MAE Final Work

A Tool to Recognize Music through Colours: MUSAIC

Pablo Domingo

January 8, 2019

Abstract

Human beings are guided mainly by sight. Sight is our primarily way of learning. For this reason, this project tries to understand sound using our sight. Converting sound into Colour.

Links

Musaic Code: https://github.com/paudom/MUSAIC

Musaic Demo: https://youtu.be/Q7Mvt5B0CZU

Contents

Chapter 1

Introduction

Everyone has a favourite group or music genre. But sometimes we discover new music and we don't know the title of the song or the name of the artist. On the present-day many music industries and other services like "Shazam" offer systems that can recognize music using fingerprints. This is usually done using machine learning.

Most of the machine learning techniques used today, are fairly complex to completely understand, included some fingerprints systems. Fingerprints first detect relevant features from songs and then use them to recognise similar patterns. Fingerprinting has demonstrated to work very well. However, we don't exactly know how the music is compared to finally detect the correct music.

Humans are mainly guided by their senses, primarily his vision and hearing. With this idea in mind could be useful to use a system to convert the sounds we hear into images to make sound more intuitive, more visual. This way sounds can be compared more easily by us, and understand how fingerprinting work their way around to compare music.

The conversion from sound to colour is not new. In fact, there are already a lot of papers that explain possible solutions for that, one of them being [2] from where this project is influenced. But apart from achieving this conversion, the addition would be to use those colours to compare music.

The aim of this project is to combine two worlds, these being the music and the data analysis, to develop a tool that allows us to first, convert music into images, and secondly use it to compare visually two songs. Finally analyse this results to evaluate if could be used to recognise music. To accomplish this goal the intention is first extract key components of sound, then transform those into colour properties and finally convert those colours into mosaics that would be our standard format to compare music.

To sum up, this project propose an alternative method to compare music by using coloured images. Depending on the results, then evaluate if this method can be extrapolated and used to recognise music automatically.

Chapter 2

Theoretical Background

Sound and light are very similar in terms of wavelength. Perceptible sound waves to human ears oscillate approximately between 20Hz and 20KHz whereas light waves perceptible to human eyes oscillate between 390THz and 750THz. Knowing that both waves share the same domain (frequency) it is mathematically possible to map the frequency rate of audible band into visible light band. Such a mapping requires first the knowledge of some music characteristics.

2.1 Notes & Octaves in Music

Starting with sound, we could take a piano keyboard as reference. Piano's keyboards are structure with eight octaves, each one containing twelve notes or frequencies. Starting at 27.5Hz, one octave include are all the notes existing between the fundamental (in this case 27.5) and the double so 55Hz. And so on, until the last frequency on the keyboard.

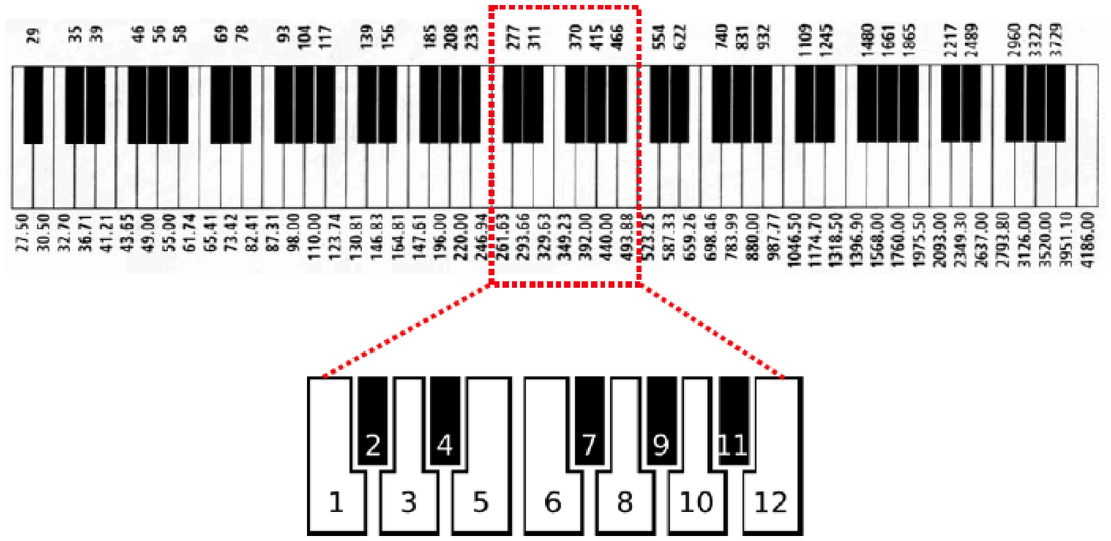


Figure 2.1: Notes and Octaves in a Piano Keyboard

Nonetheless in music, not all instruments are pianos. In fact, greater frequencies than 4186Hz can be reached. However, frequencies over 4186 Hz are considered not tonal and are attributed to hi-hats, and other sounds similar that can be estimated as noise. The same happens with frequencies below 27.5Hz. Apart from not being very noticeable by human ears, this tones are so large that pitch algorithms need a lot of samples. In addition, capturing those frequencies would mean to work with a higher sample rate and in consequence requires more computational power. That's why almost all fingerprints algorithms work with low sample rates. First because doesn't require a lot of data to get sufficient results, and second because over 4186 Hz there are not much consistent information. In the sense that, once over those frequencies, features (like pitch, maximum of energy and others) are difficult to detect with precision and its variance is normally very high. Taking this into account and that mostly all commercial music is sampled with 44100KHz, could be a good option to use a sample rate of 8820 KHz, five times lower than the original.

2.2 Feature Extraction

As we have seen, frequency is a good parameter to take into account. It has a relation with the notes that are played but also can be mapped to colours. For this reason, a reliable feature to extract from the music could be the pitch, or in other words the tone or frequency. But with only the frequency we don't get enough features from the sound. Fingerprints algorithms often use also a power component. In our case, we can use the energy of the signal, as [2] suggest.

2.2.1 Pitch Detection for Music

The detection of pitch in sound is very well know for human voice, but not that much for music. That is why, most of popular methods to detect pitch, such as autocorrelation or the minimum difference function, are discarded for this project. We are more interested in methods that consider frequency as the main ingredient. This is why cepstrum analysis can be used in this case. The primary advantage is that it does not need center clipping.

$$x[n] \to TF\{x\} \to X[k] \to |X[k]| \to \log|X[k]| \to TF^{-1}\{\log|X[k]|\} \to Cx[n] \tag{2.1}$$

Once the coefficients are computed the position where the pitch is located can be found by (2.2)

$$Npitch = argmax\left(Cx[n]\right) \tag{2.2}$$

We have to take into account that Npitch is bounded between Nmin and Nmax that in our case would be (2.3)

$$Nmax = \frac{SampleRate}{F_{minRef} = 27.5}, \quad Nmin = \frac{SampleRate}{F_{MaxRef} = \frac{SampleRate}{2}} \tag{2.3}$$

And finally find the pitch using (2.4)

$$F0 = \frac{SampleRate}{Npitch} \tag{2.4}$$

An example of the cepstrum domain can be seen in Figure (2.2)

