),
Vilson Vieira da Silva Junior (Cod.ai),
Antônio Carlos Silvano Pessotti (Unimep),
Débora Cristina Corrêa,
Osvaldo N. Oliveira Jr. (USP)

September 20, 2017

Abstract

This document displays the Python code in the MASS framework. The code is accessible as Python scripts in the main repository [1], and this PDF is made available because it might facilitate browsing the implementations. Check the final consideration in this document for further directions.

1 Sections

Here is the code related to each section of the MASS article [2].

Python implementation of equations in Section 2

```python
import numpy as n
from scipy.io import wavfile as w

# auxiliary functions __n and __s.
# These only normalize the sonic vectors and
# write them as 16 bit, 44.1kHz WAV files.
def __n(sonic_array):
    """Normalize sonic_array to have values only between -1 and 1"""

    t = sonic_array
    if n.all(sonic_array==0):
        return sonic_array
    else:
        return ( (t-t.min()) / (t.max() -t.min()) )*2.-1.

def __s(sonic_array=n.random.uniform(size=100000), filename="asound.wav", f_s=44100):
    """A minimal approach to writing 16 bit WAVE files.
```

```
    One can also use, for example:
        import sounddevice as S
        S.play(array) # the array must have values between -1 and 1"""

    # to write the file using XX bits per sample
    # simply use s = n.intXX(__n(sonic_array)*(2**(XX-1)-1))
    s = n.int16(__n(sonic_array)*32767)
    w.write(filename, f_s, s)


############ Sec. 2.1 Duration
# relation between the number of samples and the sound duration
f_s = 44100  # sample rate
Delta = 3.7   # duration of Delta in seconds

Lambda = int(f_s*Delta)  # number of samples
# Eq. 1
T = n.zeros(Lambda)  # silence with ~Delta, in seconds

# write as a PCM file (WAV)
__s(T, 'silence.wav')

############ Sec. 2.2 Loudness
Lambda = 100  # 100 samples
T = n.random.random(Lambda)  # 100 random samples

# Eq. 2 Power of wave
pow1 = (T**2.).sum()/Lambda

T2 = n.random.normal(size=Lambda)
pow2 = (T2**2.).sum()/Lambda  # power of another wave

# Eq. 3 Volume difference, in decibels, given the powers
V_dB = 10.*n.log10(pow2/pow1)

# Eq. 4 double the amplitude => gains 6 dB
T2 = 2.*T
pow2 = (T2**2.).sum()/Lambda
V_dB = 10.*n.log10(pow2/pow1)
is_6db = abs(V_dB - 6) < .05  # is_6db is True

# Eq. 5 double the power => gains 3 dB
pow2 = 2.*pow1
V_dB = 10.*n.log10(pow2/pow1)
is_3dB = abs(V_dB - 3) < .05  # is_3dB is True

# Eq. 6 double the volume => gains 10 dB => amplitude * 3.16
V_dB = 10.
A = 10.**(V_dB/20.)
T2 = A*T  # A ~ 3.1622776601

# Eq. 7 Decibels to amplification conversion
A = 10.**(V_dB/20.)
```

```python

############ Sec. 2.3 Pitch
f_0 = 441
lambda_0 = f_s//f_0
cycle = n.arcsin(n.random.random(lambda_0))  # random samples
# Eq. 8 Sound with fundamental frequency f_0
Tf = n.array(list(cycle)*1000)  # 1000 cycles

# normalizing to interval [-1, 1]
__s(Tf,'f_0.wav')


############ Sec. 2.4 Timbre
L = 100000.  # sample number of sequences (Lambda)
ii = n.arange(L)
f = 220.5
lambda_f = f_s/f
# Eq. 9 Sinusoid
Sf = n.sin(2.*n.pi*f*ii/f_s)
# Eq. 10 Sawtooth
Df = (2./lambda_f)*(ii % lambda_f)-1
# Eq. 11 Triangular
Tf = 1.-n.abs(2.-(4./lambda_f)*(ii % lambda_f))
# Eq. 12 Square
Qf = ((ii % lambda_f) < (lambda_f/2))*2-1

Rf = w.read("22686__acclivity__oboe-a-440_periodo.wav")[1]
# Eq. 13 Sampled period
Tf = Rf[n.int64(ii) % len(Rf)]


############ Sec. 2.5 The spectrum of sampled sound
Lambda = 50
T = n.random.random(Lambda)*2.-1.
C_k = n.fft.fft(T)
A_k = n.real(C_k)
B_K = n.imag(C_k)
w_k = 2.*n.pi*n.arange(Lambda)/Lambda

# Eq .14 Spectrum recomposition in time
def t(i):
    return (1./Lambda)*n.sum(C_k*n.e**(1j*w_k*i))

# Eq. 15 Real recomposition
def tR(i):
    return (1./Lambda)*n.sum(n.abs(C_k)*n.cos(w_k*i-n.angle(C_k)))

# Eq. 16 Number of paired spectrum coefficients
tau = int( (Lambda - Lambda % 2)/2 + Lambda % 2-1 )

# Eq. 17 Equivalent coefficients
F_k = C_k[1:tau+1]
F2_k = C_k[Lambda-tau:Lambda][::-1]
```

```python

# Eq. 18 Equivalent modules of coefficients
ab = n.abs(F_k)
ab2 = n.abs(F2_k)
MIN = n.abs(ab-ab2).sum()  # MIN ~ 0.0

# Eq, 19 Equivalent phases of coefficients
an = n.angle(F_k)
an2 = n.angle(F2_k)
MIN = n.abs(an+an2).sum()  # MIN ~ 0.0

# Eq. 20 Components combination in each sample
w_k = 2*n.pi*n.arange(Lambda)/Lambda

def t_(i):
    return (1./Lambda)*(A_k[0]+2.*n.sum(n.abs(C_k[1:tau+1]) *
                        n.cos(w_k*i-n.angle(C_k)) + A_k[Lambda/2] *
                        (1-Lambda % 2)))


############ Sec. 2.6 The basic note
f = 220.5  # Herz
Delta = 2.5  # seconds
Lambda = int(2.5*f_s)
ii = n.arange(Lambda)

# Eq. 21 Basic note (preliminary)
ti_ = n.random.random(int(f_s/f))  # arbitrary sequence of samples
TfD = ti_[ii % len(ti_)]

# Eq. 22 Choose any waveform
Lf = [Sf, Qf, Tf, Df, Rf][1]  # We already calculated these sequences

# Eq. 23 Basic note
TfD = Lf[ii % len(Lf)]


############ Sec. 2.7 Spatialization: localization and reverberation
zeta = 0.215  # meters
# considering any (x,y) localization
x = 1.5  # meters
y = 1.  # meters
# Eq. 24 Distances from each ear
d = n.sqrt((x-zeta/2)**2+y**2)
d2 = n.sqrt((x+zeta/2)**2+y**2)
# Eq. 25 Interaural Time Difference
ITD = (d2-d)/343.2  # segundos
# Eq. 26 Interaural Intensity Difference
IID = 20*n.log10(d/d2)  # dBs

# Eq. 27 DTI and DII application in a sample sequence (T)
Lambda_ITD = int(ITD*f_s)
IID_a = d/d2
T = 1-n.abs(2-(4./lambda_f)*(ii % lambda_f))  # triangular
```

```python
T2 = n.hstack((n.zeros(Lambda_ITD), IID_a*T))
T = n.hstack((T, n.zeros(Lambda_ITD)))

som = n.vstack((T2, T)).T
w.write('stereo.wav', f_s, som)
# mirrored
som = n.vstack((T, T2)).T
w.write('stereo2.wav', f_s, som)

# Eq. 28 Object angle
theta = n.arctan(y/x)

# Reverberation is implemented in 3.py
# because it makes use of knowledge of the next section


############ Sec. 2.8 Musical uses
Delta = 3.  # 3 seconds
Lambda = int(Delta*f_s)
f1 = 200.  # Hz
foo = n.linspace(0., Delta*f1*2.*n.pi, Lambda, endpoint=False)
T1 = n.sin(foo)  # sinusoid of Delta seconds and freq  =  f1

f2 = 245.  # Hz
lambda_f2 = int(f_s/f2)
T2 = (n.arange(Lambda) % lambda_f < (lambda_f2/2))*2-1  # square

f3 = 252.  # Hz
lambda_f3 = f_s/f3
T3 = n.arange(Lambda) % lambda_f3  # sawtooth
T3 = (T3/T3.max())*2-1

# Eq. 29 mixing
T = T1+T2+T3
# writing file
__s(T, 'mixed.wav')

# Eq. 30 concatenation
T = n.hstack((T1, T2, T3))
# writing file
__s(T, 'concatenated.wav')
```

