Music Scoring With Gabor Time-Frequency Transform

Arnold Peter Ruymgaart[*]

University of Washington
apruymgaart@gmail.com github udemy

March 20, 2021

Abstract

In this report we analyze samples of music and attempt to reproduce guitar and bass music scores from them. In order to accomplish this we carry out a time-frequency decomposition of the signal. For the latter, we make use of the Short Time Fourier Transform (STFT) also called Gabor transform. We use low, high or bandpass filters to isolate the bass or guitar from other sound.

I. Introduction & Overview

Music scoring is the process of transcribing the musical notes produced in a musical performance. This is generally not an easy task to automate. In case of guitar and bass music difficulties arise from the many different playing techniques [Su, 2019].

Musical instruments produce different frequency sounds at different times and one goal of this exercise is to develop an understanding of the use and limitations of time-frequency decomposition with STFT-Gabor transforms.

In this paper we'll briefly review the necessary theory to understand & implement the Gabor transform in Section II. Then in Section III we'll explain the code implementation in detail. The results of scoring the bass and/or guitar of two music samples are included in Section IV.

II. Theoretical Background

In order to accomplish our music scoring task, the following principles are needed and briefly reviewed:

A. Time,Frequency & Time-Frequency Analysis
B. Gabor/STF Transform
C. Gabor/STFT limitations
D. Low & high pass filters
E. Music analysis

(A) Time, Frequency & Time-Frequency Analysis. The reciprocal domain of time is frequency. Signals (functions of time t) can be decomposed into their original domain, the reciprocal domain (frequency) or some combination.

[*]Contact for further information

Time series analysis involves decomposing a signal in the time domain. Methods include Autoregression (AR) in which a signal amplitude at time t is decomposed into a function (typically a linear combination) of its previous amplitudes: $s_t = w_1 s_{t-1} + w_2 s_{t-2} + \dots$ Only time information is used so this is a time domain-only decomposition as visualized in the top-left of Figure 1.

Conversely, *Fourier frequency analysis* $f(t) \mapsto F(k)$ decomposes a (time) signal $f(t)$ into frequencies (wavenumbers k) only while retaining no time information at all. In music, if a note of some frequency is played at a certain time, it will show up in the spectrum regardless of what time it was played. The frequency decomposition is illustrated in the top-right panel of Figure 1.

Time-frequency analysis combines the decomposition into both domains. A signal $f(t)$ is decomposed into time-dependent frequencies. The traditional method is the Gabor transform visualized in the bottom right panel of Figure 1. This method applies a time window to the signal function causing all signal outside the time window to be ignored. The window is applied at different time positions through a convolution operation. The Short Time Fourier Transform (STFT) generalizes the Gabor transform in choice of window function. The Gabor transform Gaussian window is a special case of the STFT. The Gabor/STFT is further discussed in (B) below.

Another way to accomplish a time-frequency decomposition is by wavelet transform. Wavelet time-frequency decompositions allow us to circumvent some of the trade-offs and limitations of the STFT that we'll mention in (C) but are beyond the scope of this document. They are mentioned here for completeness and visualized in the bottom right panel of Figure 1. For a more detailed discussion refer to CH13.4-13.5 in [Kutz, 2013].

(B) Gabor & Short Time Fourier Transforms. We define *window function* g with a certain width along time axis τ centered at time point t:

$$g(\tau - t) \tag{1}$$

Now we modify the kernel $e^{ik\tau}$ of the Fourier transform $\int f(\tau)e^{ik\tau}d\tau$ with it to get a new transform kernel $g_{\tau,k}(\tau) = e^{ik\tau}g(\tau - t)$. The transform with this new kernel is the Gabor/STF Transform:

$$G[f(t)](t,k) = \int_{-\infty}^{\infty} e^{-ik\tau}f(\tau)g(\tau - t)d\tau \tag{2}$$

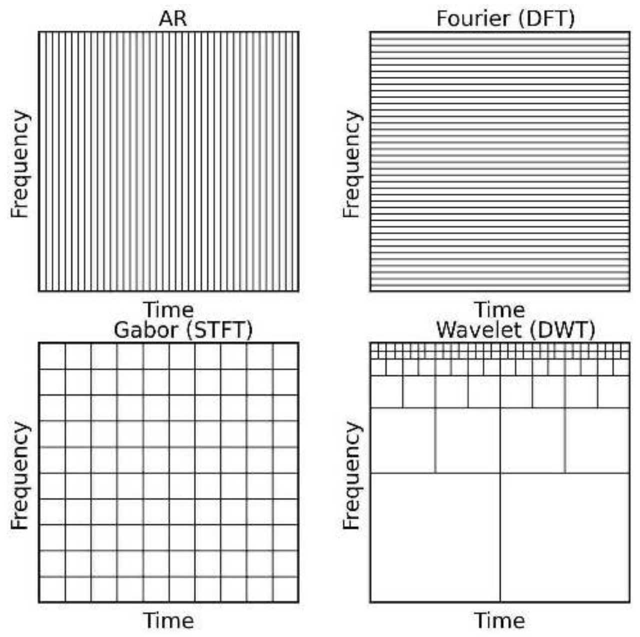
The Gabor/STF transform is invertible but we will not use the inverse in the implementation described in this document. In the above k is wavenumber (often named ω). The window function in the original Gabor transform is *Gaussian*: $g = e^{-w(t-\tau)^2}$ where w is window width parameter (not to be confused with ω).

(C) Gabor/STF Transform Limitations. Although the STFT allows decomposition of a signal into both time and frequency domain, there is a resolution trade-off. The larger the window g, the longer the wavelengths that can be detected. So a wider window (in time) can detect lower frequencies than a narrow window. But a wider time window reduces the positioning resolution. Higher time resolution means lower frequency resolution. A window the size of the entire time axis results in a "regular" Fourier transform, retaining no time information.

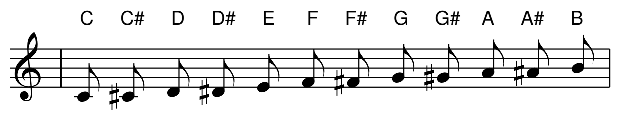
(D) High- and low-pass filters. High, low and band-pass filters work in the frequency domain. A high-pass filter set at a certain cutoff frequency removes any signal of frequencies below the cutoff frequency. A high-pass filter allows the high frequencies to pass. Similarly a low pass filter removes the high frequency components and keeps those components of the signal that have frequencies below the cutoff. A bandpass filter has two cutoffs, a low and a high cutoff, and effectively applies both a high- and a lowpass filter at the same time. There are many types of filters. A popular choice of filter is a Butterworth filter. This particular filter removes the frequencies above/below the cutoff while uniformly keeping the wanted frequencies. Other filters may not apply evenly.