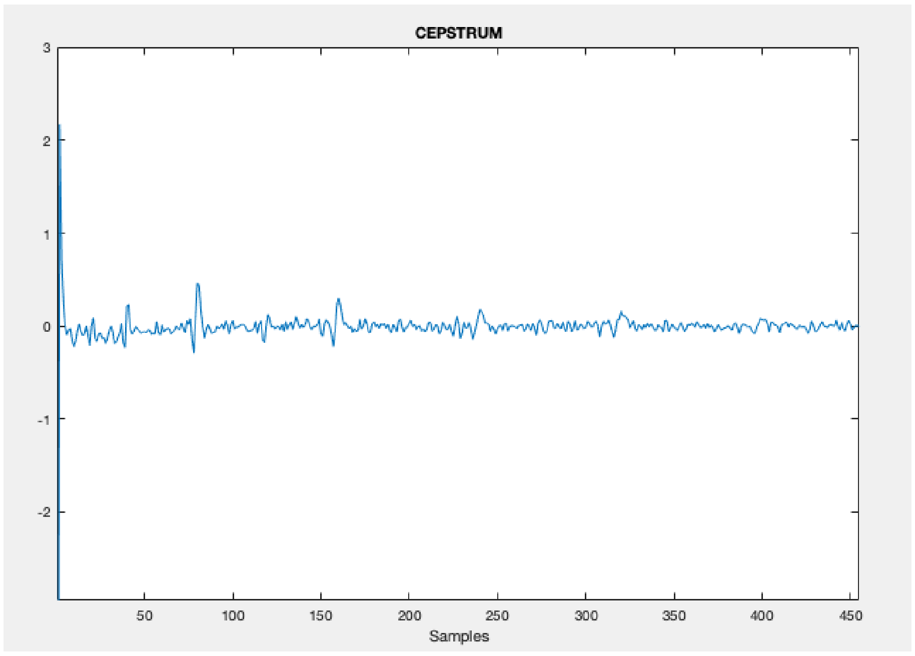


Figure 2.2: Cepstrum Domain

However a problem arises when computing pitch in music. Music rarely is monophonic (this meaning one tone at a time) and is usually polyphonic (multiple tones at a time). The study of pitch for monophonic music is very well studied, but for polyphonic music it is still a field of research. Cepstrum analysis will detect pitch but only the most predominant, or significant, usually being the lowest pitch. To compute polyphonic pitch a possible solution is to detect all the harmonics of the signal using its spectrum.

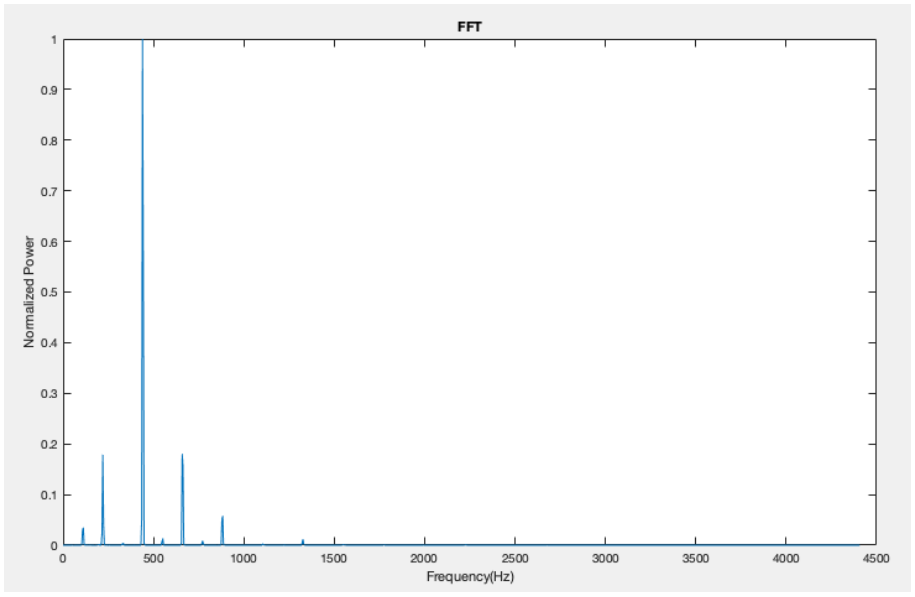


Figure 2.3: Spectrum of a Frame with it's Harmonics

Usually a salience function is used but instead we will take multiple pitches by taking the position and energy of the peaks in the spectrum. Salience function are described on [1]

$$Salience = \sum_{h=1}^{H} \left| X\left[k \cdot h\right] \right| \tag{2.5}$$

Where "H" denotes the number of harmonics considered and "X" is the spectrum of the signal. As each frequency denotes one note, one way of getting the "composed note" is to use a weighted mean by taking the energies of the harmonics, to then get a "composed note".

$$F\theta_{composed} = \sum_{f=0}^{F} w_h F\theta_f \tag{2.6}$$

Where F is the number of pitches detected. Another way could be to get the notes for each pitch separately and when the conversion to colour is effectuated, mix the resulting colours using the energy of the harmonic as a weigh.

2.2.2 Assign Pitch to Music Properties

Once we have the pitch or pitches detected, we would like to determine which note reference that pitch. To do that, the following formulas can be used to determine the octave and note for one pitch. As reference frequency 27.5Hz is taken. Octaves can be compute by (2.7)

$$Octave = \left\lfloor \log_2\left(\frac{F\theta}{F_{ref}}\right) \right\rfloor, \quad F_{ref} = 27.5Hz \tag{2.7}$$

Take into account that octave goes from 0 to 7, that is related to the 8 octaves that exist on a piano keyboard as Figure (2.1) shows. Once we have the octave we can get the note by using 2.8

$$Note = \left\lfloor \frac{\frac{\left(F\theta - 2^{Octave} \cdot F_{ref}\right)}{2^{Octave} \cdot F_{ref}}}{12} \right\rfloor, \quad F_{ref} = 27.5Hz \tag{2.8}$$

It's worth noting that the valid values for the note is between 0 and 11. This is useful to convert each note from the twelve existent in Figure (2.1), to one colour. On the following section we will propose a possible mapping.

2.2.3 Energy Computation

As we mentioned, another feature that could be useful, would be the energy of the signal.

$$Energy = \sum_{n=0}^{N-1} \left| x\left[n\right] \right|^2 \tag{2.9}$$

Because we will need a normalized energy, we would want the energy to be as uniform as possible. The equation (2.9) does not allow the uniformity that we are searching, so instead the following formula is used to compute the energy of a frame.

$$Energy = \sqrt{\sum_{n=0}^{N-1} |x[n]|^2} \tag{2.10}$$

2.3 Colour Space

Colour can be structured with colour spaces. Each one has its definition and depending on the use, ones work better for some applications than others. In this case we need to find a colour space than fits into our needs.

2.3.1 RGB

We could start by the most known colour space, RGB. However this colour space, only specifies the quantity of red, green and blue in an image, and don't have any relation with frequency. There is not a relation between the octave and the quantity of red, green or blue. For this reason, RGB is not a valid colour space to use when converting the features extracted to colour. Nevertheless RGB is a very useful colour space to display images, as it serves in most platforms as a standard. Given this situation we could use RGB to display and save the resulting Mosaics.

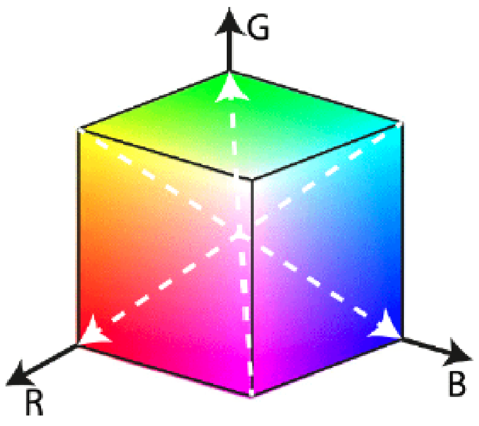


Figure 2.4: RGB Colour Space

2.3.2 HSV

HSV is an acronym for *Hue*, *Saturation* and *Value*. This colour space is more align with the way human vision perceives colour. *Hue* represents the tint of colour and is expressed on degrees. The *Saturation* represents the amount of colour in a colour. If the saturation is low then there is more white in that colour and less tint. Vice versa with more saturation, the colour contains less white, and in consequence more tint. The *Value* indicates how dark or bright a colour is.