Python implementation of equations in Section 3

```python
import numpy as n
from scipy.io import wavfile as w

# auxiliary functions __n and __s.
# These only normalize the sonic vectors and
# write them as 16 bit, 44.1kHz WAV files.
def __n(sonic_array):
    """Normalize sonic_array to have values only between -1 and 1"""

    t = sonic_array
    if n.all(sonic_array==0):
        return sonic_array
    else:
        return ( (t-t.min()) / (t.max() -t.min()) )*2.-1.

def __s(sonic_array=n.random.uniform(size=100000), filename="asound.wav", f_s=44100):
    """A minimal approach to writing 16 bit WAVE files.

    One can also use, for example:
        import sounddevice as S
        S.play(array) # the array must have values between -1 and 1"""

    # to write the file using XX bits per sample
    # simply use s = n.intXX(__n(sonic_array)*(2**(XX-1)-1))
    s = n.int16(__n(sonic_array)*32767)
    w.write(filename, f_s, s)


f_s = 44100  # Hz, sample rate

############ Sec. 3.1 Lookup table (LUT)
# at least 1024 samples in the table
Lambda_tilde = Lt = 1024

# Sinusoid
foo = n.linspace(0, 2*n.pi, Lt, endpoint=False)
S = n.sin(foo)  # a sinusoidal period with T samples

# Square:
Q = n.hstack((n.ones(Lt/2)*-1, n.ones(Lt/2)))

# Triangular:
foo = n.linspace(-1, 1, Lt/2, endpoint=False)
Tr = n.hstack((foo, foo*-1))

# Sawtooth:
D = n.linspace(-1, 1, Lt)

# real sound, import period and
# use the number of samples in the period
Rf = w.read("22686__acclivity__oboe-a-440_periodo.wav")[1]

f = 110.  # Hz
Delta = 3.4  # seconds
Lambda = int(Delta*f_s)
```

```python

# Samples:
ii = n.arange(Lambda)

# Eq. 31 LUT
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
# It is possible to use S, Q, D or any other period of a real sound
# with a sufficient length
L = Tr
TfD = L[Gamma % Lt]


############ Sec. 3.2 Incremental variations of frequency and intensity
# == FREQUENCY VARIATIONS ==
f_0 = 100.  # initial freq in Hz
f_f = 300.  # final freq in Hz
Delta = 2.4  # duration

Lambda = int(f_s*Delta)
ii = n.arange(Lambda)
# Eq. 32 linear variation
f_i = f_0+(f_f-f_0)*ii/(float(Lambda)-1)
# Eq. 33 coefficients for LUT
D_gamma = f_i*Lt/f_s
Gamma = n.cumsum(D_gamma)
Gamma = n.array(Gamma, dtype=n.int)
# Eq. 34 resulting sound
Tf0ff = L[Gamma % Lt]

# Eq. 35 exponential variation
f_i = f_0*(f_f/f_0)**(ii/(float(Lambda)-1))
# Eq. 36 coefficients for the LUT
D_gamma = f_i*Lt/f_s
Gamma = n.cumsum(D_gamma)
Gamma = n.array(Gamma, dtype=n.int)
# Eq. 37 resulting sound
Tf0ff = L[Gamma % Lt]


# == INTENSITY VARIATIONS ==
# First, make/have an arbitrary sound to
# apply the variations in amplitude
f = 220.  # Hz
Delta = 3.9  # seconds
Lambda = int(Delta*f_s)

# Sample indexes:
ii = n.arange(Lambda)

# (as in Eq. 31)
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
L = Tr
T = TfD = L[Gamma % Lt]

a_0 = 1.  # starting fraction of the amplitude
```

```python
a_f = 12.  # ending fraction of the amplitude
alpha = 1.  # index of transition smoothing

# Eq. 38 exponential transition of amplitude
A = a_0*(a_f/a_0)**((ii/float(Lambda))**alpha)
# Eq. 39 applying envelope A to the sound
T2 = A*T

# Eq. 40 linear transition of amplitude
A = a_0+(a_f-a_0)*(ii/float(Lambda))

# Eq. 41 exponential transition of V_dB decibels
V_dB = 31.
T2 = T*((10*(V_dB/20.))**((ii/float(Lambda))**alpha))


############ Sec 3.3 Application of digital filters
# See src/aux/delays.py for generating Fig. 17
# See src/aux/filters/iir.py for generating Fig. 18

# synthetic impulse response (for a "reverb", a better reverb is bellow in: Reverberation)
H = (n.random.random(10)*2-1)*n.e**(-n.arange(10))

# Eq. 42 Convolution (application of a FIR filter)
T2 = n.convolve(T, H)  # T from above

