MANUAL DE USUARIO – MUGUI

Elaborado por:

Juan Camilo Páez Guaspud

Juan Sebastián Olaya Castañeda

Presentado a:

Néstor Germán Bolívar Pulgarín

Asignatura:

Programación Orientada a Objetos (POO)

Universidad Nacional de Colombia

2025-2s

Cómo instalar las librerías (requirements.txt)

El proyecto incluye un archivo requirements.txt que contiene todas las librerías necesarias.

Abre una terminal dentro de la carpeta del proyecto:

Escribe:

pip install -r requirements.txt

Esto instala todas las librerias (audio, Tkinter, numpy, etc.).

Archivos y sus clases

1. **mvvm/Model/AudioProcessor.py**

Propósito:

- Capturar audio en tiempo real desde el micrófono y extraer una estimación robusta de la frecuencia fundamental (f0) usando **librosa.yin**.

Clase principal: AudioProcessor

- Atributos clave:
- **p: pyaudio.PyAudio** — instancia de PyAudio para acceder a dispositivos.
- **RATE** — frecuencia de muestreo (por defecto 44100 Hz).
- **CHUNK** — tamaño del bloque de audio leído por frame (por defecto 32768).
- **FORMAT** — formato de audio (paInt16).
- **CHANNELS** — número de canales (1 mono).
- **MIN_ENERGY** — umbral de energía para considerar que hay señal útil.
- **stream** — objeto stream abierto por PyAudio.
- **device_index** — índice del dispositivo de entrada (si no se pasa, se detecta automáticamente el primer dispositivo con entrada disponible).

- Métodos principales:
- **__init__(self, device_index=None, chunk=32768, rate=44100, min_energy=0.00004)**

— inicializa PyAudio, parámetros y detecta dispositivo por defecto.

- **start(self)** — abre el stream de entrada; captura excepciones si el dispositivo está ocupado o no disponible.
- **process(self)** — lee un bloque (**CHUNK**), convierte a float normalizado, calcula energía media y, si supera **MIN_ENERGY**, aplica **librosa.yin** para estimar f0. Devuelve **(freq, energy)** cuando hay detección válida o **(None, energy)** cuando no hay tono.
- **stop(self)** — cierra el stream y finaliza PyAudio.

Notas de implementación y recomendaciones:

- **CHUNK** alto mejora resolución en bajas frecuencias pero incrementa latencia.
- Ajusta **MIN_ENERGY** si hay mucho/ poco ruido de fondo.
- **librosa.yin** se configura con **fmin=30** y **fmax=1318** para cubrir rango típico de guitarra.

2. **mvvm/Model/PitchAnalyzer.py**

Propósito:

- Convertir una frecuencia (Hz) en su nota musical correspondiente y calcular posiciones (cuerda, traste) en un diapasón de guitarra estándar.

Clase principal: PitchAnalyzer

- Constantes:
- **OPEN_STRINGS** — frecuencias de las 6 cuerdas al aire (E2..E4).
- **NOTES** — lista cromática (C..B).
- **A4** — referencia 440 Hz.

- Método clave:
- **freq_to_note(self, freq)** — valida el rango, convierte a número MIDI,

determina nombre de nota y octava, calcula desviación en cents y busca todas las posiciones posibles en el diapasón (cuerdas 1..6, trastes 0..19) donde la nota coincide.

Notas:

- Devuelve una tupla **(note_name, cents, positions, freq)** donde **positions** es una lista de pares **(string, fret)**.
- Está pensado para integración con la UI del afinador y para mostrar posiciones en el **Fretboard**.

3. **mvvm/Model/MetronomeModel.py**

Propósito:

- Producir el sonido del metrónomo (click) con estrategias de fallback: usar **pygame** si está disponible, generar sonido sintético con **numpy**, o usar **winsound**/beep de sistema como último recurso.

Comportamiento principal:

- En la importación intenta inicializar **pygame.mixer**. Define banderas (**_USING_PYGAME**, **has_numpy**, **has_winsound**) según disponibilidad.
- Busca archivos **sonidos/tic.wav** y **sonidos/tac.wav** en el root del proyecto (soporta PyInstaller ajustando **sys._MEIPASS**).

Clase principal: MetronomeModel

- Atributos:
- **using_pygame**, **has_numpy**, **has_winsound** — flags de capacidades.
- **click_accent**, **click_normal** — objetos **pygame.Sound** o **None**.

- Métodos:
- **_make_sound(self, freq_hz, duration_s, volume)** — genera un arreglo

  numpy y lo convierte a **pygame.Sound** (si numpy y pygame están disponibles).

- **play_click(self, accent: bool = False)** — reproduce el sonido acorde

  al tipo de click; si falla, intenta winsound o imprime **\a** como fallback.

Notas:

- Permite que la UI del metrónomo llame **play_click** sin preocuparse por

  disponibilidad de librerías externas.

4. **mvvm/Model/reproductorModel/pista.py** y **reproductorModel/reproductor.py**

Objetivo general:

- Encapsular la representación de una pista de audio (**Pista**) y la lógica de

  reproducción (**Reproductor**) usando **pygame.mixer**.

pista.py — Clase Pista

- Propósito: representar una pista de audio y exponer metadatos útiles.
- Atributos:
- **ruta** — ruta al archivo.
- **_info** — caché del objeto devuelto por **mutagen.File**.
- Métodos / propiedades:
- **nombre** — nombre base del archivo (propiedad).
- **cargar_info()** — intenta cargar metadatos con **mutagen.File** y los

  guarda en **_info**.

- **duracion_segundos()** — devuelve **info.length** (o **0** si no hay info).

reproductor.py — Clase Reproductor

- Propósito: manejar una lista de **Pista** y control de reproducción básica.

- Atributos:
- **pistas: list[Pista]** — lista de objetos Pista.
- **indice_actual: int** — índice de la pista seleccionada.
- **volumen: float** — 0.0..1.0.
- Métodos:
- **agregar_rutas(self, rutas: list[str])** — añade rutas como **Pista**.
- **pista_actual(self)** — devuelve la **Pista** actual o **None**.
- **cargar_actual(self)** — carga la pista actual en **pygame.mixer**.
- **reproducir/pausar/continuar/detener/siguiente/anterior** — controles de reproducción; **siguiente**/**anterior** actualizan **indice_actual**.
- **establecer_volumen(self, valor: float)** — ajusta volumen y lo aplica.

Notas:

- **pygame.mixer** debe estar inicializado antes de usar este modelo; la clase intenta inicializarlo en su constructor.

5. **mvvm/Model/firebase_admin.py** (interfaz con Firebase)

Propósito:

- Proveer funciones para autenticación con Google/Firebase y operaciones básicas en RTDB. Contiene helpers para intercambiar códigos OAuth y autenticarse con el Identity Toolkit de Firebase.

Componentes y comportamientos importantes:

- **firebase_config** — diccionario con la configuración del proyecto utilizada por **pyrebase** (API key, authDomain, databaseURL, etc.).
- Inicialización: **pyrebase.initialize_app(firebase_config)** y creación de **auth** y **db** (RTDB).

Clase principal: FirebaseAdmin

- Atributos:
- **current_user** — datos del usuario autenticado (según respuesta de Firebase).
- **google_client_id**, **google_client_secret** — cargados desde **.env**.

- Métodos principales:
- **get_google_auth_url()** — genera la URL de autorización de Google. Nota: el proyecto migró de OOB a un servidor OAuth local; la función prepara el flow adecuado según la implementación.
- **exchange_code_for_id_token(code)** — intercambia el **code** de Google por **id_token** usando **https://oauth2.googleapis.com/token**.
- **sign_in_with_google(id_token)** — llama al endpoint **accounts:signInWithIdp** de Firebase Identity Toolkit para crear/validar sesión en Firebase.
- **get_current_user()**, **sign_out()** — utilidades de sesión.
- RTDB: **save_user_config(uid, config)**, **load_user_config(uid)**, **update_user_config(uid, updates)** — operaciones CRUD básicas en RTDB.

Notas y consideraciones:

- Maneja errores HTTP y devuelve mensajes amigables. Es sensible a errores tipo **invalid_client**, **invalid_grant**, y a problemas de red (timeouts).
- Requiere que **client_id**/**client_secret** estén configurados en **.env**.
- Se añadió **oauth_local_server.py** en el modelo para soportar el redirect a **http://localhost:<port>/** evitando el flujo OOB bloqueado.

6. Notas generales del Model

- Hilos y concurrencia: varios modelos (p. ej. **TunerApp** en ViewModel)

  lanzan threads que hacen uso de los modelos del **Model** (AudioProcessor,

  Reproductor). Asegúrate de:

- Detener streams y hilos correctamente en **cleanup**.
- No llamar directamente a la GUI desde threads; usa **root.after(...)**.
- Manejo de errores: los modelos intentan capturar excepciones y devolver

  valores por defecto (p. ej. **0** o **None**) para no romper la UI.

- Dependencias críticas:
- **pyaudio** / **sounddevice** / **librosa** para captura y análisis de audio
- **pygame** para reproducción y metrónomo (con fallback a winsound)
- **mutagen** para metadata de audio
- **pyrebase4** / **requests** para autenticación y RTDB

Fin de la sección **mvvm/Model**.

mvvm/View — Documentación detallada (Frontend / Vistas)

Esta sección describe las vistas (frames y diálogos) implementadas con

Tkinter dentro de **mvvm/View**. Para cada archivo se explica su propósito,

las clases principales, atributos relevantes, métodos y consideraciones de

uso (responsividad, hilos, gestión de imágenes, etc.).

7. **mvvm/View/menu.py** (MenuFrame)

Propósito:

- Pantalla de bienvenida con logo y botones para navegar a los módulos principales (Afinador, Metrónomo, Acordes, Reproductor).

Clase principal: MenuFrame (hereda **ttk.Frame**)

- Atributos clave:
- **on_tuner_clicked**, **on_metronome_clicked**, **on_reproductor_clicked**, **on_chords_clicked** — callbacks provistos por **FrameManager**.
- **_base_width**, **_base_height**, **_scale_factor** — parámetros para escalado responsivo.
- **main_frame**, **description_label**, **buttons_frame** — widgets raiz del UI.
- Manejo de imagen: carga de **imagenes/logo.png** mediante PIL, escalado seguro y retención de referencia **PhotoImage** para evitar Garbage Collection.

- Métodos importantes:
- **setup_ui()** — configura estilos ttk, layout, crea logo y botones en una cuadrícula 2x2. Usa **rowconfigure/columnconfigure** para que los

botones escalen con la ventana.

- **_create_button(parent, text, callback, row, column)** — helper para crear botones con estilo y colocarlos con **grid(sticky='nsew')**.
- **_get_logo_path()** / **_add_logo(parent, logo_path)** — localizan y cargan el logo; manejan excepciones si la imagen no existe.
- **_on_resize(event)** y **_update_fonts()** — ajustan **_scale_factor** y actualizan tamaños de fuente para mantener la interfaz legible al cambiar el tamaño de ventana.

Notas y recomendaciones:

- Los botones originalmente usaban emojis; se reemplazaron por textos para evitar problemas de encoding/visualización en distintas plataformas.
- Para que los botones no se vean aplastados, el frame de botones está configurado con pesos iguales en filas y columnas.

8. **mvvm/View/MenuManager.py**

Propósito:

- Crear y mantener la barra de menú de la aplicación (Menú Inicio, Funciones, Cuenta). Actualiza dinámicamente el menú de cuenta según el estado de autenticación.

Clase principal: MenuManager

- Atributos:
- **root**, **frame_manager**, **menu_vm** (MenuViewModel), **auth_view** (enlace a **AuthenticationView**), **menu_bar**, **account_menu**.

- Métodos principales:
- **create_menu()** — construye la estructura de menús (Inicio, Funciones, Cuenta) y la asocia con **root.config(menu=...)**.
- **update_account_menu()** — borra y reconstruye el submenú de Cuenta según **menu_vm.get_account_menu_items()** (muestra usuario/ email o acciones de iniciar sesión/registrarse). Usa **self.auth_view** para delegar diálogos.
- **set_auth_view(auth_view)** — inyección de la vista de autenticación para evitar importaciones circulares.

Notas:

- **update_account_menu** debe llamarse tras cambios de sesión (login/logout).

9. **mvvm/View/FrameManager.py**

Propósito:

- Orquestar la navegación entre frames de la aplicación. Se encarga de crear/mostrar los frames (Menu, Afinador, Metrónomo, Reproductor, Detector de Acordes) y de iniciar/parar loops de audio cuando corresponda.

Clase principal: FrameManager

- Atributos:
- **root**, **main_container**, **nav_vm** (FrameNavigationViewModel con estado de navegación y referencias a hilos/modelos activos).

- Métodos importantes:
- **show_menu()**, **show_tuner()**, **show_metronome()**, **show_reproductor()**, **show_chords()** — cada método limpia el frame actual y crea el nuevo

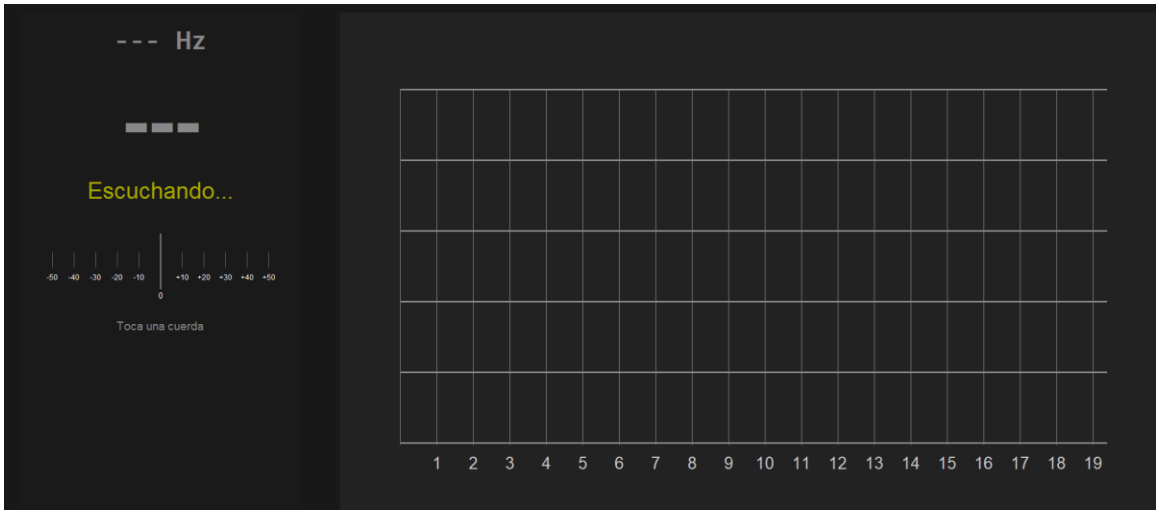
frame, ajustando **root.geometry(...)** y flags como **nav_vm.audio_running**.

- **cleanup_current_frame()** — delega en **nav_vm** la limpieza del frame anterior (p. ej. detener audio, hilos, timers).
- **_audio_loop()** — método que se ejecuta en un thread cuando se muestra el afinador: recibe resultados del ViewModel de audio y usa **root.after(0, ...)** para actualizar la GUI de forma thread-safe.

Notas:

- **FrameManager** crea instancias de viewmodels (por ejemplo **TunerApp** o **ChordDetectorViewModel**) y de las vistas correspondientes, conectando callbacks. Importante: nunca actualizar widgets directamente desde hilos (usar **root.after**).

10. **mvvm/View/TunerGUI.py**



Propósito:

- Componer la UI del afinador: muestra el **TunerCalibratorFrame** (visualizador de frecuencia y aguja) y el **FretboardFrame** (diapasón). Gestiona el reescalado de fuentes y delega actualizaciones de audio a los subcomponentes.

Clase principal: TunerGUI (hereda **Frame**)

- Atributos:
- **calibrator: TunerCalibratorFrame**, **fretboard: FretboardFrame**.
- **_base_width/_base_height/_scale_factor** para responsividad.

- Métodos relevantes:
- **on_audio_update(self, freq, note, cents, positions, energy=None)** — callback público que delega a **calibrator.update(...)** y **fretboard.update_notes(positions)** en try/except para evitar fallos.
- **_on_resize** / **_update_fonts** — propagan el factor de escala a los subcomponentes (llamando a **_update_fonts** si existe).

Notas:

- **TunerGUI** mantiene compatibilidad mediante **update_alias** para código antiguo que usara otro nombre de método.

11. **mvvm/View/afinador/TunerCalibrator.py** (TunerCalibratorFrame)

Propósito:

- Presentar la lectura de frecuencia, la nota detectada, desviación en cents y una aguja que indica afinación relativa.

Clase principal: TunerCalibratorFrame (hereda **Frame**)

- Atributos y widgets:
- **freq_label**, **note_label**, **cents_label**, **status_label** — etiquetas que muestran información principal.
- **canvas** — lienzo donde se dibuja la regla y la aguja (**needle**).
- **last_freq**, **last_note**, **last_cents** — cachés de la última lectura válida.
- **_base_width/_scale_factor** — para ajustar fuentes en **_on_resize**.

- Métodos:
- **update(self, freq, note, cents)** — mantiene la última lectura válida si la nueva es None; actualiza etiquetas y mueve la aguja llamando **move_needle(cents)** (con límites de ±50 cents).
- **move_needle(self, cents)** — calcula offset en pixeles y actualiza

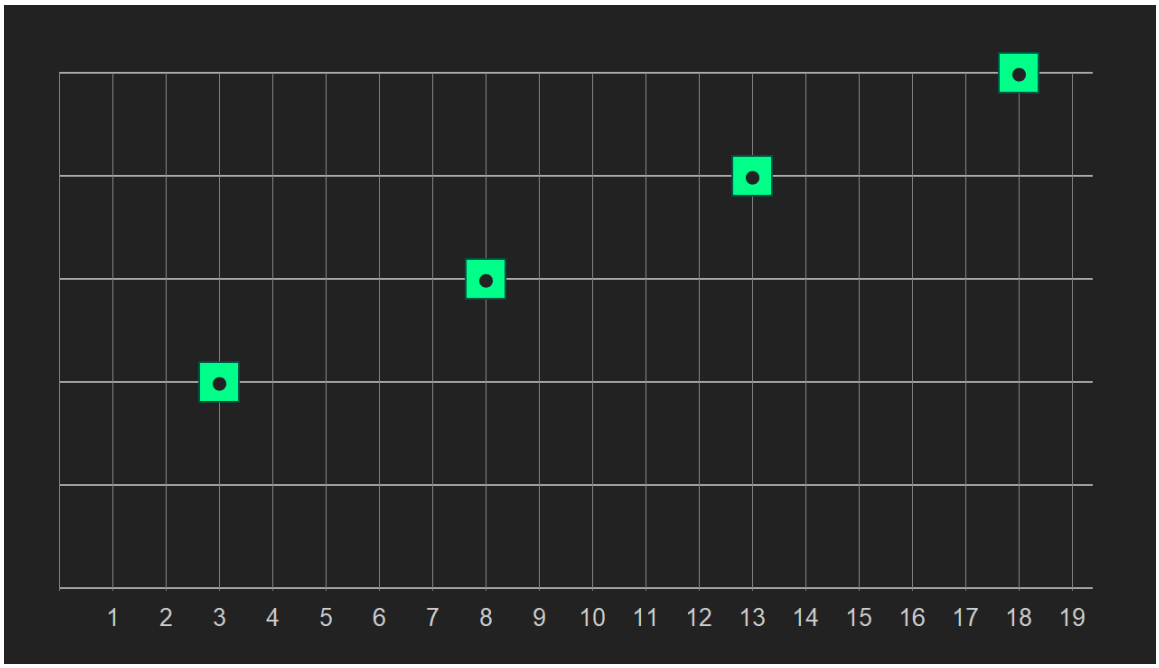
coordenadas del **needle** en el canvas.

- **_on_resize(self, event)** y **_update_fonts(self, scale_factor=None)** — recalculan tamaños de fuentes con **int(base * _scale_factor)** protegiendo contra excepciones.

Notas:

- El canvas dibuja marcas cada 10 cents; la función **update** es tolerante a lecturas faltantes y mantiene la última información visible.

12. **mvvm/View/afinador/Fretboard.py** (FretboardFrame)



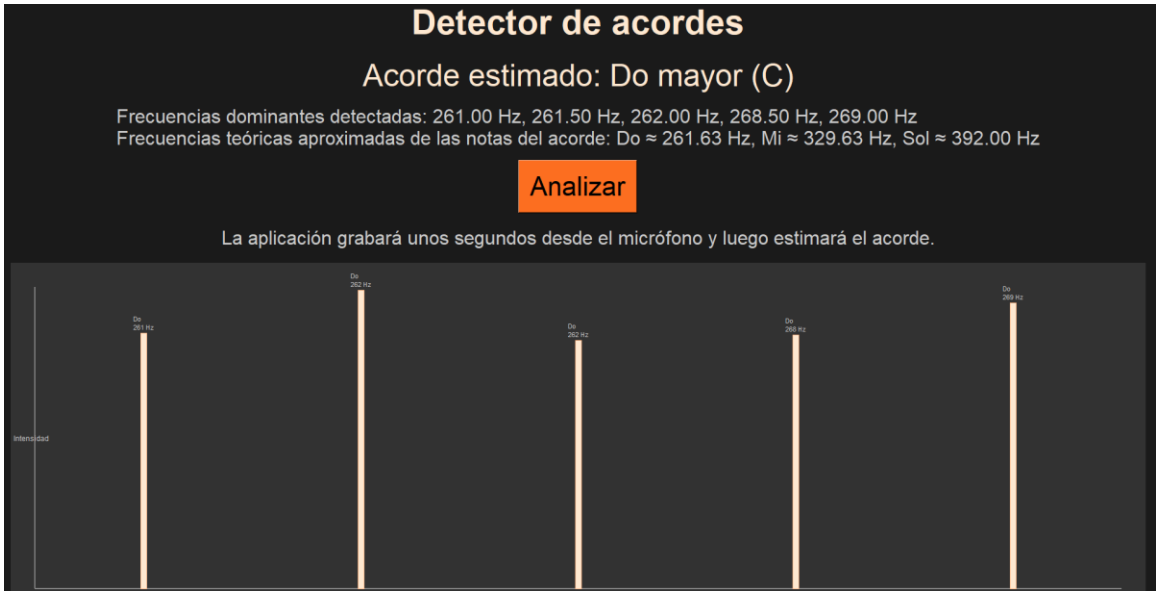
Propósito esperado:

- Mostrar un diagrama del diapasón y resaltar posiciones (cuerda, traste) sugeridas por **PitchAnalyzer**.

La interfaz pública esperada incluye métodos como **update_notes(positions)** y **_update_fonts(scale)** para integrarse con

TunerGUI y **TunerCalibrator**.

13. **mvvm/View/chords.py** (ChordDetectorView)



Propósito:

- Ofrecer una UI para detectar acordes grabando unos segundos de audio,

  mostrando el acorde estimado, detalles (notas componentes) y un diagrama de

  frecuencias.

Clase principal: ChordDetectorView (hereda **tk.Frame**)

- Atributos:
- **_view_model: ChordDetectorViewModel**, botones y labels (**chord_label**,

  detail_label, **info_label**), **canvas** para diagrama y **_base_width**/

  _scale_factor para responsividad.

- Flujo principal:
- **on_analyze_click()** — deshabilita el botón y pide al VM **analyze_once_async()**.
- **_check_result_queue()** — invocado periódicamente con **after(...)** para

  leer la **result_queue** del viewmodel; por cada resultado se actualiza

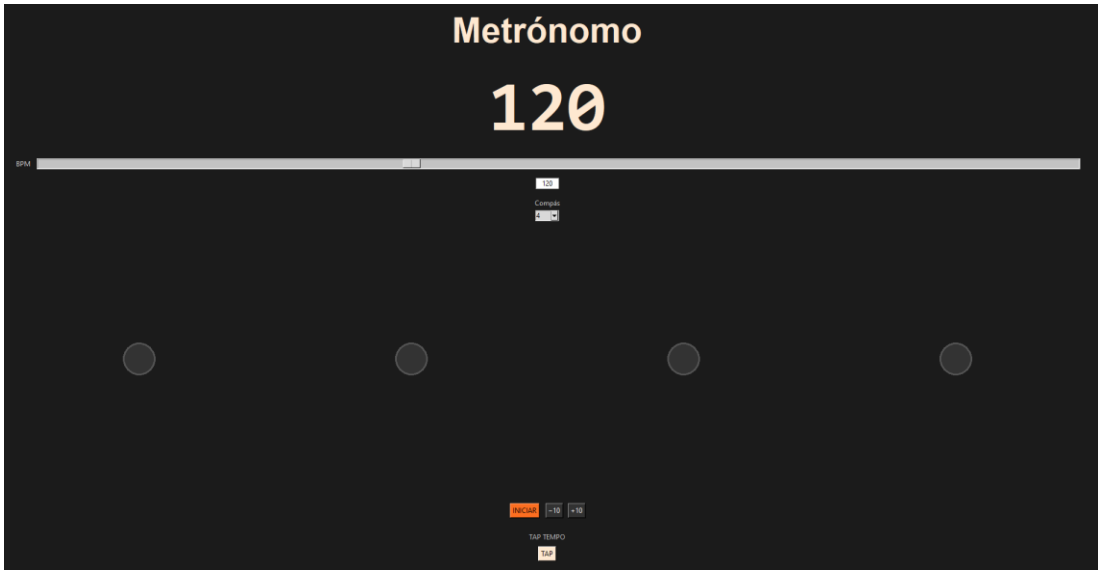
chord_label, **detail_label** y se pinta el diagrama con **_draw_frequency_diagram(freqs, labels, mags)**.

- **_draw_frequency_diagram(...)** — dibuja ejes, barras (intensidad) y etiquetas en el canvas; gestiona tamaños según **winfo_width()/winfo_height()**.

Notas:

- La vista depende de que el ViewModel entregue resultados en una cola thread-safe; por tanto, la lectura regular con **after** evita bloquear la UI.

14. **mvvm/View/metronomo.py** (MetronomeFrame)



Propósito:

- Interfaz completa del metrónomo que conecta con **MetronomeModel** y **MetronomeViewModel**, muestra BPM, compás, indicadores visuales y controles de reproducción (start/stop, tap tempo, ajustes).

Clase principal: MetronomeFrame (hereda **ttk.Frame**)

- Atributos:
- **model: MetronomeModel**, **vm: MetronomeViewModel**.
- Widgets: **bpm_label**, **slider**, **entry_bpm**, **compas** combobox,

  _lights_frame con indicadores visuales constituidos por círculos/labels.

- **_base_width/_base_height/_scale_factor** para responsividad.

- Comportamiento:
- Los indicadores (**_lights**) se recrean al cambiar el compás mediante

  _recreate_lights() y se actualizan con **_on_tick** (callback desde el VM).

- **_highlight(idx, accent)** aplica estilos temporales a un indicador para

  mostrar el pulso.

- UI -> VM: controles como **_on_slider**, **_on_entry**, **_on_compas**,

  _on_toggle, **_on_tap** llaman al ViewModel para cambiar BPM/estado.

Notas:

- La vista usa **after** y flags para sincronizar la animación con el VM y

  evita tocar audio directamente (delegando en **MetronomeModel**).

15. **mvvm/View/reproductorFrame.py** y **mvvm/View/reproductorView/reproductorUI.py**

Propósito general:

- Envolver la UI del reproductor en un frame embebible para la ventana principal. **reproductorUI** contiene los controles (play/pause/stop, barra de tiempo, lista de pistas) y es consumido por **ReproductorFrame**.

ReproductorFrame (hereda **Frame**)

- Atributos:
- **vm: ReproductorViewModel**, **vista: VistaReproductor**.
- Variables para responsividad: **_base_width/_scale_factor**.

- Métodos:
- **_on_resize** y **_update_fonts** — delegan el escalado de fuentes a **vista._update_fonts(self._scale_factor)** si existe.

Notas:

- **reproductorUI** utiliza **pygame.mixer** a través del ViewModel para cargar y reproducir pistas; asegúrate de inicializar **pygame.mixer** y de ejecutar las operaciones pesadas de E/S en hilos si quieres evitar bloquear la UI.

16. **mvvm/View/importmusic.py**

Propósito:

- Utilidad simple que muestra un **filedialog** para seleccionar un archivo de audio y lo reproduce con **pygame.mixer.music**.

Implementación:

- **MusicImporter.import_music()** — inicializa **pygame.mixer**, abre el selector de archivos y reproduce la pista seleccionada en el hilo actual.

Consideraciones:

- El bucle **while pygame.mixer.music.get_busy(): time.sleep(0.5)** bloquea el hilo actual; no usar directamente desde el hilo principal de la UI sin delegar a un hilo separado.

17. **mvvm/View/splash_screen.py**

Propósito:

- Mostrar una pantalla de bienvenida/ carga con barra de progreso antes de mostrar la ventana principal.

Clase principal: SplashScreen

- Atributos:
- **splash: Toplevel**, **progress: Canvas**, **progress_bar** (id del rectángulo), **logo_img** (PhotoImage opcional, mantener referencia para no perder la imagen).

- Métodos:
- **center_window(window, width, height)** — centra la ventana en pantalla.
- **animate_progress(value)** — anima la barra de progreso con **after** y cierra la splash al completarse llamando a **finish()**.
- **show()** — asigna **grab_set()** para bloquear interacción con la ventana principal mientras la splash está activa.

Notas:

- Usa **overrideredirect(True)** para ocultar bordes; en algunos gestores de ventanas puede impedir mover la ventana.

18. **mvvm/View/AuthenticationView.py**

Propósito:

- Envolver la interacción entre **AuthenticationViewModel** y el usuario: abrir diálogos de login/registro y actualizar la UI después de cambios de sesión.

Clase principal: AuthenticationView

- Atributos:
- **root**, **auth_vm** (AuthenticationViewModel), **menu_manager** (para refrescar menú tras cambios de sesión).

- Métodos clave:
- **initialize()** — carga la sesión guardada si el VM expone **load_session_from_file()**.
- **show_login_dialog()** — instancia **LoginDialog** (diálogo modal).
- **logout_with_confirmation()** — muestra **LogoutConfirmDialog** y llama **auth_vm.logout()** si el usuario confirma; luego actualiza el menú.
- Callbacks **_on_session_loaded** y **_on_auth_success** — actualizan el menú cuando cambian los datos de sesión.

Notas:

- **AuthenticationView** no implementa la lógica de OAuth; delega al **AuthenticationViewModel/firebase_admin** y a diálogos como **login_dialog.py** (que no editamos aquí).

19. Notas generales sobre **mvvm/View**

- Responsividad: la mayoría de frames usan **_base_width/_base_height** y un **_scale_factor** calculado en **_on_resize** para ajustar tamaños de fuente

y layout. Esto mejora la adaptabilidad a distintas resoluciones.

- Concurrencia: las vistas que reciben datos de threads (afinador, acordes) consumen resultados mediante colas y **after(...)** para evitar acceder a widgets desde hilos secundarios.
- Imágenes: siempre guardar la referencia a **PhotoImage** en el objeto (p.ej. **self.logo_img**) para evitar que se recolecte por el GC y desaparezca de la UI.
- Recomendación: evitar operaciones bloqueantes en callbacks (usar threads para I/O y cómputo intensivo) y preferir **root.after** para sincronizar.

Fin de la sección **mvvm/View**.

mvvm/ViewModel — Documentación detallada (ViewModels)

Esta sección documenta los módulos bajo **mvvm/ViewModel** que coordinan la lógica entre **mvvm/View** y **mvvm/Model**. No se incluye **authentication_vm.py** por petición del usuario.

20. **mvvm/ViewModel/TunerApp.py**

Propósito:

- Orquestar el pipeline del afinador: captura de audio (AudioProcessor), análisis de tono (PitchAnalyzer) y notificación a la interfaz gráfica.

Clase principal: TunerApp

- Atributos:
- **audio: AudioProcessor** — captura y preprocesa audio.
- **analyzer: PitchAnalyzer** — convierte frecuencia en nota, cents y posiciones de diapasón.
- **gui** — referencia al componente de UI que recibe actualizaciones

(típicamente **TunerGUI**).

- Métodos clave:
- **__init__(self)** — crea **AudioProcessor** y **PitchAnalyzer**.
- **run(self, gui)** — asigna **self.gui = gui**, arranca **audio.start()** y lanza un thread que ejecuta **audio_loop()**. También puede llamar a **gui.run()** si la GUI expone un método **run** (diseño opcional).
- **audio_loop(self)** — bucle que lee **audio.process()**, obtiene **freq** y **energy**, pide al **analyzer.freq_to_note(freq)** y notifica a la GUI mediante **gui.on_audio_update(...)** o **gui.update(...)** protegiendo con try/except. Duerme ~20ms entre iteraciones para limitar CPU.

Notas:

- El hilo de audio no debe actualizar widgets Tkinter directamente; el código actual delega a la GUI (que a su vez hace **root.after** cuando es necesario en **FrameManager**). Mantener la separación GUI/Worker es importante para evitar bloqueos o excepciones de Tcl.

21. **mvvm/ViewModel/reproductor_vm.py**

Propósito:

- Exponer la API del modelo **Reproductor** a la vista: abrir archivos, controlar reproducción y notificar periódicamente la UI sobre el estado.

Clase principal: ReproductorViewModel

- Atributos:
- **modelo: Reproductor** — instancia del modelo que gestiona pistas y reproducción (usa **pygame.mixer**).

- **_actualizaciones: list[callable]** — callbacks registrados por la UI que deben llamarse periódicamente para refrescar elementos (barra de tiempo, etiquetas).
- **_ejecutando: bool**, **_update_thread: Optional[threading.Thread]** — gestión del thread que notifica la UI.

- Métodos clave:
- **abrir_archivos()** — abre un selector (**filedialog.askopenfilenames**), añade rutas al modelo (**modelo.agregar_rutas**) y selecciona la primera.
- **reproducir()**, **pausar()**, **detener()**, **siguiente()**, **anterior()** — delegan en **modelo** y manejan el thread de actualización (**iniciar_actualizador/detener_actualizador**).
- **establecer_volumen(v)** — adapta el valor recibido por la UI (0..10) a rango 0.0..1.0 y lo envía al modelo.
- **pista_actual()**, **duracion_pista()** — utilidades para la UI.
- **posicion_reproduccion_segundos()** — intenta leer la posición usando **pygame.mixer.music.get_pos()** y convertir ms a segundos; si falla, devuelve 0.
- **iniciar_actualizador()** — crea un thread daemon que periódicamente llama a **notificar()** (cada 50ms por diseño) para que la UI se actualice.
- **detener_actualizador()** — solicita la parada del thread y espera un corto tiempo a que termine.

Notas y observaciones:

- Las operaciones de E/S (carga de archivos) y llamadas a **pygame** pueden fallar; los métodos envuelven llamadas en try/except para no romper la UI.
- Pequeño bug: **continuar()** llama a **iniciar_actualizador()** dos veces en

el código actual; se recomienda solo una invocación.

22. **mvvm/ViewModel/MetronomeVM.py**

Propósito:

- Controlar el estado del metrónomo: BPM, compás, comenzar/parar y emitir callbacks de pulso que la vista visualizará (luces/animaciones).

Clase principal: MetronomeViewModel

- Atributos:
- **model: MetronomeModel** — responsable de la generación/reproducción de sonidos del click.
- **bpm: int**, **compas: int**, **running: bool** — estado del metrónomo.
- **_thread: Optional[threading.Thread]**, **_beat_count: int** — control del loop interno.
- **on_tick: Optional[Callable[[int, bool], None]]** — callback que la vista asigna para ser notificada en cada pulso (índice de beat y si es acento).

- Métodos:
- **set_bpm(bpm)**, **set_compas(compas)** — validación y ajuste de parámetros.
- **toggle()**, **start()**, **stop()** — control del hilo de reproducción.
- **_run_loop()** — loop que calcula el delay entre pulsos (60/bpm), llama **model.play_click(accent)** y notifica **on_tick** a la vista; mantiene **_beat_count** y duerme **delay** segundos.
- **tap()** — registra taps recientes y, si hay suficientes, estima un BPM promedio para ajustar tempo.

Notas:

- El model (**MetronomeModel**) encapsula los fallbacks por disponibilidad de

librerías (pygame/numpy/winsound). El VM solo orquesta el tiempo y la notificación a la vista.

23. **mvvm/ViewModel/MenuViewModel.py**

Propósito:

- Preparar los datos necesarios por **MenuManager** para renderizar el menú de cuenta (estado logueado/deslogueado, nombre y email).

Clase principal: MenuViewModel

- Atributos:
- **auth_vm: AuthenticationViewModel** — referencia al ViewModel de autenticación (solo lectura desde aquí).

- Método clave:
- **get_account_menu_items()** — devuelve un diccionario con la forma **{'logged_in': bool, 'username': str, 'email': str}** que la vista consumirá para construir el menú dinámico.

24. **mvvm/ViewModel/FrameNavigationViewModel.py**

Propósito:

- Mantener el estado de navegación global (frame actual, hilos activos, referencia al **TunerApp**) y exponer utilidades para limpiar recursos al cambiar de pantalla.

Clase principal: FrameNavigationViewModel

- Atributos:
- **current_frame: Optional[tk.Widget]** — referencia al frame activo.

- **audio_thread: Optional[threading.Thread]**, **tuner_app: Optional[TunerApp]** — para el afinador.
- **audio_running: bool** — bandera que controla el loop de audio.

- Métodos:
- **cleanup_current_frame()** — detiene metrónomo y reproductor vía sus **vm** si están presentes, para el hilo de audio del afinador (la bandera **audio_running = False**) y llama a **join** con timeout; finalmente destruye el widget actual.
- **process_audio(callback)** — arranca **tuner_app.audio.start()** y entra en un bucle mientras **audio_running** y exista **current_frame**; por cada frame de audio llama al **analyzer.freq_to_note(...)** y ejecuta el **callback(freq, note, cents, positions, energy)** pasado (normalmente **FrameManager._audio_loop** que hace **root.after** para actualizar la UI).

Notas:

- **cleanup_current_frame** es el punto central para evitar fugas de hilos y recursos cuando se navega entre vistas. Debe capturar excepciones para tolerar estados parciales.

25. **mvvm/ViewModel/chords_vm.py**

Propósito:

- Grabar unos segundos de audio, calcular el espectro, identificar picos dominantes y mapearlos a un acorde estimado; exponer resultados en una cola (thread-safe) para que la vista los consuma.

Componentes principales:

- **ChordDetectionResult** (dataclass) — contenedor para **summary_text**, **detail_text**, **dominant_freqs**, **note_labels**, **magnitudes**.
- **ChordDetectorViewModel** — clase principal con:
- **result_queue: queue.Queue** — cola donde se ponen instancias de **ChordDetectionResult** o mensajes de error.
- **_build_chord_dictionary()** — genera un diccionario de acordes mayores y menores basados en terceras y quintas relativas.
- **_detect_dominant_frequencies(audio_data, sample_rate, n_peaks=5)** — aplica ventana, FFT y selecciona los picos más energéticos por encima de **min_freq** (50 Hz).
- **detect_chord_from_audio(audio_data, sample_rate)** — mapea picos a nombres de nota (sin octava), busca el mejor acorde en el diccionario y construye un **ChordDetectionResult** con detalles.
- **analyze_once_async()** — spawn de un worker en un hilo daemon que usa **sounddevice.rec()** para grabar **CAPTURE_SECONDS** y luego pone el resultado o mensaje de error en **result_queue**.

Notas:

- La comunicación con la vista se realiza mediante **result_queue**; la vista debe consumir la cola periódicamente (ej. con **after**) para actualizar la interfaz sin bloquear.

26. Notas generales sobre **mvvm/ViewModel**

- Patrón: los ViewModels hacen de puente entre la UI y los Modelos, exponiendo operaciones aptas para ser llamadas desde la vista y callbacks para notificar

eventos (por ejemplo **on_tick**, **on_auth_success**, **on_tick**).

- Hilos: varios VMs lanzan hilos demonio (afinador, reproductor updater, chord analyzer). Siempre manejar flags de parada y **join** con timeout en **cleanup** para evitar procesos huérfanos.
- Comunicación GUI/VM: preferir colas y **root.after** para pasar datos desde hilos de fondo a widgets Tkinter.

Fin de la sección **mvvm/ViewModel**.

Descripción breve: **MUGUI.py** (ejecutable)

Propósito:

- Punto de entrada de la aplicación. Inicializa la ventana principal (Tk), crea managers (FrameManager, MenuManager), la vista de autenticación, muestra la pantalla splash y arranca el bucle principal de Tkinter.

Comportamiento típico (resumen):

- Crear **root = tk.Tk()** y configurar estilo/tema y tamaño inicial.
- Instanciar **FrameManager(main_container)** y **MenuManager(root, frame_manager, auth_vm)**.
- Crear **AuthenticationView**, **SplashScreen** y registrar callbacks entre managers (por ejemplo **menu_manager.set_auth_view(auth_view)**).
- Mostrar la splash; al terminar, llamar **frame_manager.show_menu()**.
- Registrar **root.protocol('WM_DELETE_WINDOW', on_closing)** que delega en **frame_manager.cleanup_current_frame()** antes de **root.destroy()**.