Producción y análisis de música

Autores: Alejandro Magnorsky, Andrés Mata Suárez, Mariano Merchante

Instituto Tecnológico de Buenos Aires

Resumen

Palabras clave

Archivo de sonido; Partitura; Transformada de Fourier; Frecuencia.

I. Introducción

Existe mucho interés en el análisis de sonidos musicales. Desde el reconocimiento de las notas más significativas para fines tales como la generación de una partitura a partir de una canción, el cálculo de qué tan afinada está una grabación o incluso analizar qué escala se utiliza en cierta obra, hasta la sintetización de dichas notas para poder generar música a partir de sencillas partituras digitales. Se utilizará la escala conocida como de *Temperamento Igual*, la cual divide la escala de frecuencias en fragmentos de doce niveles diferentes, mostrados en la Tabla I. La relación de frecuencias entre notas sucesivas es siempre $2^{\frac{1}{12}}$. Cada fragmento de doce notas denota una octava, y es un múltiplo de la escala fundamental.

La sección II muestra principalmente el desarrollo del análisis, donde primero se muestra cómo analizar y reconstruir una secuencia de notas a partir de un archivo de sonido, y luego el análisis inverso, es decir, cómo construir un archivo de sonido a partir de una partitura digital.

Luego, en la seccion III se muestran algunos resultados de los programas realizados.

II. Desarrollo

A. Construcción de una partitura a partir de un archivo de sonido

El objetivo es la creación de un programa que, dado un archivo de sonido (.wav), genere una partitura. El código se divide en dos archivos o funciones, que son las que se detallan en la figuras 1 y 2.

Una señal física, como la onda del sonido, puede ser representada mediante una función del tiempo continua. Así, un tono puro con frecuencia f_0 produce una onda de la forma:

$$x(t) = \sin(f_0 t) \tag{1}$$

Notación Latina
do
do #
re
mi ♭
mi
fa
fa #
sol
la ♭
la
si ♭
si

Tabla I: Notas musicales

La ecuación (1) es la representación en el tiempo del tono puro, pero también existe una representación en la frecuencia. La música está formadas por sucesiones y superposiciones de funciones de la forma de la ecuación (1). En particular, si se considera el caso de tonos puros consecutivos, obteniendo la representación en frecuencia cada cierto intervalo de tiempo, se pueden conseguir todas las notas musicales que componen dicha música. Cabe mencionar que se trabaja con una discretización de la función continua que modela la música.

Para obtener la representación de la música en frecuencia, es decir, el espectro de frecuencias, se puede utilizar la Transformada Discreta de Fourier para cada uno de los intervalos de la función original. La Transformada Discreta de Fourier (Mathews y Fink, 1992) se define como:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n} \tag{2}$$

donde x_n es la función periódica que representa al tono puro y N es la cantidad de muestras de dicho tono.

Para comenzar, la función `getFrequencies` (figura 1) recibe la ruta del archivo de sonido que se desea procesar. Utilizando la función de Octave, `wavread`, se obtiene el vector que representa la onda del sonido, x, junto a la frecuencia de muestro de dicha señal, f_s. Considerando intervalos de tiempo de $30ms$, se divide a x en vectores de $\frac{30}{1000} f_s$ elementos, ya que esa es la cantidad de muestreos del sonido que se realizan a lo largo de $30ms$. Para cada uno de dichos vectores, se calcula la Transformada Discreta de Fourier usando la función que provee Octave, `fft`, la cual usa el algoritmo de la Transformada Rápida de Fourier. La ventaja de esto es que tiene $O(n \log n)$ en lugar de $O(n^2)$.

Cada vector X que devuelve la función `fft`, pasa a ser la entrada para otra función llamada `fftshift`. El efecto de `fftshift` es mover los valores de X para que queden centrados en torno a la frecuencia 0. Finalmente, considerando los valores en que la frecuencia es positiva, es decir, la segunda mitad del vector X, se averigüa en qué valor de la frecuencia X se maximiza. Ese valor es la frecuencia fundamental del intervalo considerado, que se guarda en el vector de frecuencias que devuelve la función en cuestión.

