Madrid, 17 de mayo de 2019

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería y Diseño Industrial

Informática

Grupo A-109

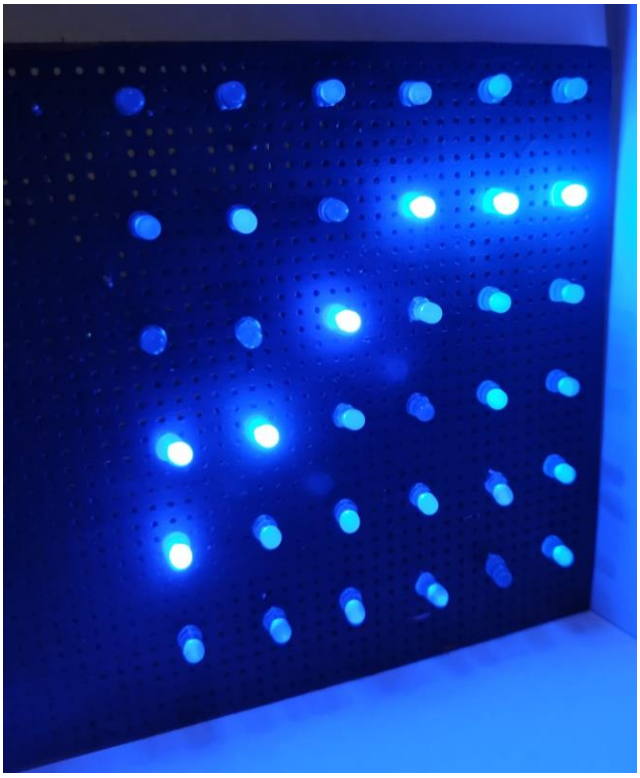
Alumnos:

Marcelo Luna Díaz m.lunad@alumnos.upm.es

Francisco Padilla De Aguiar f.padilla@alumnos.upm.es

María Lisa Salto Molodojen ml.salto@alumnos.upm.es

Analizador de espectro con Arduino y Processing



Resumen

Este proyecto consiste en analizador de espectro de leds, con dimensiones de 6x6 que se iluminan en función de las frecuencias y volumen de sonidos que recibe a través de un micrófono.

Funcionamiento

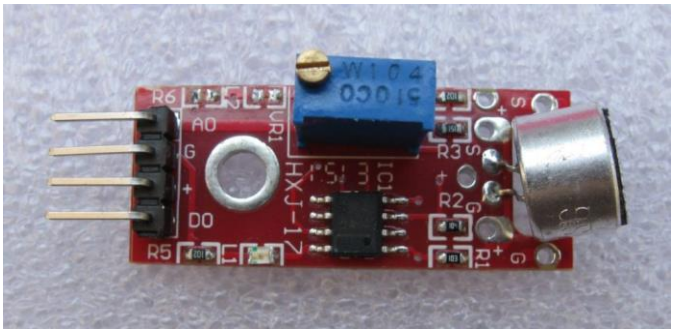
- Se muestrea el sonido del ambiente usando un micrófono.
- Es recomendable filtrar la señal recibida por el micrófono para obtener mejores resultados.
- Con los datos recibidos, la placa Arduino aplica la Transformada Rápida de Fourier (FFT por sus siglas en inglés).
- El resultado se envía al ordenador para representarlo en pantalla. También, este mismo resultado es ajustado para poder ser enviado y encender los Leds, que reaccionarán de acuerdo a las frecuencias y amplitudes obtenidas.

Marco teórico

Micrófono

Un micrófono es un sensor electroacústico que convierte el sonido (ondas sonoras) en una señal eléctrica para aumentar su intensidad, transmitirla y registrarla. Los micrófonos tienen múltiples aplicaciones en diferentes campos como en telefonía, ciencia, salud, transmisión de sonido en conciertos y eventos públicos, trasmisión de sonido en medios masivos de comunicación como producciones audiovisuales (cine y televisión), radio, producción en vivo y grabado de audio profesional, desarrollo de ingeniería de sonido, reconocimiento de voz y VoIP.

Para este proyecto, utilizamos un micrófono con un circuito integrado LM393, que contiene dos unidades iguales, diseñado para ser utilizado como comparador de voltaje de precisión.



