Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

DISEÑO de una APLICACIÓN PARA APRENDER a USAR un SINTETIZADOR (1). DISEÑO de la INTERFACE GENERAL y de los OSCILADORES.

Autor: David Lence González
Tutor: Ángel Merchán Pérez
Cotutor: Sergio Plaza Alonso

Madrid, enero 2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: DISEÑO de una APLICACIÓN PARA APRENDER a USAR un SINTETIZADOR (1). DISEÑO de la INTERFACE GENERAL y de los OSCILADORES
Enero 2026

Autor: David Lence González

Tutor: Ángel Merchán Pérez
Departamento de arquitectura y tecnología de sistemas informáticos
ETSI Informáticos
Universidad Politécnica de Madrid
Cotutor: Sergio Plaza Alonso
Centro de Tecnología Biomédica
Universidad Politécnica de Madrid

Resumen

En este Trabajo de Fin de Grado se elabora el diseño e implementación de una aplicación educativa que simula un sintetizador, con el objetivo de enseñar al usuario la síntesis de audio de una manera sencilla. El proyecto no pretende diseñar un sintetizador completo, sino que se centra específicamente en el diseño de la interfaz general y en la implementación completa de los osciladores.

Contexto y motivación

La síntesis sustractiva constituye uno de los métodos más utilizados en la creación de sonidos electrónicos desde la creación de los sintetizadores analógicos en los años 1960 [2]. Este enfoque consiste en generar señales ricas en armónicos que posteriormente se procesan mediante filtrado y modulación para obtener el timbre deseado. A pesar de su importancia histórica y su presencia en la mayoría de sintetizadores modernos, tanto hardware como software, el aprendizaje de estos conceptos presenta una barrera de entrada significativa para principiantes, especialmente por la complejidad de las interfaces y la falta de retroalimentación visual inmediata.

La motivación principal de este trabajo es crear una herramienta didáctica que permita a estudiantes y aficionados comprender de forma intuitiva cómo funcionan los osciladores y sus interacciones, proporcionando además una visualización en tiempo real de las formas de onda generadas y un sistema de ayuda para un usuario principiante. Con esta herramienta el usuario podrá no sólo oír los osciladores, sino interpretar visualmente las modificaciones del sonido que se producen al manipular los controles.

El sistema se desarrolló en Python [3], utilizando Tkinter para la interfaz gráfica [4], NumPy para procesar las señales de audio [7], SoundDevice para la reproducción de audio [6] y Matplotlib para la visualización de formas de onda [5]. La estructura modular del proyecto separa la parte de síntesis de audio (audio/), la interfaz de usuario (ui/) y algunas funciones de visualización (visual/). Esta organización facilita el desarrollo de otras funcionalidades en proyectos futuros, hasta completar el diseño de un sintetizador típico.

La aplicación cumple todos los objetivos planteados. Se ha logrado crear una interfaz sencilla que permite visualizar de forma inmediata el efecto de cada parámetro sobre la forma de onda generada. Los mensajes de ayuda y las guías facilitan el aprendizaje. El código está bien estructurado lo que hace que sea fácil de añadir las componentes restantes del sintetizador típico.

Abstract

This Bachelor's Thesis addresses the design and implementation of an educational application for teaching audio synthesis using synthesizers. The project does not intend the design of a complete synthesizer, but focuses specifically on the design of the overall interface and on the complete implementation of oscillators.

Context and Motivation

Subtractive synthesis is one of the most widely used methods in the creation of electronic sounds since the emergence of analogic synthesizers in the 1960s [2]. This approach consists of generating harmonically rich signals that are subsequently processed through filtering and modulation to obtain the desired tone. Despite its historical importance and its presence in most modern synthesizers, both hardware and software, learning these concepts represents a significant entry barrier for beginners, especially due to the complexity of interfaces and the lack of immediate visual feedback.

The main motivation of this work is to create a didactic tool that allows students and enthusiasts to intuitively understand how oscillators work and how they interact, providing real-time visualization of the generated waveforms and a help system designed for beginners. In addition to the sound produced by the synthesizer, this tool provides a visual interpretation of the effects of the different controls on the tone.

The system was developed in Python [3], using Tkinter for the graphical interface [4], NumPy for audio signal processing [7], SoundDevice for sound reproduction [6], and Matplotlib for waveform visualization [5]. The software architecture follows a modular pattern that clearly separates the audio synthesis logic (audio/), the user interface (ui/), and the visualization utilities (visual/). This approach facilitates the development of other functionalities in future projects, that will complete a full-featured synthesizer.

The resulting application meets all the proposed objectives. An educational tool was successfully created that allows users to immediately visualize the effect of each parameter on the generated waveform. The integrated help system facilitates learning, while the presets provide starting points for experimentation. The code is well structured, facilitating future extension with the remaining synthesizer components.

Tabla de contenidos

1 Introducción **1**
2 Diccionario de conceptos **2**
3 Desarrollo **3**
3.1 Estructura del proyecto 3
3.2 Módulo de audio: Osciladores 4
3.2.1 Diseño del Oscilador 4
3.2.2 Sistemas de cents y desafinación 5
3.2.2.1 Fundamento del Sistema de cents 5
3.2.3 Unison: Distribución de voces 5
3.2.3.1 Concepto de Unison 5
3.2.3.2 Distribución de voces desafinadas 5
3.2.4 Creación de los arrays que representan una onda 6
3.2.5 Implementación de formas de onda 7
3.2.5.1 Forma Senoidal 7
3.2.5.2 Forma Cuadrada 8
3.2.5.3 Forma de onda Triangular 8
3.2.5.4 Forma de Sierra 9
3.3 Interacciones entre Osciladores 10
3.3.1 Suma simple (Modo “Ninguna”) 10
3.3.2 Modulación en Frecuencia (FM) 10
3.3.2.1 Teoría de la FM 10
3.3.2.2 Índice de Modulación 11
3.3.2.3 Implementación 11
3.3.3 Modulación en Anillo: 12
3.3.4 Sincronización 12
3.4 Sistema de notas musicales 14
3.4.1 Temperamento y cálculo de frecuencias a partir de una nota 14
3.4.2 Diccionario de notas 14
3.4.3 Conversión nota a frecuencia 15
3.4.4 Integración con el sistema de osciladores 16
3.5 Interfaz de Usuario 17
3.5.1 Estructura de la interfaz Tkinter 17
3.5.2 Inicialización del sistema 18
3.5.3 Llamada a construcción de componentes 19
3.5.3.1 Controles de osciladores 19
3.5.3.2 Controles de interacción 20
3.5.3.3 Botones de presets, reproducir sonido y ayuda 20
3.5.4 Sincronización inicial 21
3.5.5 Lambda: After-Idle 21
3.5.6 Método actualizar_onda() 22
3.5.7 Método reproducir_onda() 24
3.5.8 Gestión del cierre de la aplicación 24

3.6 Widgets personalizados 25
3.6.1 Entry editables 25
3.6.2 Widget Knob 26
3.6.3 Sistema de tooltips 27
3.6.4 Help Popups 27
3.7 Visualización en tiempo real 29
3.7.1 Integración Matplotlib-Tkinter 29
3.7.2 Estrategia de actualización visual y audio 30
3.7.3 Patrón Clear-Plot-Draw 30
3.8 Teclado virtual 31
3.8.1 Construcción del teclado 31
3.8.2 Sistema de mapeo geometría - nota 32
3.8.3 Detección de clicks 32
3.8.4 Scroll Horizontal 33
3.9 Presets 34
3.9.1 Diccionario de presets 34
3.9.2 Aplicación de presets 35
3.9.3 Integración con interfaz gráfica 35
3.10 Sistema de ayuda 36
3.11 Reproducción de audio 37
3.11.1 Reproducción desde Teclado Virtual 37
3.11.2 Reproducción directa con botón “Reproducir sonido” 38
4 Resultados y conclusiones 39
4.1 Cumplimiento de objetivos 39
4.2 Conclusiones personales 39
5 Análisis de Impacto 40
5.1 Impacto personal 40
5.2 Impacto Empresarial 40
5.3 Impacto Social 40
5.4 Impacto económico 40
5.5 Impacto Medioambiental 40
5.6 Impacto cultural 41
5.7 Relación con objetivos de desarrollo sostenible. 41
5.8 Decisiones de diseño basadas en el impacto 41
6 Bibliografía 42

1 Introducción

Este Trabajo Fin de Grado se basa en el desarrollo de un sintetizador didáctico en Python, orientado a la enseñanza y experimentación de conceptos básicos de síntesis de sonido.
El trabajo se basa en un sintetizador sustractivo básico, en el que se crea un sonido rico en armónicos, que luego se filtrará y modulará para obtener el timbre deseado.

La herramienta permite al usuario modificar parámetros de los dos osciladores independientemente, experimentar con diferentes formas de interacción (unison, detune, modulación en frecuencia, modulación en anillo y sincronización) y visualizar en tiempo real el resultado de sus acciones.

Los primeros sintetizadores analógicos empezaron a usarse en 1960 y su uso se ha generalizado en la actualidad tanto en su versión hardware como software. Sin embargo, lo más común es que el principiante empiece a usarlos sin un conocimiento previo de los principios de la síntesis de sonido, y por tanto el uso de estos instrumentos se vuelve más complicado de lo que debería, y no se les saca todo el provecho

Los objetivos específicos son:
- Desarrollar una interfaz gráfica clara, intuitiva y didáctica.
- Implementar los osciladores y los modos de interacción clásicos de la síntesis de sonido.
- Permitir la visualización simultánea de las ondas individuales y la onda combinada.
- Implementar la reproducción de audio de la onda generada.
- Implementar un sistema de ayuda y guía al usuario.

2 Diccionario de conceptos

- Cent
Unidad logarítmica de intervalo musical. Un semitono equivale a 100 cents y una octava a 1200 cents. Se utiliza para expresar pequeñas variaciones de forma más fácil.

- Detune (Desafinación)
Variación de la frecuencia que se aplica a un oscilador respecto a su valor base, expresada en cents.

- Oscilador
Elemento fundamental que se encarga de generar la señal de audio. Está definido por parámetros como frecuencia, amplitud, forma de onda y fase.

- Forma de onda
Representación matemática de la señal generada por un oscilador a lo largo del tiempo. En este proyecto utilizamos ondas senoidal, cuadrada, triangular y de sierra.

- Unison
Técnica de síntesis que consiste en generar varias voces del mismo oscilador, ligeramente desafinadas (detune) entre sí, y sumarlas para obtener un sonido más denso y ancho.

- Voz
Instancia de un oscilador en el modo unison (Por ejemplo si ponemos el unison a 3 se hacen 3 instancias, es decir 3 voces). Cada voz puede tener una afinación ligeramente diferente.

- Preset
Conjunto definido de valores de los parámetros del sintetizador que produce un sonido determinado. Se utilizan como punto de partida para experimentar.

- Modulación en Frecuencia (FM)
Técnica de síntesis en la que un oscilador (modulador) modula la frecuencia de otro (portador) , generando timbres complejos y ricos en armónicos.

- Modulación en anillo (Ring Modulation)
Técnica que combina dos señales mediante su multiplicación, produciendo sonidos metálicos o de campana.

- Sincronización de osciladores (Sync)
Técnica en la que el ciclo de un oscilador esclavo se reinicia siguiendo los ciclos de un oscilador maestro, produciendo sonidos brillantes y con alto contenido armónico.

