Construction automatique de niveaux pour le jeu de rythme osu! à partir de fichiers audio

Aboujaib Alexandre - 28173

Section 1

Presentation du problème

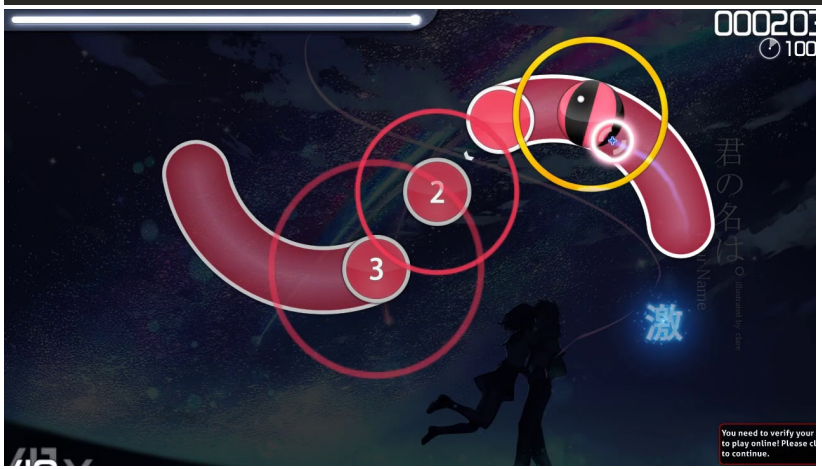
Le jeu osu!

- Jeu de rythme pour PC
- Sorti le 16 septembre 2007, adapté d'un jeu pour DS
- Environ 500 000 joueurs actifs
- Se joue avec une souris/tablette et clavier

Les beatmaps (niveaux)

A **beatmap** (sometimes called *beatmapset*) is a set of game levels (difficulties) that are composed of various hit objects and almost always represent a single song. It also includes other components, all packed in an archive with the `.osz` extension:

- the song itself, stored in MP3 or Ogg format.
- background images, or a video, acting as a playfield.
- custom hitsounds for arrangement and improved aural feedback (optional).
- storyboard with motion graphics and special effects, serving as a background story or theme for the song (optional).
- custom skin, which changes the appearance of interface and gameplay elements (optional).



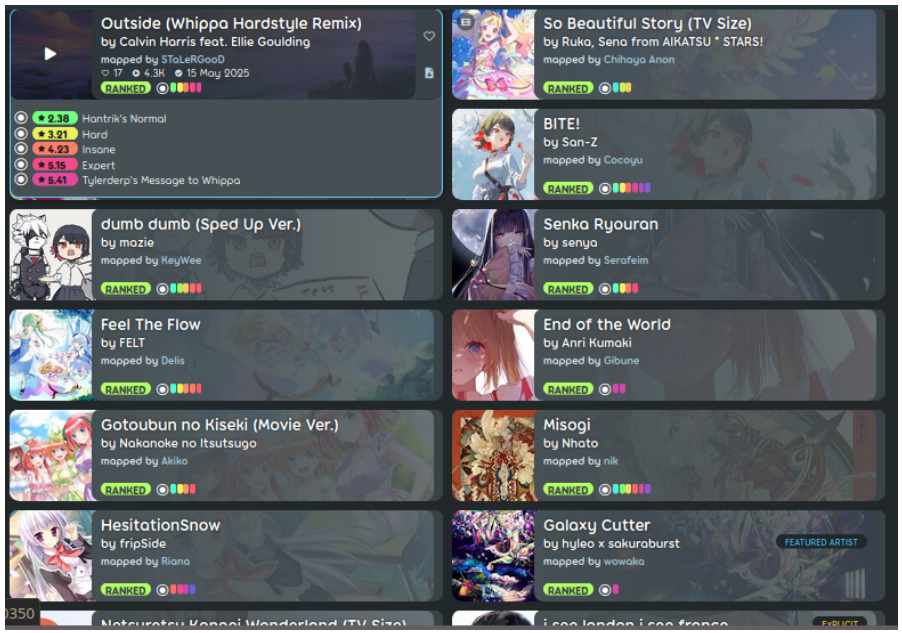
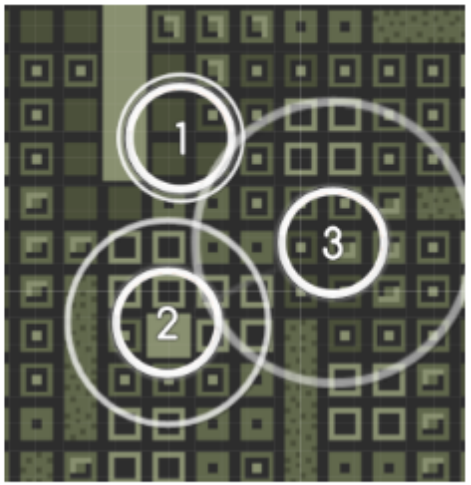
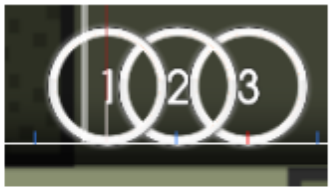
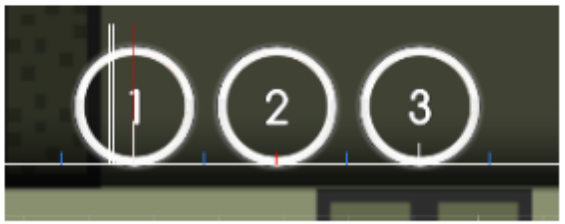