Fuentes:

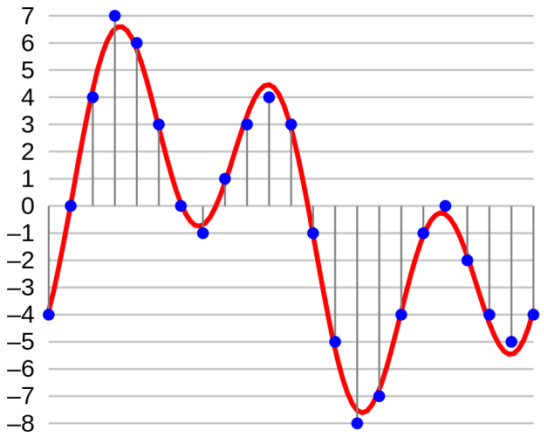
- https://es.wikipedia.org/wiki/Micr%C3%B3fono
- https://www.ecured.cu/LM393

Muestreo de señales

El muestreo digital es una de las partes del proceso de digitalización de las señales. Consiste en tomar muestras de una señal analógica a una frecuencia o tasa de muestreo constante, para cuantificarlas posteriormente. Es de especial interés el Teorema de Muestreo de Nyquist Shannon, que demuestra que la reconstrucción exacta de una señal periódica continua en banda base a partir de sus muestras, es matemáticamente posible si la señal está limitada en banda y la tasa de muestreo es superior al doble de su ancho de banda.

Dicho de otro modo, la información completa de la señal analógica original que cumple el criterio anterior está descrita por la serie total de muestras que resultaron del proceso de muestreo. No hay nada, por tanto, de la evolución de la señal entre muestras que no esté perfectamente definido por la serie total de muestras.

- Una vez satisfechos los criterios de Nyquist, la calidad de la reconstrucción de una señal en toda su banda, ya no se encuentra en función de la tasa de muestreo empleada en el proceso de muestreo.
- El proceso de muestreo no debe ser confundido con el de cuantificación, que redondea los valores obtenidos por el muestreo para su posterior codificación (generalmente en binario) y que, por tanto, puede provocar distorsión. El muestro es, desde el punto de vista matemático, perfectamente reversible una vez cumplido el criterio de Nyquist, es decir, su reconstrucción 'per se' es exacta, no aproximada. La cuantificación, por el contrario, supone una simplificación mediante redondeo o aproximación de los valores de muestreo de la señal analógica, que apenas podrán revertirse mediante técnicas de modelado de ruido a fin de alterar selectivamente la distorsión consecuencia del proceso de cuantificación en señales completamente digitalizadas, es decir, muestreadas, cuantificadas y codificadas.
- Es un error creer que los puntos que resultan del proceso de muestreo se unen en la reconstrucción mediante rectas, también denominada interpolación lineal. Esto formaría dientes de sierra en la señal obtenida, representadas por muestras con una frecuencia de valores discretos siempre menor a la continuidad analógica. Por el contrario, el proceso de cálculo de la interpolación se realiza de manera predictiva. El teorema de muestreo demuestra que toda la información de una señal contenida en el intervalo temporal entre dos muestras cualesquiera, está descrita por la serie total de muestras, siempre que la señal registrada sea de naturaleza periódica -como lo son las ondas mecánicas del sonido, o las ondas electromagnéticas-, y no tenga componentes de frecuencia igual o superior a la mitad de la tasa de muestreo; en cuyo caso no será necesario predecir la evolución de la señal entre muestras para obtener la representación continua de la señal.
- En la práctica, y dado que no existen los filtros analógicos pasa-bajo ideales, se debe dejar un margen entre la frecuencia máxima que se desea registrar, y la frecuencia crítica de Nyquist que resulta de la tasa de muestreo escogida.



Fuentes:

- https://es.wikipedia.org/wiki/Muestreo_digital
- https://es.wikipedia.org/wiki/Teorema_de_muestreo_de_Nyquist-Shannon

Filtrado de la señal: filtro de paso bajo (EMA)

El filtro EMA consiste en obtener un valor filtrado a partir de una medición mediante la aplicación de la siguiente expresión

$$A_n = \alpha M + (1 - \alpha) A_{n-1}$$

Siendo An el valor filtrado, An-1 el valor filtrado anterior, M es el valor muestreado de la señal a filtrar, y alpha es un factor entre 0 y 1.

