Síntesis FM

Antonio Bonafonte

En esta práctica:

- Se introduce la síntesis de música mediante Frecuencia Modulada
- Se utilizará para el *vibratto* y para implementar varios varios instrumentos virtuales.
- Se entenderá qué son los mensajes *MIDI* mediante una notación simplificada.

1. Fundamento de la síntesis FM

En esta práctica, se implementará un *instrumento virtual* basado en modulación de frecuencia, tal y como fue propuesto por John M. Chowning en el artículo

> *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*, Journal of the Audio Engineering Society, vol 21, no 7, 1973.

En general, el sonido asociado a una nota está formada por la fundamental y sus parciales (armónicos).

$$x(t) = E(t) \cdot \sum_{k=1}^{\infty} A_k \sin(2\pi k f_0 t + \phi_k)$$

Prescindiendo de la envolvente temporal ($E(t)$), el valor de f_0 es la frecuencia de la nota y las relaciones entre las amplitudes de los *parciales*, A_k, define el *timbre*.

- La síntesis de mejor calidad utiliza grabaciones de instrumentos. Aunque los sistemas más sencillos guardan sólo unos pocos periodos, que pueden reproducirse a distintas velocidades para obtener distintas frecuencias, las mejores librerías incluyen varias grabaciones de cada nota, según variantes interpretativas. Por ejemplo, algunas librerías para piano incluye más de 1000 grabaciones (para un total de 88 notas).
- Los métodos aditivos, implementan la ecuación anterior, *sumando* las contribuciones de M armónicos. En polifonía, con varias notas y varios instrumentos, el número de tonos a generar es elevado, lo que influye en el coste computacional
- Los métodos sustractivos parten de una señal periódica rica en armónicos (como una triangular, o un diente de sierra) y se le aplican filtros para modificar las amplitudes de los armónicos.

- La síntesis FM es un método propuesto hace casi 50 años pero que permite generar sonidos muy distintos con una computación muy limitada. El trabajo de Chowning ilustra distintas configuraciones para lograr distintos instrumentos, tales como de viento de madera o metal, etc.

Vamos a analizar el sistema más básico del artículo:

$$x(t) = A \sin\left(2\pi f_c t + I \sin(2\pi f_m t)\right)$$

siendo f_c la frecuencia *portadora*, f_m, la frecuencia moduladora y I el índice de la modulación.

La frecuencia instantánea de esta señal es

$$\frac{1}{2\pi}\frac{d}{dt}(2\pi f_c t + I \sin(2\pi f_m t)) = f_c + I f_m \cos(2\pi f_m t)$$

La idea inicial de Chowning era producir *vibrato*, con frecuencia instantánea oscilando entre $f_c \pm I f_m$. Valores típicos de f_m son unos pocos Hz ($\approx$ 5Hz), y de la extensión de vibrato (ν), expresada en semitonos, entre 0.1 y 1. La relación entre el índice de modulación I y los parámetros del vibrato es:

$$\nu = 12 \log_2 \frac{f_c + I f_m}{f_c - I f_m} \quad \Rightarrow \quad I = \frac{f_m}{f_c} \cdot \frac{2^{\nu/12} - 1}{2^{\nu/12} + 1}$$

Por ejemplo, si $\nu = 1$ semitono, $f_m = 5$Hz, $f_c = 440$Hz, el valor de I es $3.28\,10^{-4}$.

Chowning observó que al incrementar el valor de I, la percepción ya no es de un *vibrato*, sino otro tipo de sonido. En particular, veremos que tomando $f_m = M f_c$ se pueden generar sonidos armónicos con distintos timbres.