3 Desarrollo

3.1 Estructura del proyecto

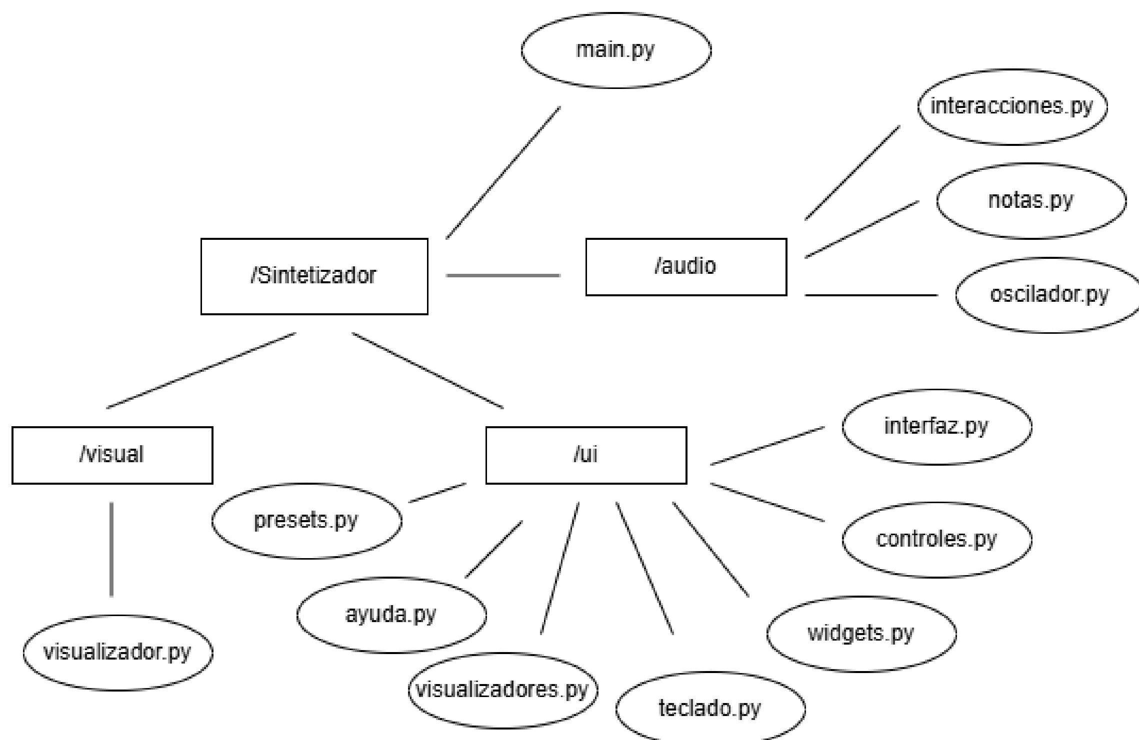
El proyecto se ha estructurado de manera modular, con el objetivo de facilitar el desarrollo, la comprensión del código y la extensión futura de la aplicación. La aplicación está organizada en tres módulos principales:

2.1.1 Módulo de Audio (audio/): Aquí está toda la síntesis y procesamiento de señales de audio, incluyendo la generación de ondas y las interacciones entre osciladores.

2.1.2 Módulo de Interfaz Gráfica (ui/): Tiene todos los componentes visuales y de interacción con el usuario, desde la ventana principal hasta los controles individuales.

2.1.3 Módulo de Visualización (visual/): Contiene algunas funciones de visualización de la onda en tiempo real.

Main.py es el punto de entrada desde donde se ejecuta la aplicación con el comando py main.py, que verifica dependencias antes de iniciar la interfaz. Este diseño por módulos, en vez de tener todo junto en el mismo archivo y carpeta, permite que estudiantes futuros añadan fácilmente nuevos componentes (filtros, envolventes, LFOs [1]) fácilmente.



3.2 Módulo de audio: Osciladores

3.2.1 Diseño del Oscilador

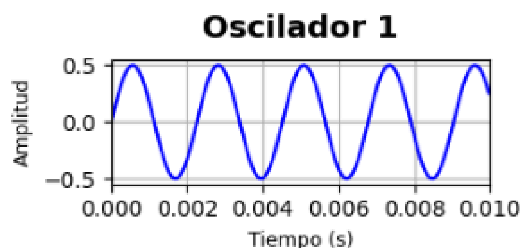
El Oscilador es la clase fundamental para la generación de sonido. Con él se crean los cuatro tipos de formas de onda básicas (senoidal, cuadrada, triangular y sierra) y también el unison y detune [8].

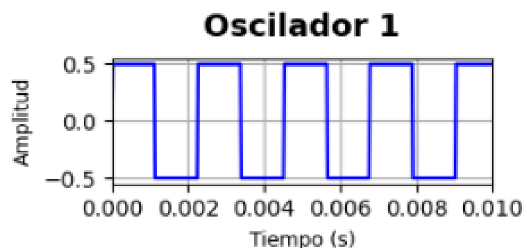
La clase se crea con los siguientes atributos principales:

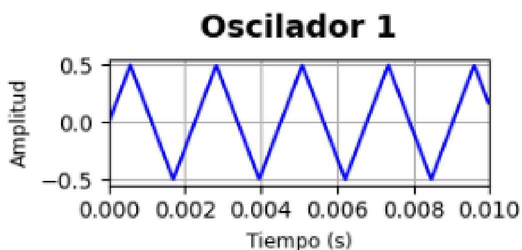
```
class Oscilador:
    def __init__(self, frecuencia=440, amplitud=1.0, forma='sinusoidal',
                 fase=0.0, sample_rate=44100, unison=1,
detune_cents=0.0):
        self.frecuencia = frecuencia       # Frecuencia base en Hz
        self.amplitud = amplitud           # Volumen (0.0 a 1.0)
        self.forma = forma                 # Tipo de onda
        self.fase = fase                   # Fase inicial
        self.sample_rate = sample_rate     # Frecuencia de muestreo (44.1
kHz)
        self.unison = unison               # Número de voces (1-8)
        self.detune_cents = detune_cents   # Desafinación (0-50 cents)
```

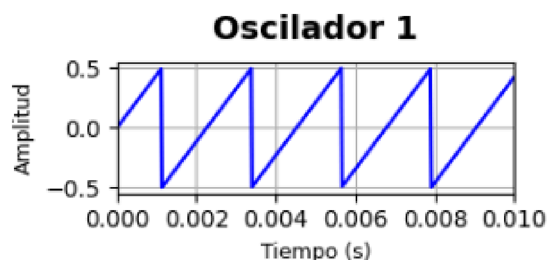
Senoidal

Cuadrada









Triangular

Sierra

Esta figura representa el oscilador 1 con las 4 formas de onda que hemos implementado en este proyecto.

3.2.2 Sistemas de cents y desafinación

En este apartado se describe cómo se producen las notas de la escala musical y cómo se afinan o desafinan

3.2.2.1 Fundamento del Sistema de cents

El sistema de cents es una unidad logarítmica de intervalo musical que divide un semitono en 100 partes iguales [8]. Esta escala se basa en la percepción logarítmica del oído humano, donde percibimos cambios de frecuencia de forma proporcional, no absoluta.

En la escala temperada moderna (equal temperament), una octava contiene 12 semitonos.

La relación fundamental es:

1 semitono = 100 cents

1 octava = 1200 cents

La fórmula para convertir un intervalo en cents a una relación de frecuencias es:

f ´ = f * 2^(c/1200)

donde:

f es la frecuencia original.

c es la desafinación (en cents).

2^(c/1200) es el factor multiplicador de frecuencia.

En la aplicación, la desafinación se limita a 0-50 cents, lo que permite variaciones suaves de ±25 cents alrededor de cada voz (aproximadamente ±1/4 de semitono), creando un efecto notable sin desafinar demasiado.

3.2.3 Unison: Distribución de voces

Aquí hablaremos de cómo generamos ondas sonoras con diferentes afinaciones

3.2.3.1 Concepto de Unison

El unison es una técnica de síntesis donde se generan múltiples instancias de la misma onda, cada una ligeramente desafinada respecto a las otras. Al sumarlas, crean un efecto de "grosor" o "anchura" [2]. Este efecto realmente se nota con muchas voces y un detune alto.

3.2.3.2 Distribución de voces desafinadas

La distribución de voces en la aplicación se realiza al generar la onda de la siguiente manera:

```
def generar(self, duracion):
    n_muestras = int(self.sample_rate * duracion)
    t = np.linspace(0, duracion, n_muestras, endpoint=False)

    if self.unison == 1:
        freq_detuned = self.frecuencia * (2 ** (self.detune_cents /
1200.0))
        return self._generar_forma(t, freq_detuned, self.fase)
    else:
        onda_total = np.zeros(n_muestras)
        detune_range = self.detune_cents
        for i in range(self.unison):
            # Distribuir uniformemente en rango [-detune_range,
+detune_range]
            detune_offset = (i / (self.unison - 1) - 0.5) * 2 *
detune_range
            freq_voice = self.frecuencia * (2 ** (detune_offset /
1200.0))
            onda_total += self._generar_forma(t, freq_voice, self.fase)
        return onda_total / self.unison
```

Para n voces (donde n=unison), el índice i va de 0 a n−1.

La desafinación para cada voz se calcula como:

$\Delta ci = (i/(n-1) - 0,5)*2*detune$

Ejemplo con unison=3 y detune=20 cents:

- Voz 0 (i=0): $\Delta c0=(0/2-0.5)\times2\times20= -20$ cents
- Voz 1 (i=1): $\Delta c1=(1/2-0.5)\times2\times20= 0$ cents (centro)
- Voz 2 (i=2): $\Delta c2=(2/2-0.5)\times2\times20= +20$ cents

Las frecuencias son:

- $f0 = fbase\times2-20/1200 \approx fbase\times0.9887$
- $f1 = fbase\times20/1200 = fbase$
- $f2 = fbase\times2+20/1200 \approx fbase\times1.0114$

La voz central siempre mantiene la frecuencia exacta de la nota, y las otras se distribuyen simétricamente a ambos lados. La salida final se normaliza dividiendo entre el número de voces: ondatotal/n. Esto previene que el volumen se incremente proporcionalmente con las voces y supere los límites permitidos por SoundDevice.

3.2.4 Creación de los arrays que representan una onda

El método generar (duracion) de la clase Oscilador se encarga de convertir los parámetros del oscilador (frecuencia, amplitud, forma) en un array concreto de muestras de audio que puede reproducirse o visualizarse.

El audio digital requiere convertir el tiempo continuo en muestras discretas. El primer paso del método es calcular cuántas muestras son necesarias para la duración deseada:

```
n_muestras = int(self.sample_rate * duracion)
```

Por ejemplo, para generar 0.5 segundos de audio a 44100 Hz:

$$n_muestras = 44100 \times 0.5 = 22050 \text{ muestras}$$

A continuación, se crea un vector t que representa los instantes temporales de cada muestra

```
t = np.linspace(0, duracion, n_muestras, endpoint=False)
```

Para el ejemplo anterior (0.5 s, 44100 Hz), t contendrá 22050 valores desde 0.0 hasta aproximadamente 0.4999 segundos.

Con el vector temporal t creado, el método delega la generación real de la onda al método generar_forma(t, freq, fase), que aplica la ecuación matemática correspondiente a cada forma de onda

El resultado es un array NumPy de n_muestras elementos que representa la señal de audio completa, listo para que se visualize en las gráficas o se reproduzca su sonido.

De esta manera el oscilador puede generar buffers de diferentes duraciones según el contexto: buffers cortos de 0.01 segundos para la visualización, o buffers más largos de 0.5 segundos para la reproducción de audio.