El resultado de un filtro exponencial EMA es una señal suavizada donde la cantidad de suavizado depende del factor alpha, como analizaremos posteriormente.

Las ventajas en cuanto a sencillez y eficiencia computacional son evidentes. El cálculo requiere una única instrucción sencilla. En cuanto a requisitos de memoria necesitamos almacenar únicamente el valor filtrado anterior. Esto supone una gran ventaja computacional frente a otros filtros que requieren guardar N valores y ejecutar cálculos sobre todos ellos.

Como todos los filtros que suavizan una señal (no es exclusivo del filtro EMA) podemos emplear el filtro exponencial como un filtro paso bajo, es decir, un algoritmo que (idealmente) deja pasar los componentes frecuenciales inferiores a una frecuencia de corte.

Podemos emplear el filtro paso bajo para eliminar el ruido de alta frecuencia superpuesto a la señal, lo que podemos emplear para mejorar la medición de sensores y las comunicaciones, entre otros usos.

El factor Alpha condiciona el comportamiento del filtro exponencial y está relacionado con la frecuencia de corte del filtro. Sin embargo una relación sencilla no es siempre posible ya que depende del tiempo de muestreo de nuestro sistema que, en principio, es desconocido y posiblemente variable entre ciclos.

De forma cuantitativa:

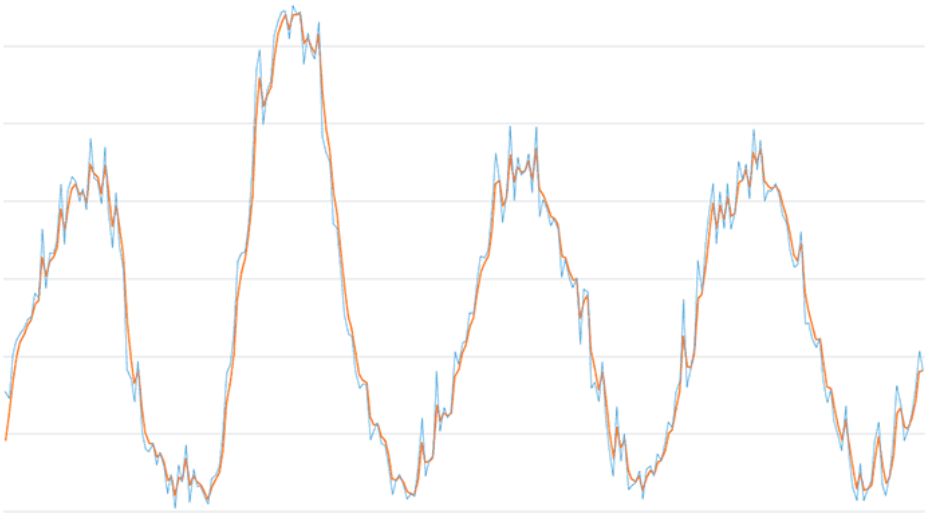
- Un valor Alpha = 1 proporciona la señal sin filtrar, ya que prescinde del efecto filtrado que proporciona la medición anterior.
- Un valor de Alpha = 0 provoca que el valor filtrado siempre sea 0, ya que prescinde la información nueva que aporta la medición al sistema.

Disminuir el factor Alpha aumenta el suavizado de la señal, pero a costa de introducir consecuencias también negativas. Por un lado, podemos eliminar componentes frecuenciales que realmente nos fueran de interés, clasificando como ruido algo que realmente era una variación real de la señal.

Por otro lado, disminuir el factor Alpha aumenta el tiempo de respuesta del sistema, es decir, el tiempo que tarda el sistema en estabilizarse ante una entrada constante. Esto se traduce en la introducción de un retraso entre la señal original y la señal filtrada.

Lógicamente el valor de Alpha adecuado dependerá de las características de nuestro sistema, de la señal muestreada, y el ruido que queramos eliminar. En principio, deberemos ajustar el valor para que resulte adecuado a nuestro montaje, siendo valores habituales 0.2-0.6.

En este proyecto, empleamos un filtro de paso bajo.