# Eq. 43  difference equation
A = n.random.random(2)  # arbitrary coefficients
B = n.random.random(3)  # arbitrary coefficients

def applyIIR(signal, A, B):
    signal_ = []
    for i, sample in enumerate(signal):
        samples_A = signal[i::-1][:len(A)]
        A_coeffs = A[:i+1]
        A_contrib = (samples_A*A_coeffs).sum()

        samples_B = signal_[-1:-1-i:-1][:len(B)-1]
        B_coeffs = B[1:i+1]
        B_contrib = (samples_B*B_coeffs).sum()
        t_i = (A_contrib + B_contrib)/B[0]
        signal_.append(t_i)
    return signal_

fc = .1
# Eq. 44 low-pass IIR filter with a single pole
x = n.e**(-2*n.pi*fc)  # fc => cutoff frequency where the resulting signal has -3dB
# coefficients
a0 = 1-x
b1 = x
# applying the filter
T2 = [T[0]]
for t_i in T[1:]:
    T2.append(t_i*a_0+T2[-1]*b1)
```

```
# Eq. 45 high-pass filter with a single pole
x = n.e**(-2*n.pi*fc)  # fc => cutoff frequency where the resulting signal has -3dB
# coefficients
a0 = (1+x)/2
a1 = -(1+x)/2
b1 = x

# applying the filter
T2 = [a0*T[0]]
last = T[0]
for t_i in T[1:]:
    T2 += [a0*t_i + a1*last + b1*T2[-1]]
    last = n.copy(t_i)


fc = .1  # now fc is the center frequency
bw = .05
# Eq. 46 Auxiliary variables for the notch filters
r = 1-3*bw
k = (1-2*r*n.cos(2*n.pi*fc)+r**2)/(2-2*n.cos(2*n.pi*fc))

# Eq. 47 band-pass filter coefficients
a0 = 1-k
a1 = -2*(k-r)*n.cos(2*n.pi*fc)
a2 = r**2 - k
b1 = 2*r*n.cos(2*n.pi*fc)
b2 = -r**2

# applying the filter
T2 = [a0*T[0]]
T2 += [a0*T[1]+a1*T[0]+b1*T2[-1]]
last1 = T[1]
last2 = T[0]
for t_i in T[2:]:
    T2 += [a0*t_i+a1*last1+a2*last2+b1*T2[-1]+b2*T2[-2]]
    last2 = n.copy(last1)
    last1 = n.copy(t_i)

# Eq. 48 band-reject filter coefficients
a0 = k
a1 = -2*k*n.cos(2*n.pi*fc)
a2 = k
b1 = 2*r*n.cos(2*n.pi*fc)
b2 = -r**2

# applying the filter
T2 = [a0*T[0]]
T2 += [a0*T[1]+a1*T[0]+b1*T2[-1]]
last1 = T[1]
last2 = T[0]
for t_i in T[2:]:
    T2 += [a0*t_i+a1*last1+a2*last2+b1*T2[-1]+b2*T2[-2]]
    last2 = n.copy(last1)
    last1 = n.copy(t_i)
```

```
############ Sec. 3.4 Noise
# See src/filters/ruidos.py for rendering Figure 19
Lambda = 100000  # Use an even Lambda for compliance with the following snippets
# Separation between frequencies of neighbor spectral coefficients:
df = f_s/float(Lambda)

# Eq. 49 White noise
# uniform moduli of spectrum and random phase
coefs = n.exp(1j*n.random.uniform(0, 2*n.pi, Lambda))

f0 = 15.  # minimum frequency which we want in the sound
i0 = n.floor(f0/df)  # first coefficient to be considered
coefs[:i0] = n.zeros(i0)

# coefficients have real part even and imaginary part odd
coefs[Lambda/2+1:] = n.real(coefs[1:Lambda/2])[::-1] - 1j * \
    n.imag(coefs[1:Lambda/2])[::-1]
coefs[0] = 0.  # no bias (no offset)
coefs[Lambda/2] = 1.  # max freq is only real (as explained in Sec. 2.5)

# Achievement of the temporal samples of the noise
ruido = n.fft.ifft(coefs)
r = n.real(ruido)
__s(r, 'white.wav')

# auxiliary variables to all the following noises
fi = n.arange(coefs.shape[0])*df # frequencies related to the coefficients
f0 = fi[i0] # first frequency to be considered

# Eq. 50 Pink noise
# the volume decreases by 3dB at each octave
factor = 10.**(-3/20.)
alphai = factor**(n.log2(fi[i0:]/f0))

c = n.copy(coefs)
c[i0:] = coefs[i0:]*alphai
# real is even, imaginary is odd
c[Lambda/2+1:] = n.real(c[1:Lambda/2])[::-1] - 1j * \
    n.imag(c[1:Lambda/2])[::-1]

ruido = n.fft.ifft(c)
r = n.real(ruido)
__s(r, 'pink.wav')


# Eq. 51 Brown(ian) noise
# the volume decreases by 6dB at each octave
fator = 10.**(-6/20.)
alphai = fator**(n.log2(fi[i0:]/f0))
c = n.copy(coefs)
c[i0:] = c[i0:]*alphai

# real is even, imaginary is odd
c[Lambda/2+1:] = n.real(c[1:Lambda/2])[::-1] - 1j * \
    n.imag(c[1:Lambda/2])[::-1]