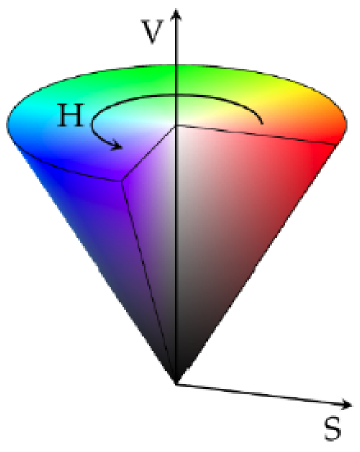


Figure 2.5: HSV Colour Space

Given this description we can propose a mapping from the features that we already have to values in the HSV space. As the Hue is measured in degrees, there are 360º possible tints.

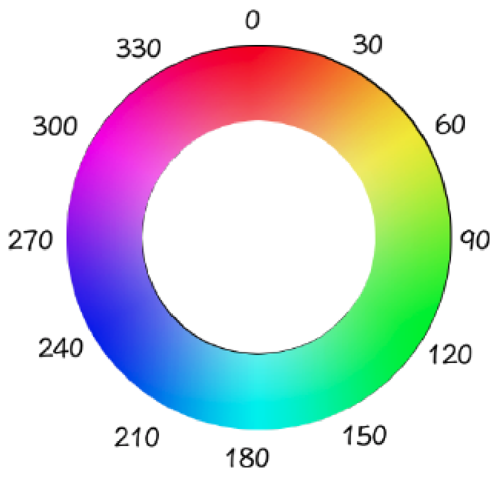


Figure 2.6: Different Values of Hue

As we have twelve notes, we can divide the radial slice in twelve parts and assign each note to a valid value of Hue.

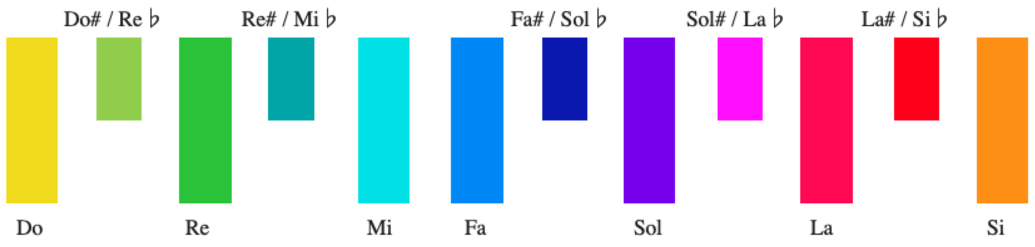


Figure 2.7: Colour assignation to each note

Chapter 3

Implementation

MUSAIC is a Matlab program that allow us to see sound. The program has three main functionalities. These being the following:

(a) Build Musaic

(b) Compare Musaic

(c) Match Musaic

The **Build** option allow us to convert any song into an image, that is stored. In addition, two histograms are shown to help you understand better the resulting image.

The **Compare** option allow us to compare two resulting images. This is useful because, as we will explain, there are different methods that can be used to convert music into colour, so makes sense to be able to compare mosaics of the same song but for different methods.

The **Match** option is the experiment to observe if this method can be used in fingerprinting.

3.1 Program Structure

MUSAIC is structured by folders. There are four folders named:

(a) Mosaics

(b) MusicFiles

(c) SampleChunks

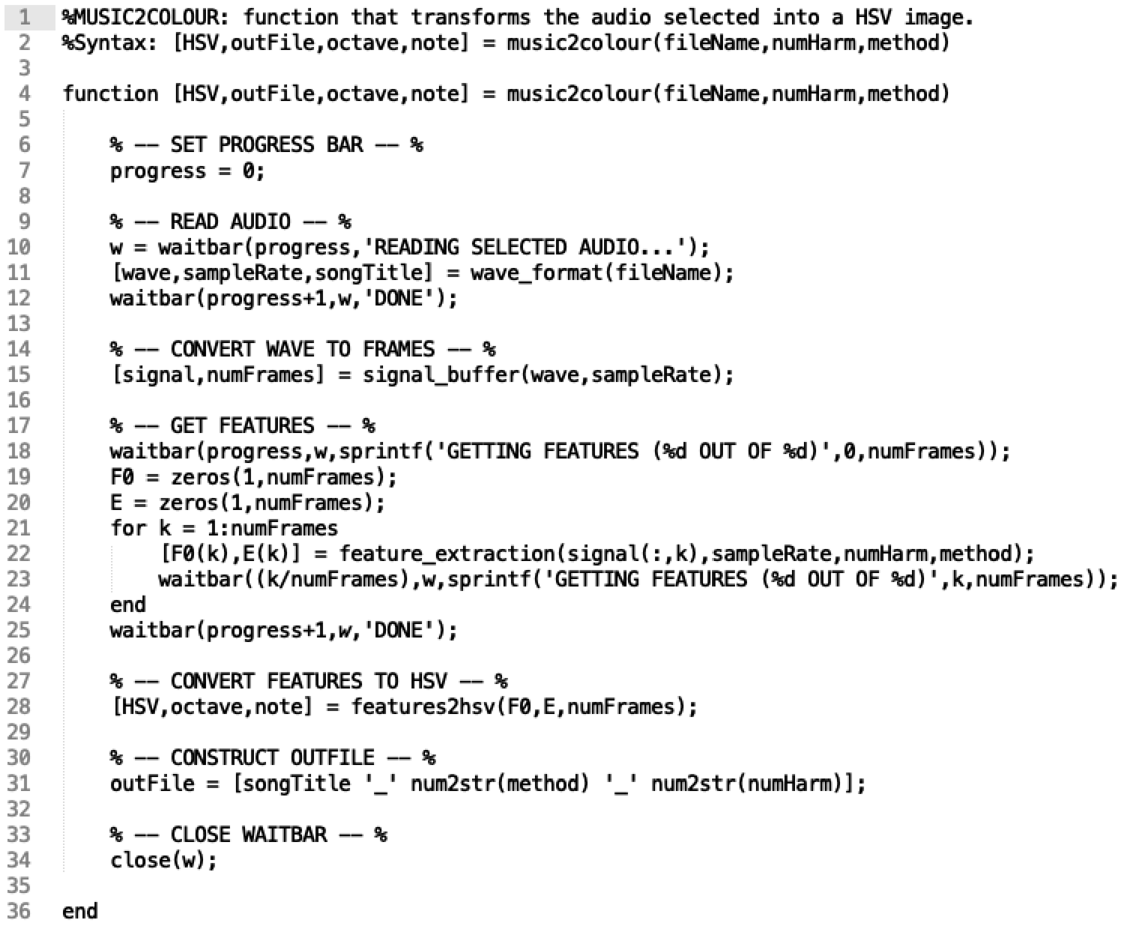
(d) Resources

On **(a)** Images of the mosaics are stored once a song has been converted. The name of the files are constructed by putting the name of the song followed by the method and the number of harmonics used. On **(b)** the music files are stored. By default there are 25 songs already stored. If the user wants to use their own songs, files need to be copied on this folder to be recognized by *MUSAIC*. On **(c)** the sample audios are stored. By default there are 15 chunks. The equivalence of this samples can be found on the *README* file, where each sample is associated with its true match. On **(d)** all the images that *MUSAIC* uses, such as the logo and other information images.

On the following sections the code will be detailed. Only the important functions will be displayed, because the GUI is only used to present and get data from the screen.