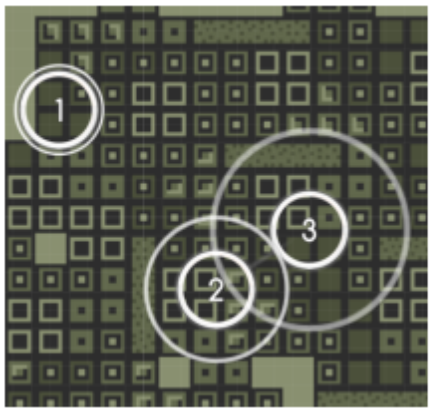
Figure 1: Liste de nouvelles beatmaps

Exemples de cercles



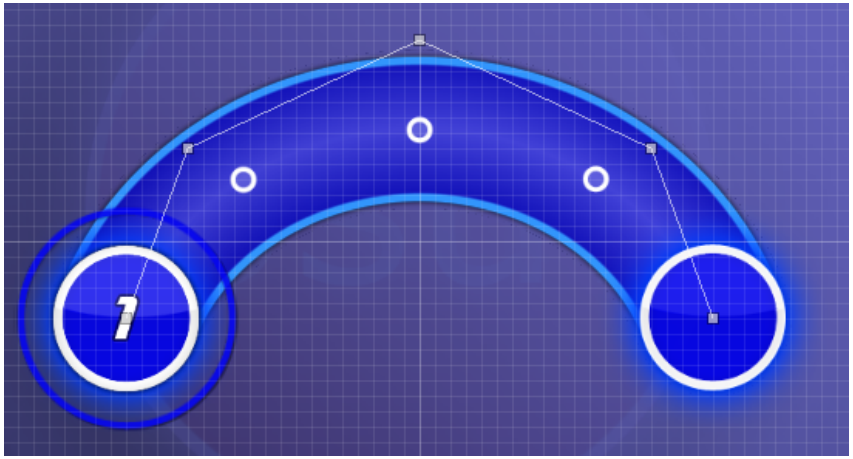


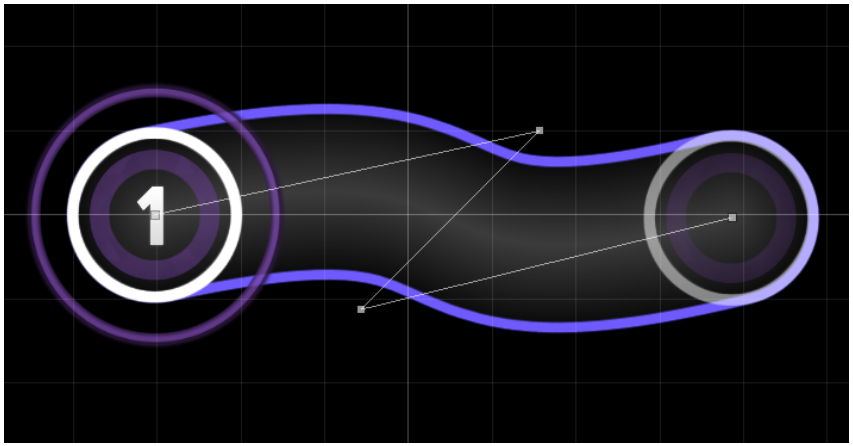




- **All circles and slider heads should be snapped to distinct sounds in the music.** Adding hit objects where there is no musical cue to justify them can result in unfitting rhythms.

Exemples de sliders





- **Hit objects must never be off-screen in 4:3 aspect ratios.** Hit objects that are even partially off-screen can create reading difficulties. Test play your beatmap to confirm this.

- **Every slider must have a clear and visible path of movement to follow from start to end.**

Formulation du problème

Nous nous proposons de **créer deux programmes** permettant **au mieux**, à partir d'un **fichier audio** donné, de **construire un niveau pour osu!**.

Section 2

Partie I : Analyse de musique

Approche en deux temps

Spécifications

Entrée : un fichier audio (format quelconque)
Sortie : des données relatives à la musique permettant le placement des notes :
`(double | (double * double)) list`

Processus retenu ici

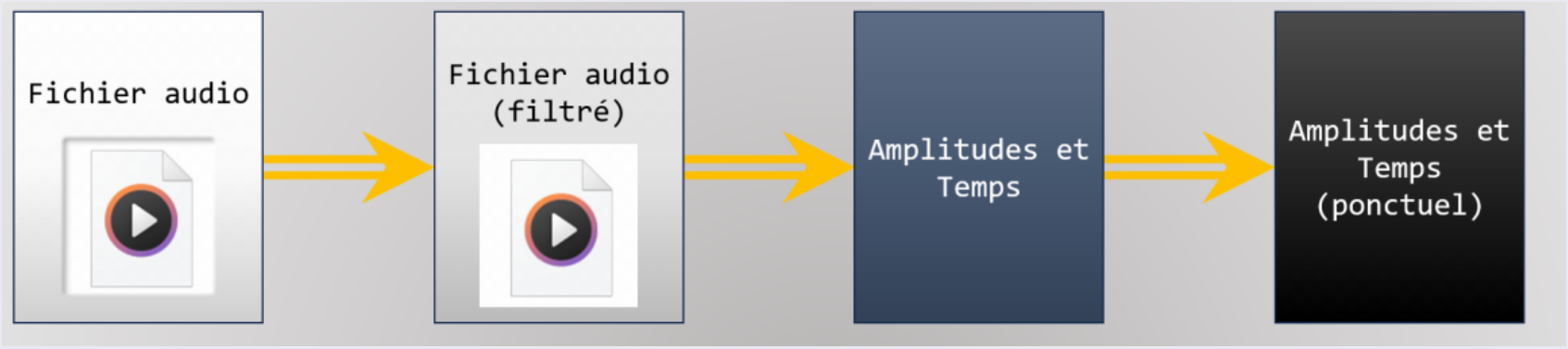
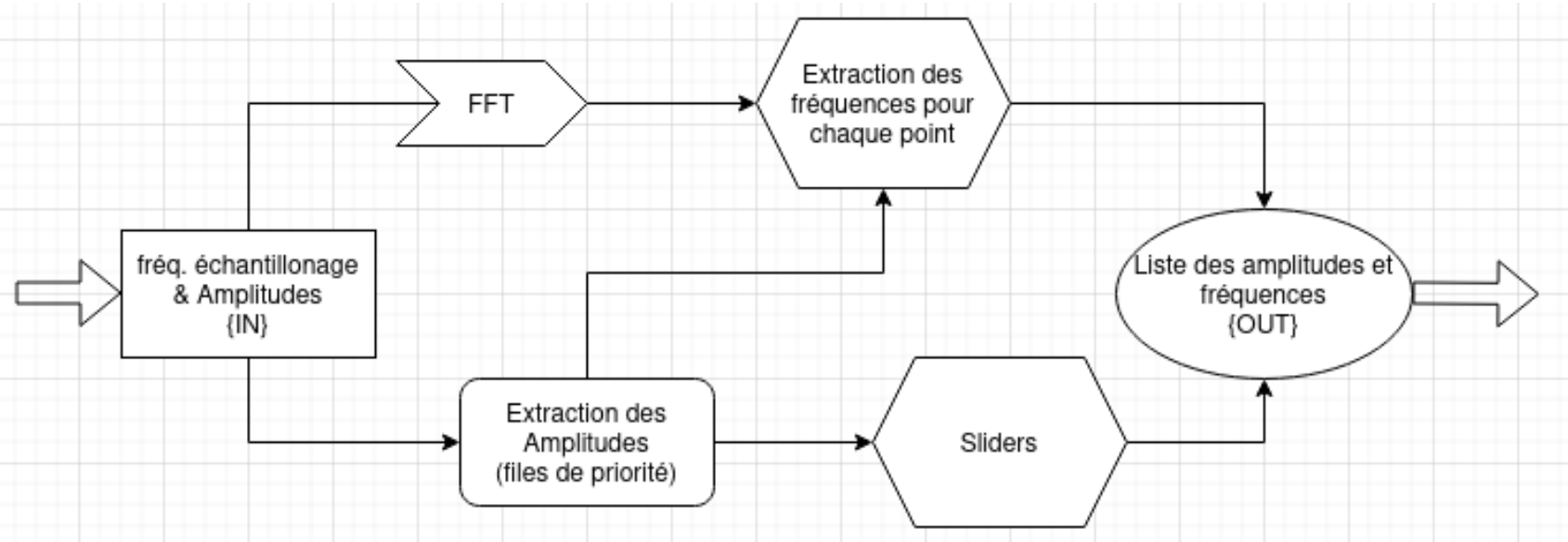