(E) Music analysis. Western music is organized into octaves which are divided by frequency into 7 full notes : A,B,C,D,E,F,G and 12 half-step notes:

Figure 1: *Time,Frequency & Time-Frequency Decomposition*



Time, frequency and time-frequency decomposition of a signal. We illustrate frequency decomposition by drawing horizontal lines dividing the frequency axis and time decomposition with vertical lines. In the top left, an autoregression (AR) decomposes a discrete time signal into a sum of previous time components. In the top right, the DFT is a frequency only decomposition. Both Gabor and Wavelet transforms decompose into both domains.

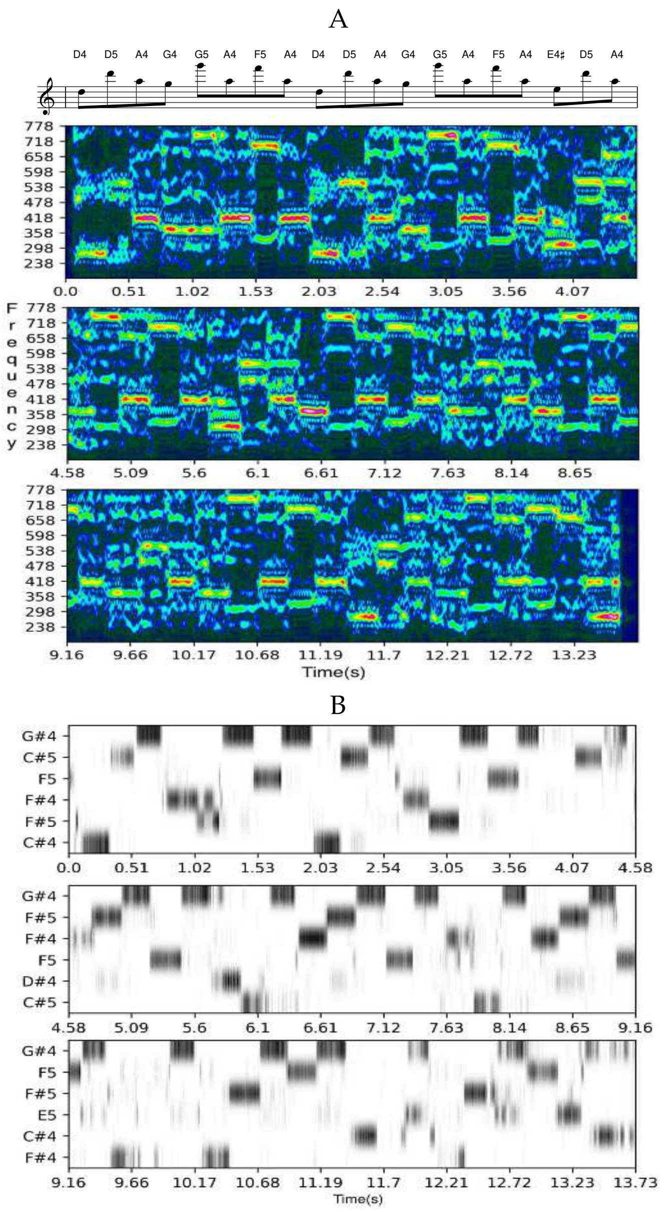


where $C\#$ is a half note up from C and equals D minor (half note down from D). The 8th full note up from any note is the same note but one octave higher (e.g. $C4, C5$). Hence the name octave. Notes are resonant with their counterparts at higher or lower octaves meaning the frequency is an integer multiple. Guitar notes start around 82 Hz (E) and range up to about 1047 Hz (highest C). Bass guitar notes range from about 40 Hz to about 300 Hz so there is overlap possible between guitar and base.

III. Algorithm Implementation & Development

The algorithm implementation (included in Appendix B) consists of a setup component that processes command line input, reads the music files, performs downsampling & filtering and a second component that is the main processing loop that produces the score.

Figure 2: *Gabor Transform & Note Density, GNR Clip*



(A) Actual notes from the guitar solo score of the song and visualization of Gabor the transform. The first row of the Gabor transform is lined up in time with the music notes. **(B)** Notes densities recovered from the transform result. Each vertical line is one instance of sampling that note. Notes are listed top to bottom in order of number of samplings (not frequency). For example, the C#5 is an octave higher than C#4 but was sampled less frequent. Almost all notes are correctly detected as listed in Table 2. GNR calls for down-tuning the guitar 1/2 step: the first D4 is detected as C#4.

Specifically, the setup portion of the script performs the following tasks:

- Read input file and process command line
- Downsample (as desired by command input)
- Calculate FFT frequencies (stored in the list $freqs$), scaling & Gabor interval
- Lowpass filter (as desired by command input)
- Highpass filter (as desired by command input)
- Segment the music into equal size pieces of length $piecelength$ (by zero padding if needed). The number of pieces is stored in the variable $nrPieces$

Note: a bandpass filter is obtained by enabling both lowpass and highpass at the same time. After downsampling and filtering, there is a playback and exit option. This option allows testing of the filtering and downsampling by listening to the result.

Frequencies & STFT/Gabor window scaling. As explained in Section II C, the size of the STF/Gabor transform window determines what frequencies can be detected. Since here we are making use of the discrete Fourier transform, the number of modes N and time length determine the wavenumbers are corresponding frequencies. The number of modes selected will equal the outer window (further discussed in STFT implementation below) in length. Therefore, the frequencies are calculated by the sampling frequency times N scaled to the Fourier domain length of 2π.

The main loop produces the music score from the signal (downsampled & filtered as selected). The loop carries out the STFT from which a time-index note dictionary is made (further explained in $getNotes()$). The dictionary is converted to a density image of the most frequently heard notes as explained further in $noteDensity()$. Example density images are shown in Figure 2 (B) and Figure 3 (B). The density image is converted to a score in $noteImg2noteList()$. Each function is further described below and global variables $nrPieces, piecelength, freqs, S$ are defined/calculated in he setup and equal number of segments, the length of each, Fourier frequencies and the signal, resp.