En azul: señal original En rojo: resultado de aplicar un filtro de paso bajo con alpha 0.6

Fuente: https://www.luisllamas.es/arduino-paso-bajo-exponencial/

Transformada de Fourier

La transformada de Fourier juega un papel muy importante en la realización y funcionamiento de este proyecto. Para obtener las distintas frecuencias que componen el sonido recibido por el micrófono, así como la energía de cada una, utilizamos una librería de Arduino: arduinoFFT.h. Para saber cómo trabaja esta librería, es bueno saber qué es la transformada de Fourier y algunos algoritmos usados para obtenerla.

1. Transformada de Fourier:

Denominada así por Joseph Fourier, es una transformación matemática empleada para transformar señales entre el dominio del tiempo (o espacial) y el dominio de la frecuencia, que tiene muchas aplicaciones en la física y la ingeniería. Es reversible, siendo capaz de transformarse en cualquiera de los dominios al otro. El propio término se refiere tanto a la operación de transformación como a la función que produce. La transformada de Fourier se utiliza para pasar una señal al dominio de frecuencia para así obtener información que no es evidente en el dominio temporal. Por ejemplo, es más fácil saber sobre qué ancho de banda se concentra la energía de una señal analizándola en el dominio de la frecuencia.

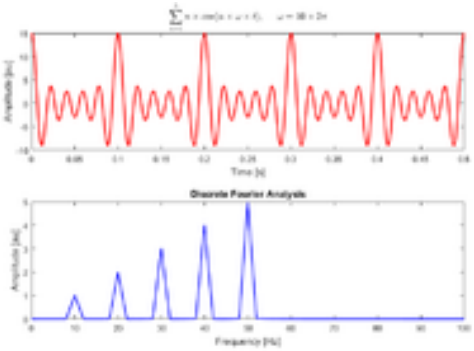
2. Transformada de Fourier discreta:

Es un tipo de transformada discreta utilizada en el análisis de Fourier. Transforma una función matemática en otra, obteniendo una representación en el dominio de la frecuencia, siendo la función original una función en el dominio del tiempo. La DFT requiere que la función de entrada sea una secuencia discreta y de duración finita. Dichas secuencias se suelen generar a partir del muestreo de una función continua, como puede ser la voz humana. Esta transformación únicamente evalúa suficientes componentes frecuenciales para reconstruir el segmento finito que se analiza. Utilizar la DFT implica que el segmento que se analiza es un único período de una señal periódica que se extiende de forma infinita; si esto no se cumple, se debe utilizar una ventana para reducir los espurios del espectro. Por la misma razón, la DFT inversa (IDFT) no puede reproducir el dominio del tiempo completo, a no ser que la entrada sea periódica indefinidamente. Por estas razones, se dice que la DFT es una transformada de Fourier para análisis de señales de tiempo discreto y dominio finito. La entrada de la DFT es una secuencia finita de números reales o complejos, de modo que es ideal para procesar información almacenada en soportes digitales. En particular, la DFT se utiliza comúnmente en procesado digital de señales y otros campos relacionados dedicados a analizar las frecuencias que contiene una señal muestreada. Un factor muy importante para este tipo de aplicaciones es que la DFT puede ser calculada de forma eficiente en la práctica utilizando el algoritmo de la transformada rápida de Fourier o FFT (Fast Fourier Transform).

3. Transformada rápida de Fourier:

Es un algoritmo eficiente que permite calcular la transformada de Fourier discreta (DFT) y su inversa. La FFT es de gran importancia en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital en general a la resolución de ecuaciones en derivadas parciales o los algoritmos de multiplicación rápida de grandes enteros. Cuando se habla del tratamiento digital de señales, el algoritmo FFT impone algunas limitaciones en la señal y en el espectro resultante ya que la señal muestreada y que se va a transformar debe consistir de un número de muestras igual a una potencia de dos. La mayoría de los analizadores de FFT permiten la transformación de 512, 1024, 2048 o 4096 muestras. El

rango de frecuencias cubierto por el análisis FFT depende de la cantidad de muestras recogidas y de la proporción de muestreo. La transformada rápida de Fourier es de importancia fundamental en el análisis matemático y ha sido objeto de numerosos estudios. La aparición de un algoritmo eficaz para esta operación fue un hito en la historia de la informática.