3.2 Build Option

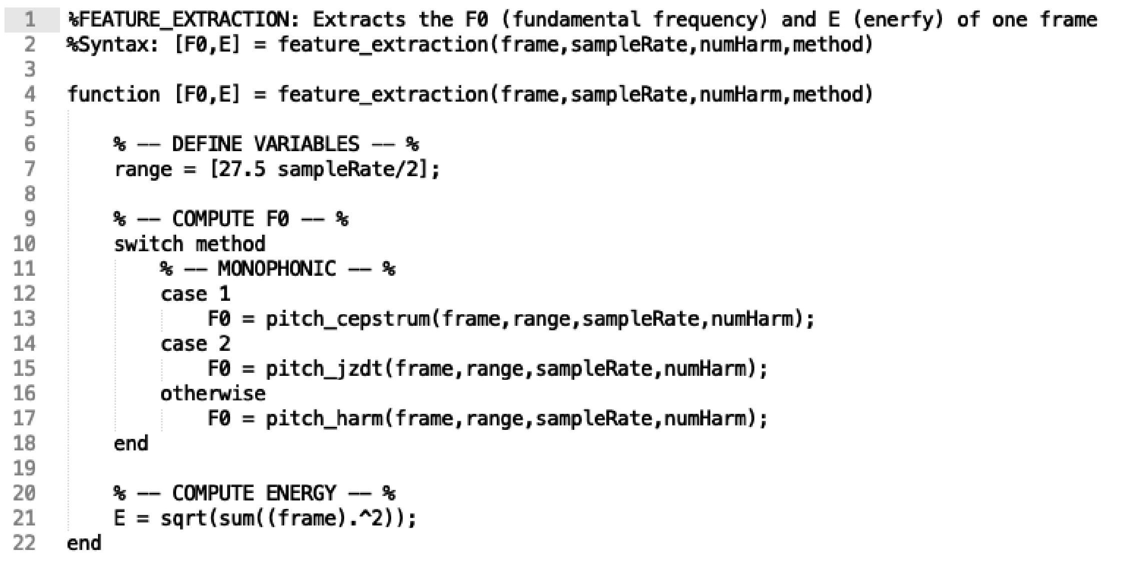
On this section the **Build Option** is explained with detail. Once a song has been selected and the button *Build* has been pressed, we get the method and number of harmonics parameters and call the function *music2colour*



```
%MUSIC2COLOUR: function that transforms the audio selected into a HSV image.
%Syntax: [HSV,outFile,octave,note] = music2colour(fileName,numHarm,method)

function [HSV,outFile,octave,note] = music2colour(fileName,numHarm,method)

    % -- SET PROGRESS BAR -- %
    progress = 0;

    % -- READ AUDIO -- %
    w = waitbar(progress,'READING SELECTED AUDIO...');
    [wave,sampleRate,songTitle] = wave_format(fileName);
    waitbar(progress+1,w,'DONE');

    % -- CONVERT WAVE TO FRAMES -- %
    [signal,numFrames] = signal_buffer(wave,sampleRate);

    % -- GET FEATURES -- %
    waitbar(progress,w,sprintf('GETTING FEATURES (%d OUT OF %d)',0,numFrames));
    F0 = zeros(1,numFrames);
    E = zeros(1,numFrames);
    for k = 1:numFrames
        [F0(k),E(k)] = feature_extraction(signal(:,k),sampleRate,numHarm,method);
        waitbar((k/numFrames),w,sprintf('GETTING FEATURES (%d OUT OF %d)',k,numFrames));
    end
    waitbar(progress+1,w,'DONE');

    % -- CONVERT FEATURES TO HSV -- %
    [HSV,octave,note] = features2hsv(F0,E,numFrames);

    % -- CONSTRUCT OUTFILE -- %
    outFile = [songTitle '_' num2str(method) '_' num2str(numHarm)];

    % -- CLOSE WAITBAR -- %
    close(w);

end
```

Figure 3.1: Function: music2colour

This function is responsible to take the selected audio, reduce its sample rate to 8820KHz, divide the audio into frames with a hamming window of 64ms using the buffer function and extract for each frame its features, being the energy and pitch, calling *feature_extraction*. Once the features are computed, the conversion to HSV is done by calling *features2hsv*. Both *feature_extraction* and *feature2hsv* are the important functions of this option. The first one (*feature_extraction*), call the functions *pitch_cepstrum*, *pitch_jzdt* and *pitch_harm* depending on the selected method.



```
%FEATURE_EXTRACTION: Extracts the F0 (fundamental frequency) and E (enerfy) of one frame
%Syntax: [F0,E] = feature_extraction(frame,sampleRate,numHarm,method)

function [F0,E] = feature_extraction(frame,sampleRate,numHarm,method)

    % -- DEFINE VARIABLES -- %
    range = [27.5 sampleRate/2];

    % -- COMPUTE F0 -- %
    switch method
        % -- MONOPHONIC -- %
        case 1
            F0 = pitch_cepstrum(frame,range,sampleRate,numHarm);
        case 2
            F0 = pitch_jzdt(frame,range,sampleRate,numHarm);
        otherwise
            F0 = pitch_harm(frame,range,sampleRate,numHarm);
    end

    % -- COMPUTE ENERGY -- %
    E = sqrt(sum((frame).^2));
end
```

Figure 3.2: Function: feature_extraction

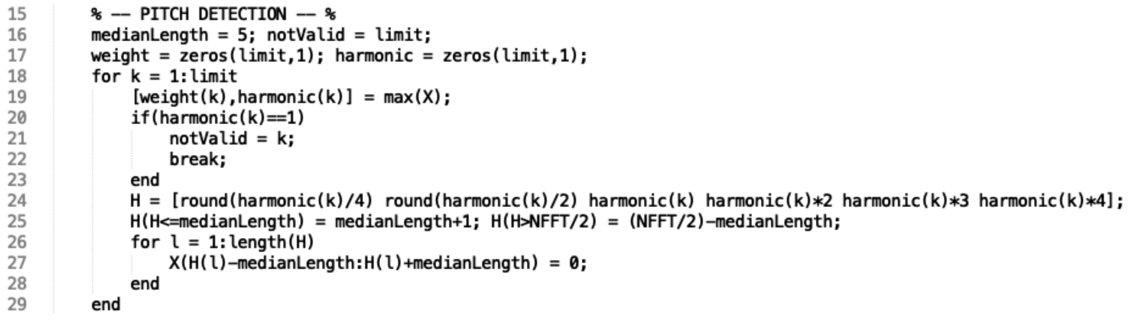
To these functions a range of acceptable pitches, the frame, the sample rate and the number of harmonics chosen are passed as input arguments.
For the Cepstrum method, the cepstrum domain is computed and normalized, using the (**??**) process. Then the location and weight of x maximums are taken from the resulting cepstrum domain, where x is the number of harmonics the user have chosen. Once we have the weights and location of these maximums, we mix them using a weighted sum (3.1).

$$\overline{X} = \frac{\sum_{l=1}^{L} x_l \cdot w_l}{\sum_{l=1}^{L} w_l} \tag{3.1}$$

For the *pitch_jzdt* instead of computing the Cepstrum, a simple Fourier Transform is computed and normalize. Then as before, the location and weights of the first x maximums are taken from the resulting fourier transform (x represents the number of harmonics chosen by the user). Finally we use a vector computed as (3.2) to get the frequencies from those location and use the weighted sum to compute a mix of the peaks.

$$\frac{SampleRate \cdot \begin{bmatrix} 0 \\ \dots \\ \frac{NFFT}{2} - 1 \end{bmatrix}}{NFFT} \tag{3.2}$$

Finally for the *pitch_harm*, the same Fourier Transform is used but instead of taking the first x maximums, the harmonics are eliminated. So in this case, the first x different frequencies are taken as pitches. This is achieved by using the following code.



```
    % -- PITCH DETECTION -- %
    medianLength = 5; notValid = limit;
    weight = zeros(limit,1); harmonic = zeros(limit,1);
    for k = 1:limit
        [weight(k),harmonic(k)] = max(X);
        if(harmonic(k)==1)
            notValid = k;
            break;
        end
        H = [round(harmonic(k)/4) round(harmonic(k)/2) harmonic(k) harmonic(k)*2 harmonic(k)*3 harmonic(k)*4];
        H(H<=medianLength) = medianLength+1; H(H>NFFT/2) = (NFFT/2)-medianLength;
        for l = 1:length(H)
            X(H(l)-medianLength:H(l)+medianLength) = 0;
        end
    end
```