3.2.5 Implementación de formas de onda

3.2.5.1 Forma Senoidal

La forma senoidal es la más simple y fundamental en síntesis:

$y(t)=A*\sin(2\pi ft+\phi)$

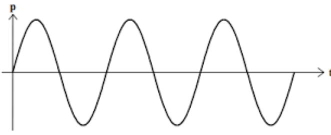
donde:

- A = amplitud
- f = frecuencia (Hz)
- t = tiempo (s)
- ϕ = fase inicial (radianes)

Implementación:

```
if self.forma == 'sinusoidal':
    return self.amplitud * np.sin(2 * np.pi * freq * t + fase)
```

El sonido que se obtiene es limpio y suave, similar a un silbido



3.2.5.2 Forma Cuadrada

La onda cuadrada se implementa usando la función sign:

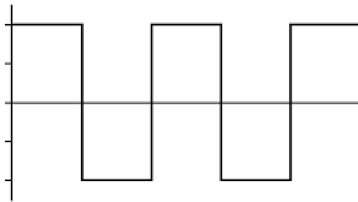
$y(t)=A \cdot sign(sign(2\pi ft+\phi))$

donde $sign(x)=\{+1 si\ x>0\ \text{ó}\ -1\ si\ x<0\}$

Implementación:

```
elif self.forma == 'cuadrada':
    return self.amplitud * np.sign(np.sin(2 * np.pi * freq * t + fase))
```

El sonido es "hueco", similar a un clarinete. Para obtener una onda cuadrada discreta a partir de una sinusoide, la función signo actúa como un comparador: cuando el seno es positivo, la salida es +1; si es negativo, es -1.



3.2.5.3 Forma de onda Triangular

La onda triangular se calcula mediante:

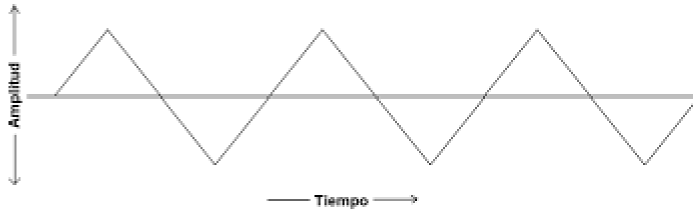
$y(t)=A \cdot 2\pi \cdot \arcsin(\sin(2\pi ft+\phi))$

La función $\arcsin(\sin(x))$ genera una onda triangular normalizada a rango $[-\pi/2,\pi/2]$. El factor 2π escala la amplitud a rango $[-1,1]$.

Implementación:

```
elif self.forma == 'triangular':
    return self.amplitud * (2 / np.pi) * np.arcsin(np.sin(2 * np.pi *
freq * t + fase))
```

El sonido producido es intermedio al que producen las ondas senoidal y cuadrada. Más suave que la cuadrada, pero con más "cuerpo" que la senoidal. Menos áspera que la onda sierra.



3.2.5.4 Forma de Sierra

La onda sierra se genera mediante la siguiente expresión:

$y(t)=A\cdot 2(ft+\phi 2\pi-\lfloor ft+\phi 2\pi+0.5\rfloor)$

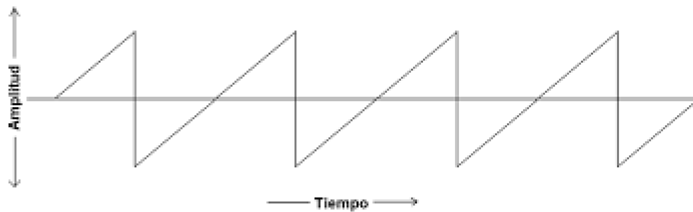
donde $\lfloor\cdot\rfloor$ es la función floor (parte entera).

Implementación:

```
elif self.forma == 'sierra':
    return self.amplitud * 2 * (freq * t + fase / (2 * np.pi) -
            np.floor(freq * t + fase / (2 * np.pi) + 0.5))
```

La expresión freq*t + fase/(2*π) genera una rampa que incrementa linealmente con el tiempo. La función floor con él + 0.5 crea la rampa, creando una forma de diente de sierra: sube linealmente y luego cae en vertical y así repetidamente

El resultado sonoro es intenso y brillante, muy rico en armónicos.



3.3 Interacciones entre Osciladores

Las interacciones entre osciladores son técnicas de síntesis para combinar las señales de dos osciladores y crear así timbres complejos que no pueden obtenerse con un solo oscilador [2]. La aplicación implementa cuatro modos de interacción: suma simple, modulación en frecuencia (FM), modulación en anillo (ring modulation) y sincronización de osciladores (sync).

3.3.1 Suma simple (Modo "Ninguna")

La forma más básica de combinar dos osciladores es mediante suma simple de sus señales [1], [2]:

$y(t)=y1(t)+y2(t)y(t)$

Esta operación es lineal y se mantienen las propiedades de ambas señales.

Implementación:

```
def ninguna(osc1, osc2, duracion):
    """Sin interacción: solo suma las ondas."""
    return osc1.generar(duracion) + osc2.generar(duracion)
```

La implementación es sencilla gracias a que NumPy soporta operaciones fáciles sobre arrays, ambos arrays de muestras se suman elemento a elemento [7].

3.3.2 Modulación en Frecuencia (FM)

A continuación describiremos la modulación en frecuencia o frecuencia modulada, que es una de las interacciones entre ondas más comúnmente utilizadas en los sintetizadores.

3.3.2.1 Teoría de la FM

La modulación en frecuencia es una técnica desarrollada por John Chowning en los años 60 que revolucionó la síntesis digital [2]. En FM, un oscilador (modulador) controla la frecuencia instantánea de otro (portadora).

La señal FM se define matemáticamente como:

$FM(t)=Acsin(\phi c(t))$

donde la fase instantánea de la portadora es:

$\phi c(t)=2\pi fct+I\cdot m(t)$

siendo:

- fc = frecuencia de la portadora (Hz)
- m(t) = señal moduladora
- I = índice de modulación
- Ac = amplitud de la portadora

3.3.2.2 Índice de Modulación

El índice de modulación I es el parámetro que controla la riqueza espectral del sonido resultante.

FM(t)=Asin(2π*fc*t+I*m(t))

Por ejemplo, para un modulador sinusoidal m(t)=A*sin(2π*fm*t)

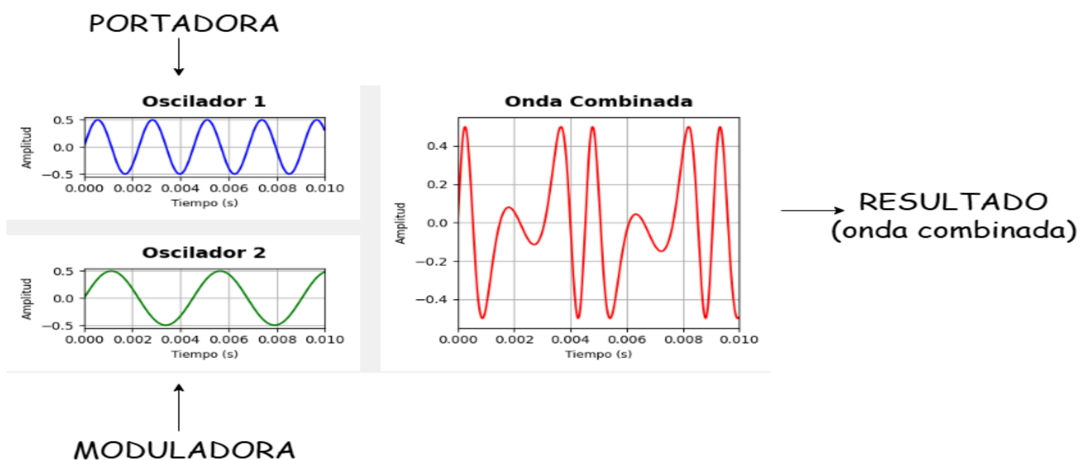
3.3.2.3 Implementación

```
def fm(osc1, osc2, duracion, indice=5.0):
    n_muestras = int(osc1.sample_rate * duracion)
    t = np.linspace(0, duracion, n_muestras, endpoint=False)
    moduladora = osc2.generar(duracion)
    fase_instantanea = 2 * np.pi * osc1.frecuencia * t + indice *
moduladora

    if osc1.forma == 'sinusoidal':
        onda_fm = osc1.amplitud * np.sin(fase_instantanea + osc1.fase)
    # ... otros casos para formas de onda
    return onda_fm
```

Análisis del código:

1. moduladora = osc2.generar(duracion): Genera la señal moduladora completa. Importante: esta señal ya está normalizada a rango aproximado [−Am,+Am].
2. fase_instantanea = 2πf_c·t + I·moduladora: Calcula la fase instantánea. Aquí hay un detalle: moduladora ya es una señal de audio (valores entre -1 y 1 aproximadamente), NO una fase. Por tanto, actúa directamente como variación de fase.

La síntesis FM clásica, tal como fue desarrollada por Chowning y popularizada por el Yamaha DX7, utiliza exclusivamente ondas sinusoidales tanto para la portadora como para la moduladora. Sin embargo, con fines educativos, nuestra implementación permite utilizar con FM utilizando otras formas de onda como portadora, para que el usuario pueda experimentar.



3.3.3 Modulación en Anillo:

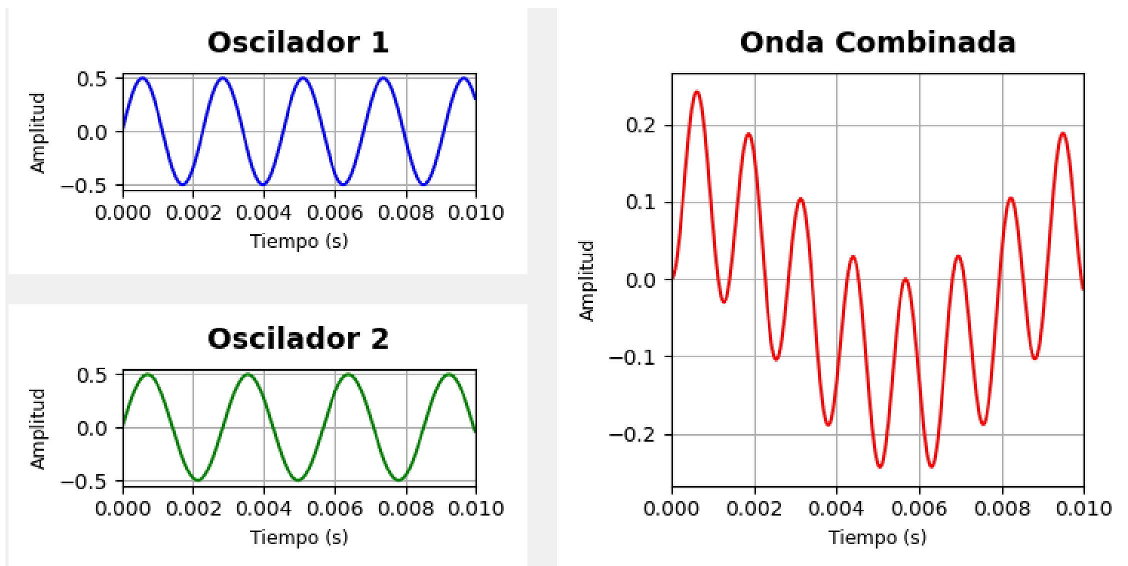
La modulación en anillo es una operación que multiplica ambas señales [2]:

yring(t)=y1(t)·y2(t)

Implementación:

```
def ring_mod(osc1, osc2, duracion):
    """Modulación en anillo: multiplica las dos ondas."""
    return osc1.generar(duracion) * osc2.generar(duracion)
```

Propiedades sonoras**:** Sonidos metálicos, campaniles, robóticos. Muy útil para efectos de ciencia ficción y timbres no tradicionales.



3.3.4 Sincronización

La sincronización de osciladores es una técnica donde un oscilador (*slave*) se "reinicia" periódicamente por cada ciclo del otro oscilador (*master*) [2].

En nuestra implementación, el reinicio del ciclo del oscilador esclavo se aproxima mediante la detección de cruces por cero ascendentes del oscilador maestro, lo cual coincide con el inicio de un nuevo ciclo en las ondas con las que trabajamos.