```

```
ruido = n.fft.ifft(c)
r = n.real(ruido)
__s(r, 'brown.wav')

ruido_marrom = n.copy(r) # it will be used for reverberation


# Eq. 52 Blue noise
# the volume increases by 3dB at each octave
fator = 10.**(3/20.)
alphai = fator**(n.log2(fi[i0:]/f0))
c = n.copy(coefs)
c[i0:] = c[i0:]*alphai

# real is even, imaginary is odd
c[Lambda/2+1:] = n.real(c[1:Lambda/2])[::-1] - 1j * \
    n.imag(c[1:Lambda/2])[::-1]

ruido = n.fft.ifft(c)
r = n.real(ruido)
__s(r, 'blue.wav')


# Eq. 53 Violet noise
# the volume increses by 6dB at each octave
fator = 10.**(6/20.)
alphai = fator**(n.log2(fi[i0:]/f0))
c = n.copy(coefs)
c[i0:] = c[i0:]*alphai

# real is even, imaginary is odd
c[Lambda/2+1:] = n.real(c[1:Lambda/2])[::-1] - 1j * \
    n.imag(c[1:Lambda/2])[::-1]

ruido = n.fft.ifft(c)
r = n.real(ruido)
__s(r, 'violet.wav')

# Eq.54 Black noise
# the volume decreases more than 6dB at each octave
fator = 10.**(-12/20.)
alphai = fator**(n.log2(fi[i0:]/f0))
c = n.copy(coefs)
c[i0:] = c[i0:]*alphai

# real is even, imaginary is odd
c[Lambda/2+1:] = n.real(c[1:Lambda/2])[::-1] - 1j * \
    n.imag(c[1:Lambda/2])[::-1]

ruido = n.fft.ifft(c)
r = n.real(ruido)
__s(r, 'black.wav')


############ Sec. 3.5 Tremolo e vibrato, AM e FM
# See src/aux/vibrato.py and src/aux/tremolo.py for rendering Figures 20 and 21
```

```
f = 220.
Lv = 2048  # size of the table for the vibrato
fv = 1.5  # vibrato frequency
nu = 1.6  # maximum semitone deviation (vibrato depth)
Delta = 5.2  # sound duration
Lambda = int(Delta*f_s)

# Vibrato table
x = n.linspace(0, 2*n.pi, Lv, endpoint=False)
tabv = n.sin(x)  # sinusoidal vibrato

ii = n.arange(Lambda)  # índices
# Eq. 55 indexes of the LUT for the vibrato
Gammav = n.array(ii*fv*float(Lv)/f_s, n.int)
# Eq. 56 samples of the oscillatory pattern of the vibrato
Tv = tabv[Gammav % Lv]
# Eq. 57 frequency at each sample
F = f*(2.**(Tv*nu/12.))
# Eq. 58 indexes of the LUT for the sound
D_gamma = F*(Lt/float(f_s))  # displacement in the table for each sample
Gamma = n.cumsum(D_gamma)  # total displacement at each sample
Gamma = n.array(Gamma, dtype=n.int)  # final indexes
# Eq. 59 the samples of the sound
T = Tr[Gamma % Lt]  # Lookup

__s(T, "vibrato.wav")


Tt = n.copy(Tv)  # same oscillatory pattern from the vibrato
# Eq. 60 Envelope of the tremolo
V_dB = 12.  # decibels variation involved in the tremolo (tremolo depth)
A = 10**((V_dB/20)*Tt)  # amplitude multiplicative factors for each sample
# Eq. 61 Application of the amplitude envelope to the original sample sequence T
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
T = Tr[Gamma % Lt]
T = T*A
__s(T, "tremolo.wav")


# the following equations are not used to synthesize sounds,
# but only to express the spectrum resulting from FM and AM synthesis
# Eq. 62 - FM spectrum, implemented in Eqs. 65-69
# Eq. 63 - Bessel function
# Eq. 64 - AM spectrum, implemented in Eqs. 70,71

fv = 60.  # typically, fv > 20Hz (otherwise one might want to use the equations above for the vibra
# Eq. 65 indexes of the LUT for the FM modulator
Gammav = n.array( ii*fv*float(Lv)/f_s, n.int )
# Eq. 66 oscillatory pattern (sample-by-sample) of the modulator
Tfm = tabv[Gammav % Lv]
f = 330.
mu = 40.
# Eq. 67 frequency at each sample
```

```
F = f+Tfm*mu
# Eq. 68 indexes of the LUT
D_gamma = f_i*(Lt/float(f_s))  # displacement in the lookup between each sample
Gamma = n.cumsum(D_gamma)  # total displacement in the lookup at each sample
Gamma = n.array(Gamma, dtype=n.int)  # indexes
# Eq. 69 FM
T = S[Gamma % Lt]  # final samples

# writing the sound file
__s(T, "fm.wav")


# AM
Tam = n.copy(Tfm)
V_dB = 12.  # am depth in decibels
alpha = 10**(V_dB/20.)  # AM depth in amplitude
# 2.71 AM envelope
A = 1+alpha*Tam
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
# 2.70 AM
T = Tr[Gamma % Lt]*A
__s(T, "am.wav")


############ Sec. 3.6 Usos musicais
# Eq. 72 Relations between characteristics
# See the musical piece Tremolos, vibratos and the frequency
# in src/pieces3/bonds.py TremolosVibratosEaFrequencia.py

# Doppler effect
v_r= 10 # receptor moves in the direction of the source with velocity v_r m/s
v_s=-80. # source moves in the direction of receptor with velocity v_s m/s
v_som=343.2
f_0=1000 # frequency of the source

# Eq. 73 Frequency resulting from the Doppler effect
f=((v_som + v_r) / (v_som + v_s)) * f_0
# after crossing of source and receptor:
f_=((v_som - v_r) / (v_som - v_s)) * f_0