Schéma du processus



Filtres physiques & Transformée de Fourier

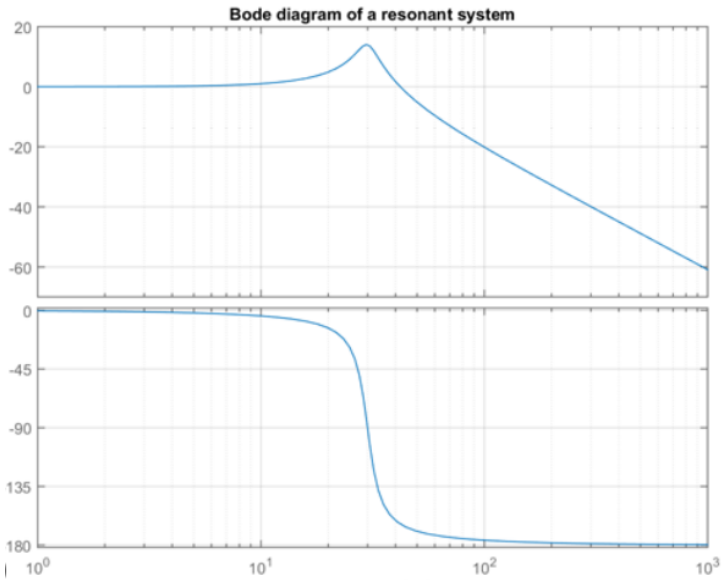


Figure 2: Un exemple de filtre

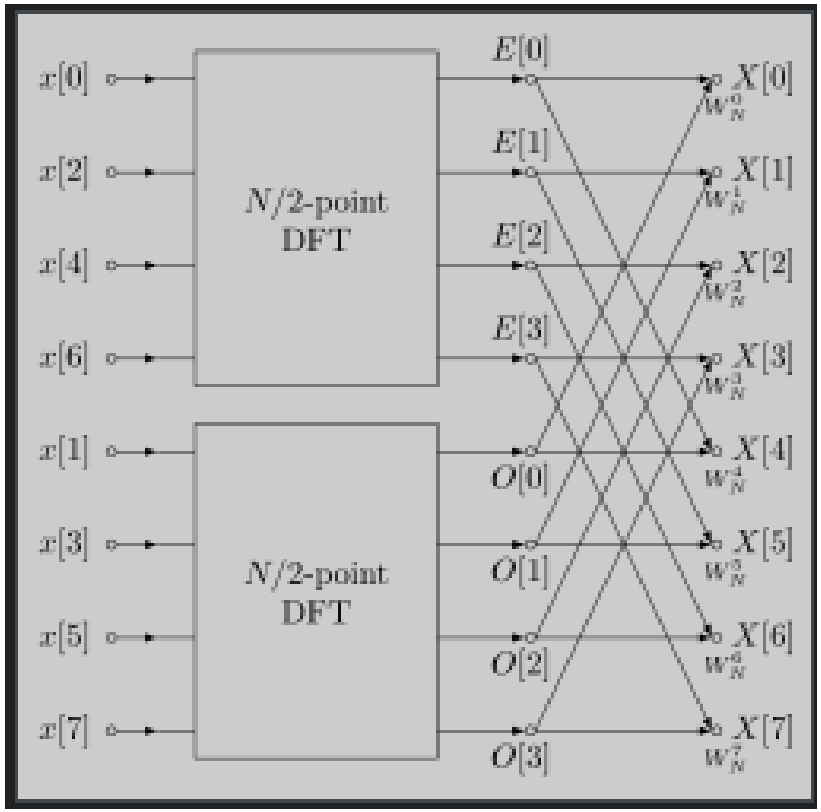
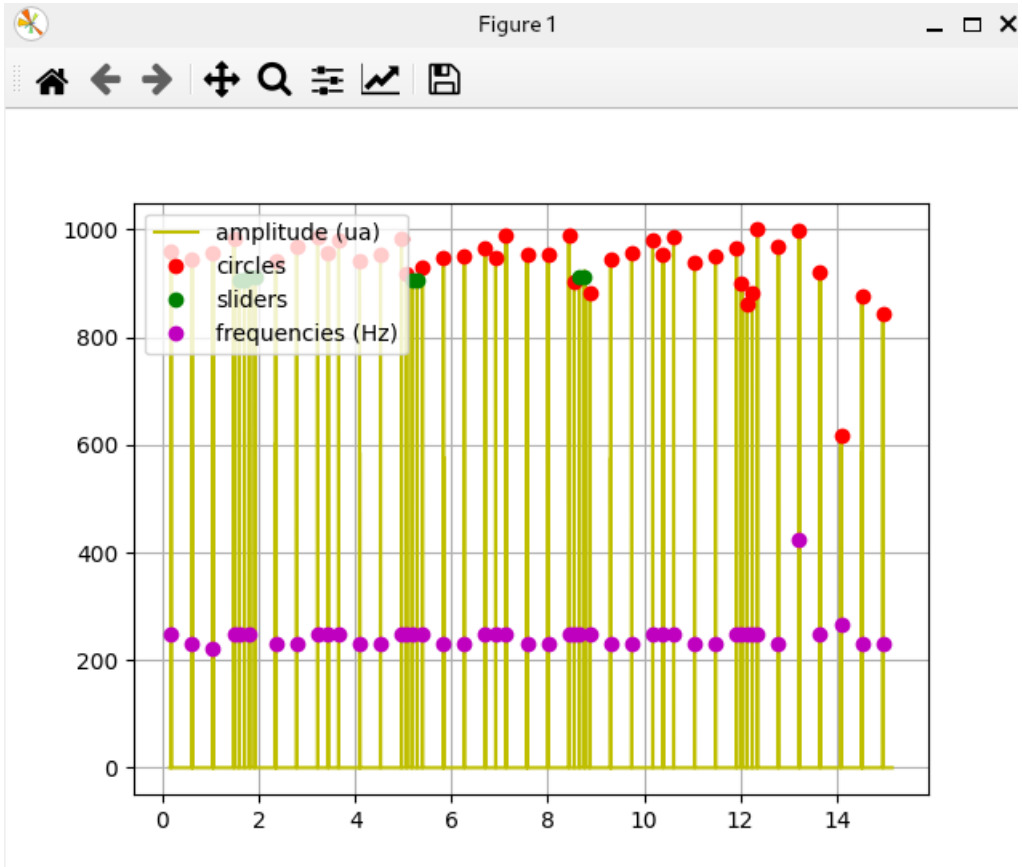