Fuentes:

- https://es.wikipedia.org/wiki/Transformada_de_Fourier
- https://es.wikipedia.org/wiki/Transformada_de_Fourier_discreta
- https://es.wikipedia.org/wiki/Transformada_r%C3%A1pida_de_Fourier

Librería arduinoFFT.h

Realizada por Enrique Condes, es una de las librerías que nos permiten aplicar la Transformada Rápida de Fourier en Arduino. Esta es la librería utilizada en este proyecto. Cuenta con las siguientes funciones:

- **arduinoFFT**(void);
- **arduinoFFT**(double *vReal, double *vImag, uint16_t samples, double samplingFrequency); Constructor
- **~arduinoFFT**(void); Destructor
- **ComplexToMagnitude**(double *vReal, double *vImag, uint16_t samples);
- **ComplexToMagnitude**();
- **Compute**(double *vReal, double *vImag, uint16_t samples, uint8_t dir);
- **Compute**(double *vReal, double *vImag, uint16_t samples, uint8_t power, uint8_t dir);
- **Compute**(uint8_t dir); Calcula la Transformada Rápida de Fourier.
- **DCRemoval**(double *vData, uint16_t samples);

- **DCRemoval**(); Remueve el componente DC de la muestra.
- **MajorPeak**(double *vD, uint16_t samples, double samplingFrequency);
- **MajorPeak**(); Busca y devuelve el valor de la frecuencia con mayor energía de la señal.
- **Revision**(void); Revisión.
- **Windowing**(double *vData, uint16_t samples, uint8_t windowType, uint8_t dir);
- **Windowing**(uint8_t windowType, uint8_t dir); Aplica una ventana sobre el array de valores. Las opciones posibles son:
    - FFT_WIN_TYP_RECTANGLE
    - FFT_WIN_TYP_HAMMING
    - FFT_WIN_TYP_HANN
    - FFT_WIN_TYP_TRIANGLE
    - FFT_WIN_TYP_NUTTALL
    - FFT_WIN_TYP_BLACKMAN
    - FFT_WIN_TYP_BLACKMAN_NUTTALL
    - FFT_WIN_TYP_BLACKMAN_HARRIS
    - FFT_WIN_TYP_FLT_TOP
    - FFT_WIN_TYP_WELCH
- **Exponent**(uint16_t value); Calcula el logaritmo de base 2 del valor dado.

Fuente: https://github.com/kosme/arduinoFFT (en inglés)

Multiplexing

Al inicio del proyecto, unos de los problemas que tuvimos que plantear era la falta de entradas en el Arduino para la cantidad de leds que queríamos utilizar, además de la necesidad de reducir el código lo máximo posible, así llegamos a la técnica del multiplexing, que consiste en utilizar una sola salida para poder controlar más de un elemento a la vez. En este caso con los leds utilizamos la técnica de charlieplexing, que fue propuesta por Charlie Allen en el año 1995 y permite controlar una matriz de Leds con muy pocos pines.

En los pines digitales de Arduino, solo podemos tener dos estados alto (HIGH) o bajo (LOW). La técnica del Charlieplexing consiste en utilizar un tercer estado que nos permita seleccionar un LED concreto. Utilizamos los pines digitales e incorporamos un nuevo estado que nos permite multiplexar, ahora los pines pueden estar en 3 estados HIGH, LOW e INPUT.

Un pin digital de Arduino puede estar en modo salida (OUTPUT) lo que indica que podemos poner en estado alto (HIGH), que suministra 5V, o en estado bajo (LOW), que suministra 0V.

Pero también puede estar en modo entrada (INPUT) lo que indica que podrá leer en que estado está el elemento conectado, alto (HIGH) o bajo (LOW).

Cuando ponemos un pin en modo INPUT, este se encuentra en estado de alta impedancia, que es una medida de resistencia en circuitos de corriente alterna, en circuitos de corriente continua la impedancia es la resistencia de ese circuito. La entrada digital que este en modo INPUT se comporta como una resistencia. Si se aplica la Ley de Ohm para calcular la intensidad que circula por esta entrada, es prácticamente despreciable, es como tener un cable desconectado, no circula intensidad.

Así logramos con tres pines manejar seis leds (al final del documento se puede ver una tabla donde vienen los números de pines necesarios para cada cantidad de leds), el único inconveniente es que no es posible encender todos los leds a la vez, para eso deberíamos usar un microchip.

