Chapter 8  Practical Exercise

Synchronizing the Frequencies of Music with Image Computation and Display in Real Time

This is a bigger project that most of the exercises associated with Digital Sound and Music. It could be considered an interdisciplinary project in the sense that it combines music, digital image creation, and computer programming in the realm of audio processing. This project was undertaken as a Senior Honors Project by a student at Wake Forest University. The student wanted a context for synthesizing his knowledge of digital imaging, sound, frequency analysis, and computational complexity. The challenge he set for himself was to write a program that plays a piece of music, continually analyzes the frequencies in small windows of time, and computes a Julia fractal that reflects the dominant frequencies over each window of time as the musical piece plays.

Fractals are structures that exhibit self-similarity. That is, the whole fractal object has a certain shape, and then components of that whole are composed of a similar shape, and those components are composed of a similar shape, and so on. Fractals appear in nature. Ferns, trees, cauliflowers, and shore lines have fractal structures.

In 1980, Benoit Mandelbrot came up with a fascinating visualization of a set of complex numbers that result from iterative computations. This visualization was given his name, the Mandelbrot fractal, shown in the two images below. The Julia fractal is computed similarly to the Mandelbrot, but it has more variations depending on input values. We leave it to the reader to investigate fractals in more depth [1] [2] [3], but we give you the algorithm below for the Julia fractal computation.

![classic Mandelbrot fractal](image1)
![Mandelbrot fractal, zoomed in](image2)

**Figure 1** Mandelbrot fractals
We chose a Julia fractal as the image to display in our project for three reasons: It is interesting to look at, you can create different versions of depending on your inputs, and computing the Julia fractal is relatively complex because the color of each pixel is computed individually each time the image is displayed. A MATLAB program for computing a Julia fractal is given at the end of this document.

In fractal computation, the user can specify a color map for the fractal (a set of colors stored in an array), choosing as many colors as desired. (MATLAB has some built in color maps that can be used.) The color of each pixel is determined by the area of the fractal being computed and the pixel's position in that area. The initial input values ($c_r$ and $c_i$ in the program) also affect the computation. Different initial values of $c_r$ and $c_i$ produce different shaped Julia fractals, as shown in the figures below.

![Figure 2 Julia fractal computed with $c_r = 0.255$ and $c_i = -0.5667$](image1.png)

![Figure 3 Julia fractal computed with $c_r = 0.18$ and $c_i = -0.59$](image2.png)
The overall scheme of our program is this:

1. Initialize the color map.
2. Compute and display the original fractal.
3. Start the music.
4. While the music is playing
   - Analyze the frequency components in the current window of \( w \) audio samples.
   - Change the color map to reflect the dominant frequency components. (How this change is made is up to the programmer. Different algorithms for color-changing could result in more or less interesting fractal displays.)
   - Recompute the fractal using the new color map, perhaps zooming in for a more interesting dynamic display. Zooming in entails resetting the area of the fractal being computed.
   - Redisplay the fractal.

You can see that the frequency analysis, resetting of the color map, and the fractal computation all have to be executed in the same loop. Importantly, they have to be executed fast enough to keep up with the music as it plays. Thus, the computation complexity and execution time of these functions must be considered.

The Julia fractal algorithm computes a color for each pixel in an \( n \times m \) pixel image. For each of these pixels, a \textit{while} loop is executed. Based on the location of the pixel, this loop could continue up to a given maximum number of iterations, or it could stop earlier. If the loop continues up to the maximum number of iterations (\( \text{max} \)), the pixel is colored black. Otherwise, the number of iterations is mapped to a position in the color map, determining the color for the pixel. Thus, in the worst case, the complexity of the Julia fractal algorithm is \( O(n \times m \times \text{max}) \).

The fast Fourier transform (FFT) can be used to analyze the frequencies present in a window of time as the music plays. The computational complexity of the FFT is \( O(w \times \log(w)) \) where \( w \) is the number of audio samples in the window of time being analyzed. The FFT has to use window sizes that are powers of 2, usually around 1024, 4096, or 8192. Choosing the size of the window is a tradeoff. The larger the window size, the less precise is the analysis of the frequencies present because the frequencies (pitches in music) can change constantly. But with a smaller window size, the analysis divides up the audible range of frequencies into fewer bands.

The algorithm that resets the color map makes use of the output of the FFT. For a window size of \( w \), the FFT returns the magnitude (and phase) of \( w/2 \) frequency bands divided evenly between 0 and the Nyquist frequency, which for a sampling rate of 44,100 Hz is 22,050 Hz. Assume that \( w = 8192 \), so we get back 4096 frequency bands. Let’s focus our attention on just the frequencies between 0 and 4091.3 Hz. This range is chosen for two reasons. It focuses on the frequencies we mostly hear in the music we intend to play. (The
highest note on an 88-key piano has a frequency of 4186.01 Hz.) Also, choosing 4090.3 Hz as the highest frequency we care about simplifies the result of our arithmetic.

To calculate how many of the frequency components (yielded from the FFT) are relevant to us, we have \( \frac{4099.3}{22050} = \frac{x}{4096} \), so \( x \) is about 760 bands of frequencies. We could have a color map of 760 colors and have a one-to-one mapping of magnitude of frequency to brightness of color, or we could group the frequencies bands to map them to a smaller number of colors in the color map. Grouping 5 frequency bands and get a color map of size 152 would be reasonable. You can see that there’s a lot of analysis, experimentation, and fiddling that could be done with sampling rate, window size, and color map size to ensure synchronization and yield different visual results.

