eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

INGENIERÍA DE TELECOMUNICACIÓN

DOCUMENTACIÓN TÉCNICA

PROGRAMA DE APOYO A LA ENSEÑANZA MUSICAL

Estudiante *Del Blanco Sierra, Eder*
Director/a *Hernáez Rioja, Inmaculada*
Departamento *Ingeniería de Comunicaciones*
Curso académico *2019-2020*

AHOLAB

Bilbao, 8 de agosto del 2020

Índice

1 Introducción

Este documento está destinado a la persona que vaya a ampliar la aplicación Signal Visualizer, o bien para quien tenga que arreglar algún fallo. Explicaré, por un lado, cómo se utiliza MATLAB App Designer, el complemento de MATLAB con el que está hecha la aplicación: cuáles son las funciones propias de los módulos y los diferentes tipos de callback, cómo se interconectan los módulos entre sí... También explicaré aspectos propios de la aplicación. También se han añadido comentarios en el propio código, explicando el funcionamiento concreto de cada parte.

Este documento es el manual del programador de Signal Visualizer. Su objetivo principal es proporcionar información sobre el diseño y la estructura del software para servir de ayuda a las personas que continuarán con el desarrollo de este software. Por un lado se describe el modo de suo de MATLAB App Designer, el complemento de MATLAB con el que está hecha la aplicación: cuáles son las funciones propias de los módulos y los diferentes tipos de callback, cómo se interconectan los módulos entre sí... A continuación se describen aspectos propios de la aplicación. Por otro lado, este manual se complementa con los comentarios añadidos en el código, explicando el funcionamiento concreto de cada parte.

2 Listado de ficheros

2.1 Árbol de directorios

Todos los ficheros *.mlapp*, correspondientes a los diferentes módulos, están guardados en el directorio de la aplicación. Las imágenes están en el mismo directorio, ya que colocarlas en otro directorio puede dar problemas. Lo mismo ocurre con los ficheros *.mat* que utiliza la aplicación. Dentro de ese directorio hay un subdirectorio, *SignalVisualizer*, con los ficheros creados durante la compilación del programa, incluyendo los instaladores finales. El subdirectorio *Help files* contiene la ayuda de usuario. Este, a su vez, contiene tres subdirectorios: *en*, *es* y *eus*, que contienen los ficheros de ayuda de usuario en inglés, castellano y euskera respectivamente. Dentro de cada uno de estos directorios hay carpetas que se corresponden con cada uno de los módulos. En cada una de ellas está el documento *.html* de ayuda correspondiente al módulo y una carpeta con las imágenes que utiliza el documento.

El árbol de directorios es el siguiente:

- SignalVisualizerApp
  - Help files
    - en
      - Generate pure tone
      - Harmonic synthesis
      - Generate square wave
      - Generate sawtooth signal
      - Generate Rosenberg pulse
      - Generate noise
      - Load audio file
      - Record audio
      - Visualization window
    - es
      - Generar tono puro
      - Síntesis de armónicos
      - Generar onda cuadrada
      - Generar diente de sierra
      - Generar pulso de Rosenberg
      - Generar ruido
      - Cargar fichero de audio
      - Grabar audio

        - ◇ Ventana de visualización
    - eus
        - ◇ Tono purua sortu
        - ◇ Harmoniko-sintesia
        - ◇ Uhin karratua sortu
        - ◇ Zerra-hortza sortu
        - ◇ Rosenberg-en pulsua sortu
        - ◇ Zarata sortu
        - ◇ Soinu-fitxategia kargatu
        - ◇ Soinua grabatu
        - ◇ Bistaratze-leioa
- SignalVisualizer
    - for_redistribution
    - for_redistribution_files_only
    - for_testing

2.2 Ficheros *.mlapp*

MATLAB App Designer guarda las aplicaciones en desarrollo en ficheros *.mlapp*. Cada una de ellas se corresponde con un módulo. Todos ellos están guardados en el directorio *SignalVisualizerApp*. En el Cuadro 1 se muestra el listado de ficheros *.mlapp*.

[HTML]F8A102 **Nombre**	**Descripción**
[HTML]F5EEC1 control_App.mlapp	Módulo de configuración
[HTML]F5EEC1 harmonics_synthesis_App.mlapp	Módulo de suma libre de tonos puros
[HTML]F5EEC1 helpApp.mlapp	Módulo de ayuda de usuario
[HTML]F5EEC1 info_App.mlapp	Módulo que muestra créditos de la aplicación
[HTML]F5EEC1 load_App.mlapp	Módulo de carga de ficheros de audio
[HTML]F5EEC1 noise_app.mlapp	Módulo de generación de ruido
[HTML]F5EEC1 record_App.mlapp	Módulo de grabación de sonido
[HTML]F5EEC1 rosenberg_App.mlapp	Módulo de generación de pulsos de Rosenberg
[HTML]F5EEC1 sawtooth_App.mlapp	Módulo de generación de señales de diete de sierra
[HTML]F5EEC1 SignalVisualizer.mlapp	Módulo principal
[HTML]F5EEC1 square_App.mlapp	Módulo de generación de ondas cuadradas
[HTML]F5EEC1 tone_App.mlapp	Módulo de generación de tonos puros
[HTML]F5EEC1 visualization_App.mlapp	Módulo de ventana de visualización

Cuadro 1: Listado de ficheros *.mlapp*

2.3 Ficheros *.png*

Son ficheros de imagen que se utilizan como iconos en algunos botones. Están en el directorio *SignalVisualizerApp*. Se han colocado en el mismo directorio que los ficheros *.mlapp* porque si se sitúan en un subdirectorio, se debe recordar a MATLAB que incluya ese subdirectorio en el entorno de ejecución cada vez que se abre. En el Cuadro 2 se ofrece un listado de los ficheros *.png*, agrupados por funcionalidad.

2.4 Ficheros *.mat*

Son ficheros de MATLAB que contienen datos que necesita la aplicación para funcionar. Existen dos tipos: los predefinidos y los datos de usuario.

Los ficheros *.mat* predefinidos se encuentran en el directorio *SignalVisualizerApp*. Existen dos:

- **espectrograma.mat:** Contiene los datos necesarios para dibujar el espectrograma de muestra que se dibuja en *control_App.mlapp*.
- **language.mat:** Es el fichero que contiene todas las traducciones. Contiene una variable de tipo *struct* por cada módulo. Cada una de estas variables *struct* contiene un campo de tipo *cell* por cada campo del módulo que requiere traducción. La primera celda de cada campo contiene la traducción en inglés, la segunda en castellano y la tercera en euskera.

  Por ejemplo, la traducción de la etiqueta del eje frecuencial del plot del módulo *tone_App.mlapp* está en *tone.frequency*. *tone.frequency*{*1*} contiene el *string* 'Frequency (Hz)', *tone.frequency*{*2*} contiene 'Frecuencia (Hz)' y *tone.frequency*{*3*} contiene 'Maiztasuna (Hz)'.

  La función *translate()* de cada módulo carga únicamente la variable *struct* correspondiente a su módulo.

El otro tipo de fichero *.mat* es el que guarda la configuración del usuario. Este fichero se llama *user_data.mat*. Cuando la aplicación se está ejecutando dentro de *MATLAB App Designer*, se guarda en el directorio *SignalVisualizerApp*. Cuando la aplicación ha sido compilada e instalada en el equipo, se guarda en el directorio *C:\Users\{Usuario}\AppData\Roaming\SV User Data*.

Este fichero contiene la variable *user_data*, de tipo *struct*, cuyos campos son los siguientes:

- **lang_ind:** Lenguaje configurado por el usuario. 1: Inglés, 2: Castellano, 3: Euskera.