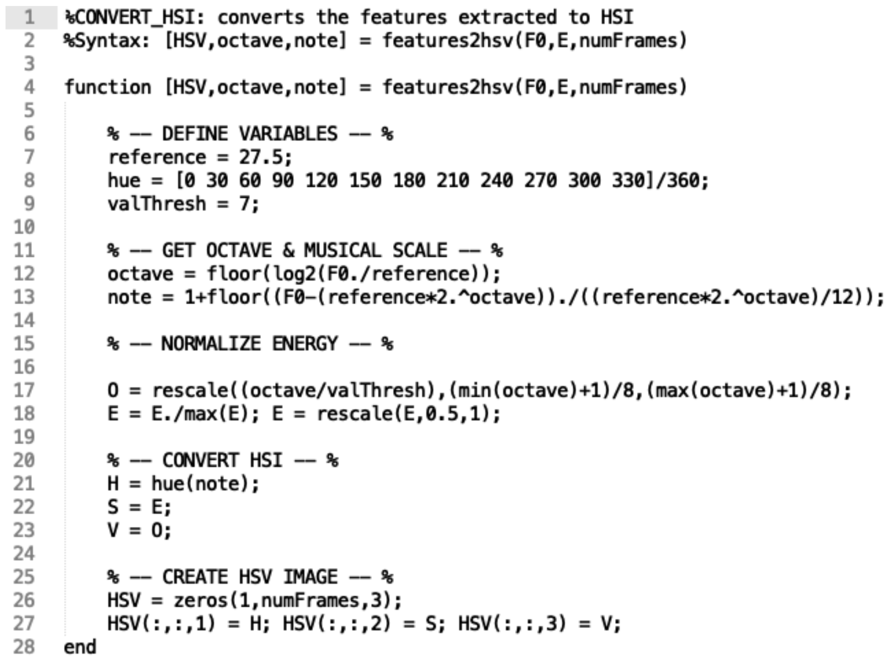
Figure 3.3: Extraction and elimination of harmonics

The *medianLength* variable is used to establish a bandwidth of how wide harmonic lobules are.

Once the pitch for one frame is computed the energy of the frame is computed using the following equation (3.3)

$$E = \sqrt{\sum_{n=0}^{N-1} x_n^2} \tag{3.3}$$

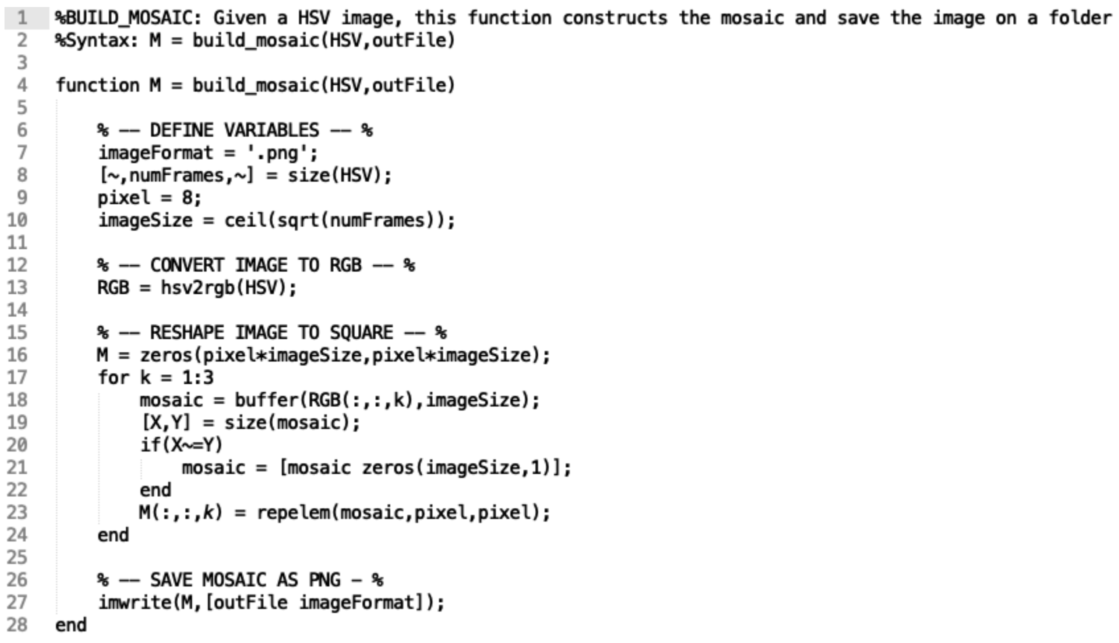
The other important function is *features2hsv* where the pitch and energy are used to compute other metrics that allow us to convert more easily sound to colour.



```
%CONVERT_HSI: converts the features extracted to HSI
%Syntax: [HSV,octave,note] = features2hsv(F0,E,numFrames)

function [HSV,octave,note] = features2hsv(F0,E,numFrames)

    % -- DEFINE VARIABLES -- %
    reference = 27.5;
    hue = [0 30 60 90 120 150 180 210 240 270 300 330]/360;
    valThresh = 7;

    % -- GET OCTAVE & MUSICAL SCALE -- %
    octave = floor(log2(F0./reference));
    note = 1+floor((F0-(reference*2.^octave))./((reference*2.^octave)/12));

    % -- NORMALIZE ENERGY -- %

    O = rescale((octave/valThresh),(min(octave)+1)/8,(max(octave)+1)/8);
    E = E./max(E); E = rescale(E,0.5,1);

    % -- CONVERT HSI -- %
    H = hue(note);
    S = E;
    V = O;

    % -- CREATE HSV IMAGE -- %
    HSV = zeros(1,numFrames,3);
    HSV(:,:,1) = H; HSV(:,:,2) = S; HSV(:,:,3) = V;
end
```

Figure 3.4: Function: features2hsv

On this function we start by using the equations (2.7) and (2.8) to convert the pitch detected into an octave and a note. Then we take the octave, note and energy and treat them as values of a HSV pixel. For the hue we use a vector containing 12 preselected colours and use vectorization to convert the notes into a hue. For the saturation (quantity of colour) we use the energy of the frame, and finally for the value (quantity of light) we use the octave. With this conversion we end up having a relation where the lower is the fundamental frequency, the darker is the colour and vice versa. Frames with more energy will be more colourful than others with less energy. For the notes will depend on where the fundamental frequency fall into the hue vector. On the Results chapter (4), some examples would be more explanatory. Before assigning this values to a HSV image, we rescale some values to make the images a little bit more visible. Energy is usually very low, and this affects the saturation. Finally we create a linear vector with pixels referencing all the frames. The values of octave and notes are also passed as output arguments to be shown with a histogram on the GUI. This way is easier for the user to understand the colours. Once we have the HSV linear image it's time to arrange the values to a regular image, so we returned to the function *music2colour* where we call now the function *build_mosaic*.



```
%BUILD_MOSAIC: Given a HSV image, this function constructs the mosaic and save the image on a folder
%Syntax: M = build_mosaic(HSV,outFile)

function M = build_mosaic(HSV,outFile)

    % -- DEFINE VARIABLES -- %
    imageFormat = '.png';
    [~,numFrames,~] = size(HSV);
    pixel = 8;
    imageSize = ceil(sqrt(numFrames));

    % -- CONVERT IMAGE TO RGB -- %
    RGB = hsv2rgb(HSV);

    % -- RESHAPE IMAGE TO SQUARE -- %
    M = zeros(pixel*imageSize,pixel*imageSize);
    for k = 1:3
        mosaic = buffer(RGB(:,:,k),imageSize);
        [X,Y] = size(mosaic);
        if(X~=Y)
            mosaic = [mosaic zeros(imageSize,1)];
        end
        M(:,:,k) = repelem(mosaic,pixel,pixel);
    end

    % -- SAVE MOSAIC AS PNG - %
    imwrite(M,[outFile imageFormat]);
end
```