Main loop

```
1: for $n = 0 : nrPieces$ do
2:     $start = piecelength \cdot (n)$
3:     $stop = piecelength \cdot (n+1)$
4:     $STFT = stft(S, start, stop)$
5:     $dN = getNotes(STFT)$
6:     $[img, noteNames] = noteDensityImg(dN, \ldots)$
7:     $musicScore = noteImg2noteList(img, \ldots)$
8:     $wholeScore = wholeScore + musicScore$
9: end for
```

The **stft** function implements a discrete STF transform of discrete signal S. In this implementation, the STFT is implemented by a windowed discrete Fourier transform

(DFT) and we use a *second window* around the STF/Gabor window to reduce the size of the result matrix in memory. The size of this second window determines the Fourier scaling. Since the Gaussian or Shannon window used here has near zero intensity some distance from its center, there is no need to DFT the entire time axis. Input is time signal $\vec{S}$ and output is a matrix of size $|\vec{S}| \times N$ where N is the number of Fourier modes which equals the length of the second window. $S[k-h : k+h]$ is a portion of the signal of length N centered at k. In the below $\odot$ indicates Hadamard (or element-wise product). Note that g is expected be zero at $\pm N/2 = \pm h$. See numpy.fft documentation for fft and $fftshift$.

stft(S,start,stop)

```
$x = np.arange(N)$
$h = \frac{N}{2}$
$g = e^{(-w(x-h)^2)}$
$STFT = [\,]$
for $k = start : stop$ do
    $ftw = fftshift(fft(S[k-h : k+h] \odot g))$
    $STFT.append(ftw)$
end for
return $STFT$
```

The **getNotes** function converts an STFT array to a dictionary of notes. Each note in the dictionary stores an array of time indices where the note was observed to be the frequency of the maximum intensity (dominant frequency corresponding to max Fourier coefficient). The function $freq2str(\ldots)$ is imported from the audiolazy python library and converts a frequency to a note string (e.g. $req2str(740)$ produces $F\#5 + 0.03$). $argmax(F_i)$ is the index i of the maximum intensity of the windowed FT F_i. $freqs[argmax(F_i)]$ is the frequency associated with the max intensity value of F_i. Getnotes also looks at the intensity of the signal in real space at position the windowed Fourier transform was computed at. If the signal is too low of an intensity, it is ignored.

getNotes(STFT, S, cut=0.001)

```
1: for $i, F_i$ in $enumerate(STFT)$ do
2:     $note_m = freq2str(abs(freqs[argmax(F_i)])$
3:     if $abs(S[j])/np.max(np.abs(S)) > cut$: then
4:         $notes[note_m].append(i)$
5:     end if
6: end for
7: return notes
```

The **noteDensity** function sorts the notes by how often they are heard in the segment. This allows for a way of discarding infrequent sounds at unusual frequencies (that likely are not part of the score). For the top n notes (as selected by input) the function subsequently draws a point (stretched to a vertical line) any time that note was heard on the time axis. See panel B in Figure 2 for example output of this function.

noteDensity(notes, seglen, top=6)

```
1: $IMG = np.zeros((top, seglen))$
2: $noteNames, nrFound = [\,], [\,]$
3: for $note\ in\ notes$ : do
4:     $nrFound.append(len(notes[note]))$
5: end for
6: $indxs = argsort(nrFound, reverse)$
7: for $g, m$ in $enumerate(indxs[0 : top])$ do
8:     $note = noteKeys[m]$
9:     $noteNames.append(note)$
10:    for $j$ in $notes[note]$ do
11:        $IMG[g, j] = 1$
12:    end for
13: end for
14: return $[IMG, noteNames]$
```

The **noteImg2noteList** function generates the score (the music notes) from the density image. Im is a $N_{notes} \times v$ matrix and is small width v of the note density image. The number of such width fractions we break the note density image into is n_{frac} while t_{seg} is segment fraction time and t_{point} is the time length of a single sample point (time length of a single point of S). For each fragment, the note is determined to be the one of most density.

noteImg2noteList(img)

```
1: musicScore = []
2: $v = len(img[0,:])/n_{frac}$
3: for $i_s$ in $0 : n_{frac}$ do
4:     $Im = img[:, (i_s v) : ((i_s + 1)v)]$
5:     $t_{seg} = t_{start} + (((i_s v) + ((i_s + 1)v))/2) \cdot t_{point}$
6:     noteDensity = []
7:     for $n$ in $0 : len(noteNames))$ do
8:         $noteDensity.append(np.sum(Im[n, :]))$
9:     end for
10:    $indx = argmax(noteDensity)$
11:    $musicScore.append([t_{seg}, noteNames[indx]])$
12: end for
13: return $musicScore$
```

noteImg2noteList returns the music score that is accumulated into the final result note list output.

IV. Computational Results

(1) GNR guitar score. We were provided with two music clips, each with an original audio sampling rate of 48kHz. The first sample was a 14 second guitar solo (only) clip by Guns N Roses (we will refer to as GNR). When downsampling, we consider the *Nyquist sampling rate* which is 2× the highest frequency of the signal. The original sampling rate was 44100Hz allowing up to 22× downsample if our guitar maximum frequency is 1000Hz. Downsampling the GNR sample 13 fold to a final sample rate (sampling frequency) of 3640.47 Hz reduced quality but the guitar solo was still clearly audible as assessed by playback. In addition, the GNR sample was high-pass filtered with Butterworth filter (discussed in Section II D) at 150Hz. With Shannon-style pulse STFT widow $w = 40$ of with 80 points (of 1024) was used with 1024 Fourier modes. This window produces similar results to a Gaussian window with parameter $w = 0.0007$ which can also be selected by command line input. The STFT along with note density image of the clip is visualized in Figure 2. The algorithm produced the music score listed in Table 1. The Guns N′ Roses sheet music calls for detuning all strings 1/2 step so a detected $C\#4$ (the first note) is a $D4$ in the sheet music. With that in mind accuracy was good, as partially assessed in Table 2.

(2) Floyd bass score. To get the bass score from our second music sample, a portion from Pink Floyd's Comfortably Numb (we will refer to as the Floyd sample) was downsampled by a factor of 15 and low-pass filtered at 150 Hz again using a Butterworth filter. From listening to the filtered/downsampled audio, it seemed the bass audio was well retained noting that bass guitar operates at low frequencies allowing more downsampling. The processing loop produced the STFT shown in Figure 3 A and note density images shown in Figure 3 B and finally, the compressed bass score listed in Table 3. The Pink Floyd bass score is mostly correct as well. In the sheet music published by the band the bass line (first 11 measures) is: $B2(\times 8), A2(\times 2)G2, F2, E2, B2(\times 5)A(\times 2), G, A, E, B(\times 4)$ while detected (as listed in Table 3) is
$B2, (A\#2, A2)G2, F\#2, B2, E2, B2, (A\#2, A2, A\#2, A2), \ldots$
In the current form, the timing of the notes is only detected if the note changes. A note change is what is recorded in the compressed output (as shown here). So if the same note is played multiple times in a row, the algorithm produces one long note. This could be remedied but this was considered to be beyond the scope of this project. With a few exceptions, the notes are correctly detected.