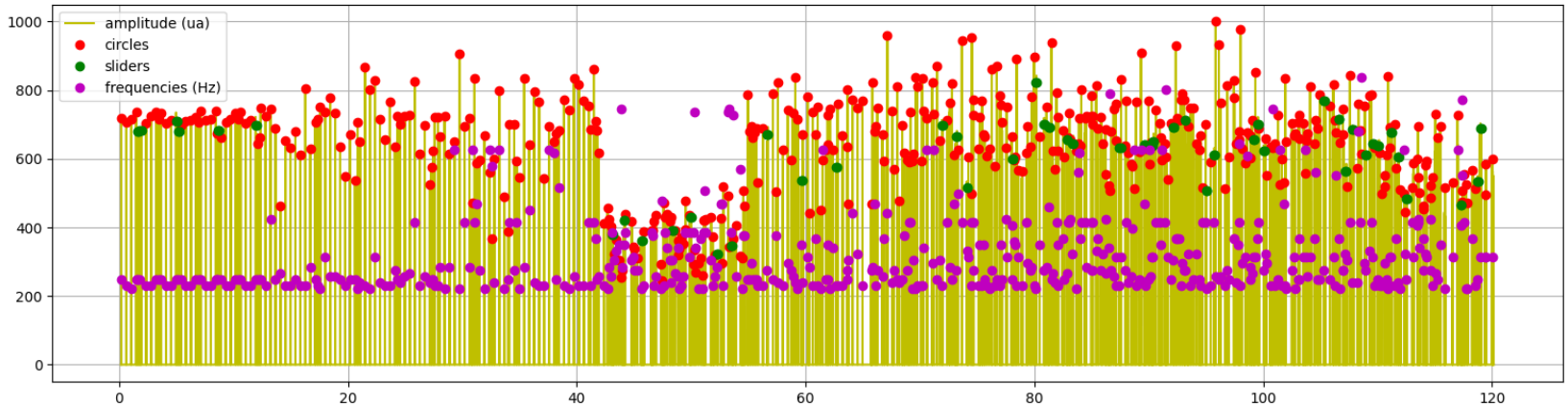
Figure 3: Illustration de la FFT

Résultats



Résultat de l'extraction sur une musique (Bad Apple) pendant 15s

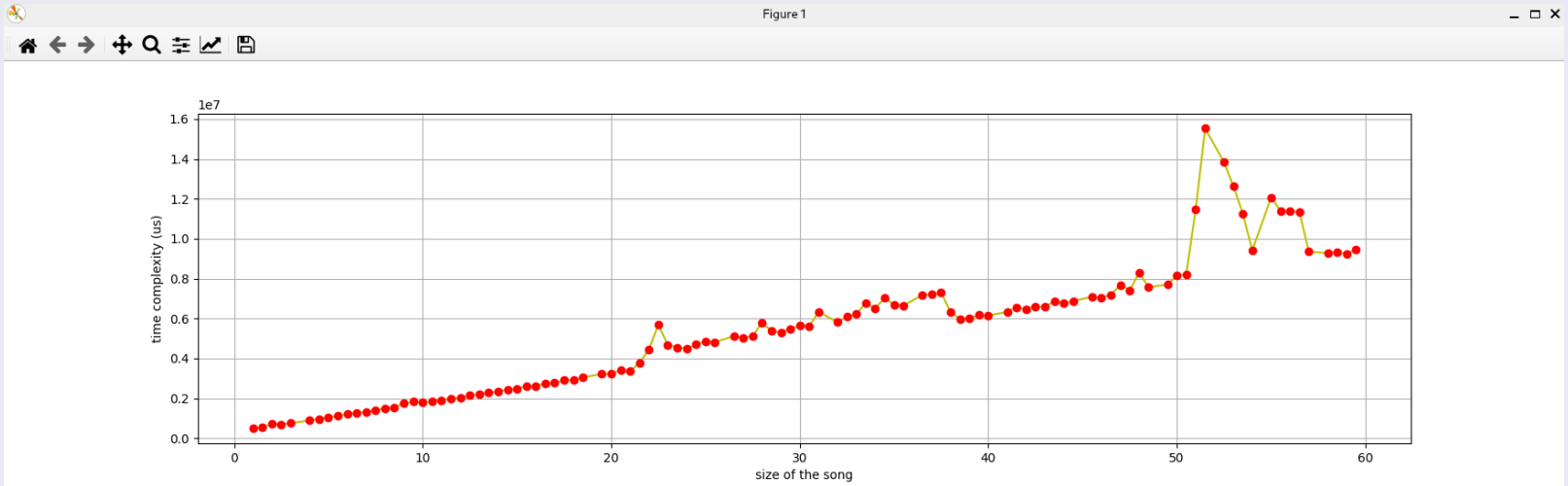
Résultats



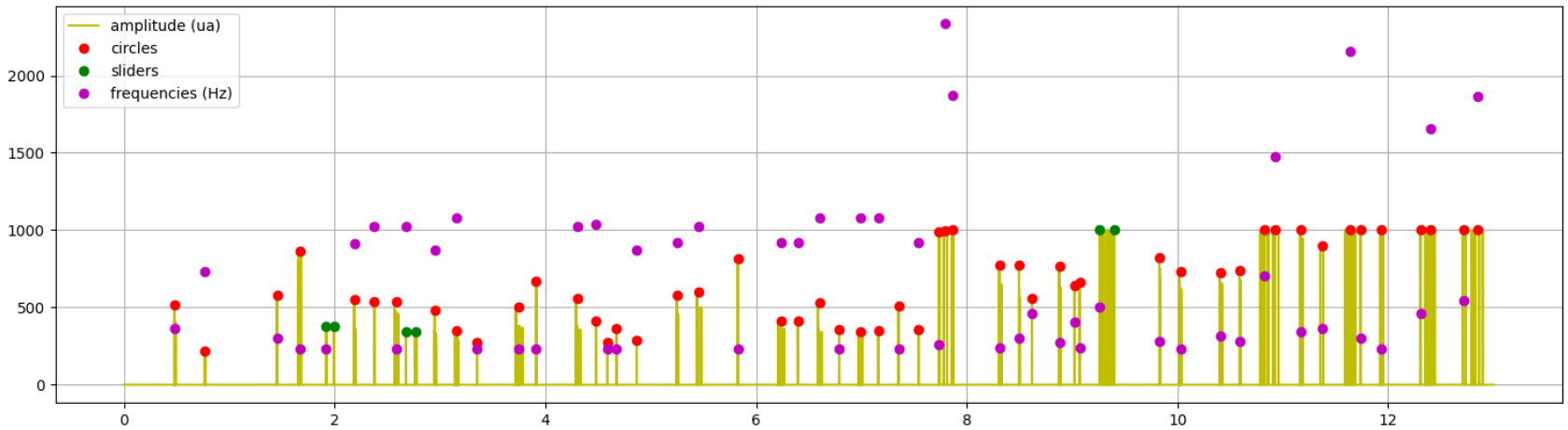
Résultat de l'extraction sur la même musique pendant 120s