[HTML]F8A102	**Nombre**	**Descripción**
[HTML]F5EEC1	colormap_autumn.png colormap_bone.png colormap_cool.png colormap_cooper.png colormap_gray.png colormap_hot.png colormap_hsv.png colormap_jet.png colormap_parula.png colormap_pink.png colormap_spring.png colormap_summer.png colormap_winter.png	Degradados que muestran las distintas opciones de color que se pueden seleccionar para los espectrogramas. Se colocan en botones de control_App.mlapp.
[HTML]F5EEC1	down.png left.png right.png up.png	Son iconos de flecha que se colocan en los botones de visualization_App.mlapp que sirven para desplazarse por la forma de onda o el espectro.
[HTML]F5EEC1	down_2_layout.png down_3_layout.png medium_3_layout.png restore_2_layout.png restore_3_layout.png up_2_layout.png up_3_layout.png	Son iconos que se colocan en los botones de visualization_Ap que sirven para ampliar el tamaño de uno de los gráficos, o para restaurar la vista inicial.
[HTML]F5EEC1	export.png	Icono que se coloca en los botones de exportación de visualization_App.mlapp.
[HTML]F5EEC1	info.png	Icono de los botones de información, situado en varios módu para abrir la ayuda de usuario.
[HTML]F5EEC1	play_button.png stop_button.png	Icono para los botones que inician o detienen la reproducción sonora de la señal. Se encuentran en los módulos de generación de señal, en los módulos de importac y en el módulo de ventana de visualización.
[HTML]F5EEC1	playSegment_button.png	Icono del botón de visualization_App.mlapp que sirve para i la reproducción desde el cursor colocado por la persona usua
[HTML]F5EEC1	record_button.png	Icono del botón de grabación de record_App.mlapp.
[HTML]F5EEC1	zoom_center.png zoom_down.png zoom_left.png zoom_right.png zoom_up.png	Iconos de los botones de zoom de visualization_App.mlapp.

Cuadro 2: Listado de ficheros *.png*

- **w_max:** Número máximo de ventanas de visualización que se permite mantener abiertas simultáneamente.
- **plot_color:** Color elegido para dibujar la señal en los plot, en formato RGB normalizado (todos los colores están en este formato).
- **bg_color:** Color elegido para el fondo de los plot.
- **start_color:** Color elegido para el cursor de inicio de una selección (utilizado en los módulos de importación y grabación).
- **end_color:** Color elegido para el cursor de fin de una selección (utilizado en los módulos de importación y grabación).
- **selection_color:** Color que se utilizará en los módulos de importación y grabación como fondo del segmento seleccionado. En la ventana de visualización se utiliza este color para dibujar la ventana.
- **spectrogram_color:** Mapa de colores elegido para el espectrograma. Se trata de un *string* con el nombre con el que MATLAB nombra dicho mapa.

3 MATLAB App Designer

En esta sección se introducirá el funcionamiento de MATLAB App Designer, desde la creación de interfaces hasta la compilación de la aplicación. En la página web de MATLAB se ofrecen varios tutoriales y ejemplos descargables que pueden resultar de ayuda.

3.1 Creación de la interfaz

En primer lugar es necesario instalar la extensión *MATLAB App Designer*. Se puede hacer desde la página web o desde el gestor de complementos en el programa. Una vez instalado, el complemento se abre con comando *appdesigner* o bien haciendo clic sobre un fichero *.mlapp*.

Es posible crear la interfaz desde cero, aunque en ocasiones es más práctico basarse en una ya hecha, si se va a hacer un módulo similar. Si es el caso, no se debe renombrar el fichero sin más. Es mejor abrir el fichero en el que se va a basar el diseño y hacer clic en *Save as...* para guardar una copia con otro nombre. De esta forma se hacen de forma automática los cambios en el código correspondientes al nombre, que son necesarios para que se ejecute problemas.

Si se decide crear una nueva, se ofrece la posibilidad de empezar una interfaz desde cero o plantillas para crear aplicaciones con varios módulos intercomunicados. Conociendo la mecánica de intercomunicación entre módulos puede ser más práctico empezar una interfaz desde cero. Más adelante se explicará cómo se hace.

La pantalla de diseño tiene dos secciones. En *Design View* se puede modificar la interfaz, cambiando su tamaño, o colocando los elementos que se muestran en la columna izquierda sobre ella. Haciendo clic sobre cada uno de los elementos se pueden editar sus propiedades en la columna de la derecha. En el menú superior se pueden editar los detalles de la aplicación y compilarla para exportarla.

En *Code View* se muestra el código del módulo. En la columna de la izquierda se muestra una lista de los *callbacks* (funciones que se ejecutan tras producirse algún evento o al interactuar con un elemento de la interfaz), de las funciones y de las variables. Desde aquí se puede acceder a ellas para editarlas, o añadir otras nuevas. En la columna de la derecha se muestra una lista de los componentes. Se pueden seleccionar para editar sus propiedades. Al hacer clic con el botón derecho sobre el nombre de un componente, aparece la opción de crear un *callback* asociado a él.

3.2 Funcionamiento de los módulos

Al ejecutar un módulo, la primera función que se ejecuta es *createComponents()*, que es la responsable de representar la interfaz de usuario. Esta función no se puede editar manualmente, sino que la crea MATLAB de forma automática al añadir, modificar o eliminar elementos en *Design View*.

Cuando se crea la ventana, en primer lugar se llama al callback Size changed. En él se gestiona la adaptación del tamaño de la interfaz y de los elementos que la componen a la resolución de la pantalla utilizada.

A continuación se ejecuta la función *startupFcn()*. En ella se incluirán las funciones necesarias para acondicionar la interfaz para su uso, así como la inicialización de las variables que lo requieran. En todos los módulos se ha incluido dentro de esta función una llamada a la función *translate()*, para traducir la interfaz al idioma configurado.

Tras ejecutarse la función *startupFcn()*, el módulo quedará a la espera de que la persona usuaria interactúe con la interfaz o de que ocurra algún evento.

3.2.1 Propiedades y funciones

En el código del módulo existen varias secciones. Por un lado están las secciones *properties*, en las que se definen las propiedades del módulo. Por otro lado están las secciones *methods*, en las que se definen las funciones. Estas secciones pueden ser de acceso público o privado.

Las propiedades y las funciones definidas dentro de secciones de acceso público podrán ser usadas tanto desde dentro de las funciones del propio módulo como desde otros módulos. Las que estén definidas dentro de secciones de ámbito privado sólo podrán ser usadas dentro del propio módulo.

Las propiedades son variables globales. Cuando su contenido es modificado dentro de una función, será guardado para poder acceder a él desde otra función. Para utilizar una propiedad dentro del propio módulo en el que está definida, se debe llamar de la siguiente forma:

$$app.\{nombre\_de\_la\_propiedad\}$$

Inicialmente, en cada módulo existe una sección *properties* de acceso privado que contiene los elementos de la interfaz, pero no es editable. Para añadir más propiedades de acceso privado, deberá crearse una nueva sección *properties* (se crea de forma automática al crear una propiedad privada). Puede crearse una nueva propiedad en *Code view*,

utilizando sección *Properties* de la columna izquierda o el menú superior. Una vez creada la sección, se pueden escribir las propiedades en ella manualmente. Basta con escribir su nombre para declararlas. Es posible asignarles un valor para inicializarlas.

Las funciones son conjuntos de línea de código. Su diferencia respecto a los *callbacks* es que las funciones no se ejecutan debido a un evento, sino que es necesario llamarlas. En las funciones definidas en un módulo, el primer argumento siempre será el propio módulo (nombrado como *app*). Si dentro de la función no se va a utilizar ningún argumento ni función del propio módulo, en lugar de *app* puede escribirse el símbolo "∼".

Supongamos una función definida como:

$$function\ output = miFuncion(app,varA,varB)$$

Se puede llamar a la función de dos formas:

$$app.miFuncion(varA,varB) \text{ o } miFuncion(app,varA,varB)$$

Ambas son válidas y funcionan de la misma forma.

3.3 Comunicación entre módulos

En *MATLAB App Designer*, los módulos que interactúan entre sí no se encuentran al mismo nivel. Una aplicación creada con este paquete abre inicialmente un único módulo. Ese módulo debe encargarse de crear otros módulos, que serán módulos dependientes de el primero. Llamaremos *módulo principal* al módulo que invoca otro módulo, y *módulo secundario* al módulo creado.

Para hacer que un módulo pase a ser un módulo secundario, se debe abrir en *App Designer* y, en el menú superior de *Code View* se debe hacer clic en *Add Input Arguments*. Se abrirá una ventana en la que se pueden añadir varios argumentos para la función *startupFcn()*. El primer argumento siempre será el módulo principal (en nuestro código se llama siempre *mainApp*). Después de este argumento se pueden añadir otros argumentos que serán pasados por el módulo principal. **A partir de este momento, este módulo no podrá ser ejecutado por sí solo, ni siquiera para hacer pruebas. Siempre deberá ser invocado por otro módulo.**

Para poder interactuar con el módulo principal, el módulo secundario deberá tener una propiedad en la que guardar el puntero al módulo principal (en nuestro código se llama *CallingApp*). En la función *startupFcn()* del módulo secundario se debe guardar *mainApp* en la propiedad *app.CallingApp*.