# initial distances:
x_0=0 # source at front of x_0
y_0=200 # height of y_0 metros

Delta=5. # duration in seconds
Lambda=Delta*f_s # number of samples
# posições ao longo do tempo, X_i=n.zeros(Lambda)
Y=y_0 - ((v_r-v_s)*Delta) * n.linspace(0,1,Lambda)

# At each sample, calculating ITD and IID as explained in the last section
# In this case, ITD e IID are == 0 because the source is centered
# Eq. 74 Amplitude resulting from the Doppler effect
# Assume z_0 meters above receptor:
z_0=2.
D=( z_0**2+Y**2  )**0.5 # distance at each PCM sample
```

```python
# Amplitude of sound related to the distance:
A_=z_0/D
# Amplitude change factor resulting from the Doppler effect:
A_DP=( (v_r-v_s)/343.2+1 )**0.5
A_DP_=( (-v_r+v_s)/343.2+1 )**0.5
A_DP=(Y>0)*A_DP+(Y<0)*A_DP_
A=A_ * A_DP

# Upon crossing, the velocities change sign:
# Eq. 75 Frequency progression
coseno=(Y)/((Y**2+z_0**2)**0.5)
F=( ( 343.2+v_r*coseno ) / ( 343.2+v_s*coseno ) )*f_0
# coefficients of the LUT
D_gamma = F*Lt/f_s
Gamma = n.cumsum(D_gamma)
Gamma = n.array(Gamma, dtype=n.int)

L = Tr  # Triangular wave
# Resulting sound:
Tdoppler = L[Gamma % Lt]
Tdoppler*=A

# normalizing and writing sound
__s(Tdoppler, 'doppler.wav')


###### Reverberation
# First reverberation period:
Delta1 = 0.15 # typically E [0.1,0.2]
Lambda1= int(Delta1*f_s)
Delta = 1.9 # total duration of reverberation
Lambda=int(Delta*f_s)

# Sound reincidence probability probability in the first period:
ii=n.arange(Lambda)
P = (ii[:Lambda1]/float(Lambda1))**2.
# incidences:
R1_=n.random.random(Lambda1)<P
A=10.**((-50./20)*(ii/Lambda))
# Eq. 76 First period of reverberation:
R1=R1_*A[:Lambda1]*ruido_marrom[:Lambda1] # first incidences

# Brown noise with exponential decay (of amplitude) for the second period:
# Eq. 77 Second period of reverberation:
Rm=ruido_marrom[Lambda1:Lambda]
R2=Rm*A[Lambda1:Lambda]
# Eq. 78 Impulse response of the reverberation
R=n.hstack((R1,R2))
R[0]=1.

# Making an arbitrary sound to apply the reverberation:
f_0 = 100.  # starting freq (Hz)
f_f = 700.  # final freq (Hz)
Delta = 2.4  # duration
Lambda = int(f_s*Delta)
```

```python
ii = n.arange(Lambda)

# (using Eq. 35 for exponential variation)
F = f_0*(f_f/f_0)**(ii/(float(Lambda)-1))
# (using Eq. 36 for the LUT indexes)
D_gamma = F*Lt/f_s
Gamma = n.cumsum(D_gamma)
Gamma = n.array(Gamma, dtype=n.int)
# (using Eq. 2.37 for making the sound)
Tf0ff = L[Gamma % Lt]

# Applying the reverberation
T_=Tf0ff
T=n.convolve(T_,R)
__s(T, "reverb.wav")


# Eq. 79 ADSR - linear variation
Delta = 5.  # total duration in seconds
Delta_A = 0.1  # Attack
Delta_D = .3  # Decay
Delta_R = .2  # Release
a_S = .1  # Sustain level

Lambda = int(f_s*Delta)
Lambda_A = int(f_s*Delta_A)
Lambda_D = int(f_s*Delta_D)
Lambda_R = int(f_s*Delta_R)

# Achievement of the ADRS envelope: A_
ii = n.arange(Lambda_A, dtype=n.float)
A = ii/(Lambda_A-1)
A_ = A
ii = n.arange(Lambda_A, Lambda_D+Lambda_A, dtype=n.float)
D = 1-(1-a_S)*((ii-Lambda_A)/(Lambda_D-1))
A_ = n.hstack((A_, D))
S = a_S*n.ones(Lambda-Lambda_R-(Lambda_A+Lambda_D), dtype=n.float)
A_ = n.hstack((A_, S))
ii = n.arange(Lambda-Lambda_R, Lambda, dtype=n.float)
R = a_S-a_S*((ii-(Lambda-Lambda_R))/(Lambda_R-1))
A_ = n.hstack((A_, R))

# Eq. 80 Achievement of a sound with the ADSR envelope
ii = n.arange(Lambda, dtype=n.float)
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
T = Tr[Gamma % Lt]*(A_)

__s(T, "adsr.wav")


# Eq. 79 ADSR - exponential variation
xi = 1e-2  # -180dB for starting fade in and ending in the fade out
De = 2*100.  # total duration
DA = 2*20.  # attack duration
DD = 2*20.  # decay duration
DR = 2*20.  # release duration
```

```
SS = .4  # fraction of amplitude in which sustain occurs

Lambda = int(f_s*De)
Lambda_A = int(f_s*DA)
Lambda_D = int(f_s*DD)
Lambda_R = int(f_s*DR)

A = xi*(1./xi)**(n.arange(Lambda_A)/(Lambda_A-1))  # attack samples
A = n.copy(A)
D = a_S**((n.arange(Lambda_A, Lambda_A+Lambda_D)-Lambda_A)/(Lambda_D-1)) # decay samples
A = n.hstack((A, D))
S = a_S*n.ones(Lambda-Lambda_R-(Lambda_A+Lambda_D))  # sustain samples
A = n.hstack((A, S))
R = (SS)*(xi/SS)**((n.arange(Lambda-Lambda_R, Lambda)+Lambda_R-Lambda)/(Lambda_R-1)) # release
A = n.hstack((A,  R))

# Eq. 80 Achievement of sound with ADSR envelope
ii = n.arange(Lambda, dtype=n.float)
Gamma = n.array(ii*f*Lt/f_s, dtype=n.int)
T = Tr[Gamma % Lt]*(A)

__s(T, "adsr_exp.wav")
```