Résultats

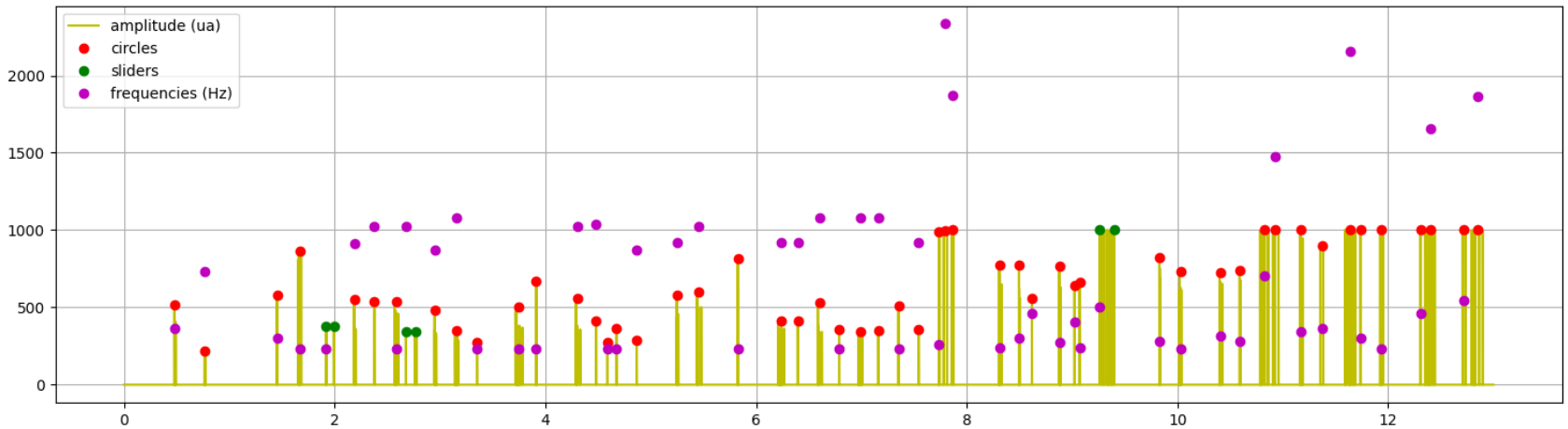
Complexité



Limites, saturation et améliorations

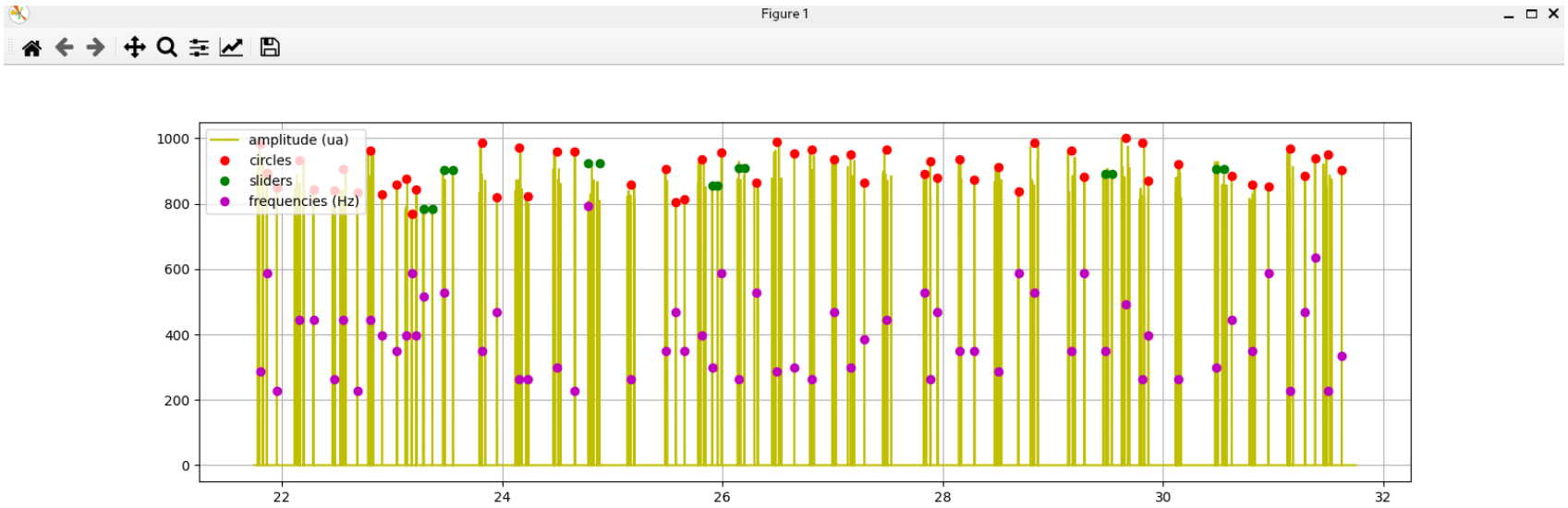


Limites, saturation et améliorations

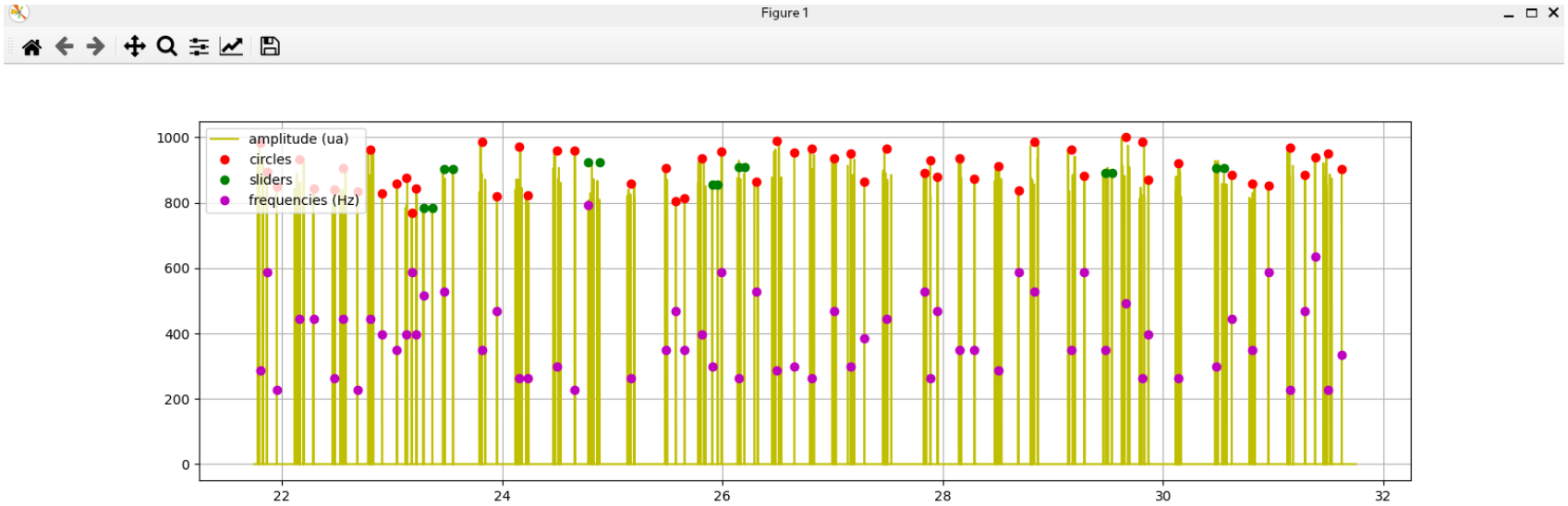


les fréquences ne sont pas correctement détéctées ici (car trop d'harmoniques)

Limites, saturation et améliorations



Limites, saturation et améliorations



une musique "dense" fait que la méthode d'extraction des amplitudes n'est pas précise

Section 3

Partie II : placement spatial

Section 4

Annexe : code Python

librairies utilisées ici

```
1   from math import *
2   import numpy as np
3   import scipy as scp
4   from scipy.io import wavfile
5   import matplotlib.pyplot as plt
6   import subprocess
7   import heapq
8   from pathlib import Path
9   from time import sleep
10  import datetime
```

quelques fonctions utiles

```python
def is_data_stereo(raw_global_data:list) -> bool:
    """
    self-explainatory
    """
    try:
        assert(raw_global_data[0][0])
    except IndexError:
        return False
    except AssertionError:
        return True
    return True

def dist_to_integer(x):
    ent = np.floor(x)
    if(ent < 0.5):
        return ent
    else:
        return (1-ent)

def is_note_within(fr1, fr2):
    if(fr1 > fr2):
        return (fr1/fr2 <= NOTE_DIST or dist_to_integer(fr1/fr2) >= OCTAVE_DIST) # same tone or octave
    else:
        return (fr2/fr1 <= NOTE_DIST or dist_to_integer(fr2/fr1) >= OCTAVE_DIST)
```