Utilizando el vector de frecuencias generado a partir de la función `getFrequencies`, la función `writePartiture` (figura 2) genera un vector de caracteres que representan los tripletes de cada una de las frecuencias.

Cabe destacar que es necesario que tanto los métodos `frequencyToNote` (figura 3) como `noteFromDistance` (figura 4), utilizados directa e indirectamente por la función `writePartiture`, trabajen con cierto grado de error, tanto al momento de decidir el intervalo de frecuencias (octava de una nota) al cual pertenece una frecuencia dada, como también al calcular la distancia en medio tonos de cada nota respecto a un *la* en la octava cuarta (*A-4* en notación de tripletes). Esto se debe a la naturaleza aproximativa del proceso de obtención de las frecuencias de un archivo de sonido.

B. Construcción de un archivo de sonido a partir de una partitura

El camino inverso al proceso descripto en el inciso II-A no presenta demasiadas complicaciones. Dado un vector compuesto por n tripletes, la función `readPartiture` (figura 5) se encarga de transformar cada uno de ellos en la nota que les corresponda usando la función `noteToFrequency` (figura 7). Una vez finalizado dicho procedimiento, se utiliza cada frecuencia f_k para construir un vector de valores $\mathbf{v^{f_k}}$, en donde el componente $v_i^{f_k}$ de dicho vector corresponde al resultado de la siguiente ecuación:

$$v_i^{f_k} = \sin(2\pi t_i f_k) \tag{3}$$

siendo t_i el i-ésimo componente de un vector tiempo t cuyos elementos van desde 0 a $30ms$ con pasos de $\frac{1}{f_s}$.

Como resultado, se dispone de n vectores $\mathbf{v^{f_k}}$ que son finalmente concatenados en uno único, listo para ser convertido a sonido mediante la función `wavwrite` que ofrece Octave.

III. Resultados

El archivo de sonido obtenido mediante el uso de `readPartiture` presenta cierto ruido entre tonos. El mismo es provocado por las discontinuidades entre los vectores $\mathbf{v^{f_k}}$ consecutivos. Por ese motivo, en la función `readPartiturev2` (figura 6),

Listing 1: Implementación de la generación de la secuencia de frecuencias de un archivo de sonido.

```
function frequencies = getFrequencies(wavFile)

% x es la onda del sonido
% fs es la frecuencia de muestreo (cantidad de muestreos por segundo).
% Es decir, un segundo equivale a fs muestras de x
% bits es la cantidad de bits de cada muestra
[x, fs, bits] = wavread(wavFile);

interval = floor(fs*30/1000);
quant = floor(length(x)/interval);

for k=1:quant
        lower = (k-1)*interval+1;
        upper = k*interval;

        X = fft(x(lower:upper));
        X = fftshift(X);
        X = X(floor(length(X)/2)+1:length(X));
        f = 0:fs/interval:fs/2-1;

        [number, pos] = max(X);
        frequencies(k) = f(pos);
endfor
plot(frequencies)
endfunction
```

Listing 2: Implementación de la escritura de una partitura dada una secuencia de frecuencias.

```
function partiture = writePartiture(frequencies)
        for i=1:length(frequencies)
                partiture(i,:) = frequencyToNote(frequencies(i));
        endfor
endfunction
```

se agrega al calculo original de $\mathbf{v^{f_k}}$, un desfase ϕ:

$$\phi = \sum_{i=0}^{k-1} 2\pi f_k (t_f + \frac{1}{f_s}) \tag{4}$$

siendo t_f la duración de un tono. De esta forma, $\mathbf{v^{f_k}}$ se obtiene calculando:

$$v_i^{f_k} = \sin(2\pi t_i f_k + \phi) \tag{5}$$

Esto hace que la concatenación de los vectores $\mathbf{v^{f_k}}$ sea continua, eliminando así el ruido producido.