Python implementation of equations in Section 4

```python
import numpy as n

# auxiliary functions __n and __s.
# These only normalize the sonic vectors and
# write them as 16 bit, 44.1kHz WAV files.
def __n(sonic_array):
    """Normalize sonic_array to have values only between -1 and 1"""

    t = sonic_array
    if n.all(sonic_array==0):
        return sonic_array
    else:
        return ( (t-t.min()) / (t.max() -t.min()) )*2.-1.

def __s(sonic_array=n.random.uniform(size=100000), filename="asound.wav", f_s=44100):
    """A minimal approach to writing 16 bit WAVE files.

    One can also use, for example:
        import sounddevice as S
        S.play(array) # the array must have values between -1 and 1"""

    # to write the file using XX bits per sample
    # simply use s = n.intXX(__n(sonic_array)*(2**(XX-1)-1))
    s = n.int16(__n(sonic_array)*32767)
    w.write(filename, f_s, s)

f_s = 44100.  # Hz, sample rate
Lambda_tilde = Lt = 1024.
foo = n.linspace(0, 2*n.pi, Lt, endpoint=False)
S_i = n.sin(foo)  # a sinusoid period of T samples

H = n.hstack

# using the content from the previous sections,
# this is a very simple synthesizer of notes
def v(f=200, d=1., tab=S_i, fv=2., nu=2., tabv=S_i):
    Lambda = n.floor(f_s*d) ii = n.arange(Lambda)
    Lv = float(len(T))

    Gammav_i = n.floor(ii*fv*Lv/f_s)  # indexes for LUT
    Gammav_i = n.array(Gammav_i, n.int)
    # variation pattern of vibrato for each sample
    Tv_i = tabv[Gammav_i % int(Lv)]

    # frequency in Hz for each sample
    F_i = f*(2.**(Tv_i*nu/12.))
    # movement inside table for each sample
    D_gamma_i = F_i*(Lt/float(f_s))
    Gamma_i = n.cumsum(D_gamma_i)  # movement in the total table
    Gamma_i = n.floor(Gamma_i)  # the indexes
    Gamma_i = n.array(Gamma_i, dtype=n.int)  # the indexes
    return tab[Gamma_i % int(Lt)]  # looking for indexes in table


############ Sec. 4.1 Tuning, intervals, scales and chords
```

```
just_ratios = [1, 9/8, 5/4, 4/3, 3/2, 5/3, 15/3, 2]
pythagorean_ratios = [1, 9/8, 81/64, 4/3, 3/2, 27/16, 243/128, 2]
equal_temperament_ratios = [2**(i/12) for i in range(12)]

f = 220  # an arbitrary frequency
just_scale = [i*f for i in just_intonations]
pythagorean_scale = [i*f for i in pythagorean_ratios]
equal_temperament_scale = [i*f for i in equal_temperament_ratios]

js = H([v(i) for i in just_scale])
__s(js, "just_scale.wav")
ps = H([v(i) for i in pythagorean_scale])
__s(js, "pythagorean_scale.wav")
es = H([v(i) for i in equal_temperament_scale])
__s(js, "equal_temperament_scale.wav")

# Microtonality
# quarter tones
epslon = 2**(1/12.)
s1 = [0., 1.25, 1.75, 2., 2.25, 4., 5., 5.25]
factors = [epslon**i for i in s1]
scale = H([v(f*i) for i in factors])
__s(scale, "quarter_tones1.wav")

epslon = 2**(1/24.)
factors = [epslon**i for i in range(24)]
scale = H([v(f*i) for i in factors])
__s(scale, "quarter_tones2.wav")

# Octave sevenths
epslon_ = 2**(1/7.)
s2 = [0., 1., 2., 3., 4., 5., 6., 7.]
factors = [epslon_**i for i in s2]
scale = H([v(f*i) for i in factors])
__s(scale, "octave_sevenths.wav")

# Eq. 81 relating note grids
# expressing octave sevenths in the quarter tone grid:
s2_ = [i*24/7 for i in s2]

# Table 1: Intervals
# using epsilon = 2**(1/12)
I1j = 0.
I2m = 1.
I2M = 2.
I3m = 3.
I3M = 4.
I4J = 5.
ITR = 6.
I5J = 7.
I6m = 8.
I6M = 9.
I7m = 10.
I7M = 11.
I8J = 12.
I_i = n.arange(13.)
```

```python

perfect_consonances = [0, 7, 12]
imperfect_consonances = [3, 4, 8, 9]
weak_dissonances = [2, 10]
strong_dissonances = [1, 11]
special_cases = [5, 6]

# the interval sums nine for inversion by traditional nomenclature
# fifth is inverted into a fourth (5+4 = 9)
# but always sums 12
# at inversions of semitones
# fifth (7) is inverted into a fourth (5) (7+5 = 12)
def inv(I):
    """Returns inversed interval of I: 0< =  I < = 12"""
    return 12-I


# harmonic interval
def intervaloHarmonico(f, I):
    return (v(f)+v(f*2.**(I/12.)))*0.5


# melodic interval
def intervaloMelodico(f, I):
    return n.hstack((v(f), v(f*2.**(I/12.))))

# Eq. 82 Symmetric scales
Ec = [0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11.]
Ewt = [0., 2., 4., 6., 8., 10.]
Etm = [0., 3., 6., 9.]
EtM = [0., 4., 8.]
Ett = [0., 6.]

# Eq. 83 Diatonic scales
Em = [0., 2., 3., 5., 7., 8., 10.]
Emlo = [1., 3., 5., 6., 8., 10.]
EM = [0., 2., 4., 5., 7., 9., 11.]
Emd = [0., 2., 3., 5., 7., 9., 10.]
Emf = [0., 1., 3., 5., 7., 8., 10.]
Eml = [0., 2., 4., 6., 7., 9., 11.]
Emmi = [0., 2., 4., 5., 7., 8., 10.]

# Eq. 84 Diatonic pattern
E_ = n.roll(n.array([2.,2.,1.,2.,2.,2.,1.]), n.random.randint(7.))
E = n.cumsum(E_)-E_[0.]


# Eq. 85 Harmonic and melodic minor scales
Em = [0., 2., 3., 5., 7., 8., 10.]
Emh = [0., 2., 3., 5., 7., 8., 11.]
Emm = [0.,2.,3.,5.,7.,9.,11.,12.,10.,8.,7.,5.,3.,2.,0.]

# Eq. 86 Harmonic series
H = [ 0, 12, 19+0.02, 24, 28-0.14, 31+0.2, 34-0.31,