Para calcular el espectro de $x(t)$ la expresamos como $x(t) = \mathrm{Im}\{x_c(t)\}$, siendo

$$x_c(t) = A\, e^{j\left(2\pi f_c t + I \sin(2\pi f_m t)\right)} = A\, e^{j2\pi f_c t}\, e^{jI \sin(2\pi f_m t)}$$

Como la señal $e^{jI \sin(2\pi f_m t)}$ es periódica, $T_m = \frac{1}{f_m}$, se puede expresar como una serie de Fourier:

$$e^{jI \sin(2\pi f_m t)} = \sum_{k=-\infty}^{\infty} c_k\, e^{j2\pi k f_m t}$$

siendo $c_k = J_k(I)$, la función de Bessel de orden k, evaluada en I:

$$\begin{aligned} c_k &= \frac{1}{T_m} \int_{-\frac{T_m}{2}}^{\frac{T_m}{2}} e^{jI \sin(2\pi f_m t)}\, e^{-j2\pi k f_m t} \mathrm{dt} \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j(I \sin(\tau) - k\tau)} \mathrm{d}\tau \\ &= \frac{1}{\pi} \int_{0}^{\pi} \cos(k\tau - I \sin\tau) \mathrm{d}\tau \equiv J_k(I) \end{aligned}$$

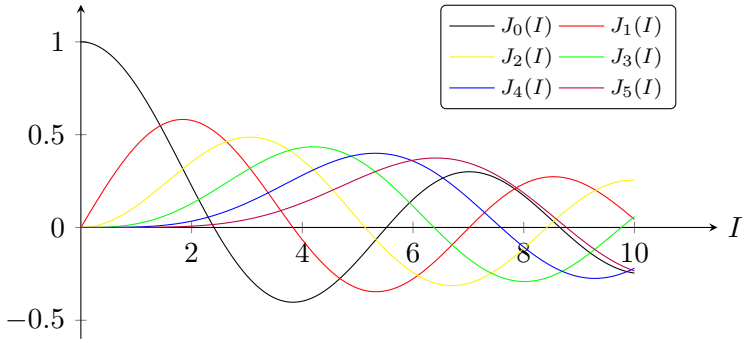
por tanto,

$$x_c(t) = A\, e^{j2\pi f_c t} \cdot \sum_{k=-\infty}^{\infty} J_k(I)\, e^{j2\pi k f_m t} = A \sum_{k=-\infty}^{\infty} J_k(I)\, e^{j2\pi (f_c + k\, f_m) t}$$

$$x(t) = \mathrm{Im}\{x_c(t)\} = A \sum_{k=-\infty}^{\infty} J_k(I)\, \sin 2\pi (f_c + k\, f_m) t$$

Como vemos, aparecen componentes frecuenciales en $f_c + k\, f_m$. Por ejemplo, si seleccionamos $f_m = f_c$, en $f_c(1+k) \quad \forall k$.

La siguiente figura muestra las 6 primeras funciones de Bessel (correspondiente a los 6 primeros componentes frecuenciales) en función del índice de modulación I. Se muestra la parte positiva pues las funciones de Bessel de orden par son pares, e impares las de orden impar.



Si $I = 0$, lógicamente todos los componentes frecuenciales salvo la fundamental son nulos. Pero según crece I, los otros componentes son mayores, el sonido es más *brillante*. La gráficas 1 del artículo citado muestran como es el espectro para distintos valores de I. Un resultado que se indica: el ancho de banda (efectivo) de la señal puede aproximarse por $2\, f_m\, (1 + I)$

En el artículo de Chowning, para simular instrumentos de viento de madera se utilizan relaciones

$$\frac{f_c}{f_m} = \frac{N_1}{N_2}$$

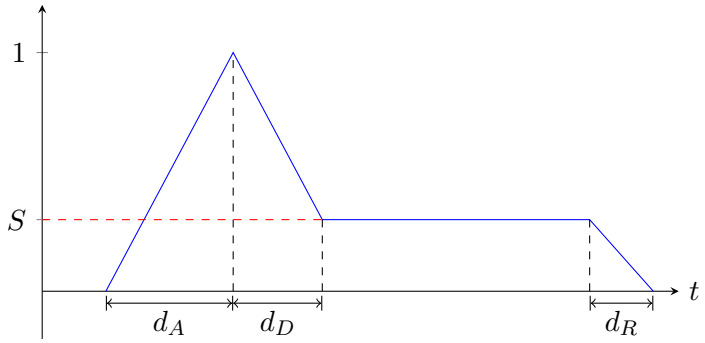
¿Cuál sería la relación de f_c y f_m con la frecuencia fundamental?