A la hora de programar, se hace una función definiendo los tres estados (ponerEstados) y otra función con los estados que tiene que tener cada led para encender el que queremos (encenderLed). Tenemos que tener en cuenta que la función encenderLed debe repetirse para cada una de las columnas que vayamos a construir:

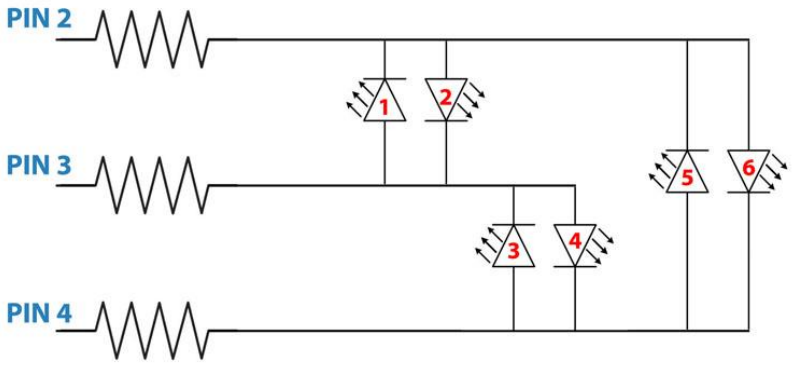
Un código de ejemplo:

```
void encenderLed1 (int led_num)
{
  switch(led_num)
  {
    case 1:
      ponerEstados(PIN_A,PIN_B,PIN_C)
      break;
    case 2:
```

```
        ponerEstados(PIN_B,PIN_A,PIN_C);
        break;
      case 3:
        ponerEstados(PIN_B,PIN_C,PIN_A);
        break;
      case 4:
        ponerEstados(PIN_C,PIN_B,PIN_A);
        break;
      case 5:
        ponerEstados(PIN_A,PIN_C,PIN_B);
        break;
      case 6:
ponerEstados(PIN_C,PIN_A,PIN_B);
        break;
      }

void ponerEstados(int pinHigh, int pinLow, int pinInput)
{
  pinMode(pinHigh,OUTPUT);
  digitalWrite(pinHigh,HIGH);
  pinMode(pinLow,OUTPUT);
  digitalWrite(pinLow,LOW);
  pinMode(pinInput,INPUT);
}
```

Por último, se sueldan los leds siguiendo el siguiente esquema:



Nº PINS	LEDS CHARLIEPLEXING
1	0
2	2
3	6
4	12
5	20
6	30
N	n^2 - n

Desarrollo del proyecto y explicación del código

En nuestro proyecto hay interacción entre el microprocesador Arduino y el ordenador, por lo tanto, es necesario programar en dos entornos diferentes: Arduino (para el procesamiento de datos) y Processing (para representar en pantalla los datos enviados desde el Arduino).

Arduino

En la primera parte del código, se incluye la biblioteca arduinoFFT.h, que nos permitirá aplicar la Transformada Rápida de Fourier. También se incluye la biblioteca SoftwareSerial.h, que nos permite utilizar otros pines además del 0 y el 1 para comunicarnos a través del puerto serie.

A continuación se definen los pines utilizados para controlar los Leds.

```
#include "arduinoFFT.h"
#include <SoftwareSerial.h>
#define N 128//numero de muestras (debe ser potencia de 2)
#define F 8000  //frecuencia de muestreo (debe ser inferior a 10000 por limitaciones del ADC de Arduino)

#define PIN_A 2
#define PIN_B 3
#define PIN_C 4

#define PIN_D 7
#define PIN_E 6
#define PIN_F 5

#define PIN_G 8
#define PIN_H 9
#define PIN_I 10

#define PIN_J 22
#define PIN_K 24
#define PIN_L 26

#define PIN_M 28
#define PIN_N 30
#define PIN_O 32