      36, 38+0.04, 40-0.14, 42-0.49, 43+0.02,
      44+0.41, 46-0.31, 47-0.12,
```

```python
      48, 49+0.05, 50+0.04, 51-0.02, 52-0.14 ]

# Eq. 86 Triads
AM = [0., 4., 7.]
Am = [0., 3., 7.]
Ad = [0., 3., 6.]
Aa = [0., 4., 8.]

def withMinorSeventh(A): return A+[10.]
def withMajorSeventh(A): return A+[11.]


############ Sec. 4.2 Atonal and tonal harmonies, harmonic expansion and modulation
# Table 2.23
def relativa(TT):
    """Returns the relative chord.

    TT is a major or minor triad at a closed and fundamental position."""
    T = n.copy(TT)
    if T[1]-T[0] == 4:  # TT is major
        T[2] = 9.  # returns minor chord a minor third bellow
    elif T[1]-T[0] == 3:  # TT is minor
        T[0] = 10.  # returns major chord a major third above
    else:
        print("send me only minor or major perfect triads")
    return T


def antiRelativa(TT):
    """Returns the anti-relative chord."""
    T = n.copy(TT)
    if T[1]-T[0] == 4.:  # major
        T[0] = 11.  # returns up minor
    if T[1]-T[0] == 3.:  # menor
        T[2] = 8.  # returns down major
    return T

# Medians
def sup(TT):
    T = n.copy(TT)
    if T[1]-T[0] == 4.:  # major
        T[0] = 11.
        T[2] = 8.  # returns major
    if T[1]-T[0] == 3.:  # minor
        T[0] = 10.
        T[2] -= 1.  # returns minor
    return T

def inf(TT):
    T = n.copy(TT)
    if T[1]-T[0] == 4.:  # major
        T[2] = 9
        T[0] = 1.  # returns major
    if T[1]-T[0] == 3.:  # minor
        T[2] = 8.
        T[0] = 11.  # returns minor
```

```python
    return T

def supD(TT):
    T = n.copy(TT)
    if T[1]-T[0] == 4.:  # major
        T[0] = 10.
        T[1] = 3.  # returns major
    if T[1]-T[0] == 3.:  # minor
        T[0] = 11.
        T[1] = 4.  # returns minor
    return T

def infD(TT):
    T = n.copy(TT)
    if T[1]-T[0] == 4.:  # major
        T[1] = 3.
        T[2] = 8.  # returns major
    if T[1]-T[0] == 3.:  # minor
        T[1] = 4.
        T[2] = 9.  # returns minor
    return T

# Main tonal functions
tonicM = [0., 4., 7.]
tonicm = [0., 3., 7.]
subM = [0., 5., 9.]
subm = [0., 5., 8.]
dominant = [2., 7., 11.]
Vm = [2., 7., 10.]  # minor chord is not dominant


############ Sec. 4.3 Counterpoint
def contraNotaNotaSup(alturas=[0,2,4,5,5,0,2,0,2,2,2,0,7,\
                                     5,4,4,4,0,2,4,5,5,5]):
    """Returns a melody given input melody

    Limited in 1 octave"""
    first_note = alturas[0]+(7,12)[n.random.randint(2)]
    contra = [first_note]

    i=0
    cont=0 # parallels counter
    reg=0 # interval register where the parallel was done
    for al in alturas[:-1]:
        mov_cf=alturas[i:i+2]
        atual_cf,seguinte_cf=mov_cf
        if seguinte_cf-atual_cf>0:
            mov="asc"
        elif seguinte_cf-atual_cf<0:
            mov="asc"
        else:
            mov="obl"

        # possibilities by consonances
        possiveis=[seguinte_cf+interval for interval in\
                                    [0,3,4,5,7,8,9,12]]
        movs=[]
```

```
        for pos in possiveis:
            if pos -contra[i] < 0:
                movs.append("desc")
            if pos - contra[i] > 0:
                movs.append("asc")
            else:
                movs.append("obl")

        movt=[]
        for m in movs:
            if 'obl' in (m,mov):
                movt.append("obl")
            elif m==mov:
                movt.append("direto")
            else:
                movt.append("contrario")
        blacklist=[]
        for nota,mt in zip(possiveis,movt):

            if mt == "direto": # direct movement
                # does not accept perfect consonances
                if nota-seguinte_cf in (0,7,8,12):
                    possiveis.remove(nota)
        ok=0
        while not ok:
            nnota=possiveis[n.random.randint(len(possiveis))]
            if nnota-seguinte_cf==contra[i]-atual_cf: # parallel
                intervalo=contra[i]-atual_cf
                novo_intervalo=nnota-seguinte_cf
                if abs(intervalo-novo_intervalo)==1: # same 3 or 6 type
                    if cont==2: # if already had 2 parallels
                        pass # another interval
                    else:
                        cont+=1
                        ok=1
            else: # oblique or opposite movement
                cont=0 # make parallels equal to zero
                ok=1
        contra.append(nnota)
        i+=1
    return contra


############ Sec. 4.4 Rhythm
# See Poli Hit Mia musical piece


############ Sec 4.5 Repetition and variation: motifs and larger units
# Ubiquitous concepts
# examples
S = [1, 2, 1.5, 3]  # a sequence of parameters, e.g. durations
S1 = S[::-1]  # reversion
S2 = [i*4 for i in S]  # expansion
S3 = [i*.5 for i in S]  # contraction
S4 = S[2:] + S[:2]