(3) Floyd guitar score. The third task was to make an attempt at producing the guitar score from the same Floyd sample. This task proved more challenging. In this task, the audio clip was downsampled between 2× and and 15× and was band-pass filtered (see Section II D) between 200 and 1300 Hz or between 300 − 1000 Hz. The scores created by the algorithm produced partial matches to the actual sheet-music but in no case was comprehensively complete. The Gabor transform and note density images are provided for reference in Appendix C along with a portion of the actual guitar solo sheet music.

One problem with this guitar solo is that is isn't truly a solo because there are three or four instruments active: the guitar, a bass (and apparently another rhythm guitar also playing the bass line) and drums. We can mostly filter out the bass and drums, especially with a 300 − 1000 Hz bandpass but some non-guitar sound remains. The lowest note in the guitar solo (according to the sheet music) should be an A3 at 220Hz while the highest not should be a G5 at 784 Hz potentially allowing a band pass filter from about 200 to 800 Hz.

Table 1: *Guitar Score Recovered From GNR Sample*

Time(s)	Note	Time(s)	Note	Time(s)	Note
0.09	C#4	4.67	F#4	8.88	G#4
0.46	C#5	4.85	F#5	9.06	F5
0.64	G#4	5.04	G#4	9.43	G#4
0.82	F#4	5.40	F5	9.61	F#4
1.19	F#5	5.59	G#4	9.98	G#4
1.37	G#4	5.95	D#4	10.35	F#4
1.56	F5	6.13	C#5	10.53	F#5
1.74	G#4	6.32	G#4	10.71	G#4
2.11	C#4	6.50	F#4	11.08	F5
2.29	C#5	6.68	F#5	11.26	G#4
2.47	G#4	7.05	G#4	11.45	C#4
2.84	F#4	7.23	F5	11.99	G#4
3.02	F#5	7.42	G#4	12.18	C#4
3.20	G#4	7.60	F#4	12.36	F#5
3.57	F5	7.97	C#5	12.73	G#4
3.75	G#4	8.15	G#4	12.91	F5
4.12	C#5	8.33	F#4	13.28	E5
4.49	G#4	8.70	F#5	13.46	C#4

Table 2: *Actual Vs Detected - GNR Sample*

Note index	1	2	3	4
Actual	D4	D5	A4	G4
Detected	C#4	C#5	G#4	F#4
Detected ($\uparrow \frac{1}{2}$)	D4	D5	A4	G4
Note index	5	6	7	8
Actual	G5	A4	F5	A4
Detected	F#5	G#4	F5	G#4
Detected ($\uparrow \frac{1}{2}$)	G5	A4	F#5	A4

Table 3: *Bass Score Recovered From Floyd Sample*

Time(s)	Note	Time(s)	Note	Time(s)	Note
0.17	B2	21.69	F#2	40.82	B2
2.90	A#2	22.03	G2	44.23	A#2
3.24	A2	23.06	F#2	44.57	B2
6.66	G2	23.74	B2	48.33	A#2
8.03	F#2	33.30	A2	48.67	A2
8.71	B2	34.67	A#2	51.41	A#2
9.39	E2	35.35	A2	52.09	F#2
10.42	B2	36.72	A#2	54.14	B2
17.93	A#2	37.06	G2	54.48	E2
18.27	A2	38.43	F#2	55.85	B2
19.98	A#2	38.77	B2	58.24	A#2
20.66	A2	40.48	A#2	58.58	B2

V. Summary and Conclusions

Considering its simplicity, the algorithm implemented here does a remarkably good job reproducing guitar and bass music scores from certain less challenging samples in a completely automated way. In the GNR sample, the algorithm nearly perfectly reproduced the music as partially summarized in Table 2 and also did a good job reproducing the bass score of the Floyd sample.

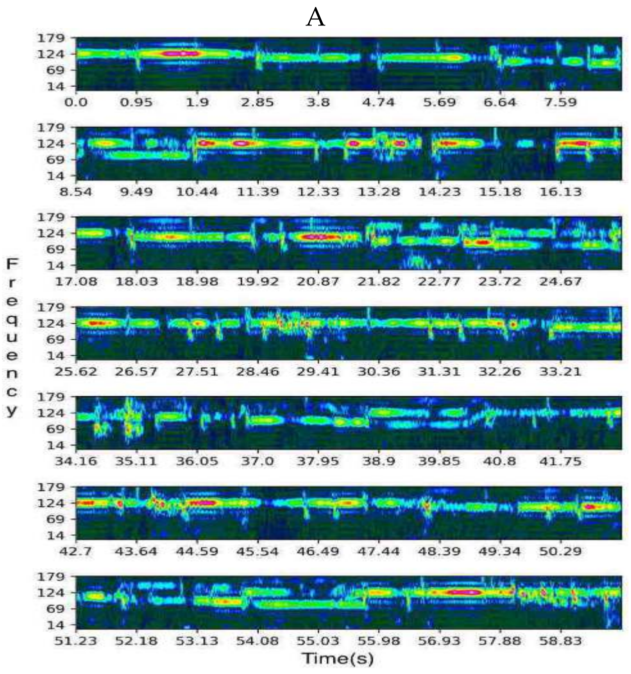
Successful scoring requires clean isolation of a single instrument. In addition, the current version only lists one note at a time and does not recognize breaks in the same note. The limitations of the algorithm become apparent when attempting to score more challenging samples such as the Floyd guitar sample. The provided code could serve as a basis for something more sophisticated that keeps track of timing and is able to recognize style variations in addition to doing a better job at isolating a single instrument. The authors of [Su, 2019] have made advances in the recognition of special guitar techniques with the use of machine learning.