Una opción que se contempla es que I cambie con el tiempo, con lo que el ancho de banda del sonido, cambia según avanza la nota (típicamente, mayor en el ataque, menor al disminuir la potencia). Para ello se proponen esquemas (como el de la figura 10) así como valores concretos de los parámetros. para producir distintos instrumentos.

2. Envolvente ADSR

Un componente básico de los sintetizadores son las envolventes. El uso más evidente es marcar la evolución temporal. Por ejemplo, en un instrumento de cuerda, la nota se produce mediante pulsación. A partir de allí no hay aporte de energía, por lo que la amplitud de la nota decae. Sin embargo, un instrumento de viento, puede mantener la aportación de energía durante toda la nota, por lo que el perfil temporal será distinto.

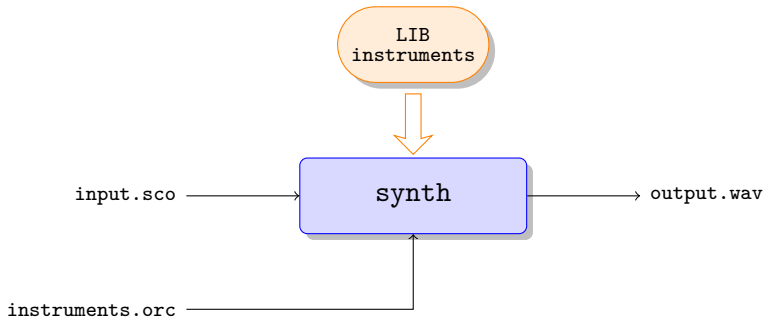
Existen distintas envolventes. En la práctica se implementa una clásica, la *ADSR*, iniciales de *attack, decay, sustain and release*. La figura muestra esta envolvente, formada por tramos rectos. Los parámetros d_A, d_D, y d_R son las duraciones (en segundos) de la distintas partes de la nota. La parte sostenida tiene amplitud S (el valor máximo es 1) y la duración de esta parte, con una interpretación en directo o mediante fichero MIDI viene determinada por el comando OFF de la nota.



Hay que indicar que estas envolventes se utilizan no sólo para amplitudes sino para controlar la evolución temporal en otros parámetros. Por ejemplo, en un sistema sustractivo, el resultado de la envolvente puede ser la frecuencia de corte de un filtro paso bajo, lográndose relaciones entre los armónicos que cambian con el tiempo. O en la síntesis FM, si se aplica al índice de modulación, pueden obtenerse un contenido espectral más rico al inicio de la nota e irse perdiendo armónicos a medida que la nota se atenúa.

3. Sintetizador *PAV*

El esquema del sistema que utilizaremos en la práctica se muestra en la siguiente figura:



El sintetizador se apoya en una librería de *instrumentos virtuales*. El objetivo de la práctica es implementar un *instrumento* de síntesis FM.

El fichero de entrada (`.sco`) contiene la partitura (*score*) a sintetizar en una notación simplificada basada en MIDI. En este fichero se indica cuando se inicia y finaliza cada nota, el *volumen* de cada nota y el *canal* MIDI que debe gestionarlo.

El fichero de entrada `instruments.orc` es un fichero de configuración de la *orquesta*: indica parámetros de configuración para cada instrumento virtual, así como la asignación entre cada uno de los *canales* MIDI que aparecen en la partitura y el instrumento que se utiliza para su síntesis.

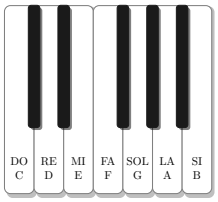
4. Código MIDI

Los ficheros `.sco` son una versión simplificada de los ficheros MIDI estándard. Al igual que estos últimos, los ficheros `.sco` están compuestos por *mensajes* MIDI, uno por línea, y constan de 5 campos (5 columnas): tiempo, evento, canal, parámetro1, parámetro2:

1. Tiempo. Es el instante temporal en que se produce un evento. Este tiempo se expresa:
   - de forma incremental, respecto tiempo del mensaje anterior;
   - de forma cuantizada en *ticks*; en nuestro caso se utiliza una resolución de 120 tpq, *ticks per quarter note*, es decir, que un valor de 120 es el valor nominal de una nota *negra*.