We can demonstrate how this is done in MATLAB.

```matlab
audio = audioread('ToccataAndFugue.wav');
window = audio(10001:18192);
fftresult = fft(window);
magnitudesOfFrequencies = abs(fftresult);
plot(linspace(1, 22050, 4096), magnitudesOfFrequencies(1:4096));
```

![Figure 4 Results of FFT of segment of 8192-sample window of "Toccata and Fugue," magnitudes of frequencies](image)

```matlab
figure;
plot(linspace(1, 4091, 760), magnitudesOfFrequencies(1:760));
```
Figure 5 Focusing in on frequencies from 1 to 4091 Hz in segment from "Toccata and Fugue"

The x-axis in the graph of Figure 5 has 760 values. This means that the frequencies from 0 to 4091 Hz are divided into 760 bands. As we make our color map, we establish a relationship between each band (or group of bands) to a color in the table. If one of these bands is dominant at a moment in time, we change the corresponding color in some way. It is up to the programmer to decide how many colors should be in the color table, what "dominant" means, and how the colors corresponding to dominant frequencies should be changed for best visual effect.

It's possible to play our very small segment of "Toccata and Fugue" as follows:

```plaintext
sound(window, 44100);
```

We can also write the segment out as a WAV file.

```plaintext
audiowrite(window, 44100, 'window.wav');
```

We did this, and then opened the WAV file in Adobe Audition. If you compare frequency analysis of this segment, you can see that it has peaks at the same places: around 1000 Hz,
Recall that in our example, we've decided to focus on 760 frequency components that range from 0 to 4091 Hz, and we want 152 colors in our color table. The figure below illustrates how we group five frequency components, determine how “dominant” they are relative to the others at this moment in time, and do something to the corresponding color in the color table.

<table>
<thead>
<tr>
<th>color’s position in color map</th>
<th>RGB values for color (initialized by programmer as desired)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frequency band</th>
<th>magnitude of frequency band (output of FFT, focusing on 1 to 4091 Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>756</td>
<td></td>
</tr>
<tr>
<td>757</td>
<td></td>
</tr>
<tr>
<td>759</td>
<td></td>
</tr>
<tr>
<td>759</td>
<td></td>
</tr>
<tr>
<td>760</td>
<td></td>
</tr>
</tbody>
</table>

Now consider how the computation has to keep up with the rate at which the music is playing. Let’s assume we’re playing a piece that is sampled at 44,100 Hz (that is, 44,100 samples per second). Suppose that we try a window size of 8192 samples for our Fourier transform. This means that we are processing \( \frac{1}{44100} \times 8192 = 0.1858 \) seconds-worth of the
music, and we have just that much time to do the Fourier transform, color map resetting, and fractal recomputation. This may or may not be possible, as you’ll discover if you try this yourself. This exercise entails 1. implementing the algorithms and making choices for how the colors react to the frequencies; and 2. experimenting with window sizes, color maps sizes, concurrency, etc. to see if the computation can keep pace with the display, giving visually interesting results.

If a straightforward algorithm doesn’t meet the challenge, threads or parallel programming could be tried as a way of speeding up computation and ensuring synchronization. However, threads introduce new issues. In particular, you have to determine which thread plays the music, which does the FFT, which resets the colormap, which recomputes the fractal, and which controls the graphic display – allowing them to share variables appropriately.

We offer an implementation and analysis of this program at Digital Sound and Music’s solution site. Your instructor can have access to the solution website. Even if you don’t want to try to implement the program yourself, you may find it interesting to look over the versions and experiments we tried.
function () = julia()
    cr = .255; %Initial values of cr and ci determine the shape of the fractal
    ci = -.5667;
    m = 300.0; %Horizontal and vertical ranges of complex number
    n = 400.0;
    hb = -2.0;
    he = 2.0;
    h = he-hb;
    vb = -1.5;
    ve = 1.5;
    v = ve - vb;
    max=1000; %Maximum number of iterations
    colorR = randi(256,1,max); %Create 3 one-dimentional array for R G B color
    colorG = randi(256,1,max);
    colorB = randi(256,1,max);
    for y=1:n %For each pixel map pixel coordinates(x,y) to complex number plane
        for x=1:m
            zr = hb +(y/n)*h;
            zi = vb + (x/m)*v;
            num_iterations=0;
            %While iteration<max and not unbounded
            while ((num_iterations < max)&&((zr*zr+zi*zi)<4.0))
                z_newr=zr*zr-zi*zi+cr;
                z_newi=2*zi*zr+ci;
                num_iterations=num_iterations+1;
                zr=z_newr;
                zi=z_newi;
            end
            if num_iterations==max %While iteration=max color=black
                answer(x,y,1) = 0;
                answer(x,y,2) = 0;
                answer(x,y,3) = 0;
            else
                answer(x,y,:) = [colorR(num_iterations+1), colorG(num_iterations+1), colorB(num_iterations+1)]; %Populate color according to iteration
            end
        end
    end
    imshow(uint8(answer));
end