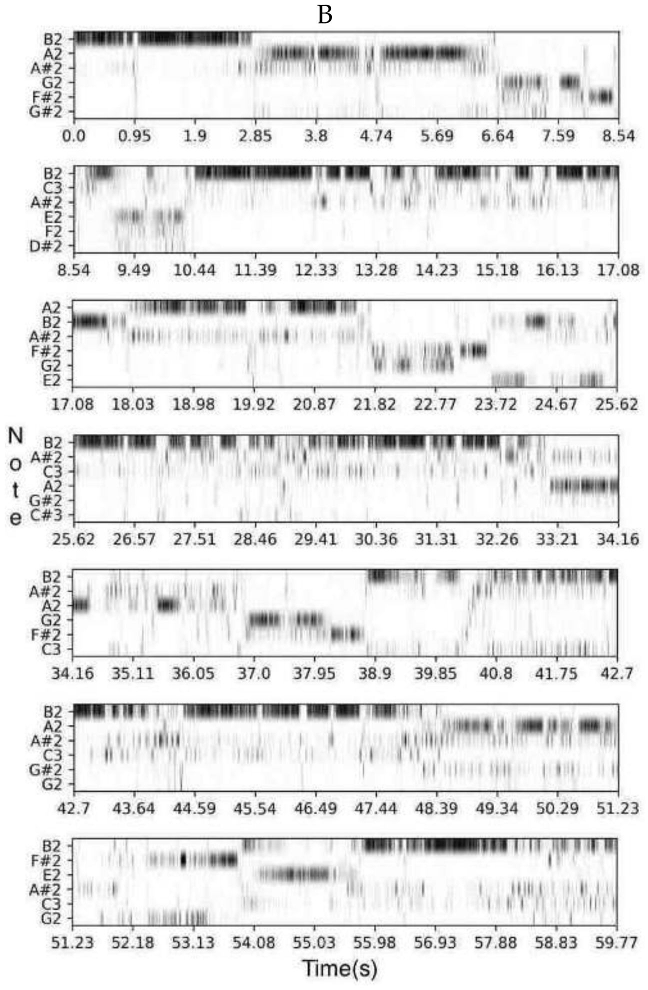
References

[Kutz, 2013] Kutz, J. Nathan (2013). Data-Driven Modeling & Scientific Computation: : Methods for Complex Systems & Big Data. *Oxford University Press, Inc. 198 Madison Ave. New York, NY, United States*, ISBN 978-0-19-966034-6.

[Su, 2019] Ting-Wei Su, Yuan-Ping Chen, Li Su and Yi-Hsuan Yang (2019). TENT: Technique-Embedded Note Tracking for Real-World Guitar Solo Recordings. *ransactions of the International Society for Music Information Retrieval, 2(1), pp. 15–28.*

Figure 3: *Gabor Transform Floyd Bass Clip*





(A) Visualization of Gabor transform of the Floyd sample filtered for the bass score (low-pass 150). **(B)** Note density image derived from (A).

A. Python functions used

The functions *stft*, *getNotes*, *noteDensityImg* and *noteImg2noteList* are explained in Section III. The functions *butter_pass* and *butter_filter* are wrappers making use of the Butterworth filter implemented in the Python-scipy signal (scipy.signal) package. The function *audioread* is a python equivalent to MATLAB audioread utilizing the Python pydub package. The pydub package is also used in *np2audio* to convert a numpy array back to audio.