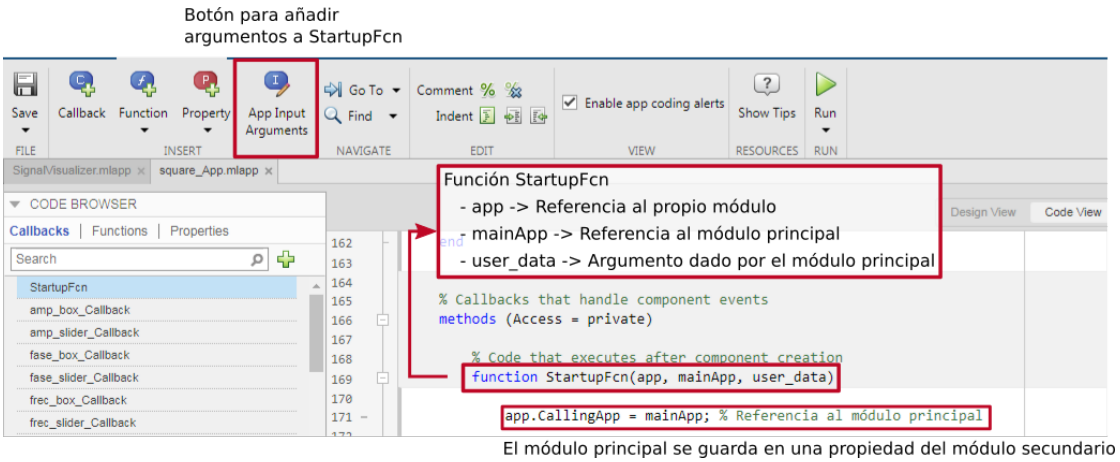

Figura 1: Ejemplo de implementación de un módulo secundario

En la figura 1 se puede ver cómo está implementado esto en el módulo *square_App.mlapp*.

Para crear un módulo secundario desde el módulo principal, en primer lugar se debe crear una propiedad en la que guardar el módulo secundario. A continuación, se debe invocar el módulo independiente de la misma forma en que se llama a una función, pasando el módulo principal como primer argumento.

A continuación vamos a ver a modo de ejemplo cómo se crea el módulo *square_App.mlapp* desde el módulo principal. Como se puede puede ver en la figura 1, la función *startupFcn()* de *square_App.mlapp* se define de la siguiente forma:

$$function\ StartupFcn(app,\ mainApp,\ user\_data)$$

Para crear el módulo *square_App.mlapp* se deberá escribir lo siguiente en el módulo principal:

$$app.DialogApp\ =\ square\_App(app,userData);$$

app.DialogApp hace referencia a una propiedad del módulo principal, en la que se guardará la aplicación secundaria. *square_App* hace referencia al módulo que se desea crear. Los argumentos, que van entre paréntesis, se corresponden con los que se han definido en la función *startupFcn()* de *square_App*. El argumento *app* hace referencia a la aplicación principal, y pasará a corresponderse con el segundo argumento de *startupFcn()* (*mainApp*). *userData* es una variable que se pasa como argumento, y se corresponderá con el tercer argumento de *startupFcn()* (*user_data*).

Una vez la aplicación principal crea la aplicación secundaria, cada una de ellas puede acceder a las funciones y propiedades públicas de la otra mediante la variable en la que se ha guardado.

3.4 Compilación

En esta sección voy a explicar cómo compilar la aplicación para crear una aplicación de escritorio independiente de la SDK de MATLAB. Existen otras opciones, como crear una aplicación de MATLAB para ejecutarla desde la SDK o una aplicación web. Estas dos últimas opciones no son relevantes para el proyecto.

Una aplicación compilada no es realmente independiente de MATLAB. No requiere que el usuario final tenga instalada la plataforma completa de MATLAB, pero debe instalar el entorno *MATLAB Runtime*, que se ofrece de forma gratuita. Se debe probar siempre la aplicación compilada, porque a veces faltan algunas funciones en *MATLAB Runtime* y habrá problemas de funcionamiento, aunque no es lo habitual.

Para realizar esta acción hay que tener instalado *MATLAB Compiler*. Para compilar la aplicación, se debe ir al módulo principal (en nuestro caso es *SignalVisualizer.mlapp*). Una vez abierto en *MATLAB App Designer*, en la *Design View*, se debe hacer clic en el botón *Share* que aparece en el menú superior y seleccionar *Standalone Desktop App*.

A continuación se abrirá *Application Compiler*, con los siguientes apartados:

- Un formulario para añadir información sobre la aplicación y cambiar el icono o la imagen que aparece al ejecutar la aplicación.
- *Additional installation folder*, donde se puede elegir la carpeta donde se instalará por defecto el programa.
- *Files required for your application to run*. Aquí se deben incluir los ficheros *.mlapp*, ficheros *.mat* e imágenes. Normalmente se incluyen de forma automática. No se deben incluir aquí los ficheros de ayuda, ya que no queremos que vayan incrustados en la aplicación.
- *Files installed for your end user*, donde se añaden ficheros que queremos que lleguen tal cual al usuario final. Son ficheros que serán colocados en el directorio *application* dentro de la carpeta de instalación.
- *Additional runtime settings*. Aquí podemos elegir que se muestre o no la consola de comandos durante la ejecución, lo cual es útil para probar el software compilado (algo que se debe hacer, porque no funciona exactamente igual). También se puede elegir crear un *log file*, aunque no resulta tan útil para saber qué falla en cada momento.

En el menú superior, en *Packaging options*, se puede seleccionar si se desea incluir *MATLAB Runtime* en el paquete de instalación o si se va a hacer un instalador sin él. En el primer caso se obtiene un instalador muy grande (aproximadamente 1 GB). En el segundo, el instalador es mucho más ligero y ofrece la opción de descargar *MATLAB Runtime* durante la instalación.

Finalmente hay que hacer clic en *Package* para empezar a compilar. En la carpeta donde están los ficheros del programa aparecerá un directorio con el nombre de la aplicación, y tres subdirectorios. El instalador final se guarda en el subdirectorio *for_redistribution*.

En ocasiones, la compilación fracasa porque, según el fichero *log*, no se ha podido abir el fichero *for_testing/SignalVisualizer.exe*. No sé a qué se debe este error, pero se suele solucionar compilando dos veces. Si no, también funciona eliminar el fichero *for_testing/SignalVisualizer.exe* y compilar una o dos veces (porque en la primera salta el error de nuevo).

4 Funcionamiento de la aplicación

En esta sección se explicará el funcionamiento de la aplicación, partiendo desde el momento en que se ejecuta.

4.1 Ejecución del módulo principal

El primer módulo que se ejecuta al abrir el programa es *SignalVisualizer.mlapp*. Como ya se ha dicho, la primera función en ser ejecutada tras la creación de la interfaz es StartupFcn. Se comprueba si existen datos de usuario previamente guardados y, en su defecto, crea unos nuevos con los valores por defecto.

Las ventanas de visualización que se creen se deben guardar en *propiedades* del módulo (consultar la subsección *Comunicación entre módulos*). Hay creadas 14 *propiedades PlotApp* por defecto, pero no se utilizarán todas, ya que el número máximo de ventanas de visualización que se pueden abrir simultáneamente está limitado en la configuración de usuario. Para referenciar estas *propiedades* se crea un *cell array* con tantos *string* como ventanas de visualización se pueden abrir al mismo tiempo, y se harán visibles los botones que se corresponden con ellas.

4.2 Llamada a un módulo secundario

Los módulos de generación y de importación se guardan en la *propiedad DialogApp*. Sólo hay una, ya que sólo se puede abrir una de estos módulos al mismo tiempo. Al seleccionar una opción en los menús de la parte superior, se ejecuta un *Menu callback*. En él, se deshabilitan todos los menús de la parte superior, para que la persona usuaria no pueda abrir más módulos. A continuación, se crea el módulo secundario que corresponda, al que se le pasa la configuración de usuario.