Implementación:

```
def sync(osc1, osc2, duracion):
    """Sincronización: osc2 reinicia la fase de osc1."""
    n_muestras = int(osc1.sample_rate * duracion)
    t = np.linspace(0, duracion, n_muestras, endpoint=False)
    onda2 = osc2.generar(duracion)
    fase_acumulada = np.zeros(n_muestras)
    fase_actual = osc1.fase

    for i in range(1, n_muestras):
        # Detectar cruce por cero ascendente
```

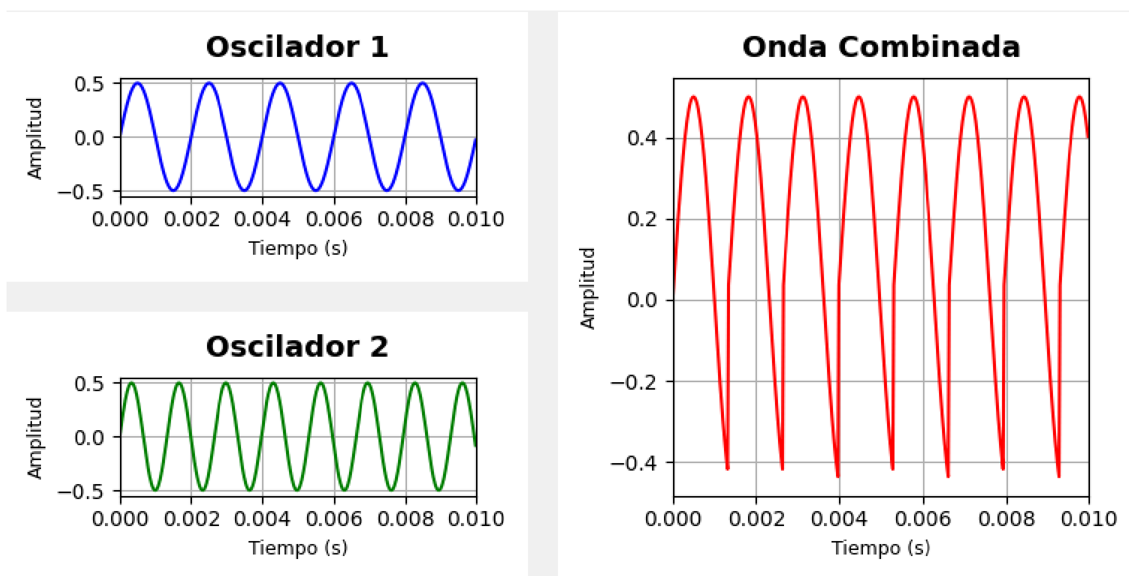
```
        if onda2[i-1] < 0 and onda2[i] >= 0:
            fase_actual = 0  # Reiniciar fase
        fase_actual += 2 * np.pi * osc1.frecuencia / osc1.sample_rate
        fase_acumulada[i] = fase_actual

    # Generar onda usando fase acumulada
    if osc1.forma == 'sinusoidal':
        return osc1.amplitud * np.sin(fase_acumulada)
    # ... otros casos
```

Primero se detecta cuando empieza un ciclo nuevo, onda2[i-1] < 0 and onda2[i] >= 0 detecta los cruces por cero ascendentes que equivalen a un ciclo.

Luego fase_actual += 2π·f/sample_rate incrementa la fase en cada muestra. El incremento por muestra es $\Delta\phi=(2\pi f)/fs$ donde fs es el sample rate.

Y cuando se detecta un cruce, fase_actual = 0 fuerza al oscilador esclavo a reiniciar su ciclo

Los sonidos que produce este tipo de interacción son muy brillantes, "cortantes", con mucha energía en agudos.



Esta figura muestra cómo funciona el sync en la aplicación, la onda combinada es la misma onda que oscilador 1 pero con el detalle que cuando oscilador 2 completa un ciclo, onda combinada se reinicia “volviendo a empezar”.

3.4 Sistema de notas musicales

La aplicación cuenta con un pequeño teclado que permite al usuario tocar notas, lo que requiere un sistema de conversión entre nombres de notas y frecuencias.

3.4.1 Temperamento y cálculo de frecuencias a partir de una nota

El sistema musical occidental moderno utiliza el temperamento igual de 12 semitonos, donde la octava se divide en 12 intervalos iguales en escala logarítmica [8]. Este sistema se basa en dos principios fundamentales:

Una octava corresponde exactamente a una duplicación de frecuencia:

$$foctava=2\cdot fbase$$

La frecuencia de cualquier nota se calcula a partir de una frecuencia de referencia (A4 = LA = 440 Hz) mediante:

$f(n)=440\cdot 2^{(n/12)}$

donde n es el número de semitonos de distancia desde A4:

- $n>0$: notas más agudas que A4
- $n<0$: notas más graves que A4
- $n=0$: A4 (440 Hz)

Hemos usado el sistema de notación alemán/anglosajón porque es más sencillo de codificar, ya que cada nota se representa con una sola letra, más un número que representa la octava. En este sistema la escala A, B, C, D, E, F, G, equivale a La, Si, Do, Re, Mi, Fa, Sol en el sistema latino.

Ejemplos de cálculo:

1. A5 (una octava arriba de A4): $n=12$ semitonos

   $f(12)=440\cdot 2^{(12/12)}=440\cdot 2=880$ Hz
2. A3 (una octava abajo de A4): $n=-12$ semitonos

   $f(-12)=440\cdot 2^{(-12/12)}=440\cdot 0.5=220$ Hz

3.4.2 Diccionario de notas

En lugar de calcular frecuencias en tiempo real, el sintetizador utiliza un diccionario que almacena las frecuencias de las 48 notas disponibles (4 octavas, desde C2 hasta B5).

Implementación:

```
NOTAS_FRECUENCIAS = {
    'C2': 65.41, 'C#2': 69.30, 'D2': 73.42, 'D#2': 77.78, 'E2': 82.41,
'F2': 87.31,
    'F#2': 92.50, 'G2': 98.00, 'G#2': 103.83, 'A2': 110.00, 'A#2':
116.54, 'B2': 123.47,
    'C3': 130.81, 'C#3': 138.59, 'D3': 146.83, 'D#3': 155.56, 'E3':
164.81, 'F3': 174.61,
    'F#3': 185.00, 'G3': 196.00, 'G#3': 207.65, 'A3': 220.00, 'A#3':
233.08, 'B3': 246.94,
    'C4': 261.63, 'C#4': 277.18, 'D4': 293.66, 'D#4': 311.13, 'E4':
329.63, 'F4': 349.23,
    'F#4': 369.99, 'G4': 392.00, 'G#4': 415.30, 'A4': 440.00, 'A#4':
466.16, 'B4': 493.88,
    'C5': 523.25, 'C#5': 554.37, 'D5': 587.33, 'D#5': 622.25, 'E5':
659.25, 'F5': 698.46,
    'F#5': 739.99, 'G5': 783.99, 'G#5': 830.61, 'A5': 880.00, 'A#5':
932.33, 'B5': 987.77
}
```

3.4.3 Conversión nota a frecuencia

La función nota_a_frecuencia() convierte una nota a frecuencia utilizando el diccionario.

```
def nota_a_frecuencia(note_name):
    """
    Convierte nombre de nota musical a frecuencia en Hz.

    Usa temperamento igual con A4 = 440 Hz.

    Args:
        note_name: Nombre en formato 'NotaOctava' (ej: 'C4', 'A#3')

    Returns:
        float: Frecuencia en Hz. Retorna 440.0 si la nota no existe.
    """
    if note_name in NOTAS_FRECUENCIAS:
        return NOTAS_FRECUENCIAS[note_name]
    return 440.0
```

Rango de frecuencias del teclado:

Nota más grave: C2 = 65.41 Hz

Nota más aguda: B5 = 987.77 Hz

Este rango cubre aproximadamente 4 octavas, suficiente para poder jugar y experimentar con él.

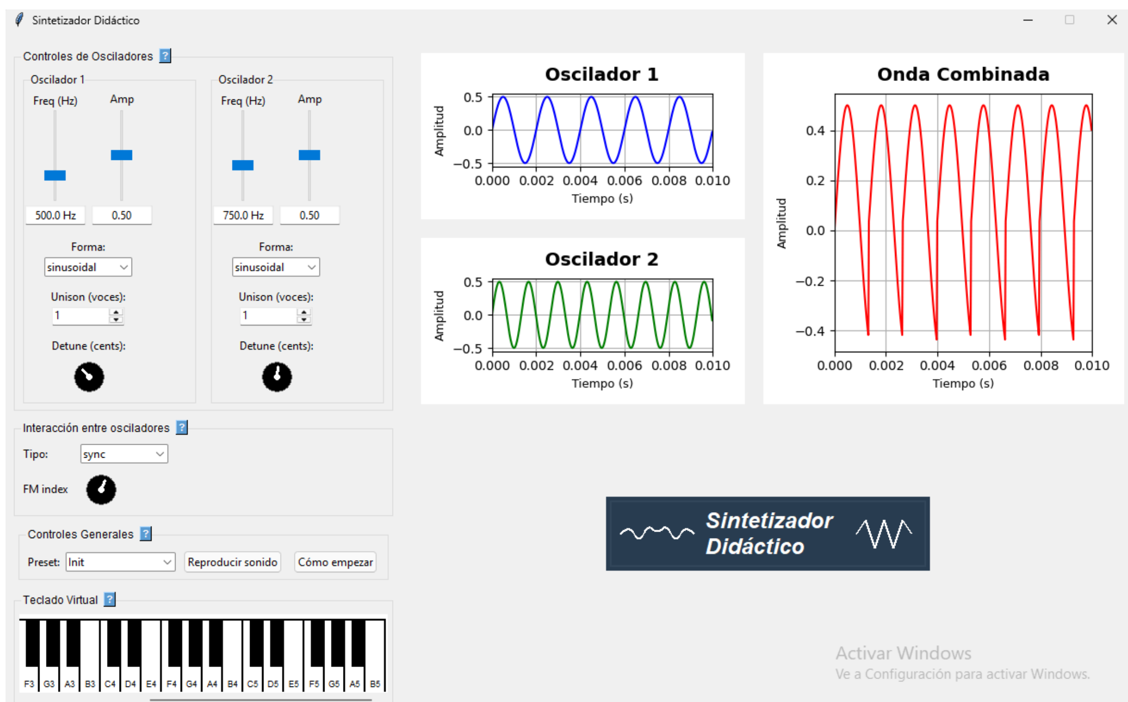
3.4.4 Integración con el sistema de osciladores

Cuando el usuario hace clic en una tecla del teclado virtual ocurre lo siguiente:

1. Detección del clic: El widget del teclado identifica qué tecla fue presionada (ej: 'C4')
2. Conversión de frecuencia: Se llama a nota_a_frecuencia('C4') → 261.63 Hz
3. Configuración del oscilador: La frecuencia se asigna a solo el osc1 mediante osc.set_frecuencia(261.63)
4. Generación de audio: Los osciladores generan las formas de onda
5. Aplicación de interacción: Se procesa la interacción seleccionada (FM, ring mod, sync, etc.)
6. Reproducción: La señal resultante se envía a sounddevice para reproducción

3.5 Interfaz de Usuario

La interfaz gráfica del sintetizador tiene un diseño basado en Tkinter. Esta interfaz es bastante didáctica pero cuenta también con precisión técnica. La interfaz cuenta con widgets personalizados, visualizadores en tiempo real y controles interactivos que muestran directamente los parámetros que se modifican en tiempo real.



3.5.1 Estructura de la interfaz Tkinter

La aplicación utiliza Tkinter como GUI (Graphical User Interface)[4], organizando la interfaz en una geometría fija de 1220×750 píxeles dividida en dos regiones funcionales:

- Región de Control (izquierda, 440px): Contiene controles de osciladores, interacciones, presets y teclado virtual.