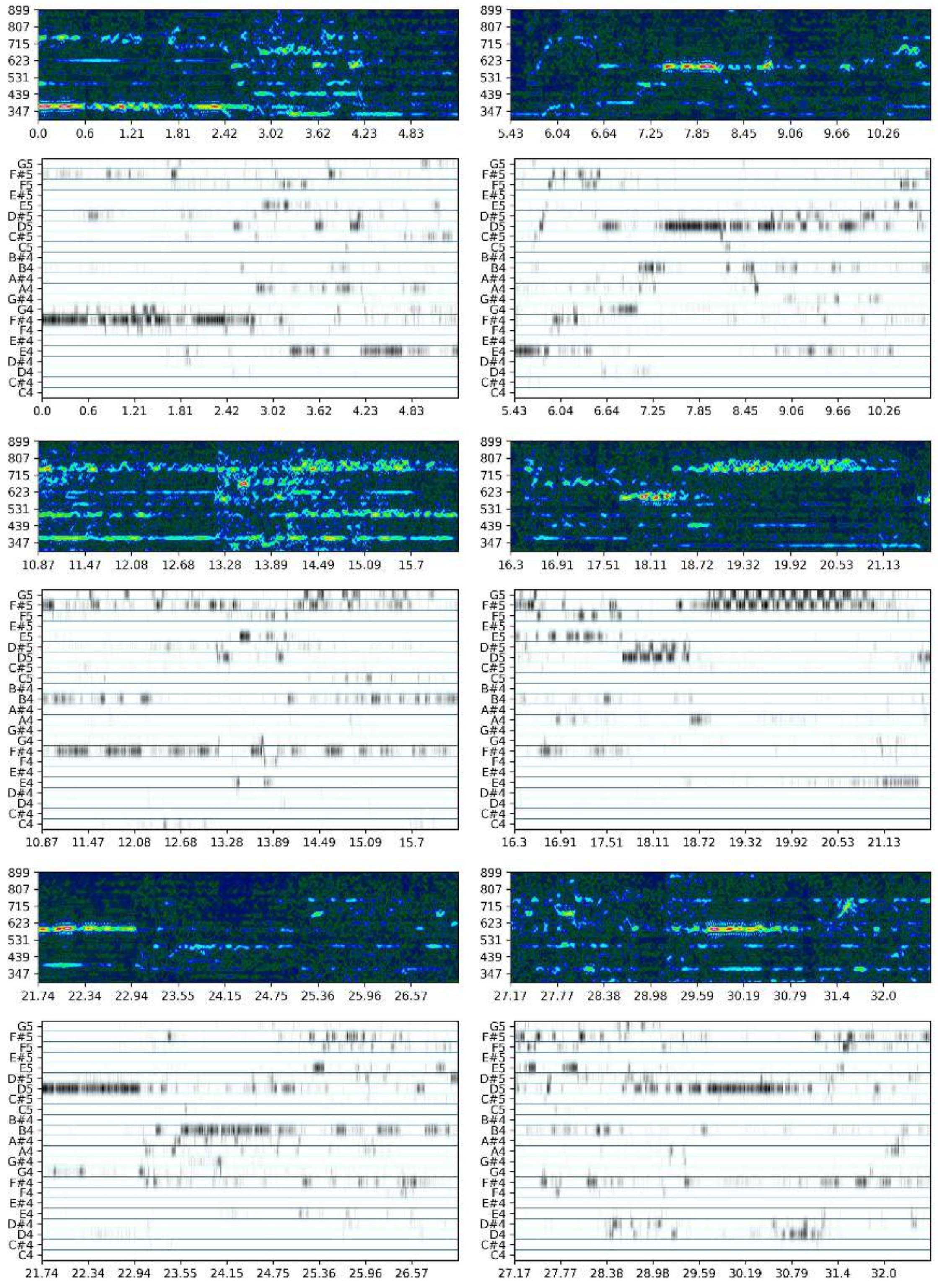
```python
# AP Ruymgaart, functions for audio file analysis
import math,scipy,os,sys,copy,numpy as np,scipy.io as sio,matplotlib.pyplot as plt
from pydub import AudioSegment
from pydub.playback import play
from pydub.utils import mediainfo
from scipy.signal import resample, butter, lfilter, freqz
from scipy.interpolate import interp1d
from scipy.io import wavfile
from audiolazy import *
fft, fftshift, ifft = np.fft.fft, np.fft.fftshift, np.fft.ifft
abs, rndm, exp, machineEpsilon = np.absolute, np.random.normal, np.exp, np.finfo(np.float32).eps
np.set_printoptions(precision=3)

def audioread(fname):
        audio, info = AudioSegment.from_file(fname), mediainfo(fname)
        return [np.array(audio.get_array_of_samples()), float(info['sample_rate'])]

def np2audio(x, sr, normalized=False):
        channels = 2 if (x.ndim == 2 and x.shape[1] == 2) else 1
        if normalized:  y = np.int16(x * 2 ** 15)
        else: y = np.int16(x)
        return AudioSegment(y.tobytes(), frame_rate=sr, sample_width=2, channels=channels)

def writeAudiofile(data,f,fout='output.wav'): wavfile.write(fout, f, data.astype(np.int16))

def ResampleLinear1D(original, targetLen):
        original = np.array(original, dtype=np.float)
        index_arr = np.linspace(0, len(original)-1, num=targetLen, dtype=np.float)
        index_floor = np.array(index_arr, dtype=np.int)
        index_ceil, index_rem = index_floor + 1,   index_arr - index_floor
        val1,val2 = original[index_floor], original[index_ceil % len(original)]
        interp = val1 * (1.0-index_rem) + val2 * index_rem
        assert(len(interp) == targetLen)
        return interp

def pad(S, plen, wh='b'):
        pd = [0.0]*plen
        if wh == 'b': return np.array( pd + list(S) + pd )
        elif wh == 'r': return np.array( list(S) + pd )
        else : return np.array( pd + list(S))

def stft(S, w=0.004, nrModes=1024, start=0,end=1000, win='shannon'):
        x, hp = np.arange(nrModes), nrModes/2
        if win == 'shannon':
                g = np.zeros(x.shape)
                g[int(hp-w):int(hp+w)] = 1.0
        else:
                g = exp(-w*(x - hp)**2)
        img = []
        for k in range(start,end):
                try:
                        win = S[k-int(hp):k+int(hp)] * g
                        ftw = fftshift(fft(win))
                except:
                        print('*ERR:', len(S), k-int(hp), k+int(hp), start, end)
                        ftw = [0] * nrModes
                img.append(ftw)
        return np.array(img)

def getNotes(img, freqs, S, cut=0.001):
        mxSig = np.max(np.abs(S))
        notes = {}
        for j,v in enumerate(img):
                indMx = np.argmax(v)        #- index of max intensity wavenumber
                mf = abs(freqs[indMx])      #- frequency of max intensity
                noteStr = freq2str(mf)      #- get note (audiolazy)
                if noteStr.find('-') > -1: #- look for e.g. C4-50% or C4+33%
                        try: note = noteStr.split('-')[0]
                        except: note = '?'
                elif noteStr.find('+') > -1:
                        try: note = noteStr.split('+')[0]
                        except: note = '?'
                try:
                        if abs(S[j])/mxSig > cut:
                                notes[note].append(j)
                except: notes[note] =[j]
        return notes

def getMultiNotes(img, freqs, S, cut=0.001, nn=10):
        mxSig = np.max(np.abs(S))
        notes = {}
        for j,v in enumerate(img):
                indsMx = list(np.argsort(v))    #- indexes of max intensity wavenumber
                indsMx = indsMx[::-1]           #- reverse argsort (descending)
                for iMx in indsMx[0:nn]:        #- loop over top 10 max intens wavnr inds
                        mf = abs(freqs[iMx])        #- frequency of max intensity
                        noteStr = freq2str(mf)      #- get note (audiolazy)
                        if noteStr.find('-') > -1: #- look for e.g. C4-50% or C4+33%
                                try: note = noteStr.split('-')[0]
                                except: note = '?'
                        elif noteStr.find('+') > -1:
                                try: note = noteStr.split('+')[0]
                                except: note = '?'
                        try:
                                if abs(S[j])/mxSig > cut:
                                        notes[note].append(j)
                        except: notes[note] = [j]
        return notes

def noteDensityImg(notes, seglen, top=6):
        IMGNOTES = np.zeros((top,seglen))
        noteNames = []
        nrFound = []
        for note in notes:
                nrFound.append( len(notes[note]) )
        indxs = list(np.argsort(nrFound))