Figure 3.5: Function: build_mosaic

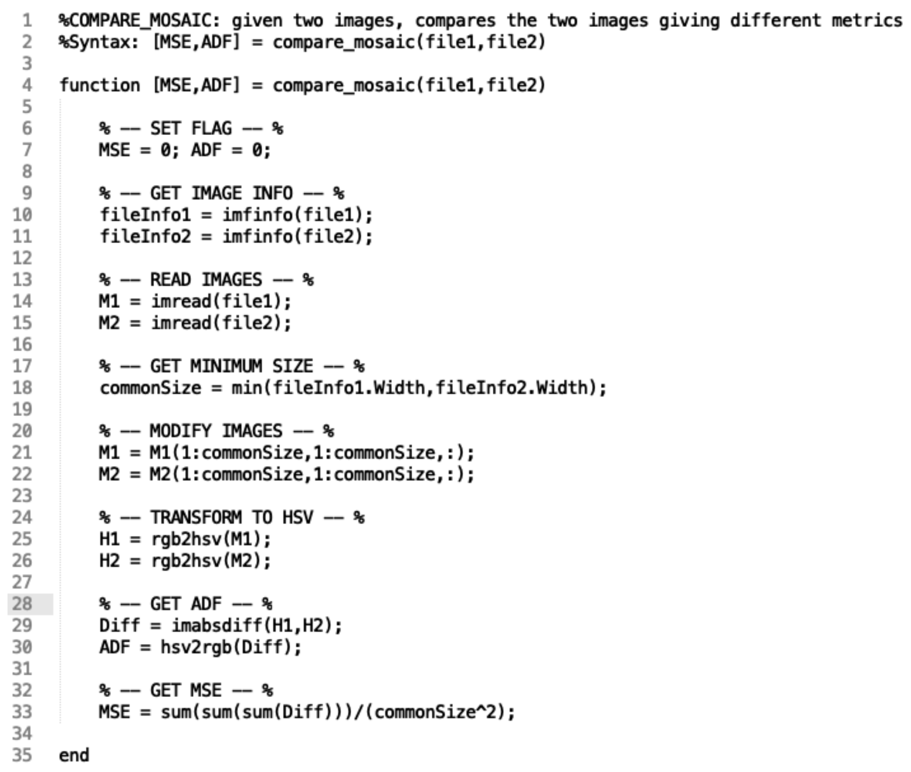
This function is responsible to arrange the linear image into a regular image. First the HSV linear image is converted to RGB. Then we compute the dimension of the image by applying this equation (3.4)

$$[W, H] = \left\lfloor \sqrt{NumFrames} \right\rfloor \cdot 8 \qquad (3.4)$$

Once we have the resulting image initialized, we use the buffer function to divide the linear RGB image into a regular image, to then use *repelem* to repeat each element of the buffer 8 times horizontally and vertically. This way, each frame will be represented by a 8 by 8 block of pixels. This increases the visibility of the colours of each frame.

3.3 Compare Option

The **Compare Option** compares musaics created with the **Build Option**. Because there are different ways of computing a mosaic, it's useful to compare different methods for the same song. Once the two mosaics have been selected using the "Select One" and "Select Two" buttons, we can start comparing the two. Then we call the function *compare_mosaic* that is responsible to compute the difference between two mosaics and show the MSE error.



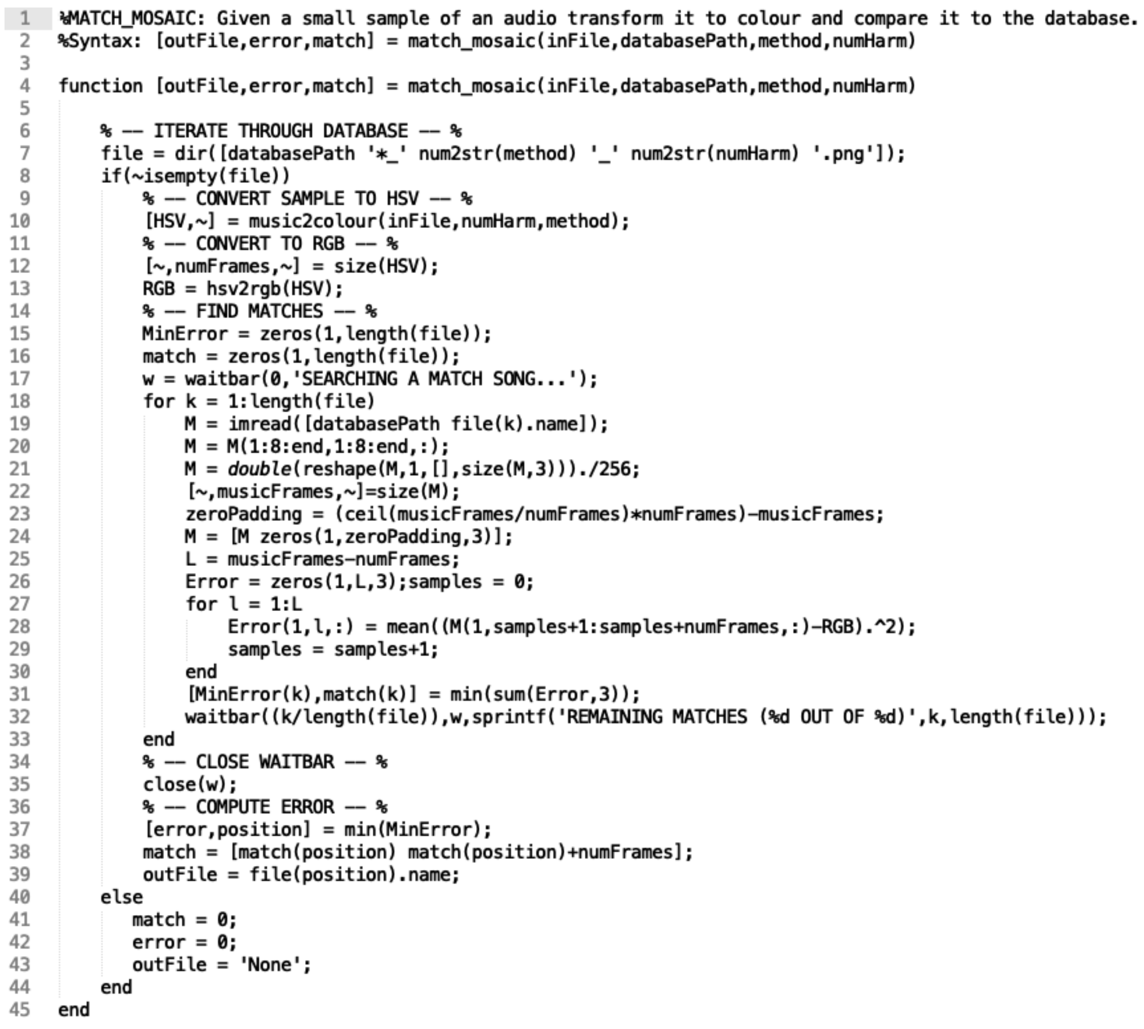
```
%COMPARE_MOSAIC: given two images, compares the two images giving different metrics
%Syntax: [MSE,ADF] = compare_mosaic(file1,file2)

function [MSE,ADF] = compare_mosaic(file1,file2)

    % -- SET FLAG -- %
    MSE = 0; ADF = 0;

    % -- GET IMAGE INFO -- %
    fileInfo1 = imfinfo(file1);
    fileInfo2 = imfinfo(file2);

    % -- READ IMAGES -- %
    M1 = imread(file1);
    M2 = imread(file2);

    % -- GET MINIMUM SIZE -- %
    commonSize = min(fileInfo1.Width,fileInfo2.Width);

    % -- MODIFY IMAGES -- %
    M1 = M1(1:commonSize,1:commonSize,:);
    M2 = M2(1:commonSize,1:commonSize,:);

    % -- TRANSFORM TO HSV -- %
    H1 = rgb2hsv(M1);
    H2 = rgb2hsv(M2);

    % -- GET ADF -- %
    Diff = imabsdiff(H1,H2);
    ADF = hsv2rgb(Diff);

    % -- GET MSE -- %
    MSE = sum(sum(sum(Diff)))/(commonSize^2);

end
```

Figure 3.6: Function: compare_mosaic

First of all we read the images using *imread*. Because music has different lengths and durations, we have to make sure that the mosaics we compare have the same size. For this reason we use the information about the width to check which image has the smallest width. Then once we know which one is it, we simply crop both images to that size. Then we transform the images to HSV, this is done because the calculations are done on the HSV space. The RGB space is only used to display the images. Once we have both mosaics in HSV we use the function *imabsdiff* that takes each element of both images and computes the absolute difference between those values. Then we convert again the resulting image to RGB. Finally to compute the MSE, we use the result of the absolute difference to simply sum all the errors between the three channels of the image and divide by the size of the mosaic. Then we simply display the ADF image and the MSE error.