   La duración concreta de un *quater note* depende del *tempo* de la canción. En esta práctica es fijo, `BPM=180`, 180 *beats* por minuto.
2. Canal: es un número que indica el instrumento que debe procesar este mensaje, según se especifica en el primer campo del fichero `instruments.orc`.
3. Evento. Los ficheros `.sco` sólo incluyen los dos comandos MIDI más importante: tecla pulsada (inicio de nota) (valor 9), y tecla liberada (fin de nota) (valor 8).
4. Los dos parámetros para este evento son, el código de la nota (escala MIDI), y la *velocidad*, que es el volumen al que ha de reproducirse la nota. Es un número de 0 a 127.

El código de nota MIDI es un número entero correspondiente a cada una de las notas de la escala temperada utilizada en la mayoría de la música occidental actual. Dicha escala divide cada octava en 12 semitonos. La figura muestra las doce teclas de una octava de piano, correspondiente a los 12 semitonos.



La expresión que relaciona el código MIDI (m) y la frecuencia fundamental (f_0) es:

$$m = K + 12 \cdot \log_2 f_0$$

de forma que un incremento en una octava se corresponde a 12 semitonos y siendo K una constante. En concreto, tomando como referencia que el código MIDI de la nota LA4 (440 Hz) es el 69, podemos reescribir la ecuación:

$$m = 69 + 12 \cdot \log_2 \frac{f_0}{440}$$

5. Instrumentos

Los instrumentos se implementan como clases C++ derivadas de la clase `Instrument`, que se muestra a continuación. En las clases derivadas debe implementar las funciones declaradas como virtuales puras: `command()` y `synthesize()`. La clase base, `Instrument` define el interfaz que utiliza el programa principal para interactuar con cualquier instrumento que se implemente.

```
const int BSIZE = 32;
const long SamplingRate = 44100;

class Instrument {
  protected:
    std::vector<float> x; //samples
    bool bActive;
  public:
  Instrument() : x(BSIZE) {}
    virtual void command(long cmd, long note, long velocity=100) = 0;
    virtual const std::vector<float> & synthesize() = 0;
    bool is_active() const {return bActive;}
    virtual ~Instrument() {};
  };

  Instrument * get_instrument(const std::string &name,
             const std::string &str_parameters = "");
};
```

El programa principal llama a la función `synthesize()` de todos los instrumentos activos pidiéndoles un buffer de `BSIZE` muestras y mezcla las generadas por los distintos instrumentos.

Por otra parte, atendiendo a la partitura MIDI, utiliza la función `command()` para informar al instrumento cuándo se inicia o finaliza una nota. La función se llama en el instante indicado en la partitura, y se le pasa los campos MIDI de evento y sus dos parámetros, tal y como se decribe en el apartado anterior.

En muchos sistemas de este tipo, los instrumentos se definen como *plugins*, es decir, software compilado de forma independiente, que el programa el programa al iniciarse detecta y carga. En la implementación que se proporciona, se simplifica la compilación mediante una librerías estáticas (como en las otras prácticas). Verá que tendrá que modificar la función `get_instrument()`, en el fichero `instrument.cpp`, para asociar el nombre que utilice en fichero `instruments.orc` y la clase C++ que lo implementa.

Se proporcional la clase `keyvalue` para facilitar la conversión de los parámetros de configuración que indique en el fichero `instruments.orc` a los tipos `float`, `int`, etc. que necesite la clase.

6. Tareas

1. Compilar los programas que se entregan. Ejecutar el programa `synth` pasándo el fichero `instruments.orc` ((o `.txt`) y `doremi.sco`. Este fichero se ha creado con el script `doremi.pl`. Edite los dos ficheros y entienda qué significan. El instrumento que ahora se utiliza, `InstrumentDumb` utiliza un parámetro `N`. ¿Qué efecto tiene en el audio de salida cambiar el valor de `N`? Localize en el código cómo se usa `N` y justifique este efecto. Entienda el propósito de las tres funciones de los instrumentos: el constructor, `command`, y `synthesize`

2. Cree un nuevo instrumento que *respete* la nota que se recibe a través de la función `command`.

   - Deberá copiar `instrument_dumb.h` e `instrument_dumb.cpp` con otro nuevo nombre y cambiar toda referencia en los ficheros a la clase `InstrumentDumb` a otro nombre que elija. Para que el programa principal pueda indentificar este nuevo instrumento, deberá editar el fichero `instrument.cpp` que es el que relaciona la clase `C++` con el nombre que se escribe en `instruments.orc`.
   - El *instrumento* más sencillo que puede crear es un tono, pero también puede optar por una función triangular o rampa (de media nula). La función `synthesize` genera unas pocas muestras y es necesario que conserve la continuidad entre llamadas a la función. Para ello deberá guardar un contador, o alguna variable relacionada con la *fase* para en cada llamada continuar con la posición anterior (veáse la nota explicativa en el siguiente apartado).