        indxs = indxs[::-1]
        noteKeys = [*notes]
        for g,m in enumerate(indxs[0:top]):
                note = noteKeys[m]
                noteNames.append(note)
                for j in notes[note]:
                        IMGNOTES[g,j] = 1
        return [IMGNOTES, noteNames]

def butter_pass(cutoff, fs, order=5, btype='low'):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype=btype, analog=False)
    return b, a

def butter_filter(data, cutoff, fs, order=5, btype='low'):
    b, a = butter_pass(cutoff, fs, order=order, btype=btype)
    y = lfilter(b, a, data)
    return y

def stftImgCrop(img, freqs, low, high):
        ret,frt = [],[]
        for k in range(len(freqs)):
                if freqs[k] >= low and freqs[k] <= high:
                        ret.append(img[:,k])
                        frt.append(freqs[k])
        return np.flipud(np.array(ret)), frt[::-1]

def stftPlot(img, freqs, t1, t2, aspect=20, modLbl=32, fname='stftPlot.png'):
        fig, ax = plt.subplots()
        xtickpos = np.linspace(0,img.shape[1],10).astype(int)
        xlabs = np.round(np.linspace(t1,t2, 10),2)
        for m in range(len(xlabs)): xlabs[m] = str(xlabs[m])
        ax.set_xticks(xtickpos)
        ax.set_xticklabels(xlabs)
        ylabs,yt = [],[]
        for m in range(len(freqs)):
                if m % modLbl == 0.0:
                        ylabs.append(int(round(freqs[m],0)))
                        yt.append(m)
        ax.set_yticks(yt)
        ax.set_yticklabels(ylabs)
        ax.imshow(abs(img), cmap='gist_ncar', aspect=aspect)
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.clf()

def signalInfo(S, freq):
        sLength = 1/freq           # single sample point length (time in seconds)
        duration = len(S)*sLength # duration is in seconds
        print('==== signal info ====\n\tLen signal (nr points)', len(S), 'sampling rate (freq Hz - cycles/sec)', freq)
        print('\tduration', duration,'\n\tPoint length (seconds)', sLength)
        return [duration, sLength]

def notesPlot(img, t1,t2, noteNames, fname='notesPlot.png', noteDict=None):
        fig, ax = plt.subplots()
        ax.set_xticks( np.linspace(0,img.shape[1],10).astype(int) )
        xlabs = np.round(np.linspace(t1,t2, 10),2)
        for m in range(len(xlabs)): xlabs[m] = str(xlabs[m])
        ax.set_xticklabels(xlabs)
        if not noteDict is None:
                imgN = np.zeros((len(noteDict),len(img[0])))
                for j,ntn in enumerate(noteNames):
                        try: imgN[noteDict[ntn], :] = img[j,:]
                        except: print('note out of given range', ntn)
                ax.imshow(np.flipud(imgN), aspect=400, cmap='Greys')
                ax.set_yticks(np.arange(len( [*noteDict] )))
                ax.set_yticklabels([*noteDict][::-1])
                for m in range(len([*noteDict])): ax.axhline(m+0.5, linewidth=0.5)
        else:
                ax.imshow(img, aspect=400, cmap='Greys')
                ax.set_yticks(np.arange(len(noteNames)))
                ax.set_yticklabels(noteNames)
                for m in range(len(noteNames)): ax.axhline(m+0.5, linewidth=0.5)
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.clf()

def noteImg2noteList(IMGNOTES, pLength, noteNames, offset=0, nSeg=15):
        musicScore = []
        intv = len(IMGNOTES[0,:])/nSeg
        for si in range(nSeg):
                seg = IMGNOTES[:, int(si*intv):int((si+1)*intv)]
                segTime = offset + ((int(si*intv) + int((si+1)*intv))/2) * pLength
                noteDensity = []
                for nn in range(len(noteNames)): noteDensity.append(np.sum(seg[nn,:]))
                musicScore.append([segTime, noteNames[np.argmax(np.array(noteDensity))] ])
        return          musicScore

def compressScore(score):
        tLast,nLast = score[0][0], score[0][1]
        compressed = [[tLast,nLast]]
        for nt in score:
                t,n = nt[0],nt[1]
                if n != nLast: compressed.append(nt)
                nLast = n
        return compressed

def processCmdArgs(arglst):
        cmd = {}
        for c in arglst:
                try:
                        elms = c.split('=')
                        cmd[elms[0]] = elms[1]
                except: pass
        return cmd

def makeNoteDictionary(strt='C4', stp='G5'):
        n, keep = 0, False
        dNotes = {}
        for octv in [1,2,3,4,5]:
                for Nt in ['C','D','E','F','G','A','B']:
                        for shrp in ['','#']:
                                note = Nt+shrp+str(octv)
                                if note == strt: keep = True
                                if keep:
                                        dNotes[note] = n
                                        n += 1
                                if note == stp: keep = False
        return dNotes
```