- Región de Visualización (derecha, 780px): Tiene tres gráficos Matplotlib implementados mediante FigureCanvasTkAgg y posicionados con geometría absoluta (place()).

La clase principal SintetizadorApp hereda de tk.Tk y gestiona la aplicación por completo.

```python
class SintetizadorApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Sintetizador Didáctico")
        self.geometry("1220x750")
        self.resizable(False, False)
```

La geometría fija evita problemas de redimensionamiento que afectarían a la nitidez y legibilidad de las formas de onda. El método update_idletasks() fuerza el cálculo de la geometría antes de la primera renderización, evitando problemas visuales durante la inicialización:

3.5.2 Inicialización del sistema

El constructor inicializa tres categorías de estado:

1. Parámetros de Temporización:

   ```
   self.duracion_visual = 0.01  # 10 ms para visualización
   self.duracion_audio = 0.5    # 500 ms para reproducción
   self.sample_rate = 44100     # Frecuencia de muestreo estándar
   ```

2. Instancias de Osciladores:

   ```
   self.osc1 = Oscilador(sample_rate=self.sample_rate)
   self.osc2 = Oscilador(frecuencia=440, forma='cuadrada',
                         amplitud=0.5, sample_rate=self.sample_rate)
   ```

   Ambos osciladores se instancian con estas configuraciones iniciales, oscilador 1 con onda sinusoidal y oscilador 2 con onda cuadrada

3. Variables de Control Tkinter:

   ```
   self.freq1 = tk.DoubleVar(value=440.0)
   self.amp1 = tk.DoubleVar(value=0.5)
   self.forma1 = tk.StringVar(value="sinusoidal")
   self.unison1 = tk.IntVar(value=1)
   self.detune1 = tk.DoubleVar(value=10.0)
   # ... (igual para osc2, interacción, preset)
   ```

   Estas variables funcionan de la siguiente manera: los widgets se vinculan a ellas mediante sus parámetros textvariable o variable, estableciendo sincronización automática. Cuando un control cambia una variable, todos los widgets vinculados se actualizan sin código explícito de sincronización.

3.5.3 Llamada a construcción de componentes

La interfaz se construye mediante llamadas a funciones especializadas que crean los distintos componentes:

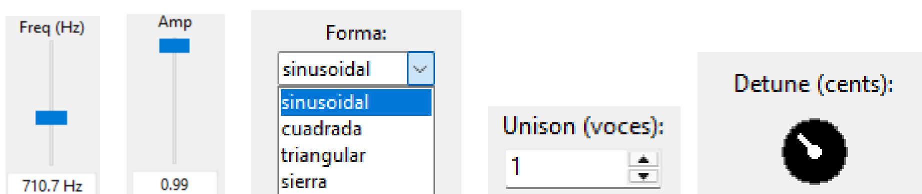
```
crear_controles_osciladores(self.left_frame, self.freq1, self.amp1,
                             self.forma1, self.unison1, self.detune1,
                             self.freq2, self.amp2, self.forma2,
                             self.unison2, self.detune2,
                             lambda:
self.after_idle(self.actualizar_onda))

crear_controles_interaccion(self.left_frame, self.interaccion,
                             self.fm_index,
                             lambda:
self.after_idle(self.actualizar_onda))

self.crear_botones(self.left_frame)

self.keyboard_canvas, self.keyboard_hbar = crear_teclado(
    self.left_frame, self.key_map, self.note_to_rect, self.tocar_nota)
```

3.5.3.1 Controles de osciladores

```
def crear_controles_osciladores(parent, freq1, amp1, forma1, unison1,
detune1, freq2, amp2, forma2, unison2, detune2, update_callback)
```

Esta función, definida en ui/controles.py, recibe las 10 variables Tkinter que controlan los parámetros de ambos osciladores (frecuencia, amplitud, forma, unison y detune para cada uno) más un callback de actualización. Construye dos paneles verticales con:

- Sliders verticales (ttk.Scale) para frecuencia (20-2000 Hz invertido) y amplitud (0.0-1.0)
- Entry widgets personalizados que permiten entrada numérica precisa
- Combobox para selección de forma de onda (sinusoidal, cuadrada, triangular, sierra)
- Spinbox para número de voces unison (1-8)
- Knob para detune (0-50 cents)

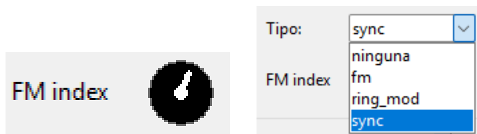


3.5.3.2 Controles de interacción

```
def crear_controles_interaccion(parent, interaccion, fm_index, update_callback):
```

También en ui/controles.py, esta función construye el panel de selección de modo de síntesis:

- Combobox para los 4 modos: ninguna, FM, ring modulation y sync
- Knob para el índice FM (0-20), el cuál solo tiene efecto cuando la interacción seleccionada es FM.



3.5.3.3 Botones de presets, reproducir sonido y ayuda

El método crear_botones(self, parent) es el método que construye el panel de controles generales:

Este método encapsula:

- Selector de presets: combobox que carga configuraciones predefinidas desde el diccionario PRESETS importado de ui/presets.py.
- Botón de reproducción: ejecuta reproducir_onda() que genera un array de audio de 0.5 segundos con los parámetros actuales y lo reproduce mediante SoundDevice.
- Botones de ayuda: se llama del módulo ui/ayuda.py que abren ventanas TopLevel con texto para ayuda y explicaciones.

Más adelante se muestran imágenes de estos botones.

3.5.4 Sincronización inicial

Antes de la primera visualización, la aplicación fuerza el cálculo de geometría y programa la actualización inicial:

```
self.update_idletasks()
self.after(60, self.actualizar_onda)
```

update_idletasks() procesa todos los eventos de diseño pendientes (cálculo de tamaños, posicionamiento de widgets) sin entrar en el bucle principal. Esto asegura que cuando actualizar_onda() consulte dimensiones de canvas para procesar gráficos, estas ya estén calculadas correctamente.

El retraso de 60ms mediante after() permite que se complete el procesado inicial antes de la primera actualización de las gráficas Sin este retraso, el primer redibujado podría ocurrir sobre un canvas aún no completamente inicializado, causando un comportamiento inesperado.

3.5.5 Lambda: After-Idle

Todos los callbacks de actualización utilizan after_idle() en lugar de invocación directa:

```
lambda: self.after_idle(self.actualizar_onda)
```

Si múltiples controles cambian en la misma iteración del bucle de eventos (por ejemplo, al cargar un preset que modifica 10+ parámetros simultáneamente), after_idle() encola solo una llamada a actualizar_onda() para el siguiente ciclo, en lugar de ejecutar 10+ llamadas a actualizar_onda().

Lo que ocurre es lo siguiente: El usuario cambia N controles y se encolan N callbacks con after_idle(), luego Tkinter detecta múltiples llamadas iguales pendientes y las junta todas en una sola, se finaliza el ciclo actual de eventos y luego se ejecuta una única llamada a actualizar_onda() en vez de n llamadas a actualizar_onda(). Así, por ejemplo, se reduce mucho el tiempo de carga de los presets.

3.5.6 Método actualizar_onda()

El método actualizar_onda() (interfaz.py) implementa por completo la sincronización del modelo-vista

1. Fase 1: Sincronización de Parámetros:

   ```
   self.osc1.set_frecuencia(self.freq1.get())
   self.osc1.set_amplitud(self.amp1.get())
   self.osc1.set_forma(self.forma1.get())
   self.osc1.set_unison(self.unison1.get())
   self.osc1.set_detune(self.detune1.get())
   # Análogo para osc2
   ```

   Lee los valores actuales de las variables Tkinter y actualiza los objetos Oscilador

2. Fase 2. Generación de ondas

   ```
   onda1 = self.osc1.generar(self.duracion_visual)
   onda2 = self.osc2.generar(self.duracion_visual)

   if self.interaccion.get() == "ninguna":
       onda_comb = ninguna(self.osc1, self.osc2, self.duracion_visual)
   elif self.interaccion.get() == "fm":
       onda_comb = fm(self.osc1, self.osc2, self.duracion_visual,
   indice=self.fm_index.get())
   elif self.interaccion.get() == "ring_mod":
       onda_comb = ring_mod(self.osc1, self.osc2,
   self.duracion_visual)
   elif self.interaccion.get() == "sync":
       onda_comb = sync(self.osc1, self.osc2, self.duracion_visual)
   else:
       onda_comb = onda1
   ```

Genera las ondas individuales de cada oscilador y la combinada según la interacción seleccionada.
Inmediatamente después, se genera una versión para reproducción de audio con mayor duración:

```
if self.interaccion.get() == "ninguna":
    self.onda_comb = ninguna(self.osc1, self.osc2,
self.duracion_audio)
elif self.interaccion.get() == "fm":
    self.onda_comb = fm(self.osc1, self.osc2, self.duracion_audio,
indice=self.fm_index.get())
elif self.interaccion.get() == "ring_mod":
    self.onda_comb = ring_mod(self.osc1, self.osc2,
self.duracion_audio)
elif self.interaccion.get() == "sync":
    self.onda_comb = sync(self.osc1, self.osc2,
self.duracion_audio)
else:
    self.onda_comb = self.osc1.generar(self.duracion_audio)
```

Aquí se separa la visualización (10ms, para un procesamiento rápido) de la reproducción (500ms, duración más alta para tener suficiente tiempo para oír correctamente el sonido reproducido). Almacenar self.onda_comb() como atributo permite que el método reproducir_onda() acceda a la señal completa sin regenerarla, evitando retrasos al presionar el botón de reproducción.

3. Fase 3: Actualización de Visualizadores
   Para cada gráfico se ejecuta la misma secuencia de configuración y renderizado.

```
tiempo = np.arange(len(onda1)) / self.sample_rate

self.ax1.clear()
self.ax1.set_title("Oscilador 1", fontsize=14, fontweight='bold',
pad=10)
self.ax1.set_xlabel("Tiempo (s)", fontsize=9)
self.ax1.set_ylabel("Amplitud", fontsize=9)
self.ax1.plot(tiempo, onda1, color='blue')
self.ax1.set_xlim(0, self.duracion_visual)
self.ax1.grid(True)
self.fig1.subplots_adjust(bottom=0.18, top=0.85)
self.fig1.tight_layout()
self.canvas1.draw()
```

   El método clear() elimina trazados previos, pero preserva la configuración de ejes. La reconstrucción completa del título, etiquetas y grid en cada actualización asegura consistencia visual incluso si Matplotlib funciona de manera inesperada.

El eje temporal se calcula de la siguiente manera:
np.arange(len(onda1))/ self.sample_rate genera valores exactos en segundos desde 0 hasta la duración de la señal.
set_xlim(0, self.duracion_visual) fija el rango visible independientemente del número de muestras, normalizando la escala entre diferentes frecuencias de muestreo.
tight_layout() ajusta automáticamente márgenes para evitar que se superpongan las etiquetas con los bordes del canvas, mientras que subplots_adjust(bottom=0.18, top=0.85) crea márgenes específicos que garantizan espacio para título y labels.

Los tres gráficos se distinguen visualmente mediante colores: azul (oscilador 1), verde (oscilador 2), rojo (onda combinada).

3.5.7 Método reproducir_onda()

El método reproducir_onda() (interfaz.py) gestiona la salida de audio con normalización previa:

```
if self.onda_comb is not None:
    max_abs = np.max(np.abs(self.onda_comb))
    onda_norm = (self.onda_comb / max_abs) if max_abs > 0 else
self.onda_comb
    print(f"Reproduciendo audio en dispositivo: {sd.default.device}")
    sd.play(onda_norm, self.sample_rate)
    sd.wait()
```

Este método y cómo funciona la reproducción de audio es explicado con detalle en la sección 2.11

3.5.8 Gestión del cierre de la aplicación

El proceso de cierre (interfaz.py) garantiza limpieza de recursos:

```
def on_closing(self):
    """Limpia recursos y cierra la aplicación correctamente."""
    try:
        sd.stop()
    except Exception:
        pass
    self.quit()
```

sd.stop() cancela cualquier reproducción activa antes de cerrar la ventana. El bloque try-except captura excepciones si no hay reproducción en curso (SoundDevice lanza error al detener cuando no hay stream activo), asegurando que el cierre siempre complete correctamente incluso si el estado de audio es inesperado.

self.quit() termina el bucle principal de Tkinter, permitiendo que Python finalice limpiamente y libere todos los recursos.