4.3 Cerrar el módulo secundario

Si, tras abrir un módulo de generación o de importación, la persona usuaria decide retroceder y no abrir la ventana de visualización, presionará el botón de cerrado de ventana (arriba a la derecha). Esto invocará la Close request. Antes de cerrar el módulo, se reactivarán los menús de la parte superior del módulo principal, para que sea posible abrir un nuevo módulo secundario.

4.4 Llamada a la ventana de visualización

Tras abrir un módulo de generación o de importación, para generar o cargar la señal en la ventana de visualización, se llama a la función *createPlot()*, del módulo principal. En esta función se buscará una *propiedad PlotApp* disponible para guardar la nueva ventana de visualización. Si no es posible abrir alguna más (en función del número máximo configurado), devolverá un mensaje de error. Si es posible, creará una nueva ventana de visualización, reactivará los menús de la parte superior del módulo principal y, finalmente, cerrará el módulo secundario. En la interfaz de la aplicación principal se activará el botón correspondiente al *PlotApp* que ha sido utilizado, y se le asignará un nombre que indica qué señal se está analizando en esa ventana.

4.5 Cerrar ventana de visualización

En el módulo principal, se desactiva el botón correspondiente a la ventana de visualización cerrada y se borra el texto descriptivo. Finalmente, se borra el módulo.

4.6 Cerrar aplicación

Al cerrar el módulo principal, debe cerrarse cada uno de los módulos que siga abierto. Por ello, se cierran todas las ventanas de visualización que estén abiertas, la ventana de ayuda y el cuadro de diálogo que esté abierto (el cuadro de diálogo se refiere a cualquier módulo de generación o importación, o al módulo de configuración). Finalmente, se cierra el módulo principal.

En alguna ocasión ha fallado el cierre de los módulos secundarios y no se ha podido rastrear el motivo del problema. En esos casos, la aplicación principal se cierra y deja huérfanos a los módulos secundarios. En las funciones de cierre de los módulos secundarios se previene esta excepcionalidad, de forma que puedan cerrarse sin depender del módulo principal cuando este no esté disponible.

5 Funciones comunes

En este apartado se explicarán las funciones que se encuentran en varios módulos.

5.1 infoButtonPushed(app,event)

Llama a la función *createHelp()* de la aplicación principal, para abrir una nueva ventana de ayuda o, si está ya abierta, mostrar la ayuda correspondiente al módulo actual.

5.2 generar_btm_callback(app,event)

Callback del botón generar. Llama a la función *createPlot()* del módulo principal para intentar crear una nueva ventana de visualización.

Si el número de ventanas de visualización ya abiertas es igual al máximo establecido en la configuración del usuario, muestra una advertencia y no continua. Si puede crear una nueva ventana, reactiva los menús de la aplicación principal (desactivados al abrir el módulo) y se cierra.

5.3 playButtonPushed(app, event)

Crea un *audioplayer* para escuchar la señal actual. Antes de crear el *audioplayer* elimina el offset de la señal (en los módulos en los que proceda) y normaliza la señal si esta supera la amplitud 1.

5.4 resizeelement(app,element)

Esta función redimensiona un elemento en función del nuevo tamaño de la ventana.

En esta función se desactivan los warnings. Esto se debe a que los elementos se redimensionan a lo ancho y a lo alto. Al intentar redimensionar un slider a lo ancho aparece un warning, aunque no es importante. En nuestro código se ha considerado más eficaz suprimir los warnings que comprobar si cada elemento es un slider.

Al redimensionar el tamaño de fuente, solo se tiene en cuenta el cambio de altura.

5.5 resizeWindow()

Esta función se ejecuta al abrir un módulo. Detecta el tamaño de pantalla y redimensiona la ventana en función de él.[1]

5.6 stopButtonPushed(app, event)

Detiene la reproducción del *audioplayer*, si es que está en curso.

5.7 translate(app,language)

Traduce los elementos del módulo al idioma en que está configurado el módulo principal.

Carga el elemento *struct* correspondiente al módulo, del fichero *language.mat*. Este elemento contiene varios elementos *cell* con tres campos, uno por cada idioma:

1. Inglés
2. Castellano
3. Euskera

La variable *language* contiene un número entre 1 y 3 con el mismo significado.

Esta función nunca ha fallado. No obstante, los warnings están predefinidos en las variables del módulo, por si llegase a fallar.

[1] Nota importante: En esta función (y en otro lugar del módulo) hay una variable estática con el tamaño original de la ventana. Si se modifica, hay que cambiar las variables manualmente.

6 Callbacks y funciones especiales

En esta sección se explicarán las funciones de *Callback* (las que se ejecutan al interactuar con algún elemento del módulo) y algunas funciones particulares, relacionadas con el funcionamiento de los módulos en MATLAB.

6.1 Box callback

Callback de las cajas de texto. Se debe recoger el valor contenido en ella, convertirlo a un valor numérico y comprobar:

a) Que el valor introducido es, efectivamente, un número.

b) Que el valor introducido se encuentra entre los valores acotados.

(En el apartado de cada módulo en este manual se especifican los límites de cada variable)

Una vez hecho, se debe actualizar el valor del slider correspondiente (si tiene) para que concuerden.

6.2 Button callback

Callback de los botones. Se activa al pulsar el botón.

6.3 Close request

Callback que se activa al pulsar el botón de cierre de la ventana. En esta función se deben incluir las funciones que deben ejecutarse antes de que se cierre.

6.4 Menu callback

Se ejecuta al hacer clic sobre uno de los menús o submenús desplegables que se encuentran en la parte superior de la ventana.

6.5 Slider callback

Callback de los slider. Ya que los slider sirven para marcar valores numéricos continuos, conviene redondear el resultado. También se debe actualizar la caja de texto correspondiente al slider (si tiene), para que los valores concuerden.

6.6 Size changed

Callback que se activa cuando se abre al abrir el módulo, y al cambiar el tamaño de la ventana.

Cuando se ejecuta al abrir el módulo, se llama a la función *resizeWindow()*, para adaptar la ventana (no maximizada) al tamaño de la pantalla. Esta función sólo se ejecuta la primera vez que se entra en este callback. A continuación, con un ciclo *for*, se pasan todos los elementos del módulo por la función *resizeelement()*. [2] Finalmente se guarda el nuevo tamaño de ventana, para utilizarlo como referencia en caso de que se vuelva a redimensionar la ventana.

6.7 StartupFcn

Es la primera función que se ejecuta tras la creación del módulo y la adaptación del tamaño de ventana (ver Size changed). La función en la que MATLAB crea el módulo no es editable, pero *StartupFcn* sí. Aquí se debe hacer la configuración del módulo.

Para los módulos dependientes del módulo principal (todas menos el módulo principal), se pasa como argumento *mainApp*. Se guarda como variable de la función, para poder interactuar con él. A través de la variable guardada (en *CallingApp*) se puede acceder a sus funciones y a sus variables públicas, y también a los elementos de su interfaz.

A los módulos dependientes también se les pasa la variable *user_data*, de tipo *struct*, que contiene la configuración guardada por el usuario. Contiene el idioma de la aplicación, color del plot y del espectrograma.

En la función *StartupFcn* se definen los valores por defecto de las variables, y el valor de otras variables globales. Aunque esto también puede hacerse en la sección de definición de variables del módulo.

[2] Nota importante: Los elementos que están dentro de paneles no entran en el ciclo *for* general. Se debe hacer uno específico para cada panel que haya en el módulo.

7 Funciones y variables específicas

7.1 Módulo principal

Este es el módulo correspondiente a *SignalVisualizer.mlapp*. Es el primer módulo que se ejecuta al abrir el programa, y es el que se encarga de llamar a los demás módulos.

7.1.1 Variables

DialogApp, HelpApp y PlotAppN

Son variables destinadas a guardar módulos creados, para después poder acceder a ellos. *DialogApp* guarda los módulos de generación o importación que se abran, o el módulo de configuración. Sólo existe una variable de este tipo porque **sólo puede haber una ventana de este tipo abierta**, y no se podrá abrir una nueva hasta que se cierre la ventana actual.

HelpApp guarda la ventana de ayuda.