```

En el setup se incluye un código para comprobar que todos los Leds encienden correctamente. Debido a que utilizamos la técnica Charlieplexing, es necesario usar funciones especiales para encender los Leds. Al final del setup se incluye la instrucción while(!Serial); que garantiza que el puerto serie esté conectado al entrar al loop .

```
void setup() {
   Serial.begin(9600);
    int i;
    //  para ver si se encienden bien los LEDs(matriz)
    for(int i=1; i<=6; i++)
    {
        encenderLed1(i);
        delay(100);
        encenderLed2(i);
        delay(100);
        encenderLed3(i);
        delay(100);
        encenderLed4(i);
        delay(100);
        encenderLed5(i);
        delay(100);
        encenderLed6(i);
        delay(100);

}
    while(!Serial); //Espera a que el puerto esté listo
}
```

El primer paso es tomar muestras con el micrófono. Al multiplicar por 2 el valor de analogRead(A0) estamos amplificando la señal de entrada. A continuación se le aplica un filtro de paso bajo para reducir el ruido presente en la señal.

Justo después de la declaración de las variables utilizadas en el loop, hay un bucle for que toma 10 muestras. Decidimos usarlo, ya que al inicializar la variable pb a 0, los primeros datos obtenidos eran muy bajos, crecían, y luego de unas cuantas muestras tendían a estabilizarse. Con esto logramos que los datos que luego serán usados en la FFT sean más fieles a la realidad.

```
61  void loop() {
62    int i,j;
63    int ledsc=6; //Número de LEDs por columna
64    int ledsf=6; //Número de LEDs por fila
65    double hz;
66    double vReal[N];
67    double vImag[N];
68    float pb = 0.0;
69    float mic,alfa = 0.5;
70
71    //Tomamos 10 muestras para que los datos se estabilicen
72
73    for(i=0;i<10;i++)
74    {
75      mic = 2*analogRead(A0);
76      pb = alfa * mic + (1 - alfa) * pb;
77    }
78
79    for(i=0; i<N; i++)
80    {
81      mic = 2*analogRead(A0);
82      pb = alfa * mic + (1 - alfa) * pb;
83
84      vReal[i]=pb;
85      vImag[i] = 0;
86    }
```

El resultado de realizar las siguientes tres instrucciones es la transformada de Fourier. En el vector vReal se encuentran los valores de la amplitud o energía de la onda. La variable hz, en cada iteración del bucle for, devuelve el valor correspondiente de frecuencia a cada elemento del vector vReal.

Posteriormente, se cambian de escala los valores de vReal para que vayan de 0 a 6 (número máximo de leds por fila que posee nuestro analizador de espectro) para facilitar el control de los leds.

```
88    FFT.Windowing(vReal, N, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
89    FFT.Compute(vReal, vImag, N, FFT_FORWARD);
90    FFT.ComplexToMagnitude(vReal, vImag, N);
91
92    for (i = 0; i < N/2; i++)
93    {
94      hz = ((i * 1.0 * F) / N);
95
96      vReal[i]= constrain(vReal[i], 0, 50);
97      vReal[i] = map(vReal[i], 0, 50, 0, ledsf);
98    }
```

Las siguientes instrucciones son las encargadas de la comunicación del microprocesador con el ordenador a través del puerto serie. La instrucción Serial.write(50); actúa como protocolo de comunicación: “avisa” al ordenador que luego de ese byte, se envía la información que nos interesa. Lo mismo sucede con Serial.write(51);. Del vector vReal queremos enviar 64 datos. Como el buffer de Arduino es de 64 bytes, enviamos los datos en dos partes, para evitar problemas de comunicación.

```
100     Serial.write(50);
101     for(i=0;i<32;i++)
102     {
103       Serial.write(int (vReal[i]+1));
104     }
105
106     Serial.write(51);
107     for(i=0;i<32;i++)
108     {
109       Serial.write(int (vReal[32+i]+1));
110     }
```

Al inicio del programa definimos que tomaríamos 128 muestras con las que calcularíamos la FFT. Como resultado, obtenemos 128 datos: 64 frecuencias y sus 64 amplitudes correspondientes. Con las columnas de leds representamos las frecuencias. Como solo usamos 6 columnas, hay que modificar los datos para poder representarlos a través de los leds. Para ello, reducimos las 64 frecuencias obtenidas a 6 intervalos, y para determinar su amplitud correspondiente, calculamos la media de las intensidades para cada intervalo obtenido.

```
111   //Adaptamos los datos que tenemos para poder representarlos en la matriz de leds
112
113   double media[ledsc];
114
115   int intervalo = (N/2)/ledsc;
116   int c=0;
117   int k;
118
119      for(i=0; i<(N/2); i+=intervalo)
120      {
121        media[c] = 0;
122        for (k=0 ; k< intervalo ; k++) {
123            media[c] = media[c] + vReal[i+k];
124        }
125        media[c] = media[c]/intervalo;
126        c++;
127      }
```