3.4 Match Option

The **Match Option** tries to match a sample of a song to its original origin. So in a way try to use the mosaics created to identify music with just seconds. On the folder ”SampleChunks” there are already 15 samples of songs to try how well the algorithm of matching works. First we will need to select a chunk then choose the parameters we want to use for the match. So when the user chooses a set of parameters, the algorithm will only use those mosaics computed the same way as the parameters chosen to find the match. This can be done, because the file names of the mosaics contain information of the parameters. If for some reason there were any mosaics with those characteristics, then *MUSAIC* will tell you that there is no mosaics available to find a match with those parameters. So once you choose a sample audio and select the desired parameters, the match button can be pressed. The the function *match_mosaic* is called.



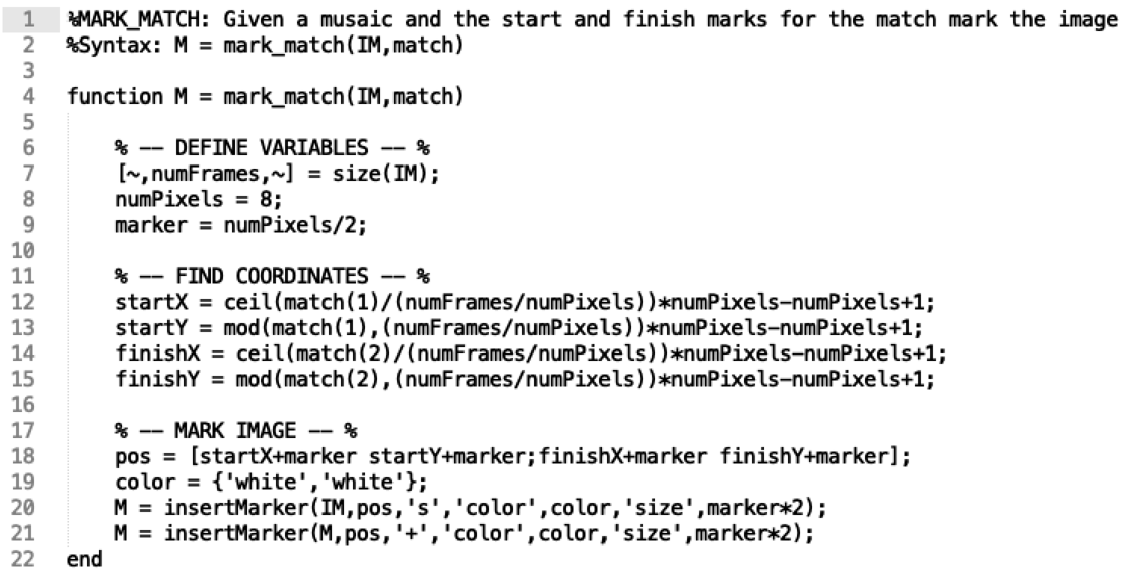
```
%MATCH_MOSAIC: Given a small sample of an audio transform it to colour and compare it to the database.
%Syntax: [outFile,error,match] = match_mosaic(inFile,databasePath,method,numHarm)

function [outFile,error,match] = match_mosaic(inFile,databasePath,method,numHarm)

    % -- ITERATE THROUGH DATABASE -- %
    file = dir([databasePath '*_' num2str(method) '_' num2str(numHarm) '.png']);
    if(~isempty(file))
        % -- CONVERT SAMPLE TO HSV -- %
        [HSV,~] = music2colour(inFile,numHarm,method);
        % -- CONVERT TO RGB -- %
        [~,numFrames,~] = size(HSV);
        RGB = hsv2rgb(HSV);
        % -- FIND MATCHES -- %
        MinError = zeros(1,length(file));
        match = zeros(1,length(file));
        w = waitbar(0,'SEARCHING A MATCH SONG...');
        for k = 1:length(file)
            M = imread([databasePath file(k).name]);
            M = M(1:8:end,1:8:end,:);
            M = double(reshape(M,1,[],size(M,3)))./256;
            [~,musicFrames,~]=size(M);
            zeroPadding = (ceil(musicFrames/numFrames)*numFrames)-musicFrames;
            M = [M zeros(1,zeroPadding,3)];
            L = musicFrames-numFrames;
            Error = zeros(1,L,3);samples = 0;
            for l = 1:L
                Error(1,l,:) = mean((M(1,samples+1:samples+numFrames,:)-RGB).^2);
                samples = samples+1;
            end
            [MinError(k),match(k)] = min(sum(Error,3));
            waitbar((k/length(file)),w,sprintf('REMAINING MATCHES (%d OUT OF %d)',k,length(file)));
        end
        % -- CLOSE WAITBAR -- %
        close(w);
        % -- COMPUTE ERROR -- %
        [error,position] = min(MinError);
        match = [match(position) match(position)+numFrames];
        outFile = file(position).name;
    else
        match = 0;
        error = 0;
        outFile = 'None';
    end
end
```

Figure 3.7: Function: match_mosaic

First all the files computed with the parameters chosen are stored using the function *dir*. Then if the variable with all the files is not empty, we take the sample audio and call the function *musci2colour* previously explained, to convert the sample audio into a linear HSV image, then its convert it to RGB using *hsv2rgb*. Then two variables are initialized to store the Minimum error and the positions of the matches. A loop is used for each file in the *file* variable, where the mosaic is read and decimate by 8 pixels vertically and horizontally. Then the function *reshape* is used to convert the mosaic image to a linear RGB image. Now both the mosaic and the sample audio can be compared. To do so, zeros are added to the mosaic linear image to make sure the length is a multiple of the length of the sample linear image. Then another loop is used to compute the error by fixing the sample linear image and moving the mosaic linear image one by one. This is achieved with the lines [27-30]. The number of iterations is determined by the line 25, where the length of the sample linear image is subtracted to the length of the mosaic linear image. The error of each iteration is computed by the following equation.

$$Error = \frac{1}{L}\sum_{l=0}^{L-1}(M-S)^2 \tag{3.5}$$

The error computed has L values with three dimensions. To compute the minimum error first a sum across the three channels is made using *sum* and then *min* is used to find the location of the minimum and its value. Once all the files have been compared to the sample, the lines [37-39] are used to find the error and position of the minimum error within the files. This is done using the minimum between minimums and for the *match* variable the position indicates where the sample has been detected to begin, then we sum its length to know the full match interval.
The error is displayed on the UI, but with the *match* variable we call the function *mark_match* where the mosaic is mark to show where the match have been found.



```
1   %MARK_MATCH: Given a musaic and the start and finish marks for the match mark the image
2   %Syntax: M = mark_match(IM,match)
3
4   function M = mark_match(IM,match)
5
6       % -- DEFINE VARIABLES -- %
7       [~,numFrames,~] = size(IM);
8       numPixels = 8;
9       marker = numPixels/2;
10
11      % -- FIND COORDINATES -- %
12      startX = ceil(match(1)/(numFrames/numPixels))*numPixels-numPixels+1;
13      startY = mod(match(1),(numFrames/numPixels))*numPixels-numPixels+1;
14      finishX = ceil(match(2)/(numFrames/numPixels))*numPixels-numPixels+1;
15      finishY = mod(match(2),(numFrames/numPixels))*numPixels-numPixels+1;
16
17      % -- MARK IMAGE -- %
18      pos = [startX+marker startY+marker;finishX+marker finishY+marker];
19      color = {'white','white'};
20      M = insertMarker(IM,pos,'s','color',color,'size',marker*2);
21      M = insertMarker(M,pos,'+','color',color,'size',marker*2);
22  end
```