PlotAppN (*PlotApp1*, *PlotApp2*, ... *PlotApp14*) guardan las ventanas de visualización que se abran. Se utilizarán tantos *PlotAppN* como el número de ventanas de visulización que se permita abrir de forma simultánea. Es decir, si se permite abrir cuatro ventanas de visualización simultáneamente, se utilizarán de *PlotApp1* a *PlotApp4*.

plotAppArray

Es un array (en realidad es una variable de tipo *cell*) que contiene las variables *PlotAppN* que se pueden utilizar (ver DialogApp, HelpApp y PlotAppN). El único propósito que tiene es facilitar la creación de bucles para comprobar los diferentes *PlotAppN* rápidamente.

plotAppCounter

Esta variable sirve para llevar la cuenta de las ventanas de visualización abiertas. Se incrementa al crear una nueva ventana de visualización. Es una variable pública, de forma que las propias ventanas de visualización se encargan de decrementarla cuando la persona usuaria la cierra.

userData

Es una variable de tipo *struct* que guarda la configuración de usuario. La configuración de usuario se guarda en un fichero, de forma que es posible cargarla cada vez que se abre la aplicación. Si no el fichero no existe o no se encuentra en la localización esperada, se cargarán en *userData* valores definidos por defecto.

Sus campos son los siguientes:

- **lang_ind:** Lenguaje configurado por el usuario. 1: Inglés, 2: Castellano, 3: Euskera.
- **w_max:** Número máximo de ventanas de visualización que se permite mantener abiertas simultáneamente.
- **plot_color:** Color elegido para dibujar la señal en los plot, en formato RGB normalizado (todos los colores están en este formato).
- **bg_color:** Color elegido para el fondo de los plot.
- **start_color:** Color elegido para el cursor de inicio de una selección (utilizado en los módulos de importación y grabación).
- **end_color:** Color elegido para el cursor de fin de una selección (utilizado en los módulos de importación y grabación).
- **selection_color:** Color que se utilizará en los módulos de importación y grabación como fondo del segmento seleccionado. En la ventana de visualización se utiliza este color para dibujar la ventana.
- **spectrogram_color:** Mapa de colores elegido para el espectrograma. Se trata de un *string* con el nombre con el que MATLAB nombra dicho mapa.

7.1.2 Funciones

createHelp()

Esta función abre la ventana de ayuda. Si ya ha sido abierta, actualiza su contenido para que muestre la ayuda del módulo que ha llamado a esta función (ver update_module()). Es una función pública, es utilizada por los módulos de generación, importación y la ventana de visualización cuando la persona usuaria pulsa el botón de información.

El argumento que recibe, **module**, indica qué módulo ha solicitado la ventana de ayuda. Es un número que responde al siguiente código:

1. Generar tono puro
2. Suma libre de tonos puros
3. Generar onda cuadrada
4. Generar diente de sierra
5. Generar pulso de Rosenberg
6. Generar ruido
7. Cargar fichero de audio
8. Grabar sonido
9. Ventana de visualización

createPlot()

Esta función se encarga de crear una nueva ventana de visualización y de habilitar un botón asociado a ella en el módulo principal. Es una función pública, por lo que puede ser utilizada por otros módulos diferentes al módulo principal. Es llamada por los módulos de generación y de importación cuando la persona usuaria presiona el botón *Generar*.

Está definida de la siguiente forma:

$success$ = createPlot(*app,signal,t,name,varargin*)

Los argumentos son los siguientes:

- **signal:** Array con la señal que se desea cargar en la ventana de visualización.
- **time:** Array de tiempos que acompaña a **signal**.
- **name:** String que contiene el nombre que se le dará a la ventana de visualización. Es un nombre descriptivo (p.e. "Tono puro.º "Diente de sierra") que se mostrará en el título de la ventana de visualización y en el botón del módulo principal que se asocie a ella.
- **varargin:** Se utiliza para notificar casos especiales (). Una sola variable o un grupo de ellas. Su uso no es obligatorio (ver documentación de MATLAB).

La mayor parte de los argumentos se pasan directamente a la ventana de visualización creada (ver startupFcn()), sin darles uso en esta función. El único que se utiliza es *name*, para nombrar el botón asociado a la ventana de visualización creada.

El valor que devuelve la función, **success**, se utiliza para indicar al módulo que ha llamado a la función si la ventana de visualización se ha creado correctamente o, en cambio, si no es posible crear una nueva porque se ha alcanzado el límite.

updateN()

Existen tres funciones update(): update1(), update2() y update3(). Son funciones públicas que son llamadas por el módulo de configuración cuando la persona usuaria presiona el botón de *Guardar cambios*. Estas funciones aplican los cambios realizados y los guardan en el fichero de configuración de usuario, de forma que puedan cargarse cuando la aplicación se utilice en el futuro.

- **update1():** Actualiza el número máximo de ventanas que se pueden abrir de forma simultánea.
- **update2():** Actualiza el color de los gráficos, lo cual incluye:
  - Color del trazo
  - Color del fondo
  - Color de inicio de selección
  - Color de fin de selección
  - Color de segmento seleccionado o enventanado
- **update3():** Actualiza el mapa de colores utilizado en los espectrogramas.

7.2 Generar tono puro

Módulo para generar tonos puros. La señal utilizada para crear tonos puros es el coseno. Es el módulo correspondiente a *tone_App.mlapp*.

7.2.1 Variables

- **Amplitud:** Amplitud del coseno, limitada entre 0 y 1.
- **Frecuencia:** Frecuencia del coseno. Sólo se permiten números enteros (para que sea más fácil de comprender para el alumnado), pero podría cambiarse. Ya que es una aplicación destinada a audio, permite una frec. máxima de 20 kHz, aunque podría subirse, ya que la *fs* utilizada es 48 kHz.

- **Fase:** Fase del coseno. Limitada entre -1 y 1 ya que, al calcular el coseno, va multiplicada por π.
- **Duración:** Duración de la señal creada. Se define en segundos, permitiendo sólo números enteros entre 1 y 30 segundos.
- **Offset:** Offset de la señal. Limitado entre -2 y 2.

7.3 Suma libre de tonos puros

Este módulo sirve para crear hasta seis tonos puros de amplitud y frecuencia independientes. Con el teclado se puede seleccionar una nota y calcular sus seis primeros armónicos. Este módulo corresponde al fichero *harmonics_synthesis_App.mlapp*.

7.3.1 Variables

- **Amplitud:** La amplitud se define para cada uno de los seis tonos puros disponibles de forma individual. Está acotada entre 0 y 1.
- **Frecuencia:** La frecuencia se define para cada uno de los seis tonos puros disponibles de forma individual. Está acotada entre 0 y 20 kHz, ya que se trata de una aplicación orientada al sonido. También sería posible subir el límite, ya que se utiliza una *fs* de 48 kHz.
- **Duración:** Se puede definir la duración de la señal entre 1 y 30 segundos. Sólo admite números enteros, pero se podría cambiar.
- **Octava:** Define la octava en la que se encuentra la nota. La octava y la nota se definen en notación anglosajona. Si tienes dudas, consulta *aquí*.

7.3.2 Funciones

```
notes_harmonics(app,note)
```

A partir de la nota seleccionada en el teclado y de la octava especificada, calcula la frecuencia fundamental de la nota dentro de una escala cromática en un sistema temperado (todos los semitonos son iguales) afinado en La 440. Después calcula los seis primeros armónicos, y asigna uno a cada uno de los seis tonos puros configurables.

Las amplitudes que se le asigna a cada armónico están definidas de forma descendente, de forma que el tono de frecuencia fundamental tenga amplitud 1, el segundo armónico

5/6, el siguiente 4/6... La única razón para ello es que el alumnado pueda observar de forma más clara la frecuencia fundamental en la representación, y que vea los armónicos diferenciados. Sólo por razones estéticas, podría cambiarse.

7.4 Generar onda cuadrada

Este módulo permite crear una onda cuadrada. Utiliza la función *square(t, duty)* para crear la onda cuadrada. Este módulo se corresponde con el fichero *square_App.mlapp*.

7.4.1 Variables

- **Amplitud:** Amplitud de la señal cuadrada, acotada entre 0 y 1.
- **Frecuencia:** Frecuencia de la señal, seleccionable entre 0 y 20 kHz. Se ha seleccionado ese límite porque es una aplicación orientada al sonido, pero podría ampliarse el límite, ya que se usa una *fs* de 48 kHz.
- **Fase:** Fase de la señal cuadrada. Puede tomar valores entre -1 y 1 porque, al calcular la señal cuadrada, se multiplica por π.
- **Ciclo activo:** Porcentaje del ciclo en que la eñal se encuentra en el nivel alto. Puede tomar valores enteros entre el 10 % y 90 %.
- **Duración:** Duración de la señal. Se pueden elegir valores enteros entre 1 y 30 segundos.
- **Offset:** Offset de la señal. Puede tomar valores entre -2 y 2.