Finalmente encendemos los leds correspondientes utilizando bucles for y las funciones especiales para controlarlos. Utilizamos un bucle for que engloba a todos los demás porque los leds se encendían y apagaban muy rápidamente, por lo que no se percibían adecuadamente los cambios que ocurrían. Este bucle for lo que hace es encender y apagar cada led 100 veces, para que sea más fácil apreciar el funcionamiento de nuestro analizador de espectro.

```
129     //Encendemos los leds
130
131      for(j=0;j<100;j++)
132      {
133       for(i=0;i<media[0];i++)
134       {
135         encenderLed1(i);
136       }
137
138       for(i=0;i<media[1];i++)
139       {
140         encenderLed2(i);
141       }
142       for(i=0;i<media[2];i++)
143       {
144         encenderLed3(i);
145       }
146
147       for(i=0;i<media[3];i++)
148       {
149         encenderLed4(i);
150       }
151       for(i=0;i<media[4];i++)
152       {
153         encenderLed5(i);
154       }
155
156       for(i=0;i<media[5];i++)
157       {
158         encenderLed6(i);
159       }
```

Processing

En primer lugar se incluye la biblioteca processing.serial para poder establecer comunicación con el microprocesador Arduino.

Se define el vector vReal, que contendrá los datos enviados por el Arduino.

En el setup definimos por qué puerto se lleva a cabo la comunicación con Arduino. Es necesario asegurarse que este puerto es el mismo al que está conectado el Arduino, además, la frecuencia de comunicación (9600 bits por segundo) debe ser igual tanto en el programa de Arduino como en el de Processing. Por último, se define el tamaño de la ventana donde aparecerá la representación de los datos recibidos.

```
1  import processing.serial.*;
2  Serial serial;
3
4  int vReal[]=new int [64];
5
6  void setup()
7  {
8    printArray(Serial.list ());
9    serial=new Serial (this, Serial.list()[0],9600);
10   size(700,640);
11 }
12
```

En el draw, primero debemos asegurarnos de que hay suficientes datos en el buffer como para empezar a hacer una lectura. Por este motivo se pide que serial.available sea mayor que 64, que es el número total de datos que queremos recibir cada vez.

Luego de verificar el buffer, se comprueba que se recibe el valor ‘2’ (es la representación en el código ASCII del 50, valor enviado por el Arduino). Si se recibe el ‘2’, es porque los siguientes datos son los valores de la amplitud enviados por el Arduino. Para evitar problemas con el buffer, se envían los datos en dos grupos. El procedimiento es igual para el segundo bloque de datos, pero recibiendo el valor ‘3’ (representación en el código ASCII del 51).

```
13 void draw()
14 {
15   int i,j;
16
17   if(serial.available()>64)
18   {
19     while(serial.read()!='2');  //Esperamos a recibir este dato por parte de Arduino para guardar los valores
20     for(i=0;i<32;i++)
21     {
22       vReal[i]=serial.read();
23     }
24     while(serial.read()!='3');
25     for(i=0;i<32;i++)
26     {
27       vReal[i+32]=serial.read();
```

Luego de rellenar el vector vReal, podemos hacer una representación gráfica de los datos. Para ello, utilizando dos bucles for anidados, se dibujan 64 rectángulos, cuya longitud depende de los valores de vReal. Cada uno de los 64 rectángulos representa una frecuencia distinta. Por la manera en que tomamos los parámetros, el rango de frecuencias mostrado va desde 0 hercios hasta un máximo de 4000 hercios, en intervalos de aproximadamente 62 hercios.

```
30
31    //Representamos gráficamente los resultados
32
33    background(255);
34    fill(0,100,100);
35
36    for(i=0;i<64;i++)
37    {
38      for(j=0;j<=vReal[i];j++)
39        rect(0, i*10, 100*j, 10);
40    }
41  }
```

La imagen mostrada a continuación es un ejemplo de lo que se observa al ejecutar el programa de Processing ya comunicándose con Arduino: