Music Information Retrieval (MIR)

Academic work developed in the course "Signal processing" with the objective of developing an MIR (Music Information Retrieval) algorithm to determine the BPM of a song through an excerpt of a song.

For this we use the website "tunebat.com" as a BPM reference for the respective songs and the audacity software for recording the songs and spectral analysis of them.

The algorithm was developed in MATLAB to filter out certain frequencies (if necessary) and providing as output a series of graphs to analyse the filter and the final signal.

Developed by:

Henrique Machado

- https://www.linkedin.com/in/henrique-machado-757500109/

Rui Fernandes

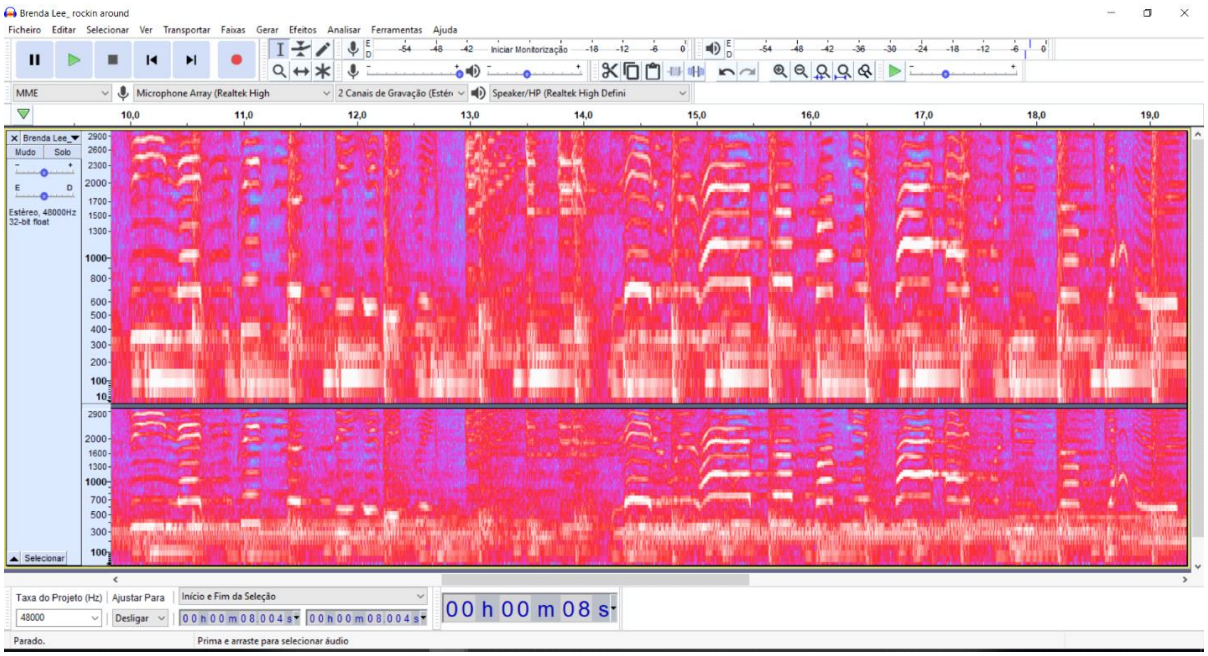
- https://www.linkedin.com/in/ruipcf/

Audacity

Spectrogram view with melody scale(MEL):

- Analyze which part of the song is periodic;
- Record this part and export it for later processing in MATLAB.

Music example ”Rocking around”



MATLAB Code

Using the function “get_BPMs”, made by us for reading the music we define as function parameters:

- the name of the file which as the music;
- the cut-off frequencies interval "[Fmin Fmax]" for the filter (if necessary);
- And if we want to show all plots(1) or just the autocorrelation plot(0).

As an output we receive the filtered signal, the sampling frequency and the respective BPM of the analyzed music.

```
clear; clc; close all;

% 1st music
fprintf('John Paesano On the Case: expected BPMs: 170\n');
[output_s1, Fs1, bpm1] = get_BPMs('john paesano_on the case.wav',0,0);

% 2nd music
fprintf('Nicky Jam, Will Smith, Era Istrefi_Live It Up : expected BPMs: 144\n');
[output_s2, Fs2, bpm2] = get_BPMs('nicky jam_live it up.wav',0,0);

% 3rd music
fprintf('Kodomo Concept 16 : expected BPMs: 94\n');
[output_s3, Fs3, bpm3] = get_BPMs('kodomo_concept 16.wav',0,0);

% 4th music
fprintf('Brenda Lee Rockin Around The Christmas Tree: expected BPMs: 67\n');
[output_s4, Fs4, bpm4] = get_BPMs('Brenda Lee_ rockin around.wav', [150 300],0);
```

When Reading the signal must be recorded at 48 kHz because the code is made to resample that signal down (if the Fs(sample frequência is lower the BPM's value won't be accurate). The objective of this work is get the BPM count for the inputed songs, and the instruments that define that value are usually instruments that produce bass sounds (below 300 Hz), we chose to resize the frequency to 8 kHz optimizing our bpm calculation program and increasing the processing speed.

In these first 3 sections of code we calculate the number of points and the time vector of the song.

```
function [output_signal, Fs, bpm] = get_BPMs(music, wn, showPlot)

[data, Fs] = audioread(music);              % read music and get data and sampling frequency
signal = data(:,2);                         % choose the data channel

%% resampling
ds = 6;
signal = resample(signal, 1, ds);           % resample de 1/ds change from 48 kHz to 8 kHz
Fs = Fs/ds;                                 % new sampling frequency

%% number of points and time vector
N = length(signal);                         % number of points
t = (0 : N-1) / Fs;                         % time vector
```

The filter chosen by us was a bandpass however it could be a lowpass. As we saw earlier, the instruments that produce bpm are instruments that produce bass sounds (below 300 Hz). Therefore, we dimension our filter to have a maximum cutoff frequency of 300 Hz.

We chose a Butterworth filter of order 2 because it allows to eliminate the unwanted frequencies caused by other instruments that produce higher frequencies, without having a great impact on the performance of the calculation since it is of low order.

The use of this filter offers us as characteristics an accentuated rolloff providing a more defined limit between the pass band and the rejected band and also a smooth pass band.

```
%% filter design and signal filtering
if(wn ~= 0)
    order = 2;
    fnyquist = Fs/2;
    wn = wn / fnyquist;                    % normalized cutoff frequency (fnyquist)
    [b,a] = butter(order,wn);              % IIR butterworth filter
    output_signal = filtfilt(b,a,signal);  % applying the filter to original signal
else
    output_signal = signal;                % signal not filtered
end
```

Autocorrelation allows us to see if the signal has a temporal structure.

```
%% autocorrelation
% compare the signal with it self with differents time shifts
 % this allows us to understand if the signal has a temporal structure
  % if it has we can determine the signal frequency
[r,lags] = xcorr(output_signal, 'coeff');
```

This function searches for the peaks in the autocorrelation result having the minimum distance (period) between peaks limit and also limiting the minimum amplitude of the signals to be processed in order to increase the accuracy and speed of the calculation.

```
%% find spikes on the correlation graph
% 60%180 = 0.33 sec represents the minimum period of time to be analised
  % because a low period will result in higher bpm's
% 'MinPeakHeight' represents the minimum value of autocorrelation (4%)
  % for which we want to count spikes
[spikes,loc] = findpeaks(r, 'MinPeakDistance', Fs*0.33, 'MinPeakHeight', 0.04);
```

In this section we calculate the distances between the various peaks obtained by the function 'FindPeaks'.

Then we calculate the frequency that corresponds to the number of beats per second and multiply it by 60 to obtain the value in BPM.

```
%% obtain BPM
bpm = zeros(length(loc)-1,1);               % inicialize array of zeros
for i=1:length(loc)-1
    period = loc(i+1)/Fs - loc(i)/Fs;       % get the distance between spikes over time = period
    bpm(i) = (1/period) * 60;               % convert time in bpm's = frequency
end
```

Through the function 'histcounts' we know how many BPM values are between the ranges of values [60.80], [80 100] … [160.180], thus realizing which range contains more bpm values, this being the most accurate to obtain results.

Knowing the most accurate interval, we use the bpm values of that interval and calculate the average bpm, this being our final result.

```
%% find the most relevant interval of bpm
[value,edges] = histcounts(bpm, 60:20:180);     % return edges (1st value of the interval range) values represents how much bmp are in each interval
[MaxBPM, iMaxBPM] = sort(value, 'descend');     % order descend by number of 'value' indexing the position of MaxBPM on array [value,edges]

% select the most relevant interval of bpms (the biggest value)
bestIntervalBPM = bpm >= edges(iMaxBPM(1)) & bpm <= edges(iMaxBPM(1)+1); %foreach bpm value we compare with the minimum and maximum value
                                                                         %of the interval (edges) and (edges+1) that has the biggest count of 'value' of bpm
                                                                         %if the condition is true the value of the position of 'bestIntervalBPM' is 1 else is 0
                                                                         %this array will be used to calculate the average bpm on the interval with the higher bpm count

imr = (MaxBPM(1)/sum(value))*100;           % get the accuracy of the most relevant interval
imr2 = (MaxBPM(2)/sum(value))*100;          % get the accuracy of the second most relevant interval
```

In this section we present the final results, the two intervals with more precision and the average number of BPM of the music. The greater the accuracy of the interval, the more bpm will be.

```
%% show results
fprintf('Interval: [%d %d]: %.1f%% Accuracy \n', edges(iMaxBPM(1)), edges(iMaxBPM(1)+1), imr);
fprintf('Interval: [%d %d]: %.1f%% Accuracy \n', edges(iMaxBPM(2)), edges(iMaxBPM(2)+1), imr2);
fprintf('Average BPM: %.1f \n\n', mean(bpm(bestIntervalBPM)));
```