```

```
# now suppose that S is a sequence of pitches
S5 = [i+7 for i in S]  # transposition
S6 = [i-12 for i in S]  # interval inversion


############ Sec 4.6 Directional structures
# See Dirracional musical piece


############ Sec. 4.7 Cyclic structures
# See 3 Trios musical pieces
# and the PPEPPS
```

2 Musical pieces

The code for the musical pieces are omitted from this PDF because it would make the document lengthy. Please see [3] and [1] to know what are the available scripts for rendering musical pieces and reach them.

3 Auxiliary files

The code for the auxiliary files (e.g. to render the figures in the article [2] are omitted from this PDF because it would make the document lengthy. Please see [3] and [1] to know what are the available auxiliary scripts and reach them.

4 Final considerations

This document exhibits the code that implements the relations in the sections of [2]. All this scripts are available in [1] with other documentations and further scripts e.g. to render musical pieces and the figures in [2]. One should also reach [3] to know about the resources in the MASS framework. This document should be available at [4].

References

[1] Fabbri (2017). Música no áudio digital : descrição psicofísica e caixa de ferramentas. MSc dissertation. Available at: `http://www.teses.usp.br/teses/disponiveis/76/76132/tde-19042013-095445/publico/RenatoFabbri_ME_corrigida.pdf`

[2] ISO: 226. (2003). Normal Equal-Loudness Level Contours.

[3] ISO: 226. (2003). Normal Equal-Loudness Level Contours.

[4] ISO: 226. (2003). Normal Equal-Loudness Level Contours.