3. Compruebe el funcionamiento de este instrumento con la partitura `doremi.sco`. Pruebe a variar los parámetros de la envolvente temporar ADSR para obtener distintos efectos. Puede probar con otras partituras de las proporcionadas en atenea.

4. A continuación cree un instrumento para la síntesis FM. Decida que parámetros será conveniente poder modificar con facilidad desde `instruments.orc` y cree variables que se leen en el constructor.

5. Configure el sistema para implementar *vibrato* y pruebe con distintas velocidades (f_m) y extensiones (ν).

6. Revise el trabajo de Chowning (pdf incluido en atenea): esquema 9 y 10, figuras 11 y 12, etc. para preparar instrumentos efectivos.

7. Seleccione los parámetros/instrumentos adecuados para algunas de las partituras proporcionadas en atena (o cree su propia composición).

En la memoria comente los *instrumentos* realizados y incluya algunos ejemplos de audio.

Puede consultar las implementaciones en `csound` de la síntesis FM, http://write.flossmanuals.net/csound/b-pitch-and-frequency/ y en general el sistema `csound` para una introducción más extensa a la *programación de audio.*

Nota: Implementación de un tono

La expresión de un tono responde a las ecuaciones

$$x(t) = \sin(2\pi f_0\, t)$$
$$x[n] = \sin(2\pi f_0\, n\, T) = \sin(2\pi F_0\, n)$$

siendo f_0 la frecuencia en Hertz, T el periodo de muestreo, y F_0 la frecuencia discreta ($0 \leq F_0 \leq 0.5$).

El cálculo de expresiones trigonométricas es costoso. Dado que un sintetizador de música trabaja a frecuencias de muestreo elevadas y en ocasiones con varias decenas de tonos simultáneos, existen métodos eficientes para su cálculo. Por ejemplo, un procedimiento habitual es precalcular la tabla sinoidal, con cierta resolución e interpolar para obtener valores intermedios.

No obstante, dada la sencillez de este sistema, y para facilitar la implementación, puede utilizarse la función trigonométrica.

Una primera aproximación es:

```
const std::vector<float> & synthesize() {
  ../..
  for (int n=0; n < x.size(); n++) {
     x[n] = A * sin(2 * M_PI * F0 * n)
  }
  ../..
  return x;
}
```

Sin embargo, recuérdese que esta función se llama repetidamente, en bloques de pocas muestras (32) para tener una baja latencia. Por tanto, lo que se generará es la repetición de las primeras 32 muestras del tono.

Es por tanto necesario conservar el *estado* entre llamadas. Una primera opción sería declarar `n` como variable de la clase (persistente), en vez de variable local de la función, que así conservaría el valor entre llamadas. Pero la implementación anterior no serviría si la frecuencia cambia en el bucle, como por ejemplo en un glisando o trémolo pues cambiar F_0 por F_0' en una muestra provocaría un salto de fase, que dependerá del instante (n) en que se produzca. Una implementación alternativa, que además reduce el número de productos es actualizar el cambio de fase entre muestras, en este caso, sumando $\alpha = 2\pi\, F_0$:

```
const std::vector<float> & synthesize() {
  ../..
  for (int n=0; n < x.size(); n++) {
     x[n] = A * sin(phase);
     phase += alpha;
     while (phase > M_PI) phase -= 2 * M_PI;
  }
  ../..
  return x;
}
```

La inicialización de `alpha = 2 * M_PI * F0` y la fase inicial (`phase=0`), así como la amplitud `A= (float)vol/128.0F` pueden realizarse en el comienzo de cada nueva nota, en la función `command()`.