`parse music` : extraction de la liste des amplitudes et du "sample rate"

```
    # extracting data from cropped song
    sample_rate, raw_song_data = wavfile.read(song_name)
    blit = int(sample_rate*segsize) # Te

    song_data = [0 for i in range(len(raw_song_data))]

    id_start = int(offset*sample_rate)
    id_end = min(len(raw_song_data), int((offset+songlen)*sample_rate))

    a = 0
    if(is_data_stereo(raw_song_data)):
        print("Converting to mono...")
        for x in range(id_start, id_end):
            song_data[x] = raw_song_data[x][0]/2 + raw_song_data[x][1]/2

            if(x % (int(len(raw_song_data)/100)) == 0):
                print(a, "/ 100")
                a += 1
    else:
        song_data = raw_song_data

    print("\nSampleRate : ", sample_rate)
    print("SegSize : ", blit)

    # calculate the frequencies associated to the FFTs
    pfreq = scp.fft.rfftfreq(blit, 1/sample_rate)

    # left boundary of segment to crop
    current_time = offset

    # list of FFTs
    fft_list = []
    fft_list_untouched = []

    # number of samples
    k = 0
```

```
    # number of samples
    k = 0

    print("Retrieving freqs from", offset, "to", songlen+offset, "...")
    while(current_time < songlen+offset-segsize):
        # index corresponding to left boundary
        left_id = int(current_time*sample_rate)

        # index corresponding to right boundary
        right_id = int((current_time+segsize)*sample_rate)

        # calculate the fft, append it to fft_list
        pff = scp.fft.rfft(song_data[int(current_time*sample_rate):int(sample_rate*(current_time+segsize))])
        fft_list.append(pff)
        fft_list_untouched.append([ee for ee in pff])

        # just to avoid what causes 0.1 + 0.1 == 0.2 to be False
        k += 1
        current_time = offset + k*segsize
        #print(current_time)

    print("\n\nSegSize :", segsize, "\nFFT :", len(fft_list), "\nFFT[0] :", len(fft_list[0]), "\npfreq :", len(pfreq), "\n\n")
```

`cleaning` : retirer les fréquences trop basses et hautes

```
100     # ---------------------------------------- Clean song ----------------------
101     pfreq_minid = 0
102     pfreq_maxid = len(pfreq) -1
103     while(pfreq[pfreq_minid] < minfreq):
104         for t in range(len(fft_list)):
105             fft_list[t][pfreq_minid] = 0+0j
106         pfreq_minid += 1
107
108     while(pfreq[pfreq_maxid] > maxfreq):
109         for t in range(len(fft_list)):
110             fft_list[t][pfreq_maxid] = 0+0j
111         pfreq_maxid -= 1
112
113     new_times = []
114     new_freqs = []
115     new_ampls = []
116     new_kept = []
117
118     # i = time, j = freq
119     for i in range(len(fft_list)):
120         #returns a list of couples [id, value]
121         elements = heapq.nlargest(count, enumerate(fft_list[i]), key=lambda x: x[1])
122
123         for idx in range(len(elements)):
124             if(elements[idx][0] < len(pfreq)):
125                 new_times.append(offset + i*segsize)
126                 new_freqs.append(pfreq[elements[idx][0]])
127                 new_ampls.append(fft_list[i][elements[idx][0]])
128
```

`get amp distribution` : usage de files de priorité pour extraire les maxima

```python
    # ---------------------------------------- Get amp distribution ---------------------------------------- #
    new_new_amps = [0 for i in range(int(sample_rate*songlen))]
    new_new_t = [offset + i/sample_rate for i in range(int(sample_rate*songlen))]

    amp_ct = 0
    incr_a = segsize*4
    len_seg_a = int(sample_rate*incr_a)
    count_a = len_seg_a//1000
    left_0 = int(sample_rate*(amp_ct+offset))
    while(amp_ct < songlen-segsize):
        left = int(sample_rate*(amp_ct+offset))
        right = int(sample_rate*(amp_ct+offset + incr_a))

        #returns a list of couples [id, value]
        elements = heapq.nlargest(count_a, enumerate([song_data[i] for i in range(left, right)]), key=lambda x: x[1])

        amp_ct += incr_a

        for idx in range(len(elements)):
            new_new_amps[elements[idx][0]+left-left_0] = song_data[left+elements[idx][0]]

    mmxx = max(new_new_amps)
    new_new_amps = [nnw*1000/mmxx for nnw in new_new_amps]

    # localize peaks
    left_id = 0
    right_id = 0
    a_ampl = 0
    in_seg = False
    time_d = 0.035
    cur_t = 0

    last_t = -10.0

    locs = []   # amplitudes
    loct = []   # times
    for i in range(len(new_new_amps)):
        if(new_new_amps[i] > 100):
            if(not in_seg):
                in_seg = True
                left_id = i
            right_id = i
            a_ampl = max(a_ampl, new_new_amps[i])
            cur_t = 0
        else:
            cur_t += 1/sample_rate
            if(in_seg and cur_t >= time_d):
                in_seg = False
                delta_t = (right_id - left_id)/sample_rate
                if(np.abs(left_id/sample_rate - last_t) >= 0.01):            # these notes are less than 10ms apart !
                    last_t = right_id/sample_rate
                    if(delta_t < segsize*1.1):
                        locs.append(a_ampl)
                        loct.append((left_id + right_id)/(2*sample_rate) + offset)
                    else:
                        locs.append(a_ampl)
                        loct.append([left_id/sample_rate + offset, right_id/sample_rate + offset])

                a_ampl = 0
```

`get frequency distribution` : fréquence maximale pour chaque point

```
189     # ----------------------------------------- Compute freqs ------------------------
190
191     ssize_0 = segsize/3
192     locf = [] # frequencies
193     for k in range(len(locs)):
194         ktime = 0
195         ssize = ssize_0
196         if(type(loct[k]) == float):             # circle
197             ktime = loct[k]
198         else:                                   # slider
199             ktime = (loct[k][1]+loct[k][0])/2
200             ssize = max((loct[k][1]-loct[k][0])/2, ssize_0)
201
202         left_id = max(0, int((ktime-ssize/2)*sample_rate))
203
204         right_id = min(int((ktime+ssize/2)*sample_rate), len(song_data))
205
206         # calculate the fft
207         pff = scp.fft.rfft(song_data[left_id:right_id])
208
209         fmax = pfreq[0]
210         fampmax = 0
211         for i in range(1, len(pff)):
212             if(pfreq[i] > minfreq and pfreq[i] < maxfreq and fampmax < np.abs(pff[i])):
213                 fmax = pfreq[i]
214                 fampmax = np.abs(pff[i])
215
216         locf.append(fmax)
```