3.6 Widgets personalizados

La aplicación implementa tres categorías de widgets personalizados que extienden las capacidades estándar de Tkinter: controles de entrada numérica editables, controles rotatorios tipo knob, y sistema de ayuda contextual.

3.6.1 Entry editables

Los widgets ttk.Scale estándar de Tkinter proporcionan ajuste continuo, pero carecen de la precisión decimal necesaria para parámetros de audio exactos, como por ejemplo para el ajuste exacto de la frecuencia. Por eso se decidió que en la frecuencia y la amplitud se pueda introducir manualmente el valor deseado.

Las funciones crear_entry_frecuencia() y crear_entry_amplitud() (controles.py) implementan widgets que combinan slider con entry de text. Ambos siguen la misma estructura

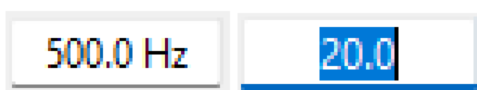
```
def crear_entry_frecuencia(parent_frame, freq_var, label_var,
scale_widget):
    """Crea un Entry editable para introducir frecuencia manualmente."""
    entry = ttk.Entry(parent_frame, textvariable=label_var, width=10,
justify='center')
    # ... (máquina de estados de foco)

def crear_entry_amplitud(parent_frame, amp_var, label_var, scale_widget):
    """Crea un Entry editable para introducir amplitud manualmente."""
    entry = ttk.Entry(parent_frame, textvariable=label_var, width=10,
justify='center')
    # ... (lógica idéntica con rango diferente)
```

Estado 1 - FocusIn: Elimina sufijos decorativos (" Hz" en frecuencia) y selecciona todo el contenido con select_range(0, tk.END).

Estado 2 - FocusOut: Se valida que el valor introducido este dentro del rango permitido de valores

```
def on_focus_out(e):
    try:
        val = float(label_var.get())
        val = max(min_val, min(max_val, val))  # Frecuencia: [20, 2000]Hz
| Amplitud: [0.0, 1.0]
        var.set(val)
        label_var.set(formato)  # "{val:.1f} Hz" | "{val:.2f}"
        scale_widget.set(val)
        update_callback()
    except ValueError:
        label_var.set(valor_anterior)  # Rollback si entrada inválida
```

Estado 3 - Return: Transfiere foco al padre mediante entry.master.focus().



3.6.2 Widget Knob

El widget Knob (widgets.py) simula un knob típico de los sintetizadores mediante tk.Canvas interactivo, haciendo que la interfaz más parecida a la de un sintetizador normal

```
 __init__(self, master, min_val=0.0, max_val=1.0, initial=0.0, size=48,
callback=None, **kwargs):
    # ... configuración
    self.radius = int(size * 0.38)  # 38% del tamaño, margen 12% por lado
    self.min_angle = -135
    self.max_angle = 135  # Arco de 270° (simula potenciómetro físico)
    self.angle = self._val_to_angle(initial)
```

Renderización:

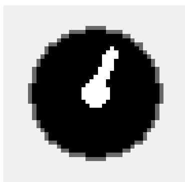
```
def _draw_knob(self):
    ang_rad = math.radians(self.angle)
    x = self.center + int(self.radius * 0.7 * math.cos(ang_rad))
    y = self.center + int(self.radius * 0.7 * math.sin(ang_rad))
    self.create_line(self.center, self.center, x, y, fill='white',
width=3)
```

El indicador se dibuja al 70% del radio (xc+0.7rcosθ, yc+0.7rsinθ), dando un margen que hace que se vea mejor.

Arrastre Vertical:

```
def _drag(self, event):
    dy = self._last_y - event.y
    delta_angle = dy * 0.8  # Factor de sensibilidad calibrado
    self.angle = max(self.min_angle, min(self.max_angle, self.angle +
delta_angle))
    self.value = self._angle_to_val(self.angle)
    if self.callback:
        self.callback(self.value)
    self._draw_knob()
```

El factor 0.8 balancea precisión y eficiencia: permite recorrer 270° con ~340px de desplazamiento vertical (la ventana es de 750px) con margen para control preciso. La actualización en tiempo real proporciona una respuesta visual inmediata.



3.6.3 Sistema de tooltips

La clase Tooltip (widgets.py) implementa con texto de ayuda para el usuario.

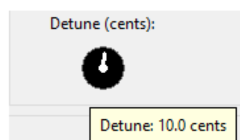
```
class Tooltip:
    def __init__(self, widget, text, delay=500, wraplength=300):
        self.widget = widget
        self.text_func = text if callable(text) else lambda: text
        self.delay = delay
        self.wraplength = wraplength
        self._id = None
        self._tip = None
        self.widget.bind("<Enter>", self._on_enter, add="+")
        self.widget.bind("<Leave>", self._on_leave, add="+")
        self.widget.bind("<Motion>", self._on_motion, add="+")
```

El parámetro text acepta tanto strings como callables, permitiendo tooltips estáticos como “Controla frecuencia” o dinámicos como el tooltip que aparece cuando mueves FM index que modifica su valor cuando tu subes y bajas el knob

```
    def _on_motion(self, _event):
        self._unschedule()
        self._schedule()

    def _schedule(self):
        self._id = self.widget.after(self.delay, self._show)
```

De esta manera se previenen tooltips "flasheantes" durante movimientos rápidos, mostrándose solo cuando el cursor permanece quieto en el mismo lugar durante delay ms.

El tooltip se implementa en TopLevel con fondo amarillo (#ffffe0)



TOOLTIP
(aparece cuando pasas el ratón por encima de un botón)

3.6.4 Help Popups

La aplicación integra ventanas de ayuda ampliada mediante botones "?" en los títulos de los LabelFrame (Títulos de los distintos grupos de controles). Se utilizan dos componentes:

Clase HelpPopup:Ventanas emergentes (tk.Toplevel) con geometría fija de 500×350 píxeles. Muestra texto mediante un widget Text en modo solo lectura (state="disabled") con Scrollbar vertical. Al invocar show(), crea la ventana con botón "Cerrar" para cerrar la ventana y volver a la interfaz principal.

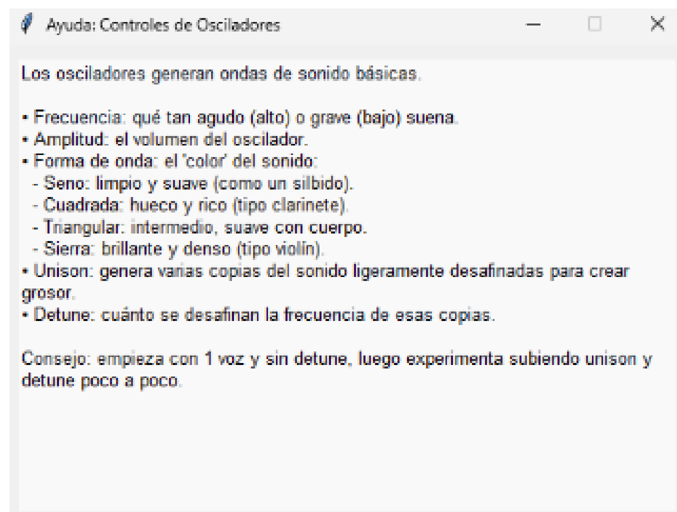
Función crear_labelframe_con_ayuda: Integra el botón "?" directamente en el título del LabelFrame mediante personalización de labelwidget

```
    frame = ttk.LabelFrame(parent, **kwargs)

    # Frame personalizado como título
    title_frame = tk.Frame(frame)
    frame.configure(labelwidget=title_frame)

    # Título + botón "?"
    ttk.Label(title_frame, text=texto_titulo,
              font=("TkDefaultFont", 9)).pack(side="left", padx=(0, 4))

    btn = tk.Label(title_frame, text="?", font=("Arial", 8, "bold"),
                   fg="white", bg="#5a9bd4", width=1, relief="raised",
                   cursor="hand2", padx=2, pady=0, borderwidth=1)
    btn.pack(side="left")

    popup = HelpPopup(parent, f"Ayuda: {texto_titulo}", texto_ayuda)
    btn.bind("<Button-1>", lambda e: popup.show())
    btn.bind("<Enter>", lambda e: btn.config(bg="#4080c0"))
    btn.bind("<Leave>", lambda e: btn.config(bg="#5a9bd4"))

    return frame, btn
```

La propiedad labelwidget permite reemplazar el texto plano del título por un widget completo (nombre del frame + botón), manteniendo la semántica de LabelFrame sin ocupar espacio adicional en el layout.

Esto se aplica en las secciones de osciladores, interacciones, teclado y presets (controles.py, teclado.py, interfaz.py), proporcionando ayuda sin saturar la interfaz.

Al pulsar este botón se despliega algo cómo:



3.7 Visualización en tiempo real

La aplicación integra visualización gráfica en tiempo real de las ondas creadas mediante la biblioteca Matplotlib combinada con Tkinter. Este sistema permite observar simultáneamente las formas de onda de ambos osciladores y su combinación resultante, proporcionando respuesta visual inmediata cuando modificamos parámetros de los osciladores.

3.7.1 Integración Matplotlib-Tkinter

La función crear_visualizadores (visualizadores.py) inicializa tres gráficos independientes utilizando FigureCanvasTkAgg [9]:

```
def crear_visualizadores(parent):
    fig1, ax1 = plt.subplots(figsize=(5, 2))
    ax1.set_title("Oscilador 1", fontsize=14, fontweight='bold', pad=10)
    ax1.set_xlabel("Tiempo (s)", fontsize=9)
    ax1.set_ylabel("Amplitud", fontsize=9)
    ax1.xaxis.set_major_locator(MultipleLocator(0.002))
    fig1.subplots_adjust(bottom=0.18, top=0.85)
    canvas1 = FigureCanvasTkAgg(fig1, master=parent)
    w1 = canvas1.get_tk_widget()
    w1.place(x=450, y=20, width=350, height=180)
    canvas1.draw()

    # [Oscilador 2 y Onda Combinada: estructura idéntica]

    return fig1, ax1, canvas1, fig2, ax2, canvas2, fig3, ax3, canvas3
```

1. Figure y Axes: plt.subplots(figsize=(5, 2)) crea la figura contenedora y los ejes del gráfico con relación de aspecto 5:2 (horizontal).
2. FigureCanvasTkAgg:Envuelve la Figure de Matplotlib en un widget Tkinter compatible. Actúa como conexión entre ambas bibliotecas.
3. Widget Tkinter: get_tk_widget() devuelve un tk.Canvas que puede posicionarse mediante geometría absoluta (place()).
4. Configuración de ejes:
    1. MultipleLocator(0.002): Marcas del eje X cada 2 ms
    2. subplots_adjust: Ajusta márgenes para maximizar área de trazado dentro de espacio limitado.
5. Posicionamiento absoluto: Coordenadas fijas mediante place():
    1. Oscilador 1: (450, 20), 350×180 px
    2. Oscilador 2: (450, 220), 350×180 px
    3. Onda Combinada: (820, 20), 390×380 px (gráfico grande central)¡

3.7.2 Estrategia de actualización visual vs audio

La aplicación mantiene dosarrays de señal con duraciones diferentes (interfaz.py:20-21):

```
self.duracion_visual = 0.01   # 10 ms para gráficos
self.duracion_audio = 0.5     # 500 ms para reproducción
```

Visualización (10 ms): Permite observar la forma de onda con suficiente resolución. Con 44100 Hz, genera 441 muestras, cantidad manejable para redibujar en cada actualización.

Audio (500 ms): Duración mínima para percibir timbre.

3.7.3 Clear-Plot-Draw

El renderizado de gráficos utiliza el patrón clear-plot-draw de Matplotlib [10]:

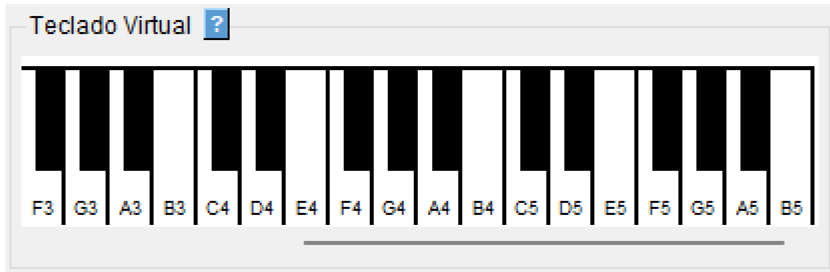
```
self.ax1.clear()
self.ax1.plot(t, onda1, linewidth=1.2)
self.ax1.set_title("Oscilador 1", fontsize=14, fontweight='bold')
self.canvas1.draw()
```

Este patrón regenera completamente el gráfico en cada actualización (clear() elimina trazados previos, plot() dibuja nuevos datos, draw() renderiza).

El comportamiento completo de actualizar_onda() (sincronización UI→osciladores, generación de señales, aplicación de interacciones) se documenta en la sección 2.5.7.

3.8 Teclado virtual

El teclado virtual permite la reproducción de notas, con una representación gráfica de un teclado de piano con 4 octavas (C2-B5, 48 teclas). La implementación utiliza el widget Canvas de Tkinter para el procesamiento de la geometría y detección de clics.



3.8.1 Construcción del teclado

La función crear_teclado (teclado.py) genera dos conjuntos de teclas con dimensiones diferentes:

```
white_width = 22
white_height = 80
black_width = 14
black_height = 50
```

Teclas blancas (notas naturales: C, D, E, F, G, A, B):

```
for octave in range(start_octave, start_octave + octaves):
    for i, note_name in enumerate(white_notes):
        full_note = f"{note_name}{octave}"
        x1 = x_offset
        y1 = y_top
        x2 = x1 + white_width
        y2 = y1 + white_height

        rect_id = canvas.create_rectangle(x1, y1, x2, y2,
                                           fill='white', outline='black',
width=2)
        note_to_rect[full_note] = rect_id
        key_map[rect_id] = full_note

        x_offset += white_width
```

Teclas negras (sostenidos): Se posicionan con desplazamientos relativos a las teclas blancas:

```
black_positions = [0.7, 1.7, 3.7, 4.7, 5.7]  # C#, D#, F#, G#, A#
for bp in black_positions:
    x1 = x_offset + int(bp * white_width) - black_width // 2
    # ...
    rect_id = canvas.create_rectangle(x1, y1, x2, y2,
                                      fill='black', outline='black')
```

El [0.7, 1.7, 3.7, 4.7, 5.7] omite posiciones tras E y B (no existen E#/B#).

3.8.2 Sistema de mapeo geometría - nota

La aplicación utiliza un diccionario para mapeo de nota↔geometría:

- key_map: {rect_id → note_name} - Permite resolver qué nota corresponde a un ID de rectángulo clicado.

```
invisible = canvas.create_rectangle(x1-2, y1-5, x2+2, y2+5,
                                                     fill='', outline='',
activefill='')
key_map[invisible_id] = full_note
```

Esta expansión de ±2-5 píxeles compensa la imprecisión del cursor, ya que antes de este cambio si no dabas clic cerca del centro de la nota a veces no se reconocía el clic.

Textos de etiqueta: Los ID de texto también se registran en key_map para evitar que clics sobre etiquetas fallen:

```
text_id = canvas.create_text(text_x, text_y, text=full_note,
                             font=("Arial", 7), fill="black")
key_map[text_id] = full_note
```

3.8.3 Detección de clicks

Se utiliza find_overlapping para detectar elementos bajo el cursor [11]:

```
def on_click(event):
    x_canvas = canvas.canvasx(event.x)
    y_canvas = canvas.canvasy(event.y)
       item   =   canvas.find_overlapping(x_canvas,   y_canvas,   x_canvas,
y_canvas)
    if item:
        rect_id = item[-1]
        if rect_id in key_map:
            note = key_map[rect_id]
            tocar_callback(note)
```

canvasx()/canvasy() convierten coordenadas del evento (relativas al widget visible) en coordenadas del canvas (considerando scroll)

find_overlapping retorna lista ordenada por profundidad (elementos más recientes al final). Al tomar item[-1], se garantiza selección de teclas negras sobre blancas en regiones de superposición, ya que las negras se dibujan después.

item[-1] siempre selecciona el objeto que está más al frente visualmente, que es el que el usuario realmente ve y quiere pulsar.

Al hacer clic, tocar_callback(note) invoca tocar_nota (interfaz.py:123), que convierte el nombre de nota a frecuencia mediante nota_a_frecuencia (ver sección 2.4) y actualiza solo el oscilador 1.

3.8.4 Scroll Horizontal

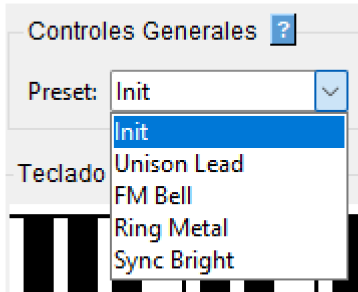
Como hacer un teclado con todas las notas ocuparía mucho espacio, se ha implementado el teclado con un scroll horizontal para poder recorrer todas las notas (C2-B5) y que el teclado siga siendo pequeño y no ocupe toda la pantalla.

```
canvas = tk.Canvas(container, bg='white', highlightthickness=0)
hbar = ttk.Scrollbar(container, orient='horizontal',
command=canvas.xview)
canvas.config(xscrollcommand=hbar.set)
canvas.config(scrollregion=canvas.bbox("all"))
```

scrollregion=canvas.bbox("all") establece área desplazable automáticamente al bounding box de todos los elementos dibujados.

3.9 Presets

El sistema de presets proporciona configuraciones predefinidas de los distintos parámetros del sintetizador que te permiten explorar de manera más fácil, aparte de producir sonidos característicos.



3.9.1 Diccionario de presets

Los presets se almacenan en un diccionario Python definido en presets.py:

```
PRESETS = {
    "Init": {
        "f1": 440.0, "a1": 0.5, "w1": "sinusoidal", "u1": 1, "d1": 0.0,
        "f2": 440.0, "a2": 0.5, "w2": "cuadrada",   "u2": 1, "d2": 0.0,
        "ix": "ninguna", "fm": 3.0
    },
    "FM Bell": {
        "f1": 440.0, "a1": 0.8, "w1": "sinusoidal", "u1": 1, "d1": 0.0,
        "f2": 440.0, "a2": 0.7, "w2": "sinusoidal", "u2": 1, "d2": 0.0,
        "ix": "fm",    "fm": 9.0
    },
    # ... más presets
}
```

Se nombran los parámetros del oscilador de la siguiente manera:

- Oscilador 1: f1 (frecuencia), a1 (amplitud), w1 (waveform), u1 (unison), d1 (detune)
- Oscilador 2: f2, a2, w2, u2, d2
- Interacción: ix (tipo de interacción), fm (FM index)

Presets incluidos:

1. Init: Configuración de referencia con osciladores básicos (sinusoidal + cuadrada, amplitud 0.5, sin unison). Sirve como punto de partida neutro.
2. Unison Lead: Sonido ancho para melodías principales mediante unison (OSC1: 5 voces con 12 cents de detune, OSC2: 3 voces con 8 cents). Forma de onda sierra en OSC1 para brillo armónico.
3. FM Bell: Se intenta simular una campana sintética usando modulación FM con índice 9.0. Ambos osciladores con forma seno.

4. Ring Metal: Efecto metálico mediante ring modulation con OSC1 (cuadrada, 440 Hz) y OSC2 (sierra, 660 Hz).
5. Sync Bright: Sincronización con OSC2 al doble de frecuencia (880 Hz vs 440 Hz). OSC1 sierra proporciona contenido armónico rico que se reinicia periódicamente, generando formas brillantes.

3.9.2 Aplicación de presets

La aplicación de un preset implica la actualización de 11 variables Tkinter en interfaz.py:

```
def aplicar_preset(self, nombre):
    p = PRESETS.get(nombre)
    if not p:
        return
    self.freq1.set(p["f1"]); self.amp1.set(p["a1"]);
    self.forma1.set(p["w1"])
    self.unison1.set(p["u1"]); self.detune1.set(p["d1"])
    self.freq2.set(p["f2"]); self.amp2.set(p["a2"]);
    self.forma2.set(p["w2"])
    self.unison2.set(p["u2"]); self.detune2.set(p["d2"])
    self.interaccion.set(p["ix"]); self.fm_index.set(p["fm"])
    self.after_idle(self.actualizar_onda)
```

3.9.3 Integración con interfaz gráfica

El selector de presets se implementa mediante un ttk.Combobox vinculado a self.preset (StringVar) en interfaz.py:

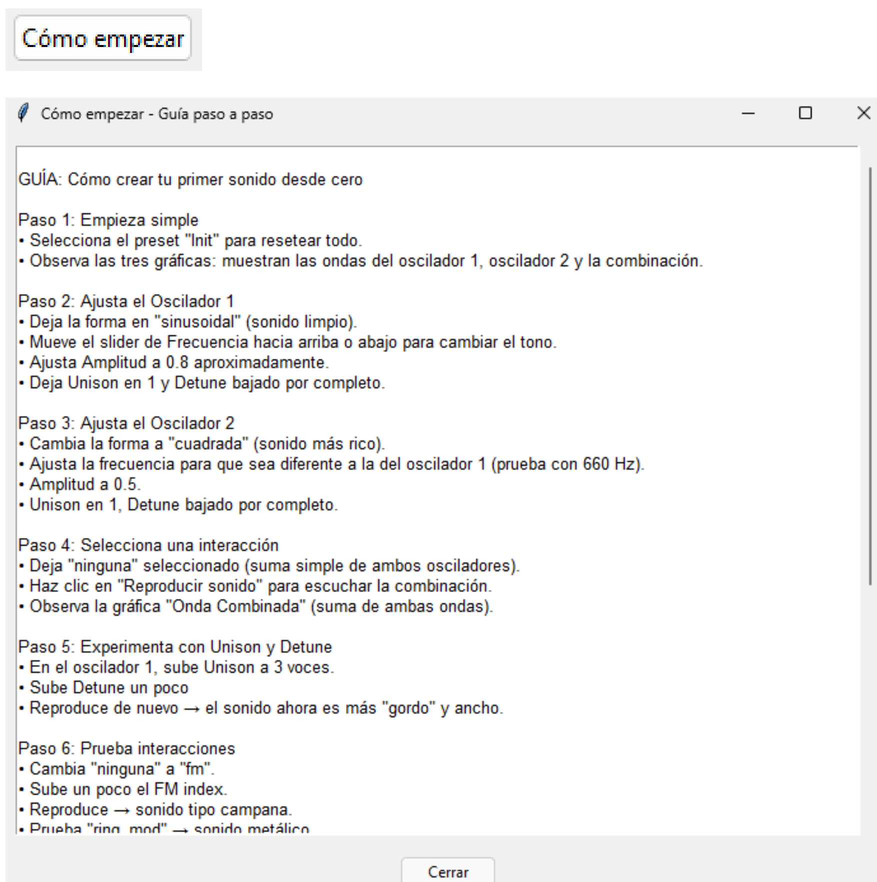
```
def crear_botones(self, parent):
    # ... (código de otros botones)

    preset_combo = ttk.Combobox(frame, textvariable=self.preset,
                                values=list(PRESETS.keys()),
                                width=14, state="readonly")
    preset_combo.bind("<<ComboboxSelected>>", self._on_preset_change)
```

3.10 Sistema de ayuda

El sistema de ayuda implementa tres niveles de documentación : tooltips instantáneos, popups por sección mediante botones '?', y un texto con una guía general estructurada. El contenido de esta guía general se almacena en TEXTO_AYUDA, TEXTO_GUIA_INICIO en ayuda.py.

```
TEXTO_AYUDA = """
Bienvenido al Sintetizador Didáctico

1) ¿Qué hace cada cosa?
- Frecuencia: controla qué tan agudo o grave es el sonido (tono).
- Amplitud: controla el volumen del oscilador.
[...]
"""

TEXTO_GUIA_INICIO = """
GUÍA: Cómo crear tu primer sonido desde cero

Paso 1: Empieza simple
• Selecciona el preset "Init" para resetear todo.
[...]
"""
```

Las funciones mostrar_ayuda y mostrar_guia_inicio crean ventanas Toplevel independientes con widgets Text de solo lectura en ayuda.py:



3.11 Reproducción de audio

El sistema de reproducción de audio utiliza SoundDevice para conversión de arrays NumPy a audio reproducible mediante hardware. Implementamos dos formas de reproducción: tocando una tecla del piano o pulsando el botón reproducir sonido.