7.5 Generar señal de diente de sierra

Este módulo sirve para crear una señal de diente de sierra. Utiliza la función *sawtooth(t,xmax)* para crear el diente de sierra. Este módulo se corresponde con el fichero *sawtooth_App.mlapp*.

7.5.1 Variables

- **Amplitud:** Amplitud de la señal de diente de sierra, acotada entre 0 y 1.
- **Frecuencia:** Frecuencia de la señal, seleccionable entre 0 y 20 kHz. Se ha seleccionado ese límite porque es una aplicación orientada al sonido, pero podría ampliarse el límite, ya que se usa una *fs* de 48 kHz.

- **Fase:** Fase de la señal. Puede tomar valores entre -1 y 1 porque, al calcular la señal, se multiplica por π.
- **Posición del máximo:** Posición que toma el máximo de la señal dentro del ciclo. Puede tomar valores entre 0 y 1, siendo 0 el comienzo del ciclo y 1 el final.
- **Duración:** Duración de la señal. Se pueden elegir valores enteros entre 1 y 30 segundos.
- **Offset:** Offset de la señal. Puede tomar valores entre -2 y 2.

7.6 Generar pulso de Rosenberg

Este módulo sirve para generar un tren de pulsos de Rosenberg. Para crear la señal, utiliza la función rosenberg(). Este módulo se corresponde con el fichero *rosenberg_App.mlapp*.

7.6.1 Variables

- **Amplitud:** Amplitud de la señal de diente de sierra, acotada entre 0 y 1.
- **Frecuencia:** Frecuencia de la señal, seleccionable entre 0 y 4.8 kHz. Se ha seleccionado ese límite por motivos del algoritmo que genera los pulsos de Rosenberg. El porcentaje mínimo de tiempo de subida o bajada que se puede elegir es el 10 %. Teniendo en cuenta que la frecuencia de muestreo utilizada es 48 kHz, el 10 % de un periodo de una señal de 4.8 kHz se corresponde con una sola muestra. Para señales de mayor frecuencia, al 10 % de un ciclo le correspondería menos de una muestra, con lo que no sería posible generar la señal correctamente.
- **Fase:** Fase de la señal. Puede tomar valores entre -1 y 1 porque, al calcular la señal, se multiplica por π.
- **Tiempo de subida:** Porcentaje del ciclo en que el pulso asciende desde 0 hasta el máximo. Puede variar entre el 10 % y 90 %.
- **Tiempo de bajada:** Porcentaje del ciclo en que el pulso desciende desde el máximo hasta 0. Puede variar entre el 10 % y el 90 %. El tiempo de subida y de bajada nunca pueden sumar más del 100 %.
- **Duración:** Duración de la señal. Se pueden elegir valores enteros entre 1 y 30 segundos.

7.6.2 Funciones

rosenberg() **Esta función ha sido creada por Matt Montag.** La función original sirve para crear un ciclo de un pulso de Rosenberg. La función ha sido modificada por Eder del Blanco para generar un tren de pulsos de Rosenberg.

Está definida de la siguiente forma:

signal = rosenberg(,*f0*,*t*,*N1*,*N2*,*fs*)

Sus parámetros son los siguientes:

- **f0:** Frecuencia fundamental del tren de pulsos de Rosenberg.
- **t:** Duración de la señal de salida en segundos.
- **N1:** Porcentaje del periodo dedicado al ciclo de subida.
- **N2:** Porcentaje del periodo dedicado al ciclo de bajada.
- **fs:** La frecuencia de muestreo utilizada.

El argumento devuelto, **signal**, es la señal generada por el algoritmo.

7.7 Generar ruido

Este módulo sirve para crear una muestra de ruido. Permite elegir entre tres tipos diferentes de ruido. La función que se utiliza para crear ruido blanco es *randn()*, la función que se utiliza para crear ruido rosa es *pinknoise()* y la que se utiliza para crear ruido marrón es **??**. Este módulo se corresponde con el fichero *noise_app.mlapp*.

7.7.1 Variables

- **Amplitud máxima:** Valor máximo que alcanza la señal de ruido.
- **Tipo de ruido:** Puede seleccionarse ruido blanco, ruido rosa o ruido marrón.
- **Duración:** Duración de la señal. Se pueden elegir valores enteros entre 1 y 30 segundos.

7.7.2 Funciones

rednoise() **Esta función ha sido creada por Hristo Zhivomirov.** Sirve para crear un array de dimensión MxN de ruido marrón (o rojo). En esta aplicación está implementada de modo que sólo se le puede pasar la dimensión N. A M se le asigna el valor 1 para obtener un array.

7.8 Importar fichero de audio

Este módulo sirve para importar ficheros de audio en la ventana de visualización. También es posible seleccionar un segmento e importarlo. Este módulo se corresponde con el fichero *load_App.mlapp*.

7.8.1 Variables

audioPosition

Lugar del audio donde se ha colocado el cursor de posición, y a partir de la cual se reproducirá la señal al pulsar el botón *Play*. Expresado en segundos, se corresponde con un valor del array de tiempos.

endPosition

Marca el lugar donde se ha establecido la posición de fin de selección. Expresado en segundos, se corresponde con un valor del array de tiempos.

selectionState

Indica el estado del proceso de selección (ver nuevaSelección()).

startPosition

Marca el lugar donde se ha establecido la posición de inicio de selección. Expresado en segundos, se corresponde con un valor del array de tiempos.

7.8.2 Funciones

corregirSeleccion()
Esta función intercambia la posición de inicio de la selección con la posición de fin.

dibujarInicioSeleccion()
Tras elegir la posición del inicio de selección, esta función dibuja el cursor de inicio sobre la señal.

dibujarSeleccion()
Cuando se ha realizado una nueva selección, esta función la representa en el plot. En primer lugar dibuja el fondo del segmento seleccionado. A continuación superpone la señal y, finalmente, dibuja los cursores de inicio y fin de posición.

loadFile()
Esta función carga el fichero de audio en el módulo. En primer lugar convierte el audio a mono promediando sus dos canales, en el caso de que se trate de un audio estéreo. A continuación, si la amplitud máxima del audio supera la unidad, lo normaliza. Finalmente, carga el audio en el reproductor de audio y representa la señal cargada.

nuevaSelección()
Esta función realiza los cambios necesarios para pasar por los distintos pasos del proceso de seleccionar un segmento de audio. Las pasos son las siguientes:

- **Paso 1 (estado 0):** Comienza el proceso de selección. Se desactivan todos los controles de la interfaz para evitar que se realice cualquier otra acción. Se prepara para que la persona usuaria indique dónde desea que comience la selección.
- **Paso 2 (estado 1):** Ya se ha seleccionado la posición de inicio, así que se desactivan los controles para marcar el inicio de la selección y se activan los que sirven para marcar el final.
- **Paso 3 (estado 2):** Se ha seleccionado la posición de fin de selección. Se desactivan los controles para seleccionar el fin de selección y se reactivan los controles de la interfaz que se anularon en el paso 1. Se vuelve al estado 0. También se comprueban posibles casos en la selección realizada:
  - **El inicio y el fin de la selección coinciden:** En este caso se anula la selección.
  - **La posición de fin es anterior a la de comienzo:** En este caso se intercambian ambas posiciones (función corregirSeleccion()).

positionLine()
Al hacer clic sobre la señal en el momento en que no hay ningún segmento seleccionado, esta función dibuja un cursor en la posición seleccionada. A continuación, carga en el reproductor de audio la parte de la señal posterior al cursor. De esta forma, la persona usuaria puede reproducir el audio a partir del cursor que ha colocado.

selectStart()
Al seleccionar la posición de inicio de la selección, se busca en el eje de tiempos el valor más cercano a la posición seleccionada. Muestra el valor elegido en la interfaz de usuario, dibuja el cursor de inicio (dibujarSeleccion()) y pasa al siguiente paso de la selección (nuevaSelección()).

selectStart()
Al seleccionar la posición de fin de la selección, se busca en el eje de tiempos el valor más cercano a la posición seleccionada. Muestra el valor elegido en la interfaz de usuario y finaliza el proceso de la selección (nuevaSelección()).

showInfo()
Esta función comprueba si el audio que se quiere cargar cumple con las restricciones (duración máxima de 5 minutos y frecuencia de muestreo (fs) mínima de 100 Hz). Si no cumple con ellas, se muestra una advertencia y se cancela la carga. En caso contrario, esta función muestra en la interfaz la duración del fichero y su fs.

7.9 Grabar audio

Este módulo permite grabar audio y cargarlo en la ventana de visualización. También permite seleccionar sólo un segmento de él para cargarlo. Este módulo se corersponde con el fichero *record_App.mlapp*.

7.9.1 Variables

Las variable específicas reseñables de este módulo son las mismas que las del módulo Importar fichero de audio.

7.9.2 Funciones

Las funciones específicas de este módulo son las mismas que las del módulo Importar fichero de audio, a excepción de las funciones loadFile() y showInfo().

7.10 Ventana de visualización

Es el módulo donde se cargan las señales de audio para ser analizadas. Permite calcular y representar la FT, STFT y el espectro de la señal de audio cargada, ampliar los gráficos y navegar por ellos, y exportar los datos de señal y espectro en ficheros de texto para analizarlos en una hoja de cálculo. También es posible exportar la forma de onda en un fichero de sonido *.wav*. Se corresponde con el fichero *visualization_App.mlapp*.

7.10.1 Zoom

El funcionamiento del zoom es un tema que merece la pena tratar en la documentación, ya que la implementación puede resultar difícil de entender, el código no resulta del todo claro y los comentarios en el código no permiten explayarse.

El zoom se aplica modificando especificando los límites mostrados en los gráficos. Cada eje tiene un factor de zoom asignado (ver zoomxFactorN,zoomyFactorN, donde se explica su funcionamiento). El uso del zoom implica ciertos aspectos que hace que su uso sea complejo.

El caso más sencillo es el zoom que se realiza en los ejes que representan amplitudes. En la función updatePlot() se establece el rango inicial que se mostrará en el eje de amplitud de la forma de onda y en el del espectro. Cuando se muestra este rango se considera que el zoom está al mínimo, y que el factor de zoom de ese eje es 1, es decir, que se muestra el 100 % del rango establecido. Al hacer zoom en el eje de amplitud, el rango mostrado disminuirá en un 10 %, por lo que el factor de zoom de ese eje disminuirá en 0.1. La primera vez que se haga zoom pasará a mostrarse el 90 % del rango inicial, la segunda se mostrará el 80 %, y así sucesivamente hasta llegar al 10 %. Se ha considerado que es un grado de ampliación suficiente para analizar en detalle la señal o el espectro, por lo que, al llegar a ese punto, se desactiva el botón de ampliar el zoom vertical. Al disminuir el zoom sucede a la inversa. Si se muestra el 10 % del rango total y se disminuye el zoom, pasará a mostrarse el 20 %, y la siguiente vez el 30 %. Al llegar al 100 %, la posición inicial, el botón para disminuir el zoom vertical se desactivará.

El caso del eje temporal es un poco más complejo. Funciona de la misma forma que el zoom de amplitud pero, al llegar al 10 % del rango inicial, no se detiene. Permite seguir haciendo zoom, pero se disminuye en pasos del 1 %. Cuando llega a mostrar el 1 % del

rango inicial, el botón de zoom horizontal se desactiva. Al disminuir el zoom funciona de la misma forma: aumenta en pasos del 1 % hasta llegar al 10 % del rango inicial, y entonces aumenta en pasos del 10 % hasta llegar al 100 %.

Al comprobar el factor de zoom para saber cuándo activar o desactivar los botones de zoom, se hace una comparación entre *float*. Por ejemplo, estamos viendo el 90 % del rango total de tiempo y queremos que, tras disminuir el zoom y ver el 100 % de la señal, se desactive el botón que permite disminuir el zoom horizontal. Poner la condición "zoomfactor ¿0.9" da problemas, debido a que son números *float*. Por eso se ha puesto "zoomfactor ¿0.91", para salvar los decimales.

Finalmente, el caso del eje frecuencial es el más complejo. El rango frecuencial total se considera que es el que va de 0 a fs/2. En la función updatePlot() se calcula un rango inicial para mostrar las frecuencias relevantes (ver la sección correspondiente a la función, para más detalle). A continuación, se calcula qué porcentaje representa este rango sobre el rango total. Lo más probable es que este porcentaje no sea un número predecible. Aún así, el mecanismo del zoom funciona igual que al ampliar el eje de tiempos: en el tramo en que se muestra del 100 % al 10 % del rango frecuencial total, el zoom se amplía o disminuye en pasos del 10 %, y en el tramo del 10 % al 1 %, en tramos del 1 %.

Al manejar el zoom en el eje de tiempos, contábamos con la ventaja de que sabíamos que el factor de zoom iba a tomar valores conocidos (0.01,0.02, ... 0.09,0.1, 0.2, ... 0.9, 1). En el caso de la frecuencia no contamos con esta seguridad, ya que el factor de zoom inicial variará en función de las componentes frecuenciales del espectro. Por este motivo nos encontraremos ccon algunas incongruencias. Por ejemplo, si el rango inicial mostrado representa un 82.3 % del rango total y disminuimos el zoom, se mostrará un 92.3 % del rango total. Debido a los límites impuestos no se permitirá disminuir más el zoom, ya que el siguiente paso sería mostrar el 102.3 %, lo que se saldría de los límites. Si, para solucionarlo, redondeamos al 100 % y mostramos todo el rango frecuencial, no podremos recuperar la vista inicial a no ser que restauremos el zoom.

El mecanismo de implementación puede ser un tema a reconsiderar. Actualmente funciona bastante bien, pero tiene algunas limitaciones, como las mencionadas. También se debería replantear el sistema de zoom inicial, y cambiarlo por otro más simple. Implementarlo de la forma en que se trata el zoom del espectro en el programa Speech Analyzer (mostrar una cuarta parte del espectro, un tercio, la mitad o espectro completo) podría ser una opción a considerar.

7.10.2 Variables

center,center2

Estas variables marcan el punto donde se ha hecho clic. **center** guarda las coordenadas al hacer clic sobre el plot UIAxes y **center2** guarda las coordenadas al hacer clic sobre UIAxes2. Estas coordenadas son utilizadas al hacer zoom, para centrarlo sobre ese punto. También se utiliza al alternar entre distintos modos, para mantener la posición del cursor, o centro del segmento enventanado (en función del modo).

mode

Sirve para determinar el modo en el que está la ventana en cada momento. Los modos son los siguientes:

1. Modo Transformada de Fourier (FT)
2. Modo Transformada de Fourier a Corto Plazo (STFT)
3. Modo Espectrograma
4. Modo STFT + Espectrograma

state

Esta variable se utiliza cuando se calcula la FT, STFT o espectrograma. Para establecer el estado del zoom, es conveniente saber el contexto en que está la aplicación en ese momento.

La variable **state** puede tomar tres valores:

0. Indica que aún no se han inicializado los límites (ver xLimN,yLinN). Este estado sólo debería darse al abrir la ventana de visualización. Tras ejecutarse la función updatePlot(), el estado pasa a 1.
1. Indica que se encuentra o viene de un modo en que se representa un espectro en UIAxes2. El eje X representa el eje de frecuencias, y el eje Y representa la amplitud del espectro.
2. Indica que se encuentra o viene de un modo en que se representa un espectrograma en UIAxes2. El eje X representa el eje temporal, y el eje Y representa el eje de frecuencias.

xLimN,yLinN

Estas variables son **yLim1**, en referencia al eje Y de UIAxes; y **xLim2** e **yLim2**, en referencia a UIAxes2. Al iniciar la ventana de aplicación, la función updatePlot() calcula la FT de la señal cargada y calcula los límites de los ejes X e Y de cada plot, de forma que sea fácil observar la información de interés. Estos serán los límites "por defecto" de los gráficos. Se permite acercar o alejar el zoom sobre esos límites. Al resetear el zoom, se volverá a esa vista "por defecto".

zoomxFactorN,zoomyFactorN

Estas variables son **zoomxFactor1** y **zoomyFactor1** (en referencia a UIAxes), y **zoomxFactor2** y **zoomyFactor2** (en referencia a UIAxes2). Sirven para guardar el factor de zoom que se está aplicando en los ejes de estos dos plot.