Figure 3.8: Function: mark_match

The main objective of this function is to set the values to call the function *insert-Marker* form the Computer Vision System Toolbox. This function takes an image, a matrix with the positions where the marker will be situated and a bunch of settings to set the appearance of the marker. For the appearance we use two markers, a square and a plus sign to make the global marker more visible. Because we want to set two markers, one for the initial point where the match is detected and the other for when the match finish, we will need two set the coordinates for two points. The variable *match* contains two values, the first one indicating the start of the match and the second the finish. Knowing that each block of 8x8 pixels represents a frame in a mosaic, then the following equations can be used to find the start and finish points of the markers. Where L indicates the number of frames that are in a mosaic.

$$Point = \begin{cases} X = \left\lfloor \frac{match(1)}{L/8} \right\rfloor \cdot 8 - 7 \\ Y = mod\left(match\left(1\right), L/8\right) \cdot 8 - 7 \end{cases} \tag{3.6}$$

Chapter 4

Results

On this chapter some mosaics will be shown and analyzed to better understand how the conversion works.

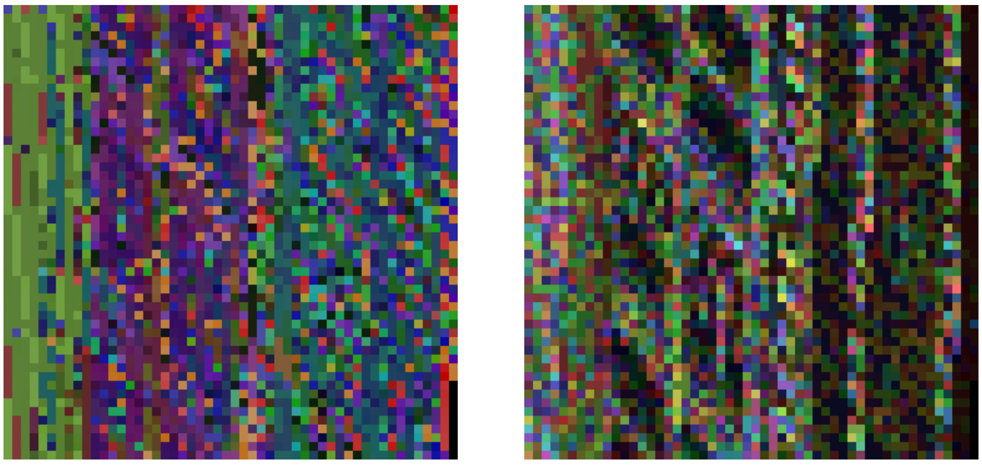

(a) Monophonic **(b)** Polyphonic

Figure 4.1: Different Methods for the same Song

These mosaics represent different methods of conversion for the same song. This is done because each method has a purpose or a different meaning. First of all, the **Monophonic** method works better with songs with strong vocal components. In addition this methods helps to see the structure of a song. In this example the song changes rhythm three times. This can be observed by the three main colours that appear on (4.1a). On the other side, **Polyphonic** method gives you more details about the song. For example on (4.1b) we can observe were the bass and treble sounds are. For example the darkest parts are indicating bass sounds meanwhile the brightest parts are indicating high frequency sounds. Apart from these details, the main objective of converting music to colour is to look at the colour patterns that are form. Music is all about patterns, so in colours they must be preserved. If we look closely at the equivalent purple part of the figure (4.1b) we can detect diagonal ramps with lower frequencies that keep repeating. Other patterns can be seen with other type of songs. Here are some examples.

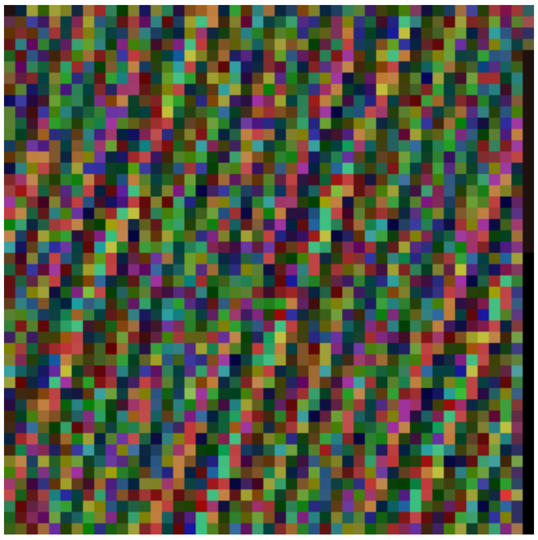


Figure 4.2: Represent by Nas

This song by Nas is very repetitive and can be seen by its diagonal lines that keep repeating the same pattern of dark and bright colours. As the notes are the same, the colours also repeat. There are of course some variances due to the voices that appear across the song.

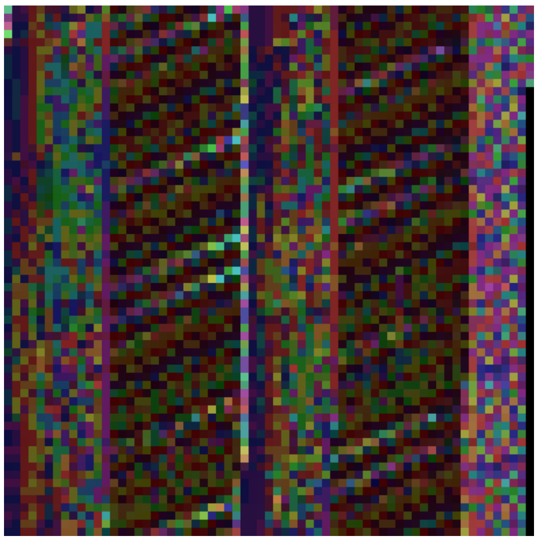


Figure 4.3: Chords by DeadMau5

On this case the different parts of the song can be observed. The chorus is easily recognizable with its diagonal patterns. Otherwise the "verses" change between each other.

Chapter 5

Conclusion

Our objective was to develop a tool that would be able to first convert music to colour, and second try to use those colours to recognize songs. The recognize can be done but has a lot of drawbacks. On this section the advantages and disadvantages will be discuss The first drawback is that colours are stored in images, and images need a lot of memory to be stored. Unless a compression was applied to the generated images for each song, storing a database to be able to recognize a lot of songs would occupy a lot of memory, thus is not efficient. Audio fingerprinting uses hashes to store data. This allows better compression of the data and of course less memory.

Another drawback is that songs can be very similar to one another. The fact that we use basically only pitch to identify music, is easily disturb by interferences, noise, or even similarities between songs. As seen on the demo, the algorithm does not always work. We could improve the algorithm, but i think we would need more information to distinguish between noise or similarities within music.

Adding to the second point, we think that this method is not very robust. Even the change of sampling rate to make the algorithm more efficient, changes some tones, that distortion the results. The addition of noise to the samples could have been tested.

Finally pitch detection for music is still not accurate, so the methods used on this algorithm are not specialized to detect multi tones. More advanced and complex methods are required to find multiple tones on one frame, and this complexity was too much for this project.

However, with the actual methods, the algorithm is capable of recognizing fairly good, simple samples and the most important, where these matches are found. Also the conversion from music to colour can be used to see the structure of a song and its patterns. Overall is a fun tool to use if you have interest in music and how the different musical patterns form interesting colours and palettes.

List of Figures

References

[1] A.Degani et al. “A Pitch Salience function derived from Harmonic Frequency deviations for Polyphonic Music Analysis”. In: *Conference on Digital Audio Effects* (Sept. 2014).

[2] Sung-Ill Kim. “A basic Study on the Conversion of Sound into Color Image using both Pitch and Energy”. In: *Journal of Fuzzy Logic and Intelligent Systems* (June 2012).