IV. Conclusiones

Referencias

Mathews, John H., Fink, Kurtis D., “Numerical Methods Using MATLAB”, Prentice Hall, 1999

Listing 3: Implementación de la conversión de una frecuencia en un triplete.

```
function n = frequencyToNote(f)

        if (f == 0)
                n = "S--";      % Silence
                return;
        endif

        f_A4 = 440;
        o_A4 = 4;
        delta = 5;
        o_interval = [ 32.07 - delta , 61.73 + delta ];
        up_distance_to_new_octave = 3;
        down_distance_to_new_octave = 10;

        octave = 1;
        while(f < o_interval(1) || o_interval(2) < f)
                o_interval = o_interval * 2;
                octave = octave + 1;
        endwhile

        sign = sign(octave - o_A4);
        v = f / f_A4;
        v = log2(v) * 12;

        distance = v;
        if (sign != 0)
                distance = distance - (octave - o_A4 - sign) * 12;
                distance = distance - sign * (down_distance_to_new_octave + up_distance_to_new_octave - 1);
        endif

        n = noteFromDistance(distance);
        n = horzcat(n, num2str(octave));

endfunction
```

Listing 4: Obtención de la nota a partir de su distancia con respecto a un *la* en la octava cuarta.

```
function note = noteFromDistance(distance)
        distance = round(distance);
        if ((distance - -9) == 0)
                note = "C-";
        elseif ((distance - -8) == 0)
                note = "C#";
        elseif ((distance - -7) == 0)
                note = "D-";
        elseif ((distance - -6) == 0)
                note = "Eb";
        elseif ((distance - -5) == 0)
                note = "E-";
        elseif ((distance - -4) == 0)
                note = "F-";
        elseif ((distance - -3) == 0)
                note = "F#";
        elseif ((distance - -2) == 0)
                note = "G-";
        elseif ((distance - -1) == 0)
                note = "Ab";
        elseif ((distance - 0) == 0)
                note = "A-";
        elseif ((distance - 1) == 0)
                note = "Bb";
        else
                note = "B-";
        endif
endfunction
```

Listing 5: Implementación de la lectura de una partitura a partir de un conjunto de tripletes.

```
function readPartiture(sps, bps, triplets)

        min_octave = 55;
        triplet_duration = 0.01;

        duration = triplet_duration * length(triplets(:,1));
        t = 0:1/sps:triplet_duration;

        wave = 0;
        for i=1:length(triplets(:,1))
                frequency = noteToFrequency(triplets(i,:));
                wave = [wave sin(2*pi*frequency*t)];
        endfor

        wavwrite(wave', sps, bps, 'audio.wav');

endfunction
```

Listing 6: Implementación mejorada de la lectura de una partitura a partir de un conjunto de tripletes.

```
function readPartiturev2(sps, bps, triplets)

        min_octave = 55;
        triplet_duration = 0.03;

        duration = triplet_duration * length(triplets(:,1));
        t = 0:1/sps:triplet_duration;

        wave = 0;
        phi = 0;
        for i=1:length(triplets(:,1))
                frequency = noteToFrequency(triplets(i,:));
                wave = [wave sin(2*pi*frequency*t + phi)];
                phi = phi + 2*pi*frequency*(t(length(t))+1/sps);
        endfor

        wavwrite(wave', sps, bps, 'audio.wav');

endfunction
```

Listing 7: Implementación de la conversión de un triplete en una frecuencia.

```
function f = noteToFrequency(triplet)

        if (triplet == "S--")
                f = 0;  % Silence
                return;
        endif

        f_A4 = 440;
        o_A4 = 4;
        up_distance_to_new_octave = 3;
        down_distance_to_new_octave = 10;

        note = substr(triplet,1,2);
        octave = str2num(substr(triplet,3,1));
        distance = distanceFromNote(note);

        sign = sign(octave - o_A4);
        if (sign != 0)
                n = distance + sign * (down_distance_to_new_octave + up_distance_to_new_octave - 1);
                n = n + (octave - o_A4 - sign) * 12;
        else
                n = distance;
        endif

        f = power(2,  n/12) * f_A4;

endfunction
```