B. Python codes

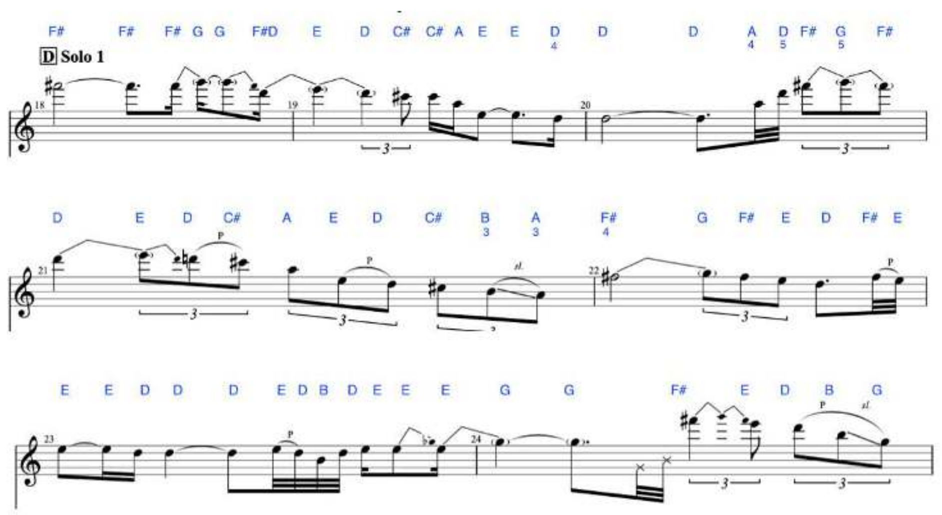
```python
# AP Ruymgaart, main music scoring script
import math,scipy,os,sys,copy,numpy as np,scipy.io as sio,matplotlib.pyplot as plt, prettytable
from scipy.signal import butter, lfilter, freqz
from sklearn.cluster import MeanShift, estimate_bandwidth
from pydub import AudioSegment
from pydub.playback import play
from pydub.utils import mediainfo
from scipy.signal import resample
from scipy.interpolate import interp1d
from audiolazy import *
fft, fftshift, ifft = np.fft.fft, np.fft.fftshift, np.fft.ifft
abs, rndm, exp = np.absolute, np.random.normal, np.exp
machineEpsilon = np.finfo(np.float32).eps
np.set_printoptions(precision=3)
from audioFunctions import *

#====== SETUP ======
print('===================================== start =================================\nInput:')
cmds = processCmdArgs(sys.argv)
for c in cmds: print('\t', c, cmds[c])
try:
        n,pi = int(cmds['modes']), np.pi
        w, wintype = float(cmds['ww']), cmds['win']
        cutoffLow, cutoffHigh = float(cmds['lowpass'].split(',')[1]), float(cmds['highpass'].split(',')[1])
        lowPass, highPass  = cmds['lowpass'].split(',')[0]=='True', cmds['highpass'].split(',')[0]=='True'
        makePlots = cmds['plots'] == 'True'
        downSample, dsmplFact = cmds['downsample'].split(',')[0]=='True', float(cmds['downsample'].split(',')[1])
        piecelengthWanted = int(cmds['seglen'])
        plScale,ntFirst,ntLast = cmds['plotscale'].split(',')[0]=='True', cmds['plotscale'].split(',')[1], \
                cmds['plotscale'].split(',')[2]
        mxNotesPerSeg = int(cmds['maxNotesSeg'])
        icut = float(cmds['icut'])
        imultinote = int(cmds['multinote'])
        resolutionPerSeg = int(cmds['resolutionSeg'])
        saveAndExit, fout = cmds['saveAndExit'].split(',')[0] == 'True', cmds['saveAndExit'].split(',')[1]
        preview = cmds['preview'] == 'True'     # preview = listen and exit
        [S,freq] = audioread(cmds['file'])
        stftPlotPars = [ int(cmds['stftPlot'].split(',')[0]), int(cmds['stftPlot'].split(',')[1]), \
                int(cmds['stftPlot'].split(',')[2]), int(cmds['stftPlot'].split(',')[3]) ]
        if plScale: nd = makeNoteDictionary(strt=ntFirst, stp=ntLast)
        else: nd = None
except:
        print('INPUT ERROR', cmds)
        exit()

#====== downsample & segment =====
if downSample:
        S = ResampleLinear1D(S, int(len(S)/dsmplFact))
        freq = freq/dsmplFact
nrPieces = int(len(S)/piecelengthWanted)
piecelength = int(len(S)/nrPieces)
reclen = piecelength*nrPieces
print(nrPieces, piecelength, reclen, len(S))
if len(S) > reclen: S = S[0:reclen]
if len(S) < reclen: S = pad(S, reclen-len(S), wh='r')

#====== FFT scaling =====
[duration, pLength] = signalInfo(S,freq)
L = (n * pLength)/2
x2 = np.linspace(-L,L,n+1)
x = x2[0:n]
scale = (2*pi)/(2*L)
nk = np.append(np.arange(0,n/2),np.arange(-n/2,0))
k = scale * nk
ks = fftshift(k)
nks = fftshift(nk)
freqs = (1/(2*pi)) * ks

#====== FILTER (if both lowPass, highPass then bandpass) =====
order = 6
if lowPass:  S = butter_filter(S, cutoffLow, freq, order, btype='low')
if highPass: S = butter_filter(S, cutoffHigh, freq, order, btype='high')

#====== info output & if preview then listen and quit ======
if downSample: print('DOWNSAMPLED to', freq, 'factor=', dsmplFact, 'original sample freq=', freq*dsmplFact)
if lowPass: print('LOW PASS FILTER', cutoffLow)
if highPass: print('HIGH PASS FILTER', cutoffHigh)
print('Clip segmented into', nrPieces, 'pieces of length', piecelength, reclen, len(S))
print('Interval width=', 2*L, 'L=',L, 'lowest freq =', 1/L, 'highest freq', 0.5/pLength)
if True:
        tbl = prettytable.PrettyTable()
        tbl.field_names = ["Wavenumber", "index", "wavelen in time (2L/n)", "freq"]
        print('==== FT info ===\n', 'Nr Fourier modes=', n)
        for j in range(len(ks)):
                if j % 64 == 0.0 and nks[j]: tbl.add_row([ ks[j], nks[j], (2*L)/nks[j],  freqs[j] ])
        print(tbl)
if preview:
        play(np2audio(S, freq))
        exit()
if saveAndExit:
        writeAudiofile(S, int(freq), fout)
        exit()

#====== MAIN PROCESSING LOOP =========
pS = pad(S, int(n/2))
wholeScore = []
for ns in range(nrPieces):
        strt,stp = piecelength*ns, piecelength*(ns+1)
        img = stft(pS, w=w, nrModes=n, start=strt+int(n/2),end=stp+int(n/2), win=wintype)
        notes = getMultiNotes(img, freqs, S, icut, imultinote)
        [densImg, noteNames] = noteDensityImg(notes, piecelength, top=mxNotesPerSeg)
        musicScore = noteImg2noteList(densImg, pLength, noteNames, offset=strt*pLength, nSeg=resolutionPerSeg)
        wholeScore += musicScore
        if makePlots:
                imgCrop, freqCrop = stftImgCrop(img, freqs, stftPlotPars[0], stftPlotPars[1])
                stftPlot(imgCrop, freqCrop, strt*pLength, stp*pLength, aspect=stftPlotPars[2], modLbl=stftPlotPars[3], \
                        fname='images/stft_seg%d.png' % (ns+1))
                notesPlot(densImg, strt*pLength, stp*pLength, noteNames, fname='images/noteDensity_seg%d.png' % (ns+1), noteDict=nd)

fout = open('images/score.txt','w')
compr = compressScore(wholeScore)
print('=== compressed ===')
for j,nt in enumerate(compr):
        print(j,nt)
        fout.write( "%5.2f, %s\n" %(nt[0],nt[1]) )
fout.close()
```

C. Additional information

Floyd sample : guitar, Gabor transform & corresponding note density plots follow below. For this sample, the function *notesPlot* was adapted to take a note scale command. The result plots the notes (within the scale, below C4 to G5) in musical sound frequency order (for easier comparison to sheet music):



Annotated sheet-music, Floyd (first portion of) guitar solo:


F# F# F# G G F#D E D C# C# A E E D D D A D F# G F#
4 4 5 5
D Solo 1
D E D C# A E D C# B A F# G F# E D F# E
3 3 4
E E D D D E D B D E E E G G F# E D B G

The algorithm can be completely configured from command line input which allowed experimentation without code changes. Example command line input:

```
rm images/*.png images/score.txt

#=== Floyd Bass ====
if true
then
python3 hw2_clean.py \
file=Floyd.m4a \
modes=1024 win='shannon' ww=40 \
lowpass=True,150.0 highpass=false,50.0 downsample=True,5 \
seglen=14000 maxNotesSeg=5 resolutionSeg=25 icut=0.01 multinote=1 \
plots=True preview=False plotscale=False,'C1','C6' \
stftPlot=40,170,30,32 saveAndExit=False,floydbass.wav
fi

#=== Floyd guitar ====
if false
then
python3 hw2_clean.py \
file=Floyd.m4a \
modes=1024 win='shannon' ww=80 \
lowpass=True,900.0 highpass=True,300.0 downsample=True,15 \
seglen=15000 maxNotesSeg=14 resolutionSeg=50 icut=0.04 multinote=4 \
plots=True preview=False plotscale=True,'C4','G5' \
stftPlot=300,900,20,32 saveAndExit=False,floydguitar.wav
fi

#=== GNR guitar ===
if false
then
python3 hw2_clean.py \
file=/Users/arnoldruymgaart/Downloads/GNR.m4a \
modes=1024 win='shannon' ww=40 \
lowpass=False,150.0 highpass=True,250.0 downsample=True,25 \
seglen=8000 maxNotesSeg=6 resolutionSeg=25 icut=0.01 multinote=1 \
plots=True preview=False plotscale=False,'C1','C6' \
stftPlot=180,780,10,32 saveAndExit=False,gnrguitar.wav
fi
```