Práctica 3: estimación de *pitch*

Antonio Bonafonte – profesores de la asignatura

marzo de 2022

Resumen

En esta práctica:

- Se implementará el algoritmo básico de estimación de pitch, basado en la autocorrelación.
- Se practicará con los conceptos de C++: clases, librería standard, `vector`, `string` `iostream`, `pair`, `copy` ...
- Se utilizará el paquete `doxygen`, para documentar clases C++ desde el propio código fuente de los programas.
- Se analizarán las opciones en línea de comando usando la librería `docopt_cpp`.

Índice

1. Introducción teórica: el *pitch* y su estimación.

En la mayor parte de los idiomas, los sonidos usados en ellos pueden dividirse en dos grandes grupos: vocales y consonantes, que se reparten, más o menos, a partes iguales. Así, por ejemplo, alrededor de un 47 % de los sonidos del castellano son vocales, y el 53 % restante, consonantes. A su vez, dentro del grupo de las consonantes, también encontramos dos grupos repartidos, aproximadamente, a partes iguales: las consonantes sonoras y las sordas. Globalmente, alrededor de tres cuartas partes de los sonidos son sonoros: bien vocales, bien consonantes sonoras. En castellano, y éste es un resultado que sirve de buena aproximación para el resto de lenguas cercanas, como las derivadas del latín, algo más del 77 % de los sonidos son sonoros, y el 23 % restante son sordos [Arias, 2016].

Los sonidos sonoros se producen al tensionar las cuerdas vocales de manera que el aire que las atraviesa lo hace de manera discontinua, produciendo una señal sonora periódica, semejante a un tren de deltas de Dirac. La frecuencia de esta señal periódica está relacionada con el *pitch*, o altura tonal, del sonido. Aunque los fonetistas y psicólogos diferencian la frecuencia fundamental (propiedad físico/matemática del sonido) del pitch (propiedad perceptiva), nosotros, y los ingenieros en general, vamos a considerarlos sinónimos. Así pues, la estimación de pitch se reduce a la determinación de la frecuencia fundamental de una señal periódica.

En las transparencias del curso colgadas en Atenea puede encontrar información detallada del pitch (Speech Signal) y su estimación (Speech Analysis). También se recomienda la lectura de los capítulos 2 y 6 de Spoken Language Processing.

En esta práctica se diseñará un sistema de estimación de pitch basado en la autocorrelación. La autocorrelación de una señal periódica es, a su vez, periódica con el mismo periodo que la señal. Dado que toda señal presenta el máximo de la autocorrelación en el origen, detectando la posición del segundo máximo somos capaces de determinar este periodo.

2. Estructura del proyecto.

Para descargar el proyecto debe acceder al repositorio de GitHub Práctica P3 de PAV. En él encontrará información de cómo realizar su descarga e instalación.

Tareas

- Acceda al repositorio GitHub de la práctica y siga las instrucciones de la práctica 2 para clonar ésta.
- Lea el documento de presentación de la página, `README.md`. En él se detallan los ejercicios que deberá resolver como resultado de esta práctica.

La estructura inicial del directorio con el proyecto es la siguiente:

```
PAV/P3
├── .gitignore
├── Makefile
├── README.md
├── meson.build
├── pitch_db
│   └── train
│       ├── orthographic.index
│       ├── rl002.f0ref
│       ├── rl002.wav
│       └── rl004.f0ref
```

```
│       ├── rl004.wav
│       ..............
│       ├── sb050.f0ref
│       └── sb050.wav
├── scripts
│   ├── meson.build
│   ├── run_get_pitch.sh
│   └── sptk_get_pitch.sh
└── src
    ├── doxyfile
    │   ├── meson.build
    │   ├── Doxyfile.in
    │   └── style.css
    ├── get_pitch
    │   ├── get_pitch.cpp
    │   ├── meson.build
    │   ├── pitch_analyzer.cpp
    │   ├── pitch_analyzer.h
    │   └── pitch_evaluate.cpp
    ├── include
    │   ├── digital_filter.h
    │   ├── ffft
    │   │   ├── .........
    │   │   ..............
    │   │   └── .........
    │   ├── filename.h
    │   ├── keyvalue.h
    │   ├── matrix.h
    │   └── wavfile_mono.h
    ├── pav
    │   ├── digital_filter.cpp
    │   ├── docopt_cpp
    │   │   ├── .........
    │   │   ..............
    │   │   └── .........
    │   ├── filename.cpp
    │   ├── keyvalue.cpp
    │   ├── meson.build
    │   └── wavfile_mono.cpp
    └── test_fft
        └── test_fft.cpp
```

En el directorio principal, aparte del subdirectorio `.git` que no se muestra en el árbol, tenemos cuatro ficheros y tres subdirectorios.

`.gitignore` Fichero de descarte y/o inclusión de ficheros y directorios en el repositorio Git.

`README.md` Fichero de presentación del repositorio GitHub, en el cual se detallan los ejercicios que deberán presentarse como resultado de esta práctica.

`Makefile` Fichero de reglas para gestionar la generación y programas de la práctica.

Aunque la gestión del software se realizará usando `make`, éste utiliza `meson/ninja` para realizar la gestión de bajo nivel

`meson.build` Fichero con las reglas usadas por `meson/ninja`.

`pitch_db` Directorio con la base de datos que se usará para *entrenar* el estimador de pitch.

`scripts` Directorio con el script que se usará para automatizar la estimación de pitch en una base de datos, `run_get_pitch.sh`.

`src` Directorio con el código fuente de la práctica, compuesto a su vez por los siguientes subdirectorios:

- `doxyfile` Ficheros usados para generar automáticamente la documentación del proyecto usando el programa `doxygen`.
- `get_pitch` Directorio con el código de los programas `get_pitch`, que realiza la estimación de pitch de un fichero de voz, y `pitch_evaluate`, que evalúa la tasa de error en la estimación de pitch de una base de datos oral.
- `include` Directorio con las cabeceras usadas por los programas del proyecto. Además incluye el subdirectorio `ffft`, con una librería (en la forma de cabeceras C++) para el cálculo de la transformada discreta de Fourier de señales reales.
- `pav` Directorio con las funciones de librería usadas por los programas del proyecto.
  Incluye el subdirectorio `docopt_cpp`, con el código de la librería de gestión de opciones y argumentos en un estilo pythónico que le hará sonreír.
- `test_fft` Directorio con el código fuente de un programa ilustrativo del funcionamiento de la librería `ffft`.

2.1. Mantenimiento de los programas y la documentación.

En esta práctica se va a realizar el mantenimiento de los programas usando un enfoque híbrido entre `make` y Meson/Ninja. Los motivos para ello, y la descripción completa del sistema de mantenimiento, pueden encontrarse en el Anexo I, pero, básicamente, consiste en usar `make` como *front-end* amigable, con Meson/Ninja realizando el trabajo duro.

Al invocar `make` sin argumentos se muestran en pantalla los posibles *targets* del fichero `Makefile`:

```
usuario