ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto fin de Carrera Grado

Grado en Ingeniería Informática de Computadores

Fundamentos del Lenguaje Csound

Realizado por
Rafael Escudero Lirio

Dirigido por
Víctor Jesús Díaz Madrigal

Departamento
Lenguajes y Sistemas Informáticos

Sevilla, septiembre de 2020 (v.0.0.1)

Resumen

El lenguaje de programación Csound está principalmente destinado a la síntesis y producción de sonido en un ámbito musical. Es un lenguaje de código abierto, accesible en la plataforma GitHub y su compilador está programado en C, de ahí su nombre. Recibe periódicamente contribuciones de desarrolladores de diferentes partes del mundo y se encuentra en su versión 6.14.0 a la fecha de realización de este trabajo. Es relativamente poco conocido al ser de uso muy específico y al estar toda su documentación dedicada a personas angloparlantes.

Al ser éstas las circunstancias del lenguaje se ha formado a su alrededor una comunidad dedicada que se agranda con el paso del tiempo, con el fin de seguir desarrollando su tecnología y de ahí la justificación del principal motivo del presente trabajo de fin de grado:

Dar a conocer los fundamentos del lenguaje Csound.

Para tal fin, se estructurará el presente trabajo a modo de guía introductoria de uso del lenguaje. Destacando las principales características de éste y priorizando la escalada progresiva de complejidad al decidir el orden de exposición de los conceptos, con intención de favorecer el aprendizaje a medida que se vaya usando el documento. Se priorizará también que el contenido se exponga de manera unitaria para favorecer el uso del presente documento como guía de consulta rápida de conceptos básicos de Csound.

Se exponen a continuación los principales objetivos del presente trabajo:

- Dar a conocer en mayor medida el lenguaje de programación Csound.
- Proporcionar una guía de aprendizaje introductorio al lenguaje Csound.
- Proporcionar un documento de consulta rápida de conceptos del lenguaje Csound.
- Proporcionar ejemplos prácticos de programación usando el lenguaje Csound a modo de demostración capacitiva del lenguaje.
- Compilar una lista de referencias a portales de contenido de Csound de manera ordenada y comentada.

Por último destacar que tecnologías como Csound invitan al trabajo colaborativo e interdisciplinar en distintos ámbitos como son en este caso la informática y la música. Es por ello fundamental dar a conocer sus diferentes usos con el fin último de ampliar el desarrollo de los conocimientos tecnológicos.

Índice general

Índice de figuras

Índice de código

CAPÍTULO 1

Introducción

1.1– ¿Qué es Csound?

Csound es un lenguaje de programación de alto nivel orientado a objetos dedicado a la síntesis, edición y producción de sonido. Su sintaxis es concreta y su compilador está codificado en lenguaje C. Entre los usos prácticos del lenguaje podemos encontrar ejemplos en la página oficial `https://csound.com/projects.html` de proyectos dedicados a la síntesis de música en directo, edición de sonido mediante efectos generados programáticamente y creación de interfaces VST o instrumentos virtuales entre otros.

Podemos referirnos a Csound como un "Compilador de Sonido".

1.2– ¿Por qué usar Csound?

Hay muchas buenas razones para usar Csound. Csound es software libre de código abierto con licencia LGPL, está en constante desarrollo y tiene una comunidad de desarrolladores que crece día a día,proporciona además un punto de conexión entre la disciplina informática y el ámbito de la música y el sonido. Es además una gran herramienta científica al facilitar la exposición y experimentación de conceptos relacionados a las ondas del sonido, con una amplia librería de funciones y objetos útiles para ello. En lo que se refiere a la composición musical, Csound se ha usado principalmente para generar música electrónica a lo largo de su historia aunque podemos encontrar a compositores de cualquier género músical o tipo de instrumento sacándole provecho al lenguaje. No es de extrañar vistas las tendencias actuales en el mundo de la música, donde para triunfar a nivel multitudinario es prácticamente imprescindible contar con un buen productor que sepa embellecer el sonido. Es frecuente ver a usuarios del lenguaje interpretando su música en directo con ayuda directa de éste. Y por último debo destacar que Csound es compatible con los principales sistemas operativos del mercado, desde Windows a iOS pasándo por distrbuciones Linux. Y como veremos más adelante en esta guía, sus funciones pueden llamarse desde el código de otros lenguajes como Python, java o C.

1.3– Breve historia de Csound

De manera resumida, Csound fue desarrollado en un principio por Barry Vercoe (véase la figura 1.1) en 1985 en el MIT[1] Media Lab. Y desde la década de los años 1990, una amplia variedad de desarrolladores ha colaborado a su código abierto, aportando además documentación, ejemplos y artículos sobre el lenguaje.

Figura 1.1: Barry Vercoe

Para hablar de los verdaderos orígenes de Csound debemos remontarnos a la década de los años 1970, a los orígenes de la producción informática de sonido. Matt Mathews creó MUSIC, el primer lenguaje informático para la generación de ondas de audio digital. En éste se basarían sus posteriores iteraciones: Music1, Music2, Music3, Music4, Music4B... Hasta llegar a Music11, desarrollado por el mentado Barry Vercoe, y del cual Csound es sucesor directo. Posteriormente el desarrollo del lenguaje ha continuado gracias a su comunidad con John Fitch de la University of Bath a la cabeza y como dueño del repositorio de código abierto en la plataforma GitHub. En la actualidad se realizan periódicamente las ICSC[2], conferencias dedicadas a Csound a nivel internacional de manera periódica, siendo la última fecha de realización el 27 de septiembre de 2019 en la actualidad del presente documento.

1.4– Las características del lenguaje

A continuación se muestra una lista de las características principales y técnicas del lenguaje:

- Usado para la síntesis, edición y análisis de sonido y música.
- Lenguaje de código abierto.
- Programación funcional.
- Licencia de distribución LGPL.
- Orientado a objetos.
- Compilador programado en lenguaje C.
- 30 años de desarrollo.
- Compatibilidad con otros lenguajes y retrocompatibilidad de versiones.

[1] Massachusetts Institute of Technology
[2] International Csound Conference

1.5– Alternativas a Csound

Voy a hablar de tres principales alternativas a Csound, siendo estos lenguajes dedicados principalmente a la síntesis, edición y análisis del sonido.

Lenguajes Gráficos

- Max/MSP
- PureData

Lenguaje Escrito

- SuperCollider

1.5.1. Max/MSP

Max es un lenguaje de programación gráfico (véase figura 1.2) dedicado a la música y sonido al igual que Csound pero a diferencia de éste, no es de código abierto. Lo desarrolla y mantiene la empresa Cycling '74 y actualmente puede probarse gratuitamente durante los primeros 30 días de uso. Podría decirse que es por excelencia el lenguaje comercial para el desarrollo de aplicaciones comerciales destinadas al ámbito musical.

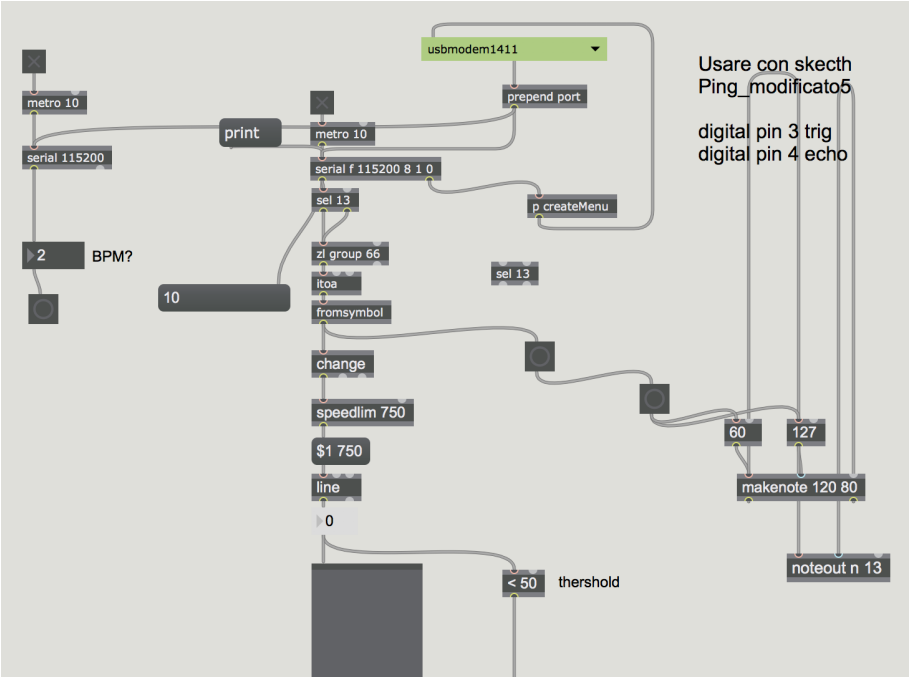

Figura 1.2: Ejemplo Max/Msp

1.5.2. PureData

PureData podría ser considerado el contraparte de código abierto a Max/MSP, pues al igual que éste se trata de un lenguaje gráfico (véase la figura 1.3) pero en esta ocasión de libre desarrollo como Csound. La principal ventaja de este lenguaje respecto a Csound es su paradigma gráfico que puede resultar atractivo a profesionales del ámbito musical o científico que no estén totalmente acostumbrados a usar los tradicionales lenguajes escritos y encuentren en PureData un acercamiento más amigable al mundo de la programación

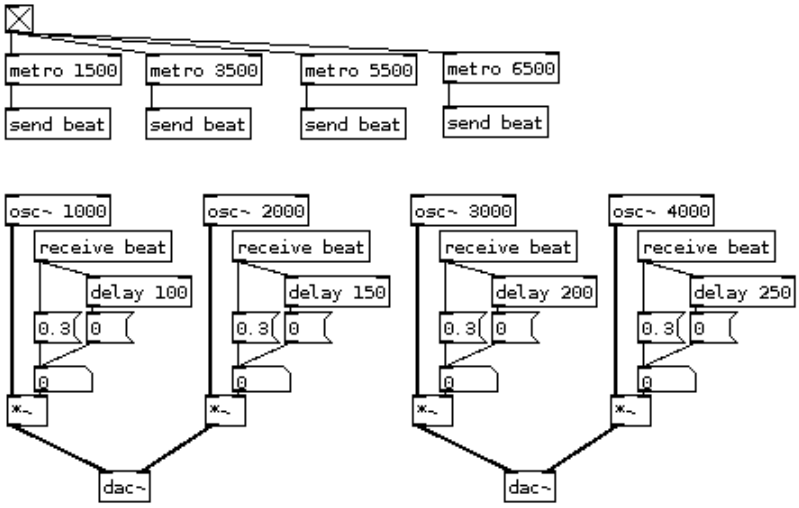

Figura 1.3: Ejemplo PureData

1.5.3. SuperCollider

SuperCollider es de entre las alternativas aquí descritas la más parecida a Csound puesto que además de tener una sintaxis de lenguaje escrito, es también de código abierto. Cuenta además con una sintaxis parecida a lenguajes muy conocidos como C o Ruby, de esta forma parecería más atractivo a iniciados y profesionales de la programación en una primera instancia. Por tanto para observar las verdaderas diferencias entre lenguajes como Csound y SuperCollider tendremos que indagar más a fondo en estos lenguajes y aprender sus características más concretas como haremos en esta ocasión con Csound.

1.6– IDEs para usar Csound

Existen varias alternativas en lo que respecta a entornos de desarrollo de Csound, a continuación se describen 3 opciones[3]:

1.6.1. CsoundQT

CsoundQT es el entorno de desarrollo predeterminado de Csound, prueba de ello es que se instala automáticamente cuando instalamos Csound en el equipo. Tiene una interfaz sencilla, una librería muy completa de ejemplos y capacidad de ampliación con extensiones. Es un buen entorno para tomar una aproximación lo más simplificada posible al lenguaje.

Versión actual: 0.9.6

Plataformas disponibles: Windows, OSX, Debian/Ubuntu.

Página principal: `http://csoundqt.github.io/`

[3]En los ejemplos de código de este documento se ha usado principalmente el IDE Cabbage, no obstante se especificará el IDE usado en cada ocasión en caso de ser necesario por diferencias de uso sustanciales.

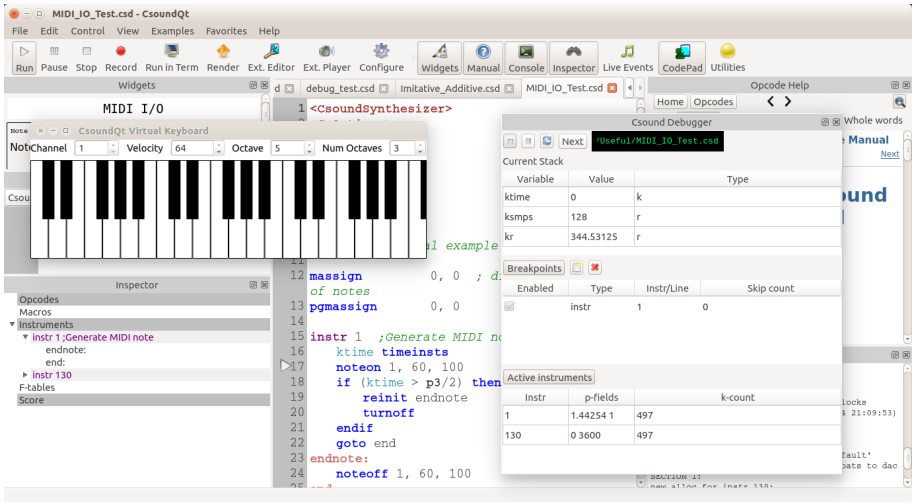

Figura 1.4: CsoundQt

1.6.2. Blue

Blue está enfocado al uso intermedio/avanzado del lenguaje Csound contando con varios plugins por defecto y una interfaz más recargada pero no por ello más difícil de usar o menos legible. Posee varias ventajas tipo framework como los Polyobject, NoteProcessors o el Orchestra manager y es de nuevo una gran opción para tener una primera toma de contacto con el Lenguaje.

Versión actual: 2.8.0

Plataformas disponibles: Windows, OSX, Linux.

Página principal: `https://blue.kunstmusik.com/`

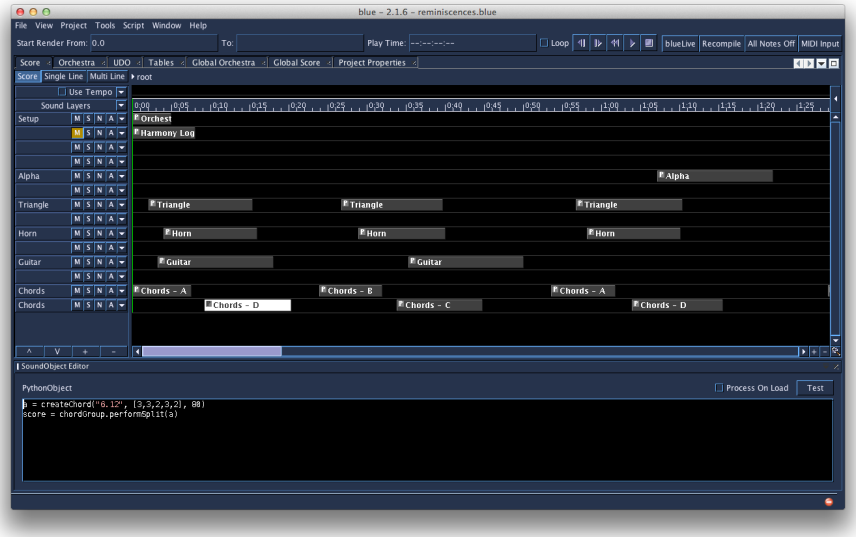

Figura 1.5: Blue

1.6.3. Cabbage

Cabbage es un IDE muy completo, posee una gran librería con utilidades de interfaz gráfica de manera que se facilita el desarrollo completo de instrumentos y plugins musicales existiendo incluso la opción de exportar el código que estamos programando como VST. Cuenta además con una interfaz personalizable y su instalador proporciona la opción de instalar la última versión de Csound si no lo teníamos instalado previamente. Por último cuenta con un instalador para sistemas android para que podamos usar los plugins que programemos con cabbage en estos sistemas.

Versión actual: 2.3.0

Plataformas disponibles: Windows, OSX, Linux.

Página principal: `https://cabbageaudio.com/`

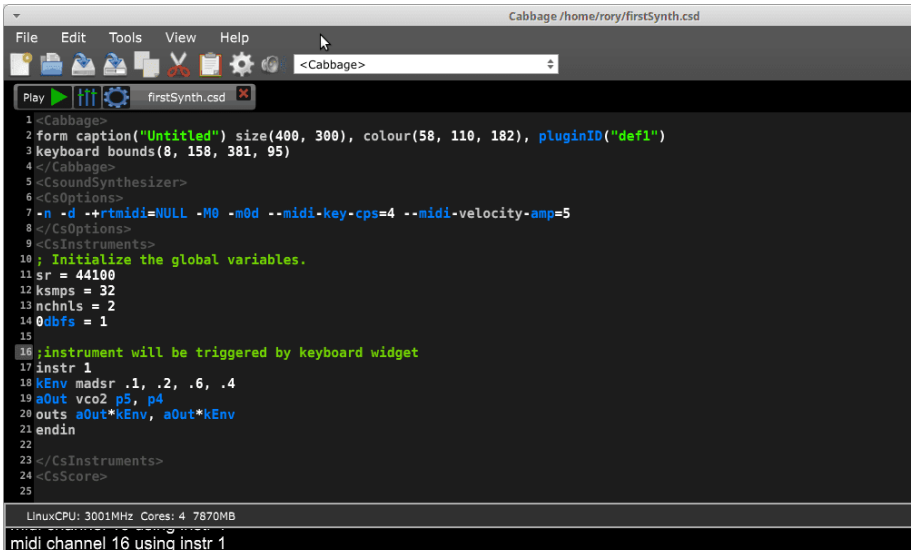

Figura 1.6: Cabbage

CAPÍTULO 2

La Sintaxis del Lenguaje

2.1– Hello Csound!

Empecemos por mostrar cómo sería el clásico *Hello World!* en Csound. Así tendremos un código básico al que hacer referencia a los largo de este capítulo[1]:

```
1  <CsoundSynthesizer>
2  <CsOptions>
3  </CsOptions>
4  <CsInstruments>
5
6  instr 1
7     prints "Hello World!%n"
8     aSin    oscils  0dbfs/4, 440, 0
9     out     aSin
10 endin
11 ;Esto es un comentario
12 </CsInstruments>
13 <CsScore>
14 i 1 0 1
15 </CsScore>
16 </CsoundSynthesizer>
```

Código 2.1: Hello World! en Csound

2.2– A tener en cuenta

Estas son algunas consideraciones básicas del lenguaje que debemos tener presentes:

- Es sensible a mayúsculas/minúsculas.
- Usa especificadores de formato al imprimir variables.
- Para realizar comentarios en el código usaremos ; al inicio de la línea o usar /* y */ para comentar varias líneas.
- Partes del código divididas en etiquetas XML.

[1] Durante el resto del presente capítulo se hará referencia a la figura de código anterior siempre que se hable de líneas concretas de código.

2.3– División por etiquetas

Csound divide la estructura de su código con etiquetas XML, empezaremos por hablar de las más básicas: <CsInstruments>y <CsScore>. Hasta llegar a etiquetas más exclusivas como <Cabbage>.

- **Etiqueta <CsInstruments>**: En esta etiqueta se incluirán las definiciones de los instrumentos que crearemos. Un poco más adelante trataremos de entender qué es un instrumento en Csound. En nuestro ejemplo la etiqueta <CsInstruments>abarca desde la línea 4 a la línea 12, y podemos observar la definición de un instrumentos entre las líneas 6 y 10.

- **Etiqueta <CsScore>**: Aquí haremos uso práctico de nuestros instrumentos, les diremos cómo ejecutarse y durante cuánto tiempo. En el código del que disponemos, la etiqueta <CsScore>abarca desde la línea 13 a la 15 y tenemos un ejemplo sencillo de ejecución en la línea 14 al que volveremos más adelante.

- **Etiqueta <CsOptions>**: Se inclurán aquí las especificaciones técnicas para interactuar con hardware u otros dispositivos.

- **Etiqueta <CsoundSynthesizer>**: Todo el código, incluidas las etiquetas mencionadas anteriormente, debe estar incluido en <CsoundSynthesizer>. Es la forma que tiene el compilador de saber dónde empieza y dónde acaba el código Csound.

- **Etiqueta especial <Cabbage>**: Es la única etiqueta que no debe estar dentro de <CsoundSynthesizer>puesto que es exlusiva de Cabbage, IDE que usaremos principalmente en este documento. En esta etiqueta incluiremos el código referente a las opciones de personalización de interfaz gráfica de nuestro programa, hablaremos más de ella en secciones referentes al uso del IDE Cabbage. Para mayor referencia y guía de uso de esta etiqueta y su contenido, visitar la sección “La etiqueta <Cabbage>”del capítulo “Cabbage: Guía de uso”.

2.4– Palabras reservadas

Las palabras reservadas son variables globales con una funcionalidad especial para Csound como delimitar bloques de código o determinando valores configurables. Pueden inicializarse a un valor determinado en nuestro código que queda posteriormente grabado en el tiempo de compilación. Veamos algunas de las palabras reservadas más comunes en Csound:

- **instr/endin**: Las palabras reservadas instr y endin sirven para determinar el comienzo y el final del bloque de código necesario para crear un instrumento en Csound. Podemos ver un ejemplo de uso en las líneas 6 y 10 de la figura de código 2.1. Además debemos asignar un nombre o identificador al instrumento acompañando a la palabra reservada instr, en nuestro ejemplo de código le damos el nombre “1” a nuestro instrumento.

- **sr**: Indica el valor del sample rate. Wl valor predeterminado es de 44100Hz, es decir, 44100 veces cada segundo. Normalmente usaremos un valor 44100 o de 48000 según la compatibilidad de la tarjeta de sonido de nuestro equipo.[2]

- **nchnls**: Se trata del número de canales de salida de audio que usamos en nuestro programa. Con nchnls = 1 conseguimos sonido mono, con nchnls = 2 stereo, nchnls = 4 cuadrafónico. (véase figura 2.1)

[2]Para mayor entendimiento de los conceptos relacionados al sampleo véase el apartado “El Sampleo y Sample Rate”del capítulo “Fundamentos del Sonido”

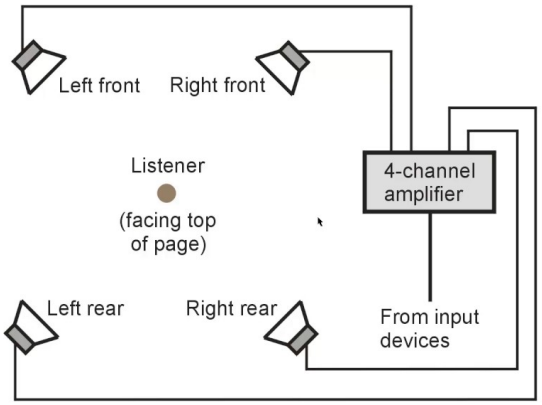

Figura 2.1: Sonido cuadrafónico

- **0dbfs**: Determina el valor relativo que usaremos en el programa como 0 decibelios y a partir del cual aumentaremos o disminuiremos los decibelios de volumen. Por defecto tiene un valor de 32767 aunque lo más lógico al usar un computador para determinar este valor es dar un valor absoluto de 0dbfs = 1 puesto que la intensidad final del sonido no depende únicamente de nuestro código sino de nuestros altavoces, la configuración de nuestra tarjeta de sonido o la configuación de nuestro IDE de Csound. Darle un valor 1 a 0dbfs nos permite tener una referencia útil y muy manejable además de ser la usada por convenio en los programas de audio digital en lugar de dar valores absolutos que bien pueden desembocar en resultados distintos a lo que deseábamos según el equipo en el que ejecutemos nuestro código. [3]

2.5– Las variables en Csound

En Csound encontramos 3 tipos principales de variable: **i**, **k** y **a**, que representan diferentes ratios de refresco para su valor. Para definir una variable de cualquiera de estos tipos basta con escribir i, k o a como primera letra del nombre de nuestra variable. Algunos ejemplos serían: iFreq, kAux, aCualquierNombre, etc...

Veamos a continuación qué implica elegir uno u otro tipo:

Cuando ejecutamos código Csound, una vez compilado el código e inicializada la configuración, el tiempo de ejecución se pasa recorriendo el código contenido de nuestros instrumentos en bucle. Podemos decir que nuestro código se recorre con una determinada frecuencia mientras se ejecuta, y dentro de esta frecuencia podemos seleccionar cómo de frecuentemente volveremos a determinar el valor de nuestra variables. Para determinar ese ratio usamos i,k y a.

- **Variables i**: Las variables refrescan su valor una única vez, cuando el instrumento es inicializado. A efectos prácticos se trata de constantes internas a los instrumentos.
- **Variables k**: Tienes una frecuencia de refresco media. Mayor a las variables i, menor a las variables a.
- **Variables a**: Tienen el mayor ratio de los 3 tipos puesto que estas variables refrescan su valor cada vez que Csound recorre su código.

[3] Para mayor entendimiento del concepto de decibelio, ir a la sección “El decibelio”del capítulo “Fundamentos del Sonido”

La principal razón para que tengamos estas diferente opciones es poder jugar con la optimización de nuestro código. Bien podríamos hacer que todas nuestras variables fuesen de tipo ‘ a’pero esto implicaría en la mayoría de los casos un aumento de carga innecesario durante el tiempo de ejecución, que puede ser crucial si nuestro objetivo pasa por ejecutar código en directo al interpretar una pieza musical.

2.6– Los Instrumentos

Los **instrumentos** serán los bloques de código básicos en los que realizaremos las operaciones de nuestro programa. Podemos tener varios **instrumentos** en la misma etiqueta <CsInstruments>de un programa y a ese conjunto de **instrumentos** lo llamaremos **orquestra**. La sintaxis base de un instrumento es:

```
1 instr 1
2 codigo...
3 endin
```

Código 2.2: Sintaxis generalizada de un Instrumento

Donde **instr** y **endin** marcan el inicio y el final del bloque de código del instrumento. Seguido a la palabra **instr** escribiremos el nombre o identificador (normalmente un número) para referirnos más tarde a nuestro instrumento.

En secciones posteriores hablaremos de la etiqueta <CsScore>y de cómo usar en ella nuestros instrumentos, de momento basta con saber que al llamar a nuestros instrumentos necesitaremos una serie de parámetros a los cuales nos referiremos como **p1**,**p2**,**p3**,**p4**... Cuando veamos estos identificadores dentro del código de los instrumentos sabremos que nos estamos refiriendo a esos parámetros de entrada.

2.7– Los Opcodes

Los Opcodes realizan diferentes funciones predefinidas, son de hecho el equivalente más parecido a una función de librería en un lenguaje de alto nivel convencional. Por ejemplo el Opcode **reverb** aplica reverberación a la señal entrante, el opcode **poscil** produce una señal oscilatoria de alta precisión. Existe una gran variedad de Opcodes (Actualmente más de 1500) con funcionalidades muy concretas y útiles en el mundo del sonido.

2.7.1. Sintaxis del Opcode

Veamos primero la sintaxis generalizada y después un ejemplo sencillo de uso de opcode. Analicemos su sintaxis para comprenderla:

```
1     aOutput opcode input1, input2, input3, ...
```

Código 2.3: Sintaxis generalizada de un Opcode

Refiriéndonos a la figura de código 2.2. **opcode** es el nombre del opcode que queremos usar, **input1**, **inpu2** e **input3** representan los diferentes atrbutos de entrada al opcode, son generalmente valores numéricos y su cantidad es indeterminada. Existen opcodes sin atributos de entrada. Por último **aOutput** es la variable de salida del opcode en la que vamos a guardar nuestro resultado esperado a la que haremos referencia más adelante en nuestro código. Podríamos resumir la sintaxis básica de uso de opcodes en: **salida** ← **función** ← **entrada/s**

```
1     aSin    oscils  0dbfs/4, 440, 0
```

Código 2.4: Ejemplo de uso del Opcode oscils

Observemos un uso práctico del opcode **oscils** en la figura de código 2.3:

- **Salida**: La variable de salida es **aSin** (variable de tipo ‘ a’) en la que quedará guardada la información de la onda generada y cuyo valor se refrescará cada vez que Csound recorra el instrumento contenedor.
- **Función**: El opcode es **oscils** cuya funcionalidad es generar una señal oscilatoria sinoidal, este opcode requiere de 3 variables de entrada de las que hablaremos a continuación.
- **Entradas**: Observamos 3 entradas, **0dbfs/4, 440, 0 → iamp, icps, iphs**. Hablemos de cada una de ellas: 0dbfs/4 (recordemos el uso de 0dbfs como palabra reservada)
  - **iamp**: Cuyo valor es 0dbfs/4. Se trata del valor de la amplitud de onda de salida, es decir, la amplitud de la onda de salida tendrá el valor de un cuarto del valor marcado en nuestro código como 0 decibelios (**0dbfs**).
  - **icps**: Frecuencia de la onda de salida en Hz. El valor en nuestro ejemplo es de 440, por lo tanto obtendremos una frecuencia de onda de salida de 440Hz.
  - **iphs**: Determina la fase de onda de la onda de salida, nuestro valor es 0 por lo que nuestra onda de salida comienza en su fase 0.[4]

Como podemos observar, una vez comprendida, la sintaxis de uso de los opcodes es sencilla pero muy concreta al no parecerse del todo a la sintaxis de lenguajes más convencionales. Para usar de forma satisfactoria los opcodes debemos tener presente la librería de referencia canónica de Csound: `http://www.csounds.com/manual/html/PartReference.html` donde encontraremos un librería muy amplia de ejemplos de uso concreto de cada uno de los opcodes disponibles.

2.8– Los Score events

Los **Score events** son las líneas de código que van dentro de la etiqueta <CsScore>. Por ejemplo, volvamos a nuestro “Hola mundo”(figura de código 2.1), revisemos el score “**i 1 0 1**”y comprendamos su sintaxis:

Para ejecutar un instrumento usaremos la siguiente sintaxis base:

[i identificador instanteInicial duración]

Siendo **identificador** el nombre del instrumento que vamos a usar, **instanteInicial** el tiempo desde el momento de ejecución del código que tarda en iniciarse nuestro instrumento y **duración** la cantidad de tiempo durante la cual se ejecuta el instrumento.

La sintaxis generalizada de un Score event de llamada a instrumento o Score **i** es:

```
1 i p1 p2 p3 p4 ...
```

Código 2.5: Sintaxis base de un Score event i

Donde **p1** será el **identificador**, **p2** el **instanteInicial**, **p3** la **duración** y a partir de **p4** serán variables de entrada opcionales que usaremos según determine el instrumento. Un ejemplo común de variable **p4** sería un valor numérico de variación de tono que se suma o resta a una de las variables internas del instrumento.

[4]Para comprender el concepto de fase de onda, acudir al capítulo “Fundamentos del sonido“ sección “La onda y sus características”

2.9– Usando lo aprendido en un Sintetizador MIDI

Para afianzar lo aprendido sobre la sintaxis de Csound, vamos a programar un sintetizador MIDI básico que haga uso de las funcionalidades más elementales del lenguaje.

2.9.1. El instrumento base <CsInstruments>

Vamos a usar el opcode **vco2**, que genera una onda oscilatoria limitada por banda. Veamos en primer lugar cómo usar este opcode: **ares vco2 kamp, kcps**. Tenemos dos atributos de entrada obligatorios, **kamp** que define la amplitud de la onda y **kcps** para determinar su frecuencia. Para nuestro ejemplo, **aOut** contedrá la información referente a una de amplitud 1 y frecuencia 440Hz. Con la línea **out aOut** se indica que el valor de salida que se devuelve al usar nuestro instrumento es el de **aOut**.

Y aquí nuestro instrumento en su forma base:

```
instr 1
aOut vco2 1, 440
out aOut
endin
```

Código 2.6: Instrumento base del sintetizador

Inicialicemos además algunas variable globales y retoquemos algunas variables internas de nuestro instrumento:

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

Código 2.7: Variables globales inicializadas

Usaremos el opcode de esta manera: **aOut vco2 iAmp, iFreq**, siendo **iFreq = p4** y **iAmp = p5**. Estas variables, p5 y p4, las determinan los valores de entrada MIDI con el objetivo de usar nuestro sintetizador de forma dinámica.

2.9.2. Las opciones de configuración <CsOptions>

Veamos qué opciones de configuración podemos añadir a nuestro sintetizador. Será importante para que el uso de señales MIDI sea funcional así que usaremos la opción **-+rtmidi=NULL -M0 -m0d**. No usaremos plgins MIDI de tiempo real en este ejemplo así daremos el valor **NULL** a **-+rtmidi**, con -M seguido de un identificador podremos seleccionar qué dispositivo vamos a usar. Como sólo tendremos disponible el dispositivo virtual de Cabbage, usaremos **-M0**.

Para relacionar las variables de nuestro instrumento como comentamos anteriormente usaremos **–midi-key-cps=4** para pasar la frecuencia de la nota MIDI actual a **p4** y **–midi-velocity-amp=5** para pasar la velocidad de pulsación a —textbfp5.

Bastará de momento con tener un entendimiento básico de la etiqueta <CsOptions>de momento, aunque profundizaremos en ella más adelante.

Quedaría así nuestro código:

```
-+rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
```

Código 2.8: Opciones de configuración

2.9.3. Ejecutando nuestro instrumento <CsScore>

La etiqueta <CsScore>contiene nuestros **score events** auqnue en esta ocasión no vamos a necesitarlos como tal, únicamente usaremos el score **f0 z** que sirve para indicar a Csound que se quede esperando eventos tanto tiempo como queramos. Esto ideal para el caso puesto que queremos que Csound escuche nuestros eventos de señal MIDI.

2.9.4. Creando una interfaz <Cabbage>

Usaremos dos widgets de forma sencilla[5]. En primer lugar un widget **form** para crear la ventana base de la interfaz, que tendrá un identificador **size(Width, Height)** para definir el tamaño de la ventana, un identificador **colour(r, g, b)** para dar un color personalizado al fondo de la ventana en formato rgb y por último un identificador **pluginid(id)** (único identificador obligatorio) para dar un nombre identificativo a la ventana y poder hacer referencia a ella en el resto del código.

En segundo lugar usaremos un widget tipo **keyboard** para crear nuestro teclado virtual que hará las veces de instrumento MIDI. Tendrá un único identificador **bounds(x, y, width, height)** para determinar la posición de coordenadas y el tamaño del teclado.

Un posible ejemplo quedaría tal que así:

```
1 form caption("Primer Sinte") size(450, 300), colour(250, 110, 20),
     pluginID("sin1")
2 keyboard bounds(30, 150, 380, 100)
```

Código 2.9: La interfaz del teclado

Y el resultado al ejecutar esta parte del código será algo parecido a esto:

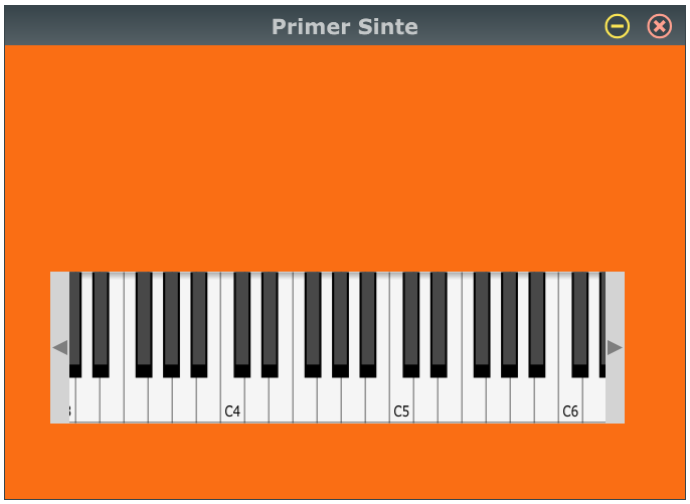

Figura 2.2: Nuestro primer sintetizador

2.9.5. El teclado básico

Si unimos todas las partes, nuestro código queda de esta manera:

```
1 <Cabbage>
2 form caption("Primer Sinte") size(450, 300), colour(250, 110, 20),
     pluginID("sin1")
3 keyboard bounds(30, 150, 380, 100)
4 </Cabbage>
```

[5]Se recomienda revisar la sección “Los widgets”del capítulo “Cabbage: Guía de uso”.

```
<CsoundSynthesizer>
<CsOptions>
-+rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
iFreq = p4
iAmp = p5
aOut vco2 iAmp, iFreq
outs aOut, aOut
endin

</CsInstruments>
<CsScore>
f0 z
</CsScore>
</CsoundSynthesizer>
```

Código 2.10: Un teclado básico funcional

Este teclado es ya ciertamente funcional pero para tener un ejemplo más completo y poder llamar 'sintetizador' a este instrumento vamos a añadirle un envelope ADSR[6]. Así conseguiremos las funcionalidades básicas de cualquier instrumento eléctrico.

2.9.6. Convirtiendo el teclado en un sintetizador

En primer lugar, para añadir un envelope ADSR, podemos seleccionar alguna de las muchas opciones que ofrece Csound en materia de opcodes. Uno de los más sencillos y que puede servirnos perfectamente es el opcode **madsr** cuya sintaxis es **kenv madsr iatt, idec, islev, irel**. Siendo **iatt** el valor de ataque, **idec** el valor de decay, **islev** el valor de sustain u **irel** el valor de release. Recordemos también que las variables i son a efectos prácticos variables constantes del instrumento, que refescarán su valor en este caso entre pulsación y pulsación de una nota.

Bastaría ahora con aplicar este envelope a nuestra onda de salida, una de tantas maneras es multiplicando la salida del opcode **madsr** a los canales de salida de nuestro instrumento de esta forma: **outs aOut*kEnv, aOut*kEnv**.

Necesitamos también dar un valor a los parámetros **iatt**, **idec**, **islev** y **irel**. Podríamos dar valores fijos aunque esto no tendría un uso demasiado funcional, por lo tanto vamos a aprovechar las opciones de interfaz que proporciona Cabbage para vincular estos valores a 4 widgets de tipo slider. Veamos cómo crear el slider para el valor de ataque:

```
rslider bounds(12, 14, 105, 101), channel("ata"), range(0, 1, 0.01, 1,
    .01), text("Ataque")
```

Código 2.11: El slider para el valor de ataque

El widget slider no es nuevo, aunque debe destacarse el identificador **channel(chan)** que se usa para otorgar al slider un nombre de canal al que se vincula su valor de salida y que más tarde podremos usar en nuestro código.

Se omite la especificación de los otros 3 sliders que añadiremos pues su sintaxis es análoga cuidándonos únicamente de dar un nombre de canal distinto en cada ocasión.

[6]Se recomienda revisar la sección "El ADSR"del capítulo "Fundamentos del sonido"

Haremos uso también del sencillo opcode **chnget** para vincular los canales que hemos creado en los sliders con variables que Csound pueda procesar. Su sintaxis es básica: **ival chnget Sname** siendo Sname el nombre del canal asociado.

Quedaría nuestro código de vinculación de la siguiente manera:

```
iAtt chnget "ata"
iDec chnget "dec"
iSus chnget "sus"
iRel chnget "rel"
```

Código 2.12: Vinculación de canales a variables

2.9.7. Últimos detalles

Para concluir vamos a añadir un par de efectos útiles a nuestro sintetizador, el **cutoff** y la **resonancia**. Una forma sencilla de conseguirlo es usando el opcode **moogladder**, cuya sintaxis es: **asig moogladder ain, kcf, kres**. Siendo **ain** la onda a la que añadir los efectos, **kcf** el valor del cuttoff y **kres** el valor de resonancia.

Para el caso vamos a darle a **ain** el valor de aOut, es decir, nuestra onda resultante de usar el opcode vco2. Para los valores de cuttoff y resonancia crearemos simplemente un par de sliders y vincularemos sus valores a variables de Csound del mismo modo que hicimos antes.

2.9.8. El resultado final

Y aquí tenemos al fin el código completo de nuestro sintetizador con efectos:

```
<Cabbage>
form caption("Primer Sinte") size(450, 300), colour(250, 110, 20),
    pluginID("sin1")
keyboard bounds(14, 88, 413, 95)
rslider bounds(12, 14, 70, 70), channel("att"), range(0, 1, 0.01, 1,
    .01), text("Ataque")
rslider bounds(82, 14, 70, 70), channel("dec"), range(0, 1, 0.5, 1,
    .01), text("Decay")
rslider bounds(152, 14, 70, 70), channel("sus"), range(0, 1, 0.5, 1,
    .01), text("Sustain")
rslider bounds(222, 14, 70, 70), channel("rel"), range(0, 1, 0.7, 1,
    .01), text("Release")
rslider bounds(292, 14, 70, 70), channel("cutoff"), range(0, 22000,
    2000, .5, .01), text("Cut-Off")
rslider bounds(360, 14, 70, 70), channel("res"), range(0, 1, 0.7, 1,
    .01), text("Resonance")
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-n -d -+rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
iFreq = p4
iAmp = p5

```

```
25 iAtt chnget "ata"
26 iDec chnget "dec"
27 iSus chnget "sus"
28 iRel chnget "rel"
29 kRes chnget "res"
30 kCutOff chnget "cutoff"
31
32 kEnv madsr iAtt, iDec, iSus, iRel
33 aOut vco2 iAmp, iFreq
34 aLP moogladder aOut, kCutOff, kRes
35 outs aLP*kEnv, aLP*kEnv
36 endin
37
38 </CsInstruments>
39 <CsScore>
40 f0 z
41 </CsScore>
42 </CsoundSynthesizer>
```

Código 2.13: Un sintetizador funcional

Con esto termina la introducción a la sintaxis del lenguaje. Como puede observarse, aunque Csound parezca algo enrevesado a primera vista y sin conociemientos previos, una vez comprendido el esquema básico y la lógica sintáctica es fácil entender y escribir código lo suficientemente elaborado como para obtener instrumentos completamente funcionales sin demasiado esfuerzo. Seguiremos profundizando en el potencial de Csound en posteriores capítulos ahora que hemos alcanzado el nivel fundamental para poder seguir las lecciones sin perdernos.

CAPÍTULO 3

Profundizando en los conceptos básicos

Profundicemos en algunos conceptos que crean normalmente confusión entre los usuarios de Csound.

3.1– Las diferencias entre variables i-rate y k-rate

Es común confundir el uso de variables **i-rate** con el de variables **k-rate**. Vamos a resolver algunas dudas y a exponer ejemplos de casos en los que usar normalmente cada tipo.

En primer lugar debemos entender que, como en cualquier lenguaje al uso, las variables en Csound se inicializan al comenzar la ejecuación. La diferencia radical que podemos encontrar a partir de este momento entre variables **i-rate** y **k-rate** es que las **i-rate** van a quedarse con este valor de inicialización. Esto es fácil de entender pero, ¿Qué pasa entonces con las variables **k-rate**?

La duración de un **k-cycle**, es decir, el tiempo (que puede medirse en cantidad de samples) que pasa desde la útima vez que se refrescaron los valores de las variables **k-rate** hasta la siguiente vez que se refrescan.

Este valor, **k-cycle**, es dinámico. Depende de la cantidad de variables tipo k de nuestro código, del valor que le demos a la palabra reservada **ksmps** y del sample rate seleccionado. Veámoslo con un ejemplos:

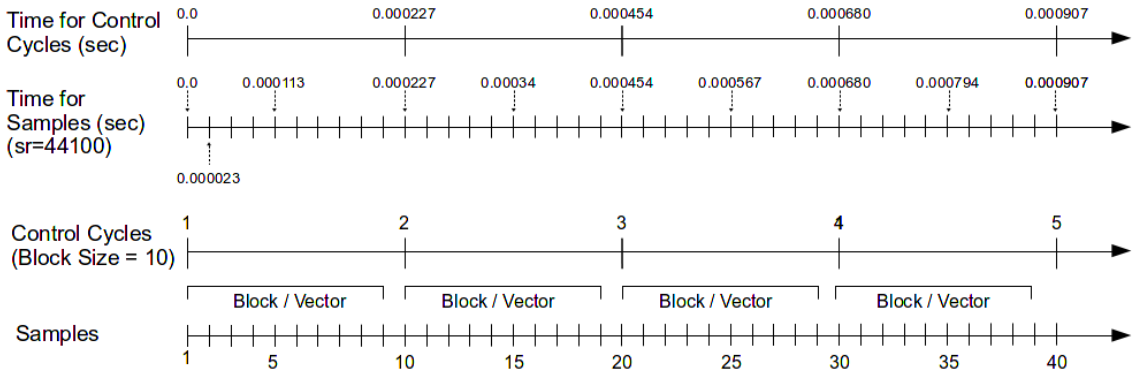

Figura 3.1: La duración de los K-Cycles

Para el gráfico anterior hemos seleccionado un sr (sample rate) de 44100 y un Block Size (cuyo equivalente en Csound sería la palabra reservada **ksmps** que mencionamos antes) con valor 10.

Sabemos que el tiempo que tarda en realizarse una muestra o sample es un segundo dividido por el número total de tomas, es decir:$1/44100$ que da 0.0000227s. Y sabemos también que el **k-cycle** dura 10 samples (también podemos llamarlos ticks), es decir: $0{,}0000227 * 10 = 0{,}000227s$.

Por lo tanto, en un código con la configuración anterior y un valor de **ksmps = 10** podremos decir que:

- Las variables **i-rate** determinarán su valor en la inicialización y lo mantendrán durante el tiempo total de ejecución del código.
- Las variables **k-rate** refrescarán su valor cada 10 samples o ticks (que podemos llamar bloque de control), en este caso, se refrescarán cada 0.000227 segundos.
- Las variables **a-rate** refrescarán su valor un vez por cada sample, siendo el ratio de refresco determinado en su totalidad por la frecuencia escogida como sr. En este ejemplo renuevan su valor cada 0.0000227 segundos.

Tener un conocimiento suficiente acerca del funcionamiento de las variables **k-rate** puede hacer que nuestro código acabe siendo mucho más eficiente y optimizado pero como nota general puede bastarnos con recordar que: Las variables **i-rate** deben usarse cuando sepamos que algo debe ser hecho una única vez y de manera puntual, y las variables **k-rate** deben usarse cuando necesitemos que algo se haga continuamente pero sepamos que ese algo no necesita ser hecho cada vez que se realiza un sample.

3.2– Las f-variables, w-variables y S-variables

Existen dos tipos de variables en csound que son algo más especiales que las vistas hasta ahora:

- **Las f-variables**: Son variables usadas por algunos opcodes (los que empiezan por **pvs**), y se usan principalmente para la realización de *Transformadas rápidas de Fourier*. Su ratio de refresco es el mismo que para las variables **k-rate** pero su valor depende de algunos parámetros de las transformadas que hemos mencionado.
- **Las w-variables**: Podemos encontrar el uso de estas variables en algunos opcodes antiguos aunque su uso es ya prácticamente por razones hereditarios por lo que no profundizaremos en ellas.
- **Las S-variables**: Son variables de tipo **String**, serán necesarias para usar algunos opcodes cuyo resultado sea una variable de este tipo.

3.3– El ámbito global y local de las variables

Las variables contenidas en el código de un instrumento son generalmente de ámbito local, es decir, podría crear una variable con el mismo nombre dentro de todos mis instrumento sin que exista ningún tipo de conflicto.

Las variables globales sin embargo deben ser únicas cada vez que escribamos un valor sobre ellas, su valor cambiará para cualquier futura lectura. Para hacer que una variable sea de tipo global debemos hacer que **g** sea la primera letra del nombre de esa variable. Algunos ejemplo de nombre de variable global serían: **gaGlobal**, **giConstante** o **gkResultado**.

Una alternativa al uso de las variables globales es el uso del opcode **chnget** para realizar conexiones de canal entre variables. Es el método usado en la etiqueta <Cabbage>para relacionar los widgets con el resto del código sin hacer uso de variables globales.

3.4– Las estructuras de control

En Csound como en la mayoría de lenguajes existen la estructuras de control: Sentencias **if-else**, bucles **while/until** y los llamados **timouts**. Vamos a explicarlas centrándonos en la peculiaridades de uso respecto al lenguaje.

3.4.1. Sentencias if-else

La forma más común de este tipo de sentencia en Csound es **If - then - [elseif - then -] else**:

```
1 if <condicion1> then
2 codigo...
3 elseif <condicion2> then
4 codigo...
5 else
6 codigo...
7 endif
```

Código 3.1: Sintaxis base de la sentencia if-else

Lo único a destacar sería que la palabra **then** debe estar en la misma línea de código que la palabra **if**, pero no ahondaremos más en este tipo de sentencia al tratarse de nociones básicas de la programación.

Csound permite también la sintaxis de lenguaje descriptivo **(a v b ? x : y)**: De ser verdadera la condición **a**, el valor devuelto es **x**. De ser falsa (y por tanto ser cierta **b**) el valor devuelto es **y**. Un ejemplo práctico de uso sería: **kRes = (kVar <1 ? 0 : 1);**. Si kVar es menor que uno se devuelve 0, se no ser así se devuelve 1.

3.4.2. Bucles While/Until

```
1 while <condicion> do
2 codigo...
3 od
4
5 until <condicion> do
6 codigo...
7 od
```

Código 3.2: Sintaxis base de los bucles while-until

Estos bucles funcionan de forma análoga, la única diferencia entre ellos es que el bucle **while** se seguirá ejecutando siempre y cuando la condicion sea verdadera y el bucle **until** se seguirá ejecutando siempre y cuando la condición sea falsa.

3.4.3. El timout

El **timout** es un opcode para generar bucles de una duración determinada.

```
1 first_label:
2         timout  istart, idur, second_label
3         reinit  first_label
4 second_label:
5 codigo...
```

Código 3.3: Sintaxis base del timout

En primer lugar **first_ label** y **second_ label** son etiquetas de referencia a las que podemos saltar desde otras partes del código. El opcode **(timout istart, idur, second_ label)** tiene tres

parámetros de entrada: **istart** el instante de inicio, **idur** la duración de timout y **second_ label** el nombre de la etiqueta de la parte del código a la que queremos saltar, en este caso por lógica la segunda.El segundo opcode necesario para el funcionamiento es **(reinit first_ label)** que da la orden directa de saltar a la parte del código referida por **first_ label**.

Entendamos el uso de **timout** con un ejemplo práctico:

```
instr 1
loop:
idur    random  .5, 3
        timout  0, idur, play
        reinit  loop
play:
kFreq   expseg  400, idur, 600
aTone   poscil  .2, kFreq, giSine
        outs    aTone, aTone
endin
```

Código 3.4: Ejemplo real de uso del timout

Hemos definido una variable **idur** a la que damos un valor aleatorio entre (0,5 y 3) mediante el opcode **random**. Acto seguido usamos el opcode **timout** para saltar desde el instante **0**, durante ese **valor aleatorio de segundos**, a la etiqueta **play** donde se ejecuta el opcode **poscil** que genera una onda de sonido. Una vez acabado ese periodo de tiempo se ejecuta el opcode **reinit** que nos hace saltar a la etiqueta **loop** y vuelta a empezar.

3.5– Los Arrays de datos

Al igual que en cualquier lenguaje con cierto nivel de complejidad, en Csound existen los **Arrays** o vectores de datos. Veamos las peculiaridades del lenguaje en el uso de este tipo de estructuras.

3.5.1. Propiedades de los Arrays

En Csound podemos pensar en cinco propiedades características al definir un Array:

- **Dimensiones**: Los elementos de un array pueden leerse mediante la sintaxis **kArr[i]** siendo **i** la posición del elemento al que queremos acceder en un array unidimensional. De igual manera podemos acceder a los elementos de un array de dos dimensiones con la sintaxis **kArr[i][j]**. De forma análoga para arrays tridimensionales.
- **i- o k-Rate**: Los arrays son variables y como tal pueden definirse como variable i-rate o k-rate.
- **Local o Global**: De la misma manera podemos hacer de nuestro array una variabe global añadiendo la letra **g** al principio del nombre.
- **Arrays de Strings**: Los arrays de Csound pueden contener variables String además de número por lo que podemos clasificarlos también de esta manera.
- **Arrays de señales digitales**: Por último podemos conectar canales y salidas de opcodes a las posiciones de un array para facilitar el trabajo con las señales de audio.

3.5.2. Opcodes útiles

Estos son algunos de los opcodes que podemos usar al trabajar con arrays:

- **init**: El opcode que usamos para inicializar un array, para su sintexis únicamente necesitamos aportar el tamaño de cada una de las dimensiones del array que estamos definiendo. Por ejemplo: **kArr[] init 5** para un array unidimensional de longitud 5.
- **fillarray**: Para añadir una serie de valores a nuestro array. Veamos un ejemplo: **kArr[] fillarray 1, 2, 3, 4, 5** que añade los valores 1, 2, 3, 4 y 5 al array.
- **genarray**: Genera un array al que se le añaden los valores comprendidos entre los valores de entrada del opcode. Un ejemplo de uso: **kArr[] genarray 1, 5** que crea un array al que se le añaden los valores 1, 2, 3, 4 y 5.
- **lenarray**: Devuelve el tamaño actual de un array. **kTam lenarray kArr** devolvería el tamaño de kArr.
- **slicearray**: Nos sirve para generar un subarray desde un índice inicial hasta un índice final del array original introducido como parámetro de entrada en el opcode. Su sintaxis base es: **kSlice[] slicearray kArr, iStart, iEnd**. Y un ejemplo de uso sería **kSub[] slicearray kArr, 0, 2** que para el array **kArr=[4,3,2,1,0]** devolvería como resultado **kSub=[4,3,2]**
- **minarray**: Este opcode devuelve el valor más pequeño de todo el array y de manera opcional su índice si especificamos una variable para guardarla. Su sintaxis base es: **kMin [,kMinIndx] minarray kArr**.
- **maxarray**: Este opcode devuelve el mayor valor de todo el array y de manera opcional su índice si especificamos una variable para guardarla. Su sintaxis base es: **kMin [,kMinIndx] minarray kArr**.
- **sumarray**: Devuelve la suma de todos los valores del array numérico. Su sintaxis es **kSum sumarray kArr** y siendo **kArr=[1,1,1,1]** resultaría **kSum = 3**.
- **scalearray**: Escala los valores de un array en referencia a un valor mínimo y a un valor máximo. Veamos un ejemplo de uso:

```
1 kArr[] fillarray values
2       scalearray kArr, kmin, kmax
```

Código 3.5: Uso del scalearray

Tenemos el array **kArr=[1,3,9,5,6]**, donde el valor más pequeño es 1 y el valor más grande es 9. Cuando hacemos uso del opcode **(scalearray kArr 1,3)** estaremos escalando con las siguientes referencias relativas: **1** $\leftrightarrow$ **1 (kmin)** y **9** $\leftrightarrow$ **3 (kmax)**. De la misma manera el resto de valores de **kArr** quedarán en referencia también a kmin y kmax. El resultado para **(scalearray kArr 1,3)** con **kArr=[1,3,9,5,6]** sería **kArr=[1, 1.5, 3, 2, 2.25]**.

- **maparray**: Aplica un opcode en formato función a cada uno de los valores del array. Su sintaxis base es: **(kRes maparray kArr, "fun")**. Siendo **fun** la función que queremos usar de entre las siguientes opciones: **abs, ceil, exp, floor, frac, int, log, log10, round, sqrt**. Por ejemplo, la operación **(kRes maparray kArr, sqrt)** aplica la función **sqrt()** a cada elemento del array **kArr** y almacena los resultados en **kRes**.

3.5.3. Operaciones con arrays

Se pueden usar los cuatro operadores básicos (**+, -, *, /**) para sumar, restar, multiplicar y dividir arrays. Si el operador se aplica entre el array y un número, la operación se realiza a todos los elementos del array, por lo que guardar los resultados en un nuevo array es una buena práctica. Por ejemplo si tenemos el array **kArr=[1,1,1]** y realizamos la operación **kSuma = Karr + 1** obtendremos **kSuma=[2,2,2]**.

Por otra parte, si realizamos la operación entre dos arrays del mismo tamaño, el cálculo tendrá en cuenta cada pareja de valores de la misma posición de cada array. Si tenemos el array **kArr1=[10,10,10]** y el array **kArr2=[1,2,3]** y realizamos la operación **kArr3 = kArr1 + kArr2** obtenemos **kArr3=[11,12,13]**.

3.6– Funciones de entrada/salida

Muchos de los opcodes de Csound pueden expresarse mediante la sintaxis funcional que se usa en tantos otros lenguajes: **fun(arg1, agr2...)**. Hablemos de algunos de los opcodes que más se usan en su formato funcional.

- **abs**: Devuelve el valor absoluto del parámetro de entrada **arg**.

  Sintaxis: **abs(arg)**

  Ejemplo: **abs(-2) = 2**

- **ceil**: Devuelve el menor número entero posible que sea mayor que **arg**.

  Sintaxis: **ceil(arg)**

  Ejemplo: **ceil(0.999) = 1**

- **exp**: Devuelve el número **e (2,718...)** elevado a la potencia de **arg**.

  Sintaxis: **exp(arg)**

  Ejemplo: **exp(2) = 7.389**

- **frac**: Devuelve la parte fraccionaria de **arg**.

  Sintaxis: **frac(arg)**

  Ejemplo: **frac(2.1234) = 0.1234**

- **int**: Devuelve la parte entera de **arg**.

  Sintaxis: **int(arg)**

  Ejemplo: **int(2.1234) = 2**

- **log**: Devuelve el logaritmo natural de **arg**.

  Sintaxis: **log(arg)**

  Ejemplo: **log(8) = 2.079**

- **sqrt**: Devuelve la raíz cuadrada de **arg**. Sintaxis: **sqrt(arg)**

  Ejemplo: **sqrt(25) = 5**

3.7– Creando un opcode (UDOs)

En Csound, un opcode creado por un usuario recibe el nombre de **UDO (User created opcode)**. Definir un **UDO** es simplemente escribir parte del código que incluiríamos en un instrumento, dentro de un bloque especial para luego poder usarlo a efectos prácticos de la misma manera que usamos cualquier opcode.

La sintaxis base para definir un UDO es:

```
1 opcode name, outtypes, intypes
2 inNames xin
3 codigo...
4 xout outNames
5 endop
```

Código 3.6: Sintaxis base de un UDO

Vamos por partes, las palabras **opcode** y **endop** marcan el inicio y el final del bloque de código en el que se define nuestro UDO. En la línea 1 de la figura de código anterior observamos: **name** que será el nombre de nuestro UDO, **outtypes** donde marcaremos el tipo de variable de las variables de salida del UDO y **intypes** donde marcaremos el tipo de variable de las variables de entrada. La sintaxis para usar **outtypes** e **inttypes** es simplemente un string con una letra por cada variable de entrada o salida especificando su tipo.

Por ejemplo, esta sería la cabecera del bloque de código de un UDO con dos variables de salida, una tipo **i-rate** y otra **tipo k-rate**; y una variable de entrada **k-rate**: **(opcode miOpcode, ik, k)**

Los opcodes **xin** y **xout** nos sirven respectivamente para recoger las variables de entrada, dándoles un nombre identificativo para usarlo en el resto del UDO, y para definir cuál o cuáles serán las variables de salida.

Por último, **inName** y **outName** representan los nombres de esas variables de entrada y salida que además deben respetar la nomenclatura de uso de tipos en Csound empezando por a, i, k, etc... Según corresponda en la defición de la primera línea del bloque.

Vamos a crear un ejemplo de UDO para realizar ecuaciones de segundo grado: $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$

```
1 opcode SegundoGrado, ii, iii
2 iVarA, iVarB, iVarC xin
3 ioutPos = ((iVarB*-1)+sqrt(iVarB^2 -(4*iVarA*iVarC)))/(2*iVarA)
4 ioutNeg = ((iVarB*-1)-sqrt(iVarB^2 -(4*iVarA*iVarC)))/(2*iVarA)
5 xout ioutPos, ioutNeg
6 endop
```

Código 3.7: UDO para ecuaciones de segundo grado

Y para usar nuestro opcode en un instrumento:

```
1 instr 1
2 iVarA = 4
3 iVarB = 12
4 iVarC = 2
5 iResPos,iResNeg SegundoGrado iVarA, iVarB, iVarC
6 print iResPos
7 print iResNeg
8 endin
```

Código 3.8: Uso del opcode SegundoGrado en un instrumento

3.8– Las macros

En Csound pueden usarse **macros** para tener una o varias líneas de código a mano y sustituirlas siempre que queramos con un nombre de referencia a elegir. Normalmente las usaremos cuando tengamos previsto repetir una serie de líneas de código a lo largo de nuestro programa o cuando hagamos referencia a algún nombre de variable o archivo para que, en caso de querer cambiar su nombre o valor, únicamente necesitemos hacerlo en la definición de la **macro** y no cada vez que aparezca en nuestro código.

3.8.1. Sintaxis de las macros

Su sintaxis es sencilla: **(#define NAME # Código o texto a sustituir #)**

Siendo **define** la palabra reservada para empezar a definir una **macro**, **NAME** el nombre con el que más tarde haremos referencia a la macro, y la parte contenida entre el segundo y tercer corchete el bloque de código o texto que queda sustituido y que aparecerá repetido a efectos de compilación cada vez que usemos la **macro**.

Para hacer referencia a la macro en otros lugares de código bastará con escribir su nombre precedido del símbolo **$**

Veamos un ejemplo de uso:

```
#define ejemploMacro # prints "Soy la macro\\n" #

instr 1
$ejemploMacro
$ejemploMacro
$ejemploMacro
endin
```

Código 3.9: Ejemplo de una macro simple

Como vemos hemos definido la macro **ejemploMacro** que contiene una única línea de código para imprimir por consola el texto "Soy la macro". Más tarde usamos esta macro 3 veces en nuestro instrumento por lo que a efectos prácticos el código **prints "Soy la macro "** se ejecutará un total de 3 veces.

3.8.2. Macros con parámetros de entrada

Podemos también definir macros con parámetros de entrada de la siguiente manera:

(#define NAME(Arg1' Arg2' Arg3...) # Código o texto a sustituir #)

Simplemente incluiremos entre paréntesis la serie de argumentos que usará nuestra macro. La separación entre nombre y nombre de argumento la marcaremos mediante el carácter ('). Para usar los argumentos dentro del bloque de código de la macro les haremos referencia precendiéndolos del símbolo **$**.

Veamos un ejemplo de uso:

```
#define ejemploMacro(arg1'arg2'arg3) #
prints "El argumento uno es %i\\n", $arg1
prints "El argumento dos es %i\\n", $arg2
prints "El argumento tres es %i\\n", $arg3
prints "Y esta es su suma %i\\n", $arg1+$arg2+$arg3
#

instr 1
$ejemploMacro(1'2'3)
endin
```

Código 3.10: Ejemplo de una macro con argumentos

3.8.3. Macros en <CsScore>

Hasta ahora hemos estado definiendo el código de nuestras macros en <CsInstruments>pero también podemos usarlas en <CsScore>. De esta manera pueden sernos útiles para realizar pruebas de testeo ordenadas y controladas o, como hicimos anteriormente, para tener un código más reutilizable y optimizado.

La sintaxis es idéntica al caso anterior así que veamos un ejemplo común de uso:

```
1  #define RIFF_1(Start'Trans)
2  #
3  i 1 [$Start ] 1    [36+$Trans]
4  i 1 [$Start+1 ] 0.25 [43+$Trans]
5  i 1 [$Start+1.25] 0.25 [60+$Trans]
6  #
7
8  $RIFF_1(0 ' 0)
9  $RIFF_1(2 ' 0)
10 $RIFF_1(4 '-5)
```

Código 3.11: Ejemplo de una macro en <CsScore>

Imaginemos para el ejemplo un instrumento que genera una determinada nota y que usa p1, p2, p3 y p4; siendo p4 un parámetro que determina la frecuencia del tono de la nota de salida del instrumento. En nuestra macro realizamos tres **Event scores** (líneas 3, 4 y 5) usando ese instrumento, cuyos p2 (momento de inicio) y p4 (tono de la nota de salida) se ven modificados por **Start** y **Trans** que son los parámetros de entrada de nuestra macro. De esta manera podemos realizar esos tres **Score events** tantas veces como queramos y únicamente introduciendo los parámetros que consideramos importantes al definir la macro.

CAPÍTULO 4

Conceptos Avanzados

4.1– El opcode ftgen

el opcode **ftgen** es muy recurrente. Nos sirve para generar una tabla de scores de entre nuestros instrumentos.

Su sintaxis base es:

```
gir ftgen ifn, itime, isize, igen, iarga [, iargb ] [...]
```

Código 4.1: Sintaxis base de ftgen

Siendo **gir** una tabla de al menos 100 posiciones, **ifn** el número de la tabla. El resto de valores se corresponden con los argumentos **p2**, **p3**, **p4** y **p5** del **f Statement** que explicaremos a continuación.

4.1.1. El f statement

Coloca valores en una tabla de scores para usar nuestros instrumentos. Está implícita en **ftgen**.

Esta es la sintaxis base:

```
f p1 p2 p3 p4 p5 ... PMAX
```

Código 4.2: Sintaxis base del f statement

Siendo **p1** el número de tabla, **p2** tiempo de activación para generar datos, **p3** tamaño de la tabla, **p4** nombre de la rutina **GEN** de generación de datos y **p5 ... PMAX** que dependerán del **GEN** que estemos usando.

Este es un ejemplo en el que rellenamos una tabla (f1) de tamaño 5 con ceros: **(f1 0 5 2 0)**

4.2– Delay y Feedback

Empecemos por explicar en qué consisten estos efectos.

4.2.1. El Delay

Podemos definir el **Delay** como un buffer de repetición de nuestra onda que va poniendo en cola los sonidos que van entrando y según las características de nuestro **Delay** los saca de una manera determinada, conformando así algunos de los efectos más conocidos en la producción musical.

El uso más básico del **Delay** es el clásico efecto “eco”pero también puede servirnos para implementar el Chorus, Flanging, Cambio tonal y otros filtros de onda.

Respecto a Csound existen varios opcodes dedicados al **Delay**, como **delayr** y **delayw**.

Empecemos por la diferencia en los nombres de estos opcodes que se usan en conjunto puesto que **delayr** es de lectura (read) y **delayw** es de escritura (write) del buffer de delay.

Veamos un ejemplo básico de uso:

```
1 aSigOut delayr 1
2         delayw aSigIn
```

Código 4.3: Sintaxis delayr y delayw

Donde **aSignIn** es la señal de entrada a la que queremos aplicar el delay y **aSignOut** es la señal de salida a la que ya se ha aplicado el efecto. El '1'que aparece como argumento es la cantidad de segundos que mide el buffer, en este caso un segundo y el hecho de definir aquí la longitud del buffer implica que tengamos que definir primero la variable de salida antes y después la variable de entrada, lo cual es en principio un poco confuso.

4.2.2. El Feedback

El delay tal y como lo hemos definido anteriormente acaba pareciendo una simple única repetición atenuada del sonido de entrada. Para encontrar el clásico sonido al que en el mundo de la edición musical nos referimos como delay debemos conocer el concepto de **feedback**.

El **feedback** será un valor residual de la señal del buffer que usaremos como parámetro de entrada del delay. Lo normal es que este valor decrezca con cada paso por el buffer de manera que el sonido acabe pareciendo cada vez más distante hasta apagarse por completo y no se produzca un bucle infinito.

Respecto a Csound podremos implementar este concepto de manera sencilla con una variable.

4.2.3. Ejemplo del efecto Delay

Veamos a continuación el código de un efecto delay con feedback y analicémoslo con el objetivo de comprenderlo mejor:

En primer lugar podemos observar tres opcodes que no habíamos visto hasta el momento. **loopseg** en la línea 16, **randomh** en la línea 17 y **interp** en la línea 18.

Expliquemos brevemente sus funcionamientos:

- **loopseg**: Genera una señal de control que puede usarse como envelope o envoltura. Tanto variable de salida como variables de entradas serán de tipo k-rate.
- **randomh**: Genera valores aleatorios al igual que el opcode **random** pero mantiene estos valores guardados durante un periodo determinado de tiempo.
- **interp**: Sirve para convertir una variable de tipo k-rate a una variable de tipo a-rate.

Una vez sabido el funcionamiento básico de cada opcode presente veamos paso por paso lo que se hace en el ejemplo:

- **Línea 13**: Hacemos uso de **ftgen** para tener preparada nuestra onda senuidad.
- **Líneas 15 a 19**: Definimos las primeras líneas de nuestro instrumento. El uso de estas líneas es el de acabar generando **aSig**, es decir, nuestra onda de entrada a la que aplicaremos el delay.
- **Línea 21**: Definimos **iFback** que tal y como explicamos en secciones anteriores hará las veces de valor de feedback, en este caso 0.7 (de ser un valor menor nuestro delay se apagaría antes).

- **Líneas 22 a 26**: Hacemos uso de **delayr** y **delayw**. Si prestamos especial atención a la línea 23 veremos que nuestro valor de entrada, **(aSig(aBufOut*iFdback))**, es efectivamente una operación en la que participan nuestra señal, su propio valor de salida (siguiente en el buffer) y el valor de feedback que acabará determinando el tiempo restante del delay.

  Como vemos también en la línea 25, el sonido que nuestro instrumento producirá será una mezcla de la onda de salida con el valor de buffer.

```
1  <CsoundSynthesizer>
2  <CsOptions>
3  -odac
4  </CsOptions>
5
6  <CsInstruments>
7
8  sr = 44100
9  ksmps = 32
10 nchnls = 1
11 0dbfs = 1
12
13 giSine ftgen 0, 0, 2^12, 10, 1
14
15   instr 1
16 kEnv  loopseg 0.5,0,0,0,0.0005,1,0.1,0,1.9,0,0
17 kCps  randomh 400, 600, 0.5
18 aEnv  interp kEnv
19 aSig  poscil aEnv, kCps, giSine
20
21 iFdback =    0.7
22 aBufOut delayr 0.3
23       delayw aSig+(aBufOut*iFdback)
24
25       out    aSig + (aBufOut*0.4)
26   endin
27 </CsInstruments>
28 <CsScore>
29 i 1 0 25
30 e
31 </CsScore>
32 </CsoundSynthesizer>
```

Código 4.4: Ejemplo completo del efecto Delay

Se invita al lector a que experimente con diversos valores en las variables mencionadas para comprender en mayor medida el concepto de delay y que posteriormente visite las referencias del documento en caso querer profundizar.

4.3– FM: La modulación de frecuencia

La modulación de frecuencia, tambien conocida como **FM (Frequency Modulation)** es la categoría de los efectos de sonido basados en la modificación de la frecuencia de la onda de sonido original. Un claro ejemplo de ello es el efecto de vibrato.

4.3.1. El vibrato

El efecto **Vibrato** en el sonido consiste en modificar la frecuencia (por tanto el tono) de la onda sonora de forma periódica y constante con valores cercanos al original. Se crea así una sensación de "vibración" del sonido.

Veamos un ejemplo de **vibrato** implementado en Csound.

```
1  <CsoundSynthesizer>
2  <CsOptions>
3  -o dac
4  </CsOptions>
5  <CsInstruments>
6  sr = 48000
7  ksmps = 32
8  nchnls = 2
9  0dbfs = 1
10
11 instr 1
12 aMod poscil 10, 5 , 1
13 aCar poscil 0.3, 440+aMod, 1
14 outs aCar, aCar
15 endin
16
17 </CsInstruments>
18 <CsScore>
19 f 1 0 1024 10 1
20 i 1 0 2
21 </CsScore>
22 </CsoundSynthesizer>
```

Código 4.5: Ejemplo completo del efecto Vibrato

Prestemos especial atención a las líneas 12 y 13 de la figura anterior. En primer lugar la línea **(aMod poscil 10, 5, 1)** produce una onda de 10Hz de amplitud y 5Hz de frecuencia. Esta será nuestra onda moduladora, la que aplicaremos sobre el sonido al que queremos añadir el efecto vibrato.

En segundo lugar tenemos la línea **(aCar poscil 0.3, 440+aMod, 1)** que produce una onda de 440Hz de frecuencia y a la que sumamos el valor de **aMod**, nuestro modulador, que afecta al valor de la frecuencia de **aCar** haciéndolo oscilar periódicamente entre 450Hz y 430Hz. Si sustituimos la línea anterior por: **(aCar poscil 0.3, 440, 1)** comprobaremos que deja de producirse el efecto **vibrato**.

Como podemos observar el efecto vibrato simple puede lograrse fácilmente en Csound, pero se invita a experimentar con los valores de **aMod** para comprobar cómo afecta una onda moduladora con distintos valores de amplitud y frecuencia.

4.4– AM: La modulación de amplitud

La modulación de la amplitud de onda o **AM (Amplitude Modulation)** consiste en hacer modificaciones en la amplitud de una onda, esto provoca lógicamente un aumento o disminución del volumen del sonido producido. Uno de los ejemplos más claros de efecto basado en el **AM** es el efecto trémolo.

4.4.1. El trémolo

Es fácil confundir el efecto trémolo con el vibrato pero como acabamos de describir, el efecto trémolo se basa en modificaciones de amplitud y no de frecuencia de la onda. No obstante, para implementar este efecto en Csound procederemos de manera similar haciendo uso de una variable moduladora(una segunda onda) que aplicaremos sobre sobre el sonido al que queremos aplicar el trémolo.

Veamos un ejemplo:

```
1  <CsoundSynthesizer>
2  <CsOptions>
3  -o dac
4  </CsOptions>
5  <CsInstruments>
6
7  sr = 48000
8  ksmps = 32
9  nchnls = 1
10 0dbfs = 1
11
12 instr 1
13 aRaise expseg 2, 20, 100
14 aModSine poscil 0.5, aRaise, 1
15 aDCOffset = 0.5
16 aCarSine poscil 0.3, 440, 1
17 out aCarSine*(aModSine + aDCOffset)
18 endin
19
20 </CsInstruments>
21 <CsScore>
22 f 1 0 1024 10 1
23 i 1 0 25
24 e
25 </CsScore>
26 </CsoundSynthesizer>
```

Código 4.6: Ejemplo completo del efecto Trémolo

Empecemos por explicar el opcode **expseg** que vemos por primera vez. Este opcode con sintaxis **(ares expseg ia, idur1, ib)** genera un valor que cambia exponecialmente a lo largo del tiempo, es decir, nos da un valor inicial **ia** que aumenta su valor exponencialmente a los largo de **idur1** segundos hasta llegar al valor **ib**. Aunque este valor sobrepase a **ib**, seguirá aumentando en el mismo ratio de exponecialidad hasta que paremos su ejecución. Es debido a este opcode y su valor **aRaise** en el ejemplo que podemos oír un aumento de la "rapidez"del trémolo a medida que pasa el tiempo.

Observemos detenidamente las líneas 14, 16 y 17 de la figura. Tenemos por una parte la onda **aModSine** que será nuestro modulador de amplitud y por otro nuestra onda original y simple de 440Hz **aCarSine**. Como vemos en la línea 17 aplicaremos un cálculo simple de multiplicación para que las amplitudes, **0.5** de **aModSine** y **0.3** de **aCarSine** se sumen y oigamos el efecto trémolo a una frecuencia de **aRaise**. Se invita al lector a ejecutar este mismo ejemplo sustituyendo

la variable **aRaise** en la línea 14 por el valor fijo 1 y se podrá observar cómo el efecto trémolo tendrá una duración constante de exactamente un segundo.

Por último en la línea 15 tenemos la variable **aDCOffset** que determina el DC Offset de la onda. Veamos con un poco más de profundidad este concepto.

4.4.2. DC Offset

El **DC Offset** es un valor añadido a la onda que en términos gráficos podemos decir que la "desplaza"respecto del valor 0.

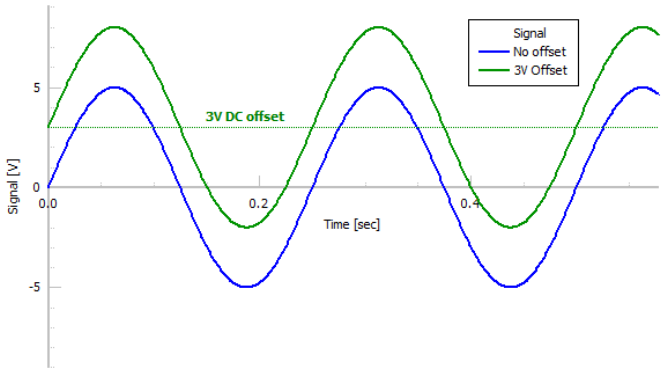

Figura 4.1: El DC Offset

El DC Offset tiene muchos usos cuando es intencionado como el de prevenir el clipping, pero en algunos casos se produce de forma indeseada por causas ajenas.

4.5– RM: La modulación en anillo

La modulación en anillo o **RM (Ring-Modulation)** es un caso especial de modulación de amplitud en el que no hacemos uso del **DC Offset**. De esta manera diremos que la amplitud variará entre +1 y -1 (respecto a 0dbfs).

Para tener un ejemplo básico de esto basta con que volvamos a la figura de código 4.6 y demos a **aDCOffset** un valor de 0.

4.6– WAVESHAPING

http://write.flossmanuals.net/csound/e-waveshaping/

4.7– Filtros de onda

Los **filtros de onda** son uno de los recursos más usados en la edición musical y sonora. Consisten principalmente en filtrar o cortar determinadas frecuencias en un sonido dejando pasar al resto. Hablemos de las tres principales categorías de **filtros de onda**. Con el concepto "frecuencia cutoff'ños referiremos a la frontera entre las frecuencias que pasan y las que corta el filtro.

4.7.1. Filtros Lowpass

Los filtros **Lowpass** dejan pasar las frecuencias de onda más bajas y filtran las más altas. Para implementar este tipo de filtros en Csound existen diversos opcodes, veamos un ejemplo con tres de ellos: **tone**, **butlp** y **moogladder**.

```
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
       prints    "tone%n"
aSig  vco2      0.5, 150
kcf   expon     10000,p3,20
aSig  tone      aSig, kcf
      out       aSig
  endin

  instr 2
       prints    "butlp%n"
aSig  vco2      0.5, 150
kcf   expon     10000,p3,20
aSig  butlp     aSig, kcf
      out       aSig
  endin

  instr 3
       prints    "moogladder%n"
aSig  vco2      0.5, 150
kcf   expon     10000,p3,20
aSig  moogladder aSig, kcf, 0.9
      out       aSig
  endin

</CsInstruments>
<CsScore>
i 1 0 3; tone
i 2 4 3; butlp
i 3 8 3; moogladder
e
</CsScore>
</CsoundSynthesizer>
```

Código 4.7: Opcodes de filtro Lowpass

En la figura anterior se definen tres instrumentos que empiezan generando la misma onda pero a la que se acaba aplicando un opcode de filtro **lowpass** distinto.

- **tone**: Este opcode se caracteriza por usar una frecuencia de cutoff no demasiado estricta, es decir, que actúa como atenuador de las frecuencias más adyacente al cutoff en lugar de cortarlas abruptamente.
- **butlp**: Usa en cambio un cutoff más agresivo que **tone** por lo que quedarán menos residuos de la frecuencias más altas en nuestro sonido.
- **moogladder**: Este opcode se basa en filtros analógicos usados en sistetizadores de la marca “Moog”. La mejor forma de describirlo sería por tanto en referencia a su contraparte original.

La mejor forma de comprender las diferencias entre estos opcodes es ejecutando el ejemplo anterior y tratando de analizar la diferencia sonora entre instrumentos.

4.7.2. Filtros Highpass

Los filtros **Highpass** dejan pasar las frecuencias de onda más altas y filtran las más bajas. Para implementar este tipo de filtros en Csound existen diversos opcodes, veamos un ejemplo con tres de ellos: **atone**, **buthp** y **bqrez**.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
       prints    "atone%n"
aSig  vco2       0.2, 150
kcf   expon      20, p3, 20000
aSig  atone      aSig, kcf
       out       aSig
  endin

  instr 2
       prints    "buthp%n"
aSig  vco2       0.2, 150
kcf   expon      20, p3, 20000
aSig  buthp      aSig, kcf
       out       aSig
  endin

  instr 3
       prints    "bqrez(mode:1)%n"
aSig  vco2       0.03, 150
kcf   expon      20, p3, 20000
aSig  bqrez      aSig, kcf, 30, 1
       out       aSig
  endin

</CsInstruments>
<CsScore>
i 1 0 3 ; atone
i 2 5 3 ; buthp
i 3 10 3 ; bqrez(mode 1)
e
</CsScore>
</CsoundSynthesizer>
```

Código 4.8: Opcodes de filtro Highpass

De forma análoga a la saección anterior, se definen tres instrumentos. Veamos sus opcodes de filtro **Highpass**.

- **atone**: De manera análoga a **tone**, este opcode se caracteriza por usar una frecuencia de cutoff no demasiado estricta, es decir, que actúa como atenuador de las frecuencias más adyacente al cutoff en lugar de cortarlas abruptamente.
- **buthp**: De manera análoga a **butlp**, **buthp** usa un cutoff más agresivo que **atone** por lo que quedarán menos residuos de la frecuencias más bajas en nuestro sonido.
- **bqrez**: Este es un opcode con múltiples modos para seleccionar, en nuestro ejemplo usamos "mode:1"pero se invita a experimentar con otros modos para comprender mejor este opcode.

4.7.3. Filtros Bandpass

Este tipo de filtros dejan pasar frecuencias comprendidas en una determinada "banda", es decir, será necesario determinar cómo de ancha será esa banda conociendo sus límites superior e inferior. Veamos la implementación de este filtro en Csound con dos opcodes: **reson** y **butbp**.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
      prints    "reson%n"
aSig  vco2      0.5, 150
kcf   expon     20,p3,10000
aSig  reson     aSig,kcf,kcf*0.1,1
      out       aSig
  endin

  instr 2
      prints    "butbp%n"
aSig  vco2      0.5, 150
kcf   expon     20,p3,10000
aSig  butbp     aSig, kcf, kcf*0.1
      out       aSig
  endin

</CsInstruments>
<CsScore>
i 1 0 3 ; reson
i 2 4 3 ; butbp
e
</CsScore>
</CsoundSynthesizer>
```

Código 4.9: Opcodes de filtro Bandpass

- **reson**: Filtro de banda basado en la resonancia.
- **butbp**: La versión de **bandpass** de los opcode **but–**. Con un cutoff relativamente agresivo.

4.8– Reverberación

http://write.flossmanuals.net/csound/e-reverberation/

CAPÍTULO 5

Haciendo Música en Directo

Este capítulo trata de dar una introducción al "live coding" en Csound. Para ello se enplicarán algunos de sus fundamentos y se usará la página web `https://live.csound.com/` (Programada y mantenida por Steven Yi, desarrollador de Csound y blue), de ahora en adelante referida como **Live Csound**, para ejemplificar los conceptos.

Live Csound usa la librería **livecode.orc** de Csound y puede abrirse en cualquier navegador commplatible con **Javascript**, **WebAudio** y **WebAssembly**. Cualquiera de los navegadores modernos más usados (Chrome, Firefox, Safari, etc..) tiene estas capacidades. Es además una **PWA (Progressive Web Application)**[1] por lo que podemos instalar una versión offline de la misma en nuestro equipo y ejecutarla mediante nuestro navegador aunque no dispongamos de conexión a internet.

5.1– Live Csound

5.1.1. Atajos de teclado

Estos son los atajos de teclado más útiles al usar la aplicación **Live Csound**, es combeniente tenerlos en mente para hacer más dinámico nuestro "Live Coding".

- **ctrl-e**: Evalúa y ejecuta el código seleccionado.
- **ctrl-enter**: Idéntico a ctrl-e.
- **ctrl-shift-enter**: Evalúa el código seleccionado y lo ejecuta tras un compás de 4/4.
- **ctrl-h**: Genera una plantilla de código del opcode hexplay().
- **ctrl-j**: Genera una plantilla de código del opcode euclidplay().
- **ctrl-;**: Convierte en comentarios las líneas seleccionadas o los descomenta si ya lo eran.
- **ctrl-alt-c**: Idéntico a ctrl-;.

5.2– Los principales opcodes

La biblioteca **livecode.orc** usada por **Live Csound** aporta los opcodes **hexbeat()** y **hexplay()** que nos servirán generar los sonidos de forma rítmica mediante valores hexadecimales.

[1] Revisar el anexo "Cómo instalar un PWA en Chrome"

5.2.1. schedule()

El opcode **schedule()** nos servirá para añadir un nuevo score event, es decir, para hacer uso de alguno de sus instrumentos. Su sintaxis siendo la misma que al escribir un score event **schedule(p1, p2, p3, p4, p5)**. Siendo **p1** el identificador, **p2** el instante de inicio, **p3** la duración, **p4** la frecuencia del sonido y **p5** la amplitud en referencia 0dbfs.

5.2.2. hexplay()

Para usar **hexplay()** es importante haber conocido previamente **schedule()** puesto que su sintaxis está implícita.

```
1 hexplay(Spat, Sinstr, idur, ifreq, iamp)
```

Código 5.1: Sintaxis base del hexplay()

Siendo **Spat** un código hexadecimal que marcará la figura rítmica en la que se activa nuestro **schedule()** implícito. **Sinstr** es p1, el nombre del instrumento. **Sinstr** es p3, la duración. **ifreq** es p4, la frecuencia. Y **iamp** es p5, la amplitud. No es necesario introducir **p2** puesto que no habrá un instante inicial particular para cada score event si contamos con un patrón rítmico.

5.3– Creando música

Empecemos por lo tanto con un sencillo ejemplo al que iremos añadiendo dificultad hasta llegar a lo más parecido a una idea musical.

5.3.1. Producir sonidos sencillos

Haremos uso del opcode **schedule()** del cual ya hemos aprendido la sintaxis. Vamos a la página `https://live.csound.com/`, seleccionamos todo (ctrl-a) y borramos todo el código que aparece como demos dejando la página en blanco. Después de eso podemos escribir la línea de código:

```
1 schedule("Sub1", 0, 2, 440, 0.5)
```

Código 5.2: Ejemplo de schedule()

Si seleccionamos esta línea de código y pulsamos **ctrl-e** haremos sonar el instrumento **"sub1"** desde el instante **0**, durante **2** segundos, a una frecuencia de **440HZ(La)**, y a un volumen relativo de **0.5** respecto a 0dbfs que va de 0.0 a 1.0.

Y así de sencillo es empezar a producir notas en **Live Csound**. Por supuesto se recomienda empezar a experimentar con los valores del **schedule()**, incluido el instrumento para el que existen esta serie de opciones predeterminadas:

- **Instrumentos tonales**: Sub1, Sub2, Sub3, Sub4, Sub5, SynBrass, Plk, Bass, VoxHumana, FM1, Noi, Wobble.
- **Percusión**: Clap, BD, SD, OHH, CHH, HiTom, MidTom, LowTom, Cymbal, Rimshot, Claves, Cowbell, Maraca, HiConga, MidConga, LowConga.

5.3.2. Creando un instrumento

No queremos vernos limitados a la librería predeterminada de instrumentos de **Live Csound** así que vamos a aprender a definir nuestro propio instrumento. Esto sigue siendo el lenguaje de programación Csound así que definir un instrumento no es algo nuevo para nosotros. Tratemos de verlo de forma sencilla:

```
1  instr Add
2    alfo = 1 + oscili(random(0.002, 0.01), random(0.2,3))
3    asig = oscili(1, p4 * alfo)
4    asig += oscili(1, p4 * 2 * alfo)
5    asig *= 0.25 * p5 * expon(1, p3, 0.001)
6    out(asig, asig)
7  endin
8
9  instr Note
10   isd = random(0, 12)
11   schedule("Add", 0, 20, in_scale(rand(array(-1,0)), isd), ampdbfs(-12)
       )
12 endin
13
14 schedule("Note", 0, 0)
```

Código 5.3: Un instrumento funcional

En primer lugar vamos a definir un bloque de instrumento llamado (Add). En (Add) vamos a definir dos ondas sonoras mediante el opcode **oscili(xamp, xcps)** donde **xamp** es la amplitud de onda y **xcps** su frecuencia.La primerá se llamará —textbfalfo y le entregaremos como parámetros de entrada dos valores aleatorios mediante el opcode **random(min, max)**.

La segunda onda será **asig** cuyos valores de entrada dependerán de **alfo** y de otras operaciones realizadas a la onda bajo un criterio musical (Se invita a experimentar realizando diferentes oprecaiones para encontrar la síntesis de sonido que más de adecúe al gusto propio).

Una vez consigamos el sonido deseado vamos a crear el instrumento **Note**, en el que generando valores aleatorios de frecuencia, ejecutaremos el instrumento **Add** siempre en una misma tonalidad musical gracias al opcode **in_ scale(ioct, idegree)** que genera una frecuencia válida de entre las notas pertenecientes a la octava **ioct**.

Al ejecutar la línea **(schedule("Note", 0, 0))** se generará una nota con las características antes descritas y que sonará durante 20 segundos mientras hace "fadeout"(su volumen se reduce progresivamente) debido a que en la línea 5 del código empezamos a reducir el valor de la onda de manera exponencial.

Ya tenemos nuestro instrumento listo, veamos cómo sacarle provecho.

5.3.3. Figuras rítmicas como valor hexadecimal

Vamos a hacer un uso básico del opcode **hexplay()** para usar el instrumento que hemos creado.

Para usar **hexplay()** tendremos que pensar en generar figuras rítmicas mediante valores hexadecimales, teniendo en mente que al pasar el valor hexadecimal a binario cada 1 será un tick en el que se ejecutará nuestro **schedule()** y cada 0 será silencio.

Tabla de conversión de valores hexadecimales:

HEX	0	1	2	3	4	5	6	7	8	9
BIN	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
DEC	00	01	02	03	04	05	06	07	08	09

HEX	A	B	C	D	E	F
BIN	1010	1011	1100	1101	1110	1111
DEC	10	11	12	13	14	15

```
1  set_tempo(100)
2  set_scale("min")
3
4  instr P1
5    hexplay("1010",
6       "BD", p3,
7       in_scale(-1, 0),
8       0.5)
9
10   hexplay("0101",
11      "Clap", p3,
12      in_scale(-1, 0),
13      0.5)
14
15   hexplay("0002",
16      "LowTom", p3,
17      in_scale(-1, 0),
18      0.5)
19
20   hexplay("f0f0",
21      "Maraca", p3,
22      in_scale(-1, 0),
23      0.5)
24 endin
25 schedule("P1", 0, 1)
```

Código 5.4: Uso del hexadecimal para las figuras rítmicas

Tengamos el código de la figura anterior como un ejemplo para entender el funcionamiento del código hexadecimal como atributo de entrada que marca las figuras de ritmo. Prestemos especial atención por tanto a las líneas 5, 10, 15 y 20 del código.

Cuando como argumento **Spat** del opcode **hexplay()** introducimos por ejemplo "1010" en valor hexadecimal podemos pensar en ello como un número binario agrupado en cuatro valores, es decir: **HEX(1010) = BIN(0001 0000 0001 0000)**.

Cuando marcamos un tempo (en este ejemplo 100) en nuestra pieza y queda el pulso determinado, diremos que cada uno de estos valores hexadecimales representará un pulso. Dicho de otra manera, si ejecutamos únicamente el **hexplay()** de las líneas 4 a 8 del código, el instrumento **BD** sonará una vez al final del pulso uno y otra vez al final de pulso tres. Esto se repetirá ad infinitum.

El **hexplay()** de las líneas 10 a 13 hará sonar el instrumento **Clap** al final de los pulsos segundo y cuarto.

Cada uno de los pulsos puede dividirse en cuatro partes de igual duración, una por cada bit de los cuatro usados para representar un valor hexadecimal.

El **hexplay()** de las líneas 15 a 18 hará sonar el instrumento **LowTom** en la tercera parte de cuarto pulso.

Y por último, el **hexplay()** de las líneas 20 a 23 hará sonar el instrumento **Maraca** cuatro veces durante el primer pulso y cuatro veces durante el tercer pulso.

hexplay() acepta cadenas hexadecimales más cortas y más largas que cuatro dígitos como las mostradas en los ejemplos, es por ello que se recomienda encarecidamente abrir **Live Csound** y empezar a experimentar por uno mismo con los **hexplay()** para empezar a compreder su concepto rítmico.

5.3.4. Ideas musicales con nuestro instrumento

Una vez introducido el concepto de figuras rítmicas hexadecimales podemos volver a nuestro instrumento casero y usarlo de la siguiente manera por ejemplo:

```
instr P1
  hexplay("000f",
     "Note", 0, 0)
endin

schedule("P1", 0, 1)
```

Código 5.5: Usando hexplay() con nuestro instrumento

Se ejecutará una vez por cada una de las cuatro parte del cuarto pulso. Al generar, como explicamos antes, una nota con frecuencia aleatoria cada vez; estaremos fabricando acordes arpegiados aleatorios con nuestro instrumento de forma automática cada cuarto pulso del tempo que hayamos elegido.

CAPÍTULO 6

Cabbage: Guía de uso

Cabbage es un IDE para el lenguaje Csound. Es de código abierto y está desarrollado por Rory Walsh.

Será el principal IDE que usaremos a lo largo de este documento puesto que además de contar con todas las comodidades necesarias para el funcionamiento del lenguaje, aporta además funcionalidades para crear interfaces gráficas para nuestro software de manera muy simple pero vistosa.

Se presenta el siguiente capítulo con la intención de dar una guía básica referencial de funcionamiento a la que acudir en caso de ser necesario durante el curso del resto de contenidos.

6.1– Instalación de Cabbage

Cabbage puede instalarse en sistemas Windows, OSX y Linux. Posee incluso un instalador en versión beta para sistemas Android.

Pasos para la instalación en Windows y OSX:

- Acudir a la página `https://cabbageaudio.com/download/` donde encontraremos los enlaces de descarga.
- Seleccionar la versión adecuada para nuestro sistema, en este caso Windows u OSX.

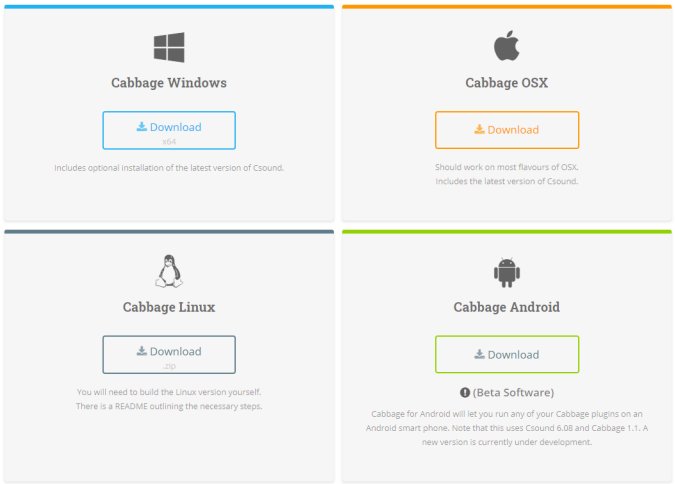

Figura 6.1: Versiones disponibles

- Ejecutamos el archivo descargado y seguimos los pasos de instalación. Para los instaladores de Windows y OSX se incluye una instalación automática del lenguaje Csound en nuestro sistema por lo que una vez instalado Cabbage todo estará listo para usar.

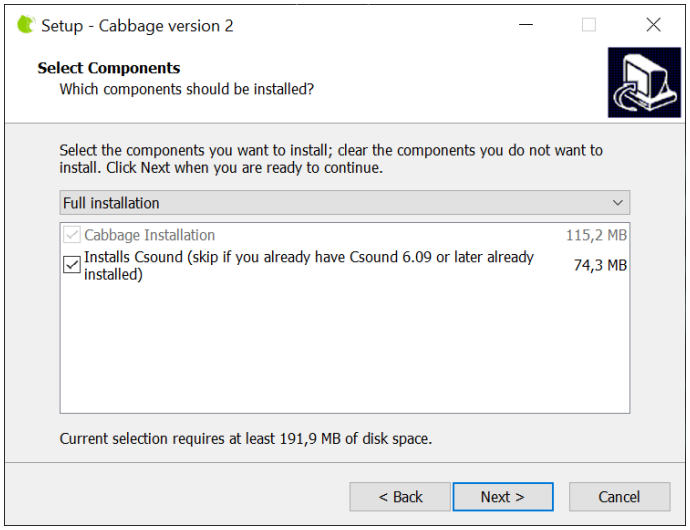

Figura 6.2: Pasos de la instalación

- Por último podemos ejecutar el acceso directo que se instala automáticamente en nuestro escritorio y empezar a usar Csound. Como se observa, Csound tiene una instalación muy fácil en estos sistemas.

6.2– Opciones del IDE

La extensión de los archivos de código Csound es **.csd**. Por supuesto, Cabbage puede abrir y ejecutar estos archivos además de que podemos encontrar una enorme librería de ejemplos que trae la instalación del IDE por defecto:

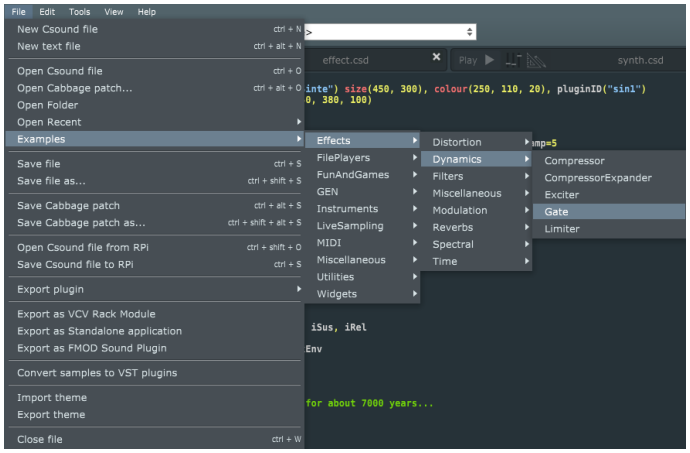

Figura 6.3: Ejemplos disponibles en el IDE

6.2.1. Creando un nuevo archivo

Para crear un nuevo archivo .csd basta con clicar el icono correspondiente en la barra superior(el primero empezando por la izquierda) o clicando en **File>New Csound File**. Aparecerá una ventana con cuatro posibles opciones:

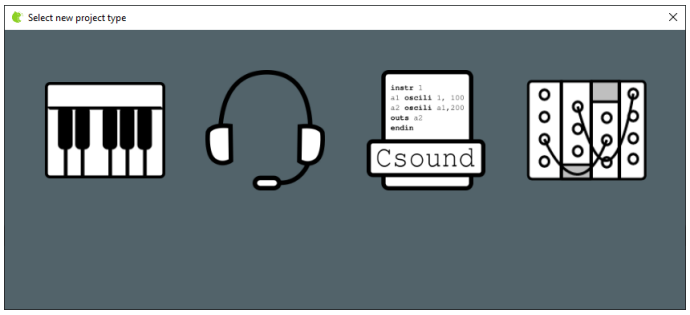

Figura 6.4: Sintetizador, Efecto, Archivo Csound y VCV Rack

Se trata de cuatro plantillas que nos aporta Cabbage para facilitarnos el desarrollo de nuevos instrumentos, hablemos de cada una de estas opciones:

- **Sintetizador**: Aporta el código fundamental de un teclado funcional sin efectos, del cual podemos partir como base para crear nuestros sintetizadores y añadir la serie de modificadores que deseemos.
- **Efecto**: Aporta el código de un efecto básico de ganancia. En base a este código podemos crear el efecto que deseemos para poder modificar a gusto las ondas de sonido generadas por nuestros instrumentos.
- **Archivo Csound**: Genera un archivo .csd completamente vacío, es la opción que escogeríamos si no tenemos predilección por las demás o si nuestro objetivo es crear un software no totalmente convencional a lo que suele verse en Csound.
- **VCV Rack**: Esta es la plantilla más novedosa hasta la fecha en Cabbage. Nos da facilidades para exportar nuestro código como módulos **VCV Rack** y al usarla genera el código base de un efecto de ganancia modular listo para ser exportado y usado en softwares de estación de trabajo digital (EAD) o (DAW) por sus siglas en inglés.

6.3– La etiqueta <Cabbage>

A diferencia del resto de etiquetas, la etiqueta <Cabbage>es exclusiva al IDE y proporciona funcionalidades para el diseño de la interfaz de usuario. Veamos algunos de sus usos:

6.3.1. Los Widgets

Llamaremos a los diferentes elementos de interfaz gráfica que aporta Cabbage, widgets. Podemos dividirlos en dos tipos: interactivos (botones, sliders, barras de selección, etc...) y no interactivos (imágenes, indicadores, etc...).

Empecemos con un ejemplo de uso de un slider para comprender la sintaxis:

```
1 <Cabbage>
2    rslider bounds(10, 10, 100, 100), range(0, 1, .5)
3 </Cabbage>
```

Código 6.1: Ejemplo básico de un widget

En nuestra figura vemos que para hacer uso de un widget empezamos por escribir su nombre identificativo de tipo, en este caso **rslider**. Mástarde podemos definir una serie de identificadores para personalizar nuetro widget. Para especificar la posición de nuestro widget y su tamaño usaremos **bounds(x, y, width, height)**. En el caso de nuestro ejemplo estamos posicionando nuestro slider en las coodernadas XY (10,10) y le estamos dando un tamaño de 100*100 píxeles.

Podemos usar también el identificador **range(min, max, value, skew, incr)**. Sus valores min y max marcan el mínimo y máximo valor del slider en cuestión, el resto de parámetros son opcionales. **value** indica el valor inicial del slider. **skew** puede usarse para determinar la salida de datos del slider de forma no lineal, su valor predeterminado es 1 pero al darle un valor por ejemplo de 0.5 conseguiríamos una salida de datos exponencial. **incr** determina el tamaño de los pasos incrementales que da el slider a usarlo, por ejemplo con un valor 0.4, de un valor cualquiera del slider a sus adyacentes habría necesariamente una distacia en valor a 0.4.

En nuestro ejemplo hemos creado un slider cuyo rango de valores va del 0 al 1 y cuyo valor inicial es 0.5.

6.4– Algunos Widgets útiles

Veamos algunos de los widgets más usados en Cabbage:

6.4.1. Form

El widget **Form** nos sirve para crear la ventana de nuestra interfaz de usuario. Sobre este widget colocaremos el resto de elementos de nuestra interfaz.

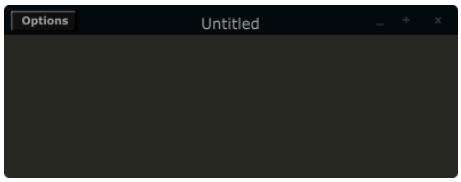

Figura 6.5: Widget: Form

Este es un ejemplo básico de **Form**:

```
1 form size(450, 500), caption("Mi Form"), pluginID("mif1"), colour(40,
     40, 12)
```

Código 6.2: Ejemplo de widget: Form

Siendo **size()** el tamaño de la ventana, **caption()** el String que muestra la cabecera de la ventana, **pluginID()** el identificador del widget y **colour()** su color en formato rgb.

6.4.2. Check Box

El widget **CheckBox** nos sirve para marcar una casilla que por ejemplo active y desactive la generación de una de las ondas de salida de nuestros instrumentos.

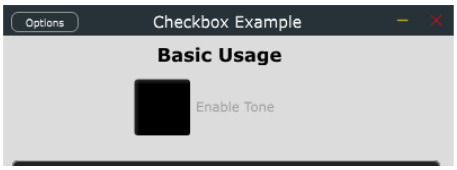

Figura 6.6: Widget: CheckBox

Este es un ejemplo básico de **CheckBox**:

```
checkbox bounds(116, 38, 150, 50), channel("checkbutton1"), text("
    Enable Tone", "Disable Tone")
```

Código 6.3: Ejemplo de widget: CheckBox

Siendo **bounds()** la posición y tamaño del widget en la ventana, **channel()** el identificador que vinculará el **checkBox** con una variable de nuestro código y **text()** El texto que aparece junto al **CheckBox** mientras esté y no esté pulsado.

6.4.3. Button

El widget **Button** crea un botón al que podemos dar cualquier tipo de uso como por ejemplo el de inicio de grabación o activación de algún instrumento desde un instante determinado.

Figura 6.7: Widget: Button

Este es un ejemplo básico de **Button**:

```
button bounds(116, 38, 150, 50), channel("button1"), text("Enable Tone
    ", "Disable Tone")
```

Código 6.4: Ejemplo de widget: Button

Siendo sus funciones de funcionamiento análogo a lo visto previamente en el **CheckBox**.

6.4.4. Keyboard

El widget **Keyboard** nos sirve para desplegar un teclado de formato occidental. Es realmente útil si queremos programar un instrumento sintetizador o si queremos tener una vía rápida de ejecutar samples de forma visual y práctica.

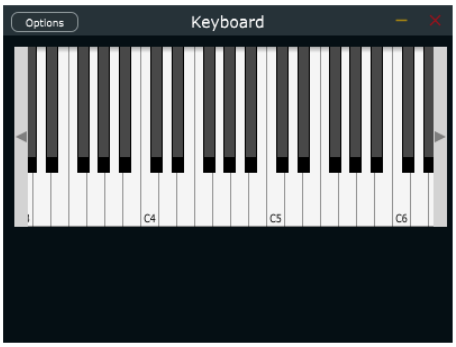

Figura 6.8: Widget: Keyboard

Este es un ejemplo básico de **Keyboard**:

```
keyboard bounds(10, 10, 385, 160), identchannel("widgetIdent")
```

Código 6.5: Ejemplo de widget: Keyboard

Siendo **identchannel()** el identificador del widget respecto al código Csound de manera análoga al funcionamiento de **channel()**.

6.4.5. Signal Display

El widget **SignalDisplay** nos servirá, como su propio nombre indica, para mostrar señales de onda de manera visual.

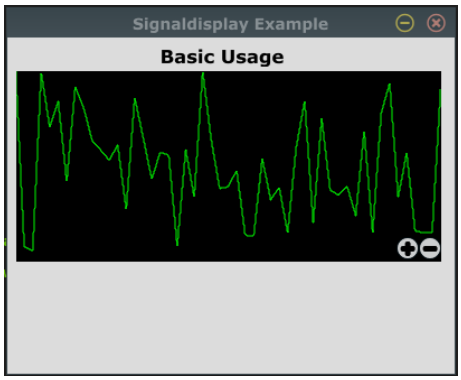

Figura 6.9: Widget: Signal Display

Este es un ejemplo básico de **SignalDisplay**:

```
signaldisplay bounds(8, 30, 380, 170), colour("lime"), backgroundcolour
    ("black"), displaytype("waveform"), signalvariable("aSig")
```

Código 6.6: Ejemplo de widget: SignalDisplay

Siendo **displaytype()** el tipo de onda a mostrar de entre las siguientes opciones: 'spectrogram', 'spectroscope', 'waveform'y 'lissajous'. Y **signalvariable()** donde instroduciremos la variable con la onda que queremos que se muestre.

6.5– Exportando nuestros instrumentos

Una vez hemos terminado de programar el código de un instrumento, necesitamos alguna manera de hacer que ese código sea útil en el mundo real. Para ello Cabbage ofrece una serie de opciones de exportación del instrumento para que podamos usarlos donde queramos (ya sea sobre un software de terceros o de forma unitaria) y saquemos provecho de ellos.

Estas son las opciones de exportación de Cabbage:

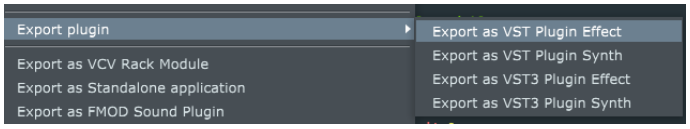

Figura 6.10: Las opciones de exportación del IDE

En primer lugar averigüemos qué es un **VST**:

Un plugin **VST** (**Virtual Studio Technology**) es una interfaz, en este caso digital, capaz de simular un instrumento o aportar un módulo entrada/salida para añadir efectos de sonido. Para usar un **VST** necesitamos un software compatible con este formato que sirva de base y desde el que ejecutemos nuestro **VST**. También es interesante destacar que **VST3** es el formato más moderno de **VST** con un código fuente más robusto y de fiar. El cual es también una de las opciones de exportación de Cabbage.

Será nuestro estándar de exportación, aunque Cabbage ofrece otras opciones como:

- **VCV RackModule**: Siendo parecido al formato VST, es el tipo de módulo del que hablábamos en la sección de plantillas de creación de archivos en Cabbage. Es un formato de código abierto con una amplia comunidad y una documentación robusta en su web `https://vcvrack.com/Fundamental`

- **Standalone application**: Para generar un archivo ejecutable .exe que podremos usar en cualquier momento siempre y cuando tengamos Cabbage instalado en el equipo. Es interesante si por ejemplo codificamos un instrumento que pueda usarse como tal sin necesidad de otros softwares como un teclado digital.

- **FMOD Sound Plugin**: El formato usados por FMOD, **https://www.fmod.com/** y que está dedicado a la composición de sonido para juegos. Usando este formato, podremos generar plugins directamente compatibles con FMOD Studio.

En cualquier caso, Cabbage generará los archivos convenientes de exportación incluidos archivos .csd y .dll en el caso de sistemas windows. Será importante que mantengamos todos los archivos generados para un instrumento en el mismo directorio y así evitemos conflictos en la ejecución.

CAPÍTULO 7

Fundamentos del Sonido

7.1– Introducción

Este capítulo tiene como función dar una breve introducción a la teoría física del sonido, en concreto a los conceptos fundamentalmente necesarios para entender los ejemplos expuestos en esta guía de Csound. Se presenta por ello como capítulo anexado o capítulo extra de modo que sirva de referencia rápida en otras partes del documento y de manera que un lector con manejo en estos términos pueda saltar su contenido cómodamente.

7.2– El Audio Digital

Para definir el audio digital debemos empezar por saber qué es el sonido:

7.2.1. ¿Qué es el sonido y cómo se transmite?

Sonido: "*Sensación producida en el órgano del oído por el movimiento vibratorio de los cuerpos, transmitido por un medio elástico, como el aire.*"[1]

A ese movimiento vibratorio que se transmite y viaja por el medio podemos llamarlo "Onda de Sonido". Y la forma más simple de describir un movimiento vibratorio, es decir, la onda más simple de todas; es mediante la forma senoidal:

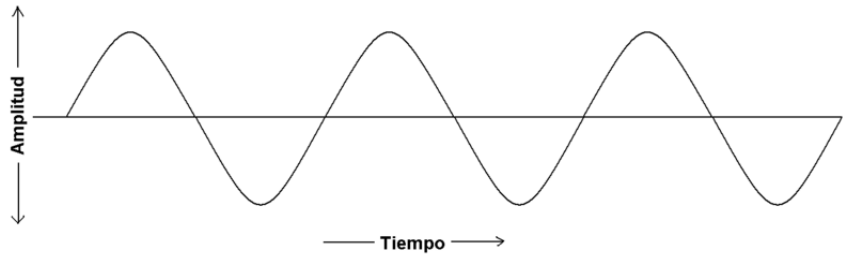

Figura 7.1: Onda Senoidal

[1]REAL ACADEMIA ESPAÑOLA: Diccionario de la lengua española, 23.ª ed., [versión 23.3 en línea]. ¡https://dle.rae.es¿[05 de julio de 2020].

Como sabemos, los medios transmisores están formador por moléculas que ocupan un determinado espacio. Podemos por lo tanto imaginar a una molécula que describe el movimiento vibratorio descrito anteriormente. Podríamos también dedir que cuando la molécula sobrepasa el punto inicial o punto 0 que definimos en la gráfica, la molécula está empujando al resto de moléculas que encuentra en su camino. De forma análoga, cuando la posición de la molécula tiene un valor menor al inicial decimos que la molécula está tirando del resto de moléculas de su entorno.

De esta manera se produce la transmisión del sonido.

7.2.2. La onda de sonido y sus características

Quedaba definida la onda de sonido en el apartado anterior. Si a continuación le añadimos la información de esa vibración de la que hablábamos es constante se producirá lo que llamamos "Onda Periódica".

Toda onda periódica posee 4 características:

- **Periodo**: Es la cantidad de tiempo que tarda la forma de la onda en repetirse, lo llamaremos T y lo expresaremos en segundos.
- **Amplitud**: Distancia máxima de los puntos de la onda respecto a la posición de eje Y (eje "Tiempo'én la figura). Podemos definirla también como la fuerza con la que las moléculas del medio consiguen empujar o tirar del resto de moléculas de su entorno.

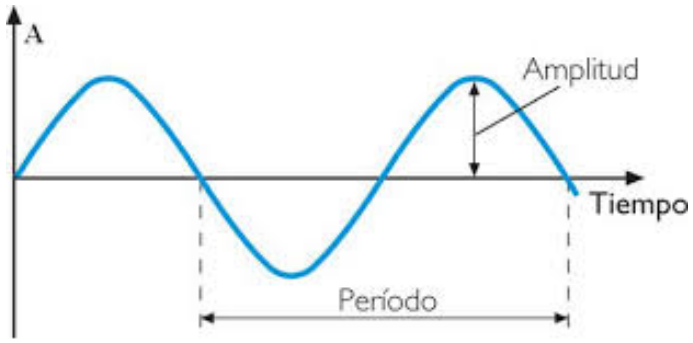

Figura 7.2: Periodo y Amplitud

- **Frecuencia**: La frecuencia de una onda expresa la cantidad numérica de veces que repite su movimiento durante un tiempo determinado. Si la definimos respecto al periodo decimos que es la cantidad numérica de periodos (o repeticiones de la forma de la onda) que ocurren durante un segundo. Se mide en Hercios o Hz, la representaremos con f y podremos calcularla fácilmento puesto que es la inversa del periodo:
  - Frecuencia = 1/Periodo
  - Periodo = 1/Frecuencia
- **Longitud de Onda**: Se trata de la distancia que va del punto inicial al punto final del recorrido de la onda marcada por un periodo. Se mide en metros.
- **Fase**: Punto de partida de la onda. Podemos observar al representar la onda en un gráfico y fijarnos en que el valor inicial del eje Y que no tiene que ser nesesariamente 0.

7.2.3. El Sampleo y Sample Rate

Para representar de manera digital una onda de sonido acústica necesitamos convertir sus valores analógicos a digitales. Esta onda tendrá un valor distinto por cada instante de tiempo y para conseguir recoger de alguna manera estos datos en un computador necesitaremos el concepto de sampleo.

El sampleo consiste en recoger un número determinado de valores en formato digital de una onda sonora por cada segundo de duración.

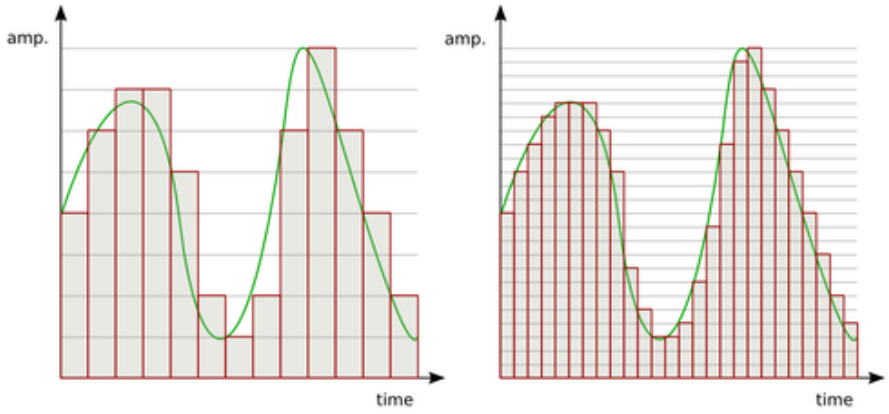

Figura 7.3: Ejemplo de sampleo de una onda

Aquí entra el concepto de "sample rate", que determina la frecuencia de muestreo en la recogida de datos de la onda. En la figura anterior podemos observaruna misma onda de sonido a la que se le realiza un sampleo primero con sample rate menor (izquierda), y con un sample rate mayor (derecha).

Por último debe destacarse que un sample rate mayor no implica necesariamente una coversión digital más feaciente para nosotros a un nivel pragmático puesto que el nivel máximo de frecuencia de datos sonoros que puede captar en oído humano ronda los 20Khz.

Contamos además con que debe respetarse el **Teorema de muestreo de Nyquist-Shannon**:

"Para representar una onda de manera digital que contenga frecuencias de hasta X Hz, es necesario usar un sample rate de al menos 2X muestras por segundo."

De otra manera, los valores digitales no representarían la onda de manera correcta dando lugar al aliasing y al muestreo incorrecto.

Se muestra a continuación una figura de ejemplo de sampleo en la que se toma un sample rate de 40000Hz. De la primera onda, que es de 10KHz observamos que recogemos los datos suficiente cada segundo como para captar toda la información contenida en ella. De hecho podemos observar también que con un sample rate de 20KHz en lugar de 40KHz también captaríamos toda la información necesaria como para recoger la onda al completo en su formato digital cumpliéndose así el Teorema de muestreo de Nyquist-Shannon.

Sin embargo la onda del segundo gráfico tiene una frecuencia de 30KHz y esto implica que con un sample rate de 40KHz como el mostrado, daría resultado una conversión de onda errónea. Dando incluso para este ejemplo una muestra de onda digital idéntica a la del primer gráfico. Sería necesario un sample rate de al menos 60KHz para recoger feacientemente los datos y conseguir una conversión digital satisfactoria de la onda.

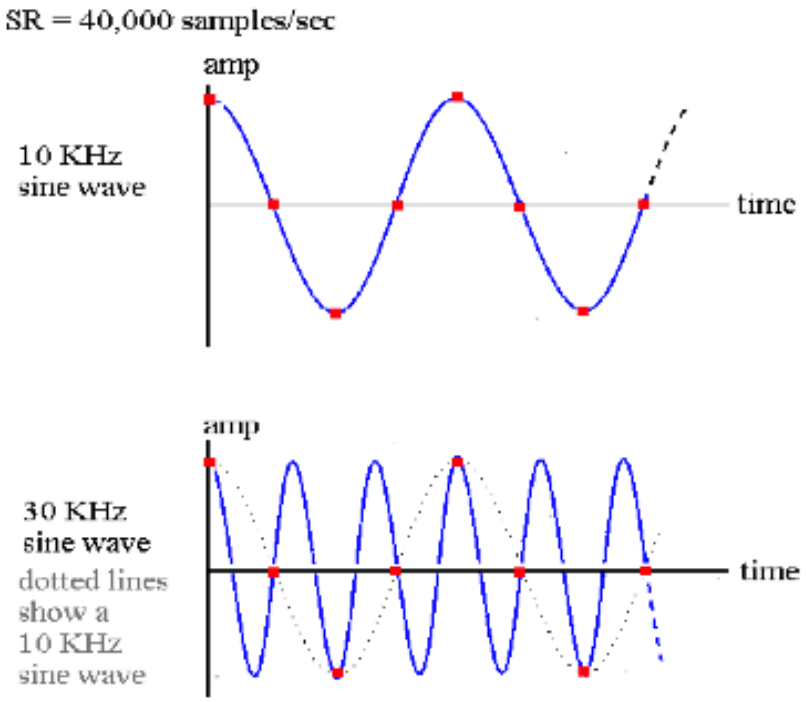

Figura 7.4: Un mismo sample rate para ondas distintas

7.3– Conceptos interesantes

Hablemos a continuación de algunos conceptos no necesariamente intrínsecos al sonido digital pero interesantes para entender mejor los conceptos necesarios para el uso adecuado de Csound.

7.3.1. El decibelio

El decibelio o 'db' es una unidad de medida que representa la intensidad de un sonido. Es la décima parte de un belio y siempre que hablamos de decibelio lo hacemos respecto a un valor de referencia preestablecido de intensidad, normalmente el marcado por el límite por debajo del oído humano en su capacidad para oír: $I_0 = 10^{-12} W/m^2$ que se da en los 1000Hz.

La fórmula para calcular los decibelios es: $10 * log_{10} * \frac{I}{I_0}$, es por tanto una fórmula logarítmica que depende de su valor de referencia I_0. Para una relación $\frac{I}{I_0}$ de 10 tenemos 10db, para una relación de 100 tendremos 20db, para 1000 30db, etc...

Un dato útil a tener en cuenta es que al doblar la amplitud de una onda de sonido obtenemos una diferencia de +6db. De manera análoga si partimos por la mitad el valor de la amplitud de la onda obtenemos un cambio de -6db.

7.3.2. El ADSR

ADSR son las siglas de **A**ttack, **D**ecay, **S**ustain y **R**elease en una onda y conforman la opción más común de envelope o envolvente sonoro, es decir, proporcionan parámetros para poder controlar una onda de sonido.

Veamos cada uno de estos parámetros para entenderlos en conjunto:

- **Attack**: El Attack o Ataque sería lo ocurrido antes de que la onda decaiga y se estabilice. Por ejemplo, un golpe de platillo produce un sonido con mucho ataque, el sonido de una nota tocada en una flauta dulce tendría normalmente poco ataque.

- **Decay**: El decay o decaimiento es lo sucedido entre el ataque ensu máximo punto y la fase estable de la onda. Al rasgar las cuerdas de una guitarra con una púa se produce un sonido con bastante decay.

- **Sustain**: El sustain o sostenibilidad es lo referido a la parte estable de la onda, su duración e intensidad máxima. Un golpe de caja tiene poco sustain, una nota tocada al aire en la cuerda de un bajo eléctrico tiene mucho sustain.

- **Release**: El release es la parte de la onda comprendida entre la fase estable o de sustain y la llegada al valor 0 de intensidad. Los instrumentos de cuerda tienen por lo gerenar una fase de release notable.

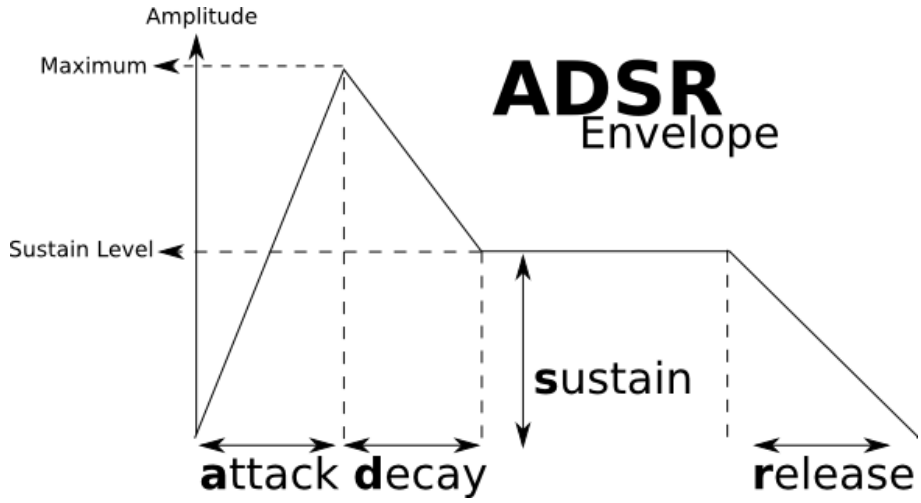

Figura 7.5: El envolvente ADSR de una onda

Es común en el sonido digital poder modificar estos parámetros a placer en instrumentos como los sintetizadores o en prácticamente cualquier instrumento con componentes eléctricos.

7.3.3. El Cutoff y la resonancia

Veamos este par de conceptos útiles para completar y mejorar nuestro sonido:

- **Cuttoff**: Se trata de un filtro de frecuencias. Normalmente se usa para bloquear determinados rangos de frecuencias altas, siendo esto de tipo LP (LOW PASS) aunque en instrumentos modernos pueden encontrar Cutoffs de tipo HP (HIGH PASS) que bloquearían en este caso frecuencias más bajas de lo deseado.

- **Resonancia**: Es el efecto que ocurre por ejemplo cuando la mesa que sujeta unos altavoces en marcha empieza a vibrar mientras algunos sonidos son producidos, querrá decir que el sonido que sale de esos altavoces producen una frecuencia coincidente con la frecuencia de resonancia de la tabla de la mesa y el sonido por tanto se ve amplificado. En el audio digital puede encontrarse la resonancia como un efecto más a añadir a nuestro arsenal de producción musical.

7.3.4. La ganancia o gain y sus diferencias con el volumen

Podríamos definir el volumen en el audio digital así: "Cómo de ruidoso es algo **después** de salir por el amplificador de nuestro equipo de sonido."

La ganancia sin embargo sería así: "Cómo de ruidoso es algo **antes** de pasar por nuestros efectos y preamplificadores."

La diferencia fundamental, además del tecnicismo de cuándo ocurre cada cosa, es que una variación en la ganacia produce una variación en el tono del sonido mientras que una variación en el volumen no produce cambios de tono.

7.4– Efectos más usados o conocidos

CAPÍTULO 8

Bibliografía Comentada

Se presenta la bibliografía comentada de cada referencia usada en el presente documento.

8.1– Csound FLOSS Manual

- **Tipo de fuente**: Libro online
- **Última actualización**: Marzo de 2015
- **Dificultad**: Recomendado para principiantes.
- **Autor/es**: Joachim Heintz, Iain McCurdy, Andres Cabrera, Alex Hofmann, Alexandre Abrioux, Rory Walsh, Martin Neukom, Jim Aikin, Jacques Laplat, Menno Knevel, Bjorn Houdorf, Christopher Saunders, Oeyvind Brandtsegg, Oscar Pablo di Liscia, Peiman Khosravi, Steven Yi, Stefano Bonetti, Victor Lazzarini, Ed Costello, François Pinot, Tarmo Johannes, Nicholas Arner, Nikhil Singh, Richard Boulanger, Michael Gogins, Anton Kholomiov.

8.1.1. Referencia

Comunidad de desarrolladores del lenguaje Csound (marzo de 2015). Csound FLOSS Manual. FLOSS Manuals. `http://write.flossmanuals.net/csound/preface/`

8.1.2. Comentario

El **Csound FLOSS Manual** es la principal fuente bibliográfica de este documento[1]. Ha sido escrito por el núcleo de la comunidad de desarrolladores del código abierto de Csound y podría decirse sin miedo a equivocarse que es la fuente más completa y accesible para aprender sobre el lenguaje en la red.

Su estructura es la de un libro electrónico y la gran mayoría de sus capítulos cuenta con ejemplos de código completos e ilustrativos. Tiene además un capítulo dedicado por completo a explicar conceptos básicos sobre el sonido y el mundo de la edición sonora, aportando una buena base sólida del conocimiento previo que se recomendaría tener antes de probar un lenguaje dedicado al sonido.

Con todo y por estar realizado de mano directa por algunos de los principales autores del código fuente de Csound, **Csound FLOSS Manual** resulta ser un compendio actual de todos los conocimientos del lenguaje y es por ello que se recomienda encarecidamente su estudio y comprensión si se quiere aprender realmente los fundamentos de Csound.

[1] Se ha sido especialmente extenso en el comentario de esta referencia por ser la principal fuente bibliográfica.

Csound FLOSS Manual ha servido como base al presente documento tanto en su estructura como en su metodología para presentar los conocimiento.

En ocasiones **Csound FLOSS Manual** puede resultar algo tedioso precisamente por la completitud de sus ejemplos y explicaciones, cosa que se ha tratado de solventar en el presente documento al simplificar alguna explicaciones, pero es precisamente por ello que lo debido es recurrir al **Csound FLOSS Manual** para empezar a profundizar realmente en lo aprendido tras revisar los conocimientos aquí mostrados.

8.1.3. Estructura

Csound FLOSS Manual tiene 13 capítulos principales más 2 capítulos extra que resumiremos a continuación para tener una referencia útil acerca de dónde buscar para profundizar en cada concepto:

- **01 BASICS**: Conceptos fundamentales sobre el sonido y su procesamiento. Muy útil incluso para el que sólo esté interesado en el mundo del sonido y no necesariamente en Csound.
- **02 QUICK START**: Información más básica sobre el lenguaje y sus IDEs. Cómo ejecutar programas, cómo exportarlos, etc...
- **03 CSOUND LANGUAGE**: Fundamentos del lenguaje. Su sintaxis y las diferentes propiedades básicas como tipos de variables o funciones.
- **04 SOUND SYNTHESIS**: Conceptos físicos aplicados a la síntesis del sonido. Posee ejemplos más complejos para complementar las explicaciones algo más academicas.
- **05 SOUND MODIFICATION**: Capítulo dedicado principalmente a las capas de envoltura y efectos de sonido mediante filtros.
- **06 SAMPLES**: Capítulo dedicado a la lectura y escritura de archivos y a su consecuente aplicación en lo referente a los datos del sonido.
- **07 MIDI**: Dedicado a lo referente al MIDI (Musical Instrument Digital Interface) dando una extensa explicación acerca de cómo vincular nuestros instrumentos físicos o virtuales a nuestro código.
- **08 OTHER COMMUNICATION**: Capítulo corto pero interesante acerca de cómo combinar Csound con OSC y projectos con Arduino.
- **09 CSOUND IN OTHER APPLICATIONS**: Capítulo dedicado a explicar cómo combinar Csound con otros lenguajes y tecnologías dedicadas como PureData o Ableton Live.
- **10 CSOUND FRONTENDS**: Nos da una revisión media sobre los principales entornos de programación entre los que podemos elegir para usar Csound.
- **12 CSOUND AND OTHER PROGRAMMING LANGUAGES**: Como el propio título indica, se nos explica cómo y con qué sintaxis podemos compilar código Csound en diferentes lenguajes como Python o Haskell.
- **13 EXTENDING CSOUND**: Capítulo corto que nos da una pequeña introducción acerca de cómo aportar al código abierto del lenguaje mediante, por ejemplo, la creación de nuevos opcodes.
- **OPCODE GUIDE**: Capítulo extra que aporta información más extensiva acerca del uso y funcionamiento de los opcodes.

- **APPENDIX**: Por último el apéndice, que aporta recomendaciones de nomenclatura, un glosario corto y una librería de enlaces con webs de información intereseante sobre Csound y el mundo del sonido.

8.2– The Canonical Csound Reference Manual

- **Tipo de fuente**: Manual online
- **Última actualización**: Enero de 2020
- **Dificultad**: Necesario conocimiento previo.
- **Autor/es**: Barry Vercoe, Comunidad de Csound.

8.2.1. Referencia

Comunidad de Csound (Enero de 2020). The Canonical Csound Reference Manual. Csound. `https://csound.com/docs/manual/index.html`

8.2.2. Comentario

The Canonical Csound Reference Manual es el manual más extenso de información sobre Csound en la red. Es por ello algo menos accesible para principiantes en el lenguaje pero el mejor compendio de referencias si necesitamos información acerca de, por ejemplo, un opcode concreto. Este manual ha sido creado por los desarrolladores del lenguajes Csound y se actualiza conjuntamente con el propio lenguaje para reflejar las nuevas características del código fuente.

En referencia a este documento, ha servido para completar y complementar conocimientos en el uso y sintaxis concretos de algunos métodos y opcodes, y descripción de algunos conceptos del lenguaje que sólo pueden encontrarse en el propio manual. Se recomienda su uso como guía de referencia siempre que se sepa previamente qué se está buscando.

Su contenido se divide en tres partes principales, una primera describiendo conceptos de Csound, una segunda extendiendo este conocimiento e introduciendo conceptos como la generación y edición de señales, y una tercera a modo de biblioteca de referencia de todos los opcodes (incluyendo los obsoletos) disponibles en Csound. Posee además una biblioteca de archivos descargables con cientos de ejemplos de programación y apartados con datos útiles como una tabla de conversión de valores de onda a nota musical.

8.2.3. Estructura

- **I. Overview**: Da una base fundamental del uso del lenguaje, como punto a favor posee un enlace a una página de la misma guía cada vez que se menciona algún opcode o palabra reservada por lo queda bien estructurado.
- **II. Opcodes Overview**: Ofrece información extensa acerca del uso de los principales opcodes de Csound dividiendo el capítulo según sus tipos (De control de instrucciones, de edición de señales de audio, relacionadas con el MIDI, etc...)
- **III. Reference**: Biblioteca de referencia de los diferentes opcodes y operadores del lenguaje que se recomienda tener siempre a mano. El resto de manual hace constante referencia a esta sección.
- **Apéndices**: Ofrece información útil sobre diversos temas relacionados al sonido o a la sintaxis concreta de Csound.

8.3– Cabbage Docs

- **Tipo de fuente**: Documentación online
- **Última actualización**: Febrero de 2020
- **Dificultad**: Recomendado para principiantes.
- **Autor/es**: Rory Walsh, Iain McCurdy, Gordon Boyle.

8.3.1. Referencia

Walsh R.(febrero de 2020). Cabbage Docs. Cabbage. `https://cabbageaudio.com/docs/introduction/`

8.3.2. Comentario

Cabbage Docs es la fuente bibliográfica de todo conociemiento referente al uso del IDE Cabbage y a sus Widgets. Ha sido escrita por Rory Walsh, principal desarrollador y autor de Cabbage, por lo que puede considerarse una fuente fiable de conocimiento. Entre sus secciones ofrece una corta introducción a Csound que se recomienda usar como repaso al lenguaje, una sección dedicada a explicar el uso de Cabbage como IDE que cuenta con buenos ejemplos prácticos, y una sección a modo de biblioteca de referencia de uso de los diferentes Widgets existentes la cual se recomienda usar como API de ejemplos de uso específico del IDE.

8.3.3. Estructura

Cabbage Docs tiene cuatro secciones:

- **Beginners(Csound)**: Da una introducción al lenguaje Csound.
- **Using Cabbage**: Da una introducción a cómo sacar provecho del IDE Cabbage al usar Csound
- **Advanced Features**: Extiende algo más sobre el uso de Csound y sus conceptos de uso avanzado.
- **Cabbage Widgets**: Sirve de biblioteca de referencia de los diferentes Widgets aportados por Cabbage.

8.4– CS Csound: Página oficial

- **Tipo de fuente**: Página Web
- **Última actualización**: Junio de 2020
- **Dificultad**: Recomendado para principiantes.
- **Autor/es**: Comunidad de Csound.

8.4.1. Referencia

Comunidad de Csound(Junio de 2020). CS Csound. Csound. `https://csound.com/index.html`

8.4.2. Comentario

Página web oficial de Csound en la que se comparten noticias y nuevas publicaciones. Tiene una muy buena introducción al lenguaje la cual ha servido de base para el primer capítulo de este documento. Se recomienda el uso esta web como nexo de los diferentes contenidos de Csound puesto que también pueden encontrarse enlaces dedicados a la referencia de otros portales de contenido.

8.5– Repositorio csound-live-code

- **Tipo de fuente**: Documentación de repositorio
- **Última actualización**: Mayo de 2019
- **Dificultad**: Necesario conocimiento previo.
- **Autor/es**: Steven Yi.

8.5.1. Referencia

Yi S.(Mayo de 2019). csound-live-code doc. Github. `https://github.com/kunstmusik/csound-live-code/blob/master/doc/intro.md`

8.5.2. Comentario

Documentación del repositorio **csound-live-code** del desarrollador Steven Yi, colaborador de Csound y autor de blue y la mayor parte del contenido sobre **Live Coding** en Csound.

La documentación del repositorio ha servido como base para el capítulo "Haciendo música en directo"de este documento.

Se recomienda revisar esta fuente para ahondar en el concepto de codificación en vivo de Csound aunque sería preferible llegar a este contenido con conocimientos previos sobre el lenguaje.

8.6– Canal de youtube: Steven Yi

- **Tipo de fuente**: Canal de Youtube
- **Última actualización**: Junio de 2020
- **Dificultad**: Necesario bastante conocimiento previo.
- **Autor/es**: Steven Yi.

8.6.1. Referencia

Yi S.(Junio de 2020). Canal de youtube: Steven Yi. Youtube. `https://www.youtube.com/channel/UCiO3nb3sN5GsZUIyxrgXh0Q`

8.6.2. Comentario

Canal personal de youtube de Seteven Yi al que sube ejemplos de uso de **Live Coding Csound** haciendo música en directo. Se recomienda bastante conocimiento previo antes de visitar el canal puesto que los ejemplos son únicamente visuales y no van dirigidos como contenido pedagógico sino más bien divulgativo acerca de lo que se puede hacer con Csound.

Respecto a este documento ha servido como base de algunos de los ejemplos de código mostrados.

CAPÍTULO 9

Anexos

9.1– Cómo instalar un PWA en Chrome

En primer lugar visitamos la página web del PWA Y hacemos click es la pestaña de opciones. Acto seguido vamos a la opción: **Más herramientas>Crear acceso directo...**.

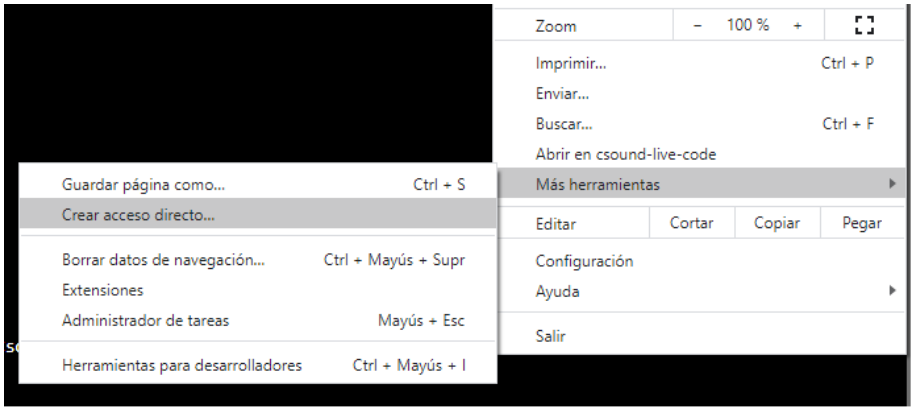

Figura 9.1: Crear acceso directo a un PWA

Chrome nos preguntará por confirmación para realizar la instalación. Bastará con pulsar en "Aceptar"y se creará un acceso directo en nuestro escritorio que podremos ejecutar siempre que queramos.

CAPÍTULO 10

Definición de objetivos

Estos son los objetivos generales del presente TFG. Se agrega a cada uno de ellos una serie de objetivos específicos:

- Dar a conocer en mayor medida el lenguaje de programación Csound.
  - Expresar el contenido de los capítulos en formato divulgativo.
  - Transmitir de manera feaciente las ventajas y desventajas de Csound en un capítulo introductorio.
- Proporcionar una guía de aprendizaje introductorio al lenguaje Csound.
  - Mostrar los contenidos en orden por dificultad, para que se pueda seguir el documento linealmente.
  - Mostrar, al menos superficialmente, varios ámbitos del lenguaje como el "Live Coding", la sintaxis base del lenguaje, y otras posibilidades.
  - Dar una introducción práctica al IDE Cabbage.
- Proporcionar un documento de consulta rápida de conceptos del lenguaje Csound.
  - Mostrar los contenidos de forma unitaria, proporcionando además un índice ordenado y pragmático.
  - Proporcionar anexos con contenido relacionado como los fundamentos del sonido y otros conceptos útiles.
- Proporcionar ejemplos prácticos de programación usando el lenguaje Csound a modo de demostración capacitiva del lenguaje.
  - Acompañar la mayoría de lecciones con al menos un ejemplo de código real.
  - Dar referencias a otros portales de contenido de Csound que cuenten con una librería de ejemplos prácticos.
- Compilar una lista de referencias a portales de contenido de Csound de manera ordenada y comentada.
  - Realizar una bibliografía comentada, centrada especialmente en las principales fuentes del documento sirviendo así de guía para saber por dónde seguir para extender conocimientos.