`get sliders` : fusionner les notes trop proches en des 'sliders'

```
218     # ---------------------------------------- Merge ---------------------------------------- #
219
220     k = 0
221     while(k < len(locs)):
222         delta_t = 0
223         if(type(loct[k]) == float):
224             delta_t += loct[k]
225         else:
226             delta_t += (loct[k][0] + loct[k][1])/2
227
228         if(type(loct[k-1]) == float):
229             delta_t -= loct[k-1]
230         else:
231             delta_t -= (loct[k-1][0] + loct[k-1][1])/2
232         if(k > 0 and np.abs(delta_t) < segsize and np.abs(locs[k] - locs[k-1]) < 50 and is_note_within(locf[k], locf[k-1])):
233             loct[k-1] = [loct[k-1], loct[k]]
234             locs[k-1] = (locs[k-1] + locs[k])/2
235             loct[k] = -1
236             locs[k] = -1
237             locf[k] = -1
238             loct.remove(-1)
239             locs.remove(-1)
240             locf.remove(-1)
241         k += 1
242
```

draw : afficher les résultats

```
244     # ---------------------------------------- Plot ---------------------------------------- #
245
246     plt_loct_all = []
247     plt_loct = []
248     plt_locs = []
249     plt_slidt = []
250     plt_slids = []
251     for i in range(len(loct)):
252         if(type(loct[i]) == float):
253             plt_loct_all.append(loct[i])
254             plt_loct.append(loct[i])
255             plt_locs.append(locs[i])
256         else:
257             plt_loct_all.append(loct[i][0])
258             plt_slidt.append(loct[i][0])
259             plt_slidt.append(loct[i][1])
260             plt_slids.append(locs[i])
261             plt_slids.append(locs[i])
262
263     plt.plot(new_new_t, new_new_amps, "y-", label="amplitude (ua)")
264     plt.plot(plt_loct, plt_locs, "ro", label="circles")
265     plt.plot(plt_slidt, plt_slids, "go", label="sliders")
266     plt.plot(plt_loct_all, locf, "mo", label="frequencies (Hz)")
267     plt.legend(loc="upper left")
268
269     '''plt.plot(new_times, new_freqs)
270     plt.plot(new_times, [elt*1000/mx for elt in new_ampls])
271     plt.plot(new_times, new_kept, "bo")'''
272     plt.grid()
273     plt.show()
274
275     # ---------------------------------------- Write ---------------------------------------- #
276
277
278     f = open("result_bad_apple[90].txt", "w")
279     f.write("Song name : " + song_name + "\n")
280     f.write("Start : " + str(offset) + "\n")
281     f.write("End   : " + str(offset+songlen) + "\n\n")
282
283     f.write("Hit Objects : \n")
284     for ct in loct:
285         f.write(str(ct))
286         f.write("\n")
287
288     f.close()
289
290
291     return (loct, locs)
```

quelques données de test

```
'''
# c-type
SONG_LEN = 7
OFFSET = 0.042
BPM = 149.3
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/ctype.mp3")
'''
'''
# tetris_2
SONG_LEN = 14
OFFSET = 0
BPM = 157
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/tetris_2.wav")
'''
'''
# test
SONG_LEN = 1
OFFSET = 0
BPM = 240
SEGSIZE = 1/(BPM/60)
'''
'''
# gmtn
SONG_LEN = 5
OFFSET = 1.652
BPM = 155
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/furioso melodia.mp3")
'''
'''
# E
SONG_LEN = 15
OFFSET = 2.641
BPM = 155
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/rushe.mp3")
'''
'''
# Tsubaki
SONG_LEN = 20
OFFSET = 35.659
BPM = 199
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/TSUBAKI.mp3")
'''
```

```
'''
# Owen 1/2
SONG_LEN = 20
OFFSET = 1.008
BPM = 157
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/owen(157.00024-1008).mp3")
'''
'''
# Owen 2/2
SONG_LEN = 7
OFFSET = 25.466
BPM = 157
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/owen(157.00024-1008).mp3")
'''

# death
SONG_LEN = 10
OFFSET = 21.750
BPM = 180
SEGSIZE = 1/(BPM/60)
wavved_song = convert_to_wav("songs/Night of Knights.mp3")

'''
# Bad apple
SONG_LEN = 15
OFFSET = 0.152
BPM = 138
SEGSIZE = 1/(BPM/60)
#wavved_song = convert_to_wav("songs/Bad apple (138-152).mp3")
wavved_song = convert_to_wav("songs/Bad apple (138-152)[filtered].wav")
'''
'''
# Freedom dive
SONG_LEN = 7
OFFSET = 1.058
BPM = 222.22
SEGSIZE = 1/(BPM/60)
#wavved_song = convert_to_wav("songs/Freedom Dive (222.22-1058).mp3")
wavved_song = convert_to_wav("songs/Freedom Dive (222.22-1058)[filtered].wav")
'''
'''
# Mevalogania
SONG_LEN = 7
OFFSET = 7.984
BPM = 240
SEGSIZE = 1/(BPM/60)
#wavved_song = convresult_bad_appleert_to_wav("songs/Megalovania(240-7984).mp3")
```

interface

```
408  '''
409  SONG_LEN = 0 # length of the song, in seconds
410  OFFSET = 0 # offset of the 1st note (aka time offset of the first red bar), in seconds
411  BPM = 0 # BPM
412  wavved_song = convert_to_wav("insert_song_name_here.wav")
413  '''
414  # Do not touch
415  DIVIDER = 4 # note divider
416  SEGSIZE = 1/(BPM/60)
417  NOTE_DIST = (2**(1/4))
418  OCTAVE_DIST = 0.05
419
420  # keep_highest(song_name, offset, songlen, segsize, count, output_name, minfreq=110, maxfreq=5000, ampthr=250):
421  (loct, locs) = keep_highest(wavved_song, OFFSET, SONG_LEN, SEGSIZE/DIVIDER, 1, "Zblit.wav", minfreq=220, maxfreq=3000, ampthr=800)
422
423  '''
424  minfreq and maxfred are thresholds for frequency analysts (anything outside of [minfreq, maxfreq] will not be accounted for)
425  ampthr is a threshold for amplitude (arbitrary unit)
426  '''
427
```