3.11.1 Reproducción desde Teclado Virtual

El método tocar_nota en interfaz.py gestiona la reproducción de audio al pulsar el teclado. El estado original se mantiene después de tocar la nota.

```
def tocar_nota(self, note_name):
    f1_prev = self.osc1.frecuencia
    f2_prev = self.osc2.frecuencia
    f_note = nota_a_frecuencia(note_name)

    osc1_loc = Oscilador(frecuencia=f_note, amplitud=self.amp1.get(),
                        forma=self.forma1.get(), ...)
    osc2_loc = Oscilador(frecuencia=self.freq2.get(), ...)

    # Generación dual: visual (10ms) + audio (500ms)
    onda_comb_vis = fm(osc1_loc, osc2_loc, self.duracion_visual, ...)
    # ... actualizar gráficas ...

    onda_play = fm(osc1_loc, osc2_loc, self.duracion_audio, ...)
    max_abs = np.max(np.abs(onda_play))
    onda_norm = (onda_play / max_abs) if max_abs > 0 else onda_play
    sd.play(onda_norm, self.sample_rate)
    sd.wait()

    # Restaurar frecuencias originales
    self.osc1.set_frecuencia(f1_prev)
    self.osc2.set_frecuencia(f2_prev)
    self.actualizar_onda()
```

Osciladores locales: Se crean instancias Oscilador temporales en lugar de modificar self.osc1/self.osc2 para evitar efectos secundarios. La nota solo afecta OSC1 (frecuencia=f_note), mientras OSC2 mantiene su configuración (frecuencia=self.freq2.get()).

Las líneas finales restauran frecuencias previas (f1_prev, f2_prev) y vuelven a sincronizar la visualización mediante actualizar_onda(). Sin esta restauración, cada pulsación de tecla modificaría permanentemente los sliders de frecuencia, rompiendo coherencia entre interfaz y modelo.

Se generan dos ondas con duraciones diferentes

duracion_visual = 0.01 s Visualización correcta sin saturar gráficas.

duracion_audio = 0.5 s Suficiente tiempo para captar el sonido.

Así optimizamos el rendimiento ya que Matplotlib renderiza ventanas cortas rápidamente, mientras el audio proporciona duración suficiente.

Normalización de la onda:

```
max_abs = np.max(np.abs(onda_play))
onda_norm = (onda_play / max_abs) if max_abs > 0 else onda_play
sd.play(onda_norm, self.sample_rate)
```

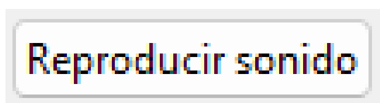
3.11.2 Reproducción directa con botón "Reproducir sonido"

El método reproducir_onda en interfaz.py reproduce el array precalculado self.onda_comb generado por actualizar_onda:

```
    def reproducir_onda(self):
        if self.onda_comb is not None:
            max_abs = np.max(np.abs(self.onda_comb))
            onda_norm = (self.onda_comb / max_abs) if max_abs > 0 else
            self.onda_comb
            print(f"Reproduciendo audio en dispositivo:
            {sd.default.device}")
            sd.play(onda_norm, self.sample_rate)
            sd.wait()
```

SoundDevice detecta automáticamente hardware disponible mediante PortAudio. La línea de debug printf(Reproduciendo audio en dispositivo: {sd.default.device}") facilita resolución de problemas en sistemas con múltiples interfaces de sonido.

Se llama a este método cuando pulsamos el siguiente botón de la interfaz gráfica:



4 Resultados y conclusiones

4.1 Cumplimiento de objetivos

El proyecto ha cumplido su objetivo principal: desarrollar una aplicación educativa que permita comprender los fundamentos de la síntesis de audio con osciladores digitales.

Objetivos técnicos alcanzados:

- Generación de formas de onda: Se implementaron las cuatro formas básicas (sinusoidal, cuadrada, triangular y sierra) mediante expresiones matemáticas.
- Osciladores: Dos osciladores independientes con control de frecuencia (20-2000 Hz), amplitud (0.0-1.0), forma de onda, unison (1-8 voces) y detune (0-50 cents).
- Modos de síntesis: Cuatro modos implementados (suma, FM, ring modulation, sync)
- Interfaz gráfica: GUI Tkinter con controles intuitivos , visualización en tiempo real mediante Matplotlib, y teclado virtual de 4 octavas.
- Sistema de ayuda: Tooltips , botones "?" integrados y guía paso a paso.

4.2 Conclusiones personales

Este proyecto me ha resultado bastante interesante de hacer ya que a mí me gusta bastante la música electrónica y haciendo este proyecto he aprendido muchas cosas del tema musical y el origen y funcionamiento del sonido electrónico.

También durante el desarrollo he aprendido bastante de lenguaje Python, de programación orientada a objetos y de diseño de interfaces

5 Análisis de Impacto

5.1 Impacto personal

A nivel musical se aprenden muchos conceptos y fundamentos lo cual es interesante para alguien que le gusta la música.

Desarrollo de conceptos en DSP (Digital signal processing), síntesis de audio, estructuras modulares y desarrollo de interfaces gráficas. Mejora en habilidades de resolución de problemas como optimización de latencia, sincronización y comunicación técnica.

También el proyecto ha supuesto una inversión de tiempo notable.

5.2 Impacto Empresarial

Este proyecto puede ser un recurso educativo para formación en empresas de audio sin inversión en hardware. Cuenta con un software libre (Python, NumPy, SoundDevice, Matplotlib) elimina costes de licencias y hace que sea totalmente gratis. La arquitectura modular permite la continuación o mejora del proyecto fácilmente.

La aplicación está orientada a la educación, no a la producción profesional de música.

5.3 Impacto Social

Una herramienta gratuita para el usuario, que puede ser utilizada en un ordenador de gama baja eliminando así barreras económicas para alguien que quiere aprender cómo funciona la síntesis de audio.

Una desventaja es que hay una curva de aprendizaje inicial. Además,el usuario tiene que tener conocimientos para la instalación de las librerías necesarias.

5.4 Impacto económico

Coste cero del programa ya que tiene librerías de libre acceso y funciona en ordenadores sin interfaces de audio profesionales.

El programa no genera ningún ingreso.

5.5 Impacto Medioambiental

No se produce ninguna huella ecológica relacionada con hardware ya que solo se utiliza software y tiene un consumo energético mínimo. Para utilizar el

programa necesitas un ordenador cuya fabricación sí tiene un impacto medioambiental.

5.6 Impacto cultural

Se utilizan técnicas de síntesis clásicas como Chowning con el FM. La aplicación está en español lo que podría suponer un problema para usuarios no hispanoparlantes.

5.7 Relación con objetivos de desarrollo sostenible.

ODS 4 [13] - Educación de Calidad: Desarrolla competencias técnicas en programación y procesamiento de audio (Meta 4.4).

ODS 9 [13] - Industria, Innovación e Infraestructura: Aplica investigación en DSP y síntesis de audio a herramienta práctica (Meta 9.5). La estructura modular facilita mejoras e innovación

ODS 10 [13] - Reducción de Desigualdades: Elimina desigualdades de acceso a educación musical/técnica (Meta 10.2). Interfaz en español reduce desigualdades lingüísticas.

ODS 12 [13] - Producción y Consumo Responsables: No requiere adquisición de hardware educativo, requiere software reutilizable, evitando residuos electrónicos (Meta 12.5).

5.8 Decisiones de diseño basadas en el impacto

Python y bibliotecas estándar: Maximiza accesibilidad (Python es el lenguaje más enseñado globalmente).

Tkinter: Minimiza tamaño de instalación y facilita ejecución en sistemas limitados.

Estructura modular: Permite extensión sin reescribir código.

6 Bibliografía

[1] J. O. Smith, Introduction to Digital Filters with Audio Applications. W3K Publishing, 2010.

[2] C. Roads, The Computer Music Tutorial. MIT Press, 1996.

[3] Python Software Foundation, “Python Documentation.” [Online]. Available: https://docs.python.org/

[4] “Tkinter Documentation.” [Online]. Available: https://docs.python.org/3/library/tkinter.html

[5] “Matplotlib Documentation.” [Online]. Available: https://matplotlib.org/

[6] “Sounddevice Documentation.” [Online]. Available: https://python-sounddevice.readthedocs.io/

[7] "NumPy Documentation." [Online]. Available: https://numpy.org/doc/

[8] R. C. Benson, Music: A Mathematical Offering. Cambridge University Press, 2007.

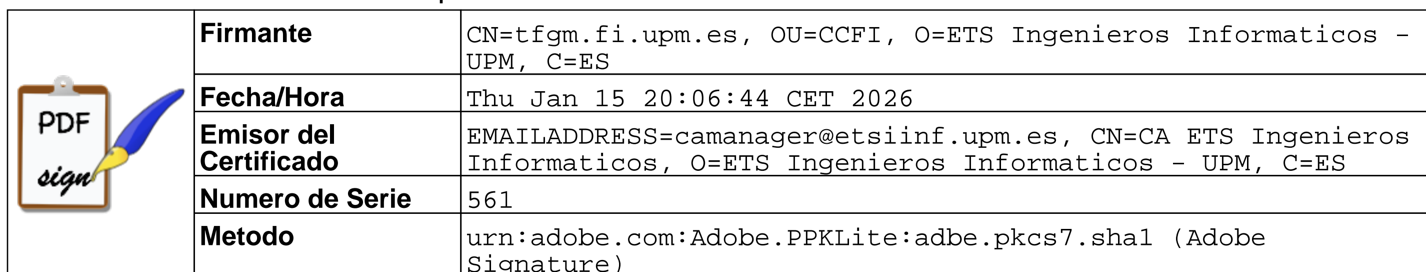
[9] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90-95, 2007

[10] N. P. Rougier, "Matplotlib tutorial," en *SciPy 2014 Conference*, 2014. [Online].
Disponible: https://www.labri.fr/perso/nrougier/teaching/matplotlib/

[11] F. Lundh, "An Introduction to Tkinter," PythonWare, 1999. [Online]. Disponible: https://effbot.org/tkinterbook/

[13] ODS: https://www.un.org/sustainabledevelopment/es/sustainable-development

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Thu Jan 15 20:06:44 CET 2026
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)