El factor de zoom indica el porcentaje que representa el rango que se está mostrando con respecto al rango total. Por ejemplo, si se ha cargado una señal de un segundo de duración y se está mostrando la forma de onda entre 0.5 y 0.75 segundos, se están visualizando 0.25 segundos de señal. Con respecto a la duración total, esto representa un 25 %. Por ello, el factor de zoom en el eje X de UIAxes (eje temporal del gráfico de forma de onda) es 0.25. El mismo concepto se aplica en el eje de frecuencias y el de amplitud.

El factor de zoom disminuye al ampliar el zoom, ya que implica mostrar una región más pequeña, y sucede a la inversa al encoger el zoom.

7.10.3 Funciones

appDataTip()

Esta función es llamada cuando se hace clic sobre el gráfico de forma de onda en el modo FT o en el modo espectrograma. Al hacer clic, se marca un punto con un cursor y se escriben en pantalla sus valores X e Y.

Esta función se encarga de dibujar el cursor sobre la señal y de escribir los valores sobre el gráfico.

addDataTipSpectrum()

Esta función es llamada cuando se hace clic sobre el gráfico del espectro (no sirve para el espectrograma). Al hacer clic, se marca un punto con un cursor y se escriben en pantalla sus valores X e Y.

Esta función se encarga de dibujar el cursor sobre la señal y de escribir los valores sobre el gráfico.

calculateFT()
Esta función se encarga de calcular la FT de la señal. A diferencia de updatePlot(), esta función no se ejecuta al inicio, sino cuando se selecciona el modo FT tras haber visitado otros modos, y no se encarga de calcular los límites.

calculateSTFT()
Esta función se encarga de calcular la STFT del segmento de señal seleccionado, y de representarla en el gráfico inferior.

calculateSpectrogram()
Esta función se encarga de calcular el espectrograma de la señal y de representarlo en UIAxes2 o UIAxes3, en función de lo que corresponda.

layout2()

Esta función se encarga de pasar de la vista de 3 plot utilizada en el modo STFT + espectrograma, a la vista de 2 plot, utilizada en los modos FT, STFT y espectrograma.

layout3()

Esta función se encarga de pasar de la vista de 2 plot, utilizada en los modos FT, STFT y espectrograma, a la vista de 3 plot utilizada en el modo STFT + espectrograma.

lockButtons()

Esta función se encarga de bloquear los controles de usuario mientras dure el cálculo del espectro o del espectrograma, y de reactivarlos cuando termine. El objetivo de bloquearlos es que, durante la espera, la persona usuaria se dé cuenta de que la aplicación está ocupada, y no solicite más cálculos.

plotCenterLine()

Esta función es llamada cuando se hace clic sobre el gráfico de forma de onda en el modo que muestra la STFT o en el que muestra la STFT y el espectrograma. Al hacer clic, se desplaza la ventana sobre la señal y se selecciona el segmento que será enventanado para calcular la STFT.

Esta función se encarga de representar la ventana aplicada sobre la señal y de recalcular el STFT teniendo en cuenta el segmento seleccionado.

resizeLayout()

Al cambiar el tamaño de la ventana, esta función se encarga de recalcular el tamaño y la posición que tendrán los plot en las distintas vistas.

startupFcn()

Esta función StartupFcn merece una mención especial para explicar los distintos parámetros que recibe:

- **user_data:** Contiene la configuración personalizada. Es una variable de tipo *struct*, cuyos campos se han explicado en la sección Ficheros *.mat*.
- **signal:** La señal que será representada y analizada.
- **t:** *Array* de tiempos que acompaña a **signal**.
- **windowN:** Número correspondiente a la variable PlotApp en la que se ha guardado la ventana de visualización actual (ver DialogApp, HelpApp y PlotAppN). Se utiliza cuando se cierra la ventana de visualización, para que pueda localizar y desactivar el botón del módulo principal que tiene asignado.
- **name:** Título que se le da a la ventana de visualización, para describir el tipo de señal que se está analizando.
- **varargin:** Este es un parámetro especial de MATLAB. No es necesario pasarlo siempre que se llame una función, y puede contener una o más variables. Actualmente, está siendo utilizado para pasar variables *strings* que indiquen ciertos casos particulares en los que hay que tener especial cuidado al establecer los límites del eje Y al representar la forma de onda (ver updatePlot()).

updatePlot()
Esta función es llamada desde startupFcn(). Se encarga de calcular la FT de la señal la primera vez, y de establecer los límites "óptimos"para ver bien la forma de onda y el espectro.

En el eje de tiempos de la forma de onda se muestra la duración total de la misma. Los límites del eje de amplitud dependen de si es un caso especial o no, en cuyo caso se notificará mediante un *string* en el parámetro *varargin*:

- En el caso más general no se pasa ningún parámetro *varargin*. Los límites del eje Y se corresponden con el valor máximo y el mínimo de la forma de onda, redondeados a los valores enteros próximos.
- Si sólo se quiere visualizar la parte positiva (valores entre 0 y 1), se pasa el *string* 'Positive' dentro del parámetro *varargin*. Actualmente, esta opción sólo se utiliza para representar los pulsos de Rosenberg, por estética, ya que no tienen valores negativos. Esto se hace así porque, si se aplica el caso general, MATLAB establece el límite inferior en -1.
- Si se va a visualizar una grabación de audio o la señal de un fichero de audio, los límites se establecerán entre -1 y 1. Esto se indica pasando el *string* 'SoundFile' en el parámetro *varargin*. Se hace así porque, si se aplica el caso general, MATLAB puede establecer los límites en -2 y 2, cuando se sabe que el valor máximo no supera la unidad.

En el gráfico del espectro, los límites del eje de amplitud se corresponden con el valor mínimo del espectro (0) y el valor máximo. El caso del eje de frecuencias es más complejo. Para intentar hallar cuál es la región frecuencial que resulta de interés para la persona usuaria, se trata de hallar la región donde la potencia del espectro alcanza un valor mínimo de 10^{-2}. Tal vez este método no sea el más adecuado, porque a veces este criterio no es válido para hallar el rango que nos interesa.

Los límites establecido para los ejes X e Y de ambas gráficas se guardarán para ser utilizados más adelante, para restablecer la vista original tras hacer zoom.

7.11 Ventana de ayuda

Este módulo muestra los documentos de ayuda de usuario. Al instalar la aplicación, la carpeta *Help files* que contiene el árbol de documentos de ayuda se localiza dentro del directorio *application*, la cual se encuentra dentro del directorio de instalación del programa. Este módulo tiene un componente HTML que permite visualizar los ficheros *.html*, y botones para alternar entre ellos. Este módulo se corresponde con el fichero *helpApp.mlapp*.

Ha sido necesario crear este fichero porque, a pesar de que MATLAB ofrece un navegador propio, este no funciona en la versión de *MATLAB Runtime* existente cuando se creó la aplicación. Al parecer, olvidaron incluir algunas funciones necesarias para su funcionamiento y, aunque permite visualizar documentos, la navegación entre ellos es complicada.

Se han utilizado botones para navegar entre los ficheros de ayuda, en lugar de utilizar URL que los comuniquen. Esto se debe a que el elemento HTML de *MATLAB App Designer* sólo soporta URLs que referencian a la propia página, pero no permite acceder a un documento HTML a través de otro.

7.11.1 Funciones

update_help()
Esta función carga en el elemento HTML del módulo de ayuda el documento HTML correspondiente al módulo indicado.

update_module()

Es una función pública que es llamada por el módulo principal tras comprobar que el módulo de ayuda se encuentra abierto. Llama a la función update_help() y carga la ayuda del módulo correspondiente, que se indica en el parámetro *module*.

7.12 Módulo de configuración

Este módulo permite cambiar la configuración personalizada de la aplicación, y guardar los cambios en un fichero para los cambios se mantengan la próxima vez que se abra la aplicación. Este módulo se corresponde con el fichero *control_App.mlapp*.

7.13 Módulo de información

Este módulo muestra la información acerca de las personas que han participado en la creación de la aplicación. Contiene un panel de texto por cada idioma disponible, con la información traducida. Al abrirlo, muestra el panel correspondiente al idioma que se esté utilizando. Este módulo se corresponde con el fichero *info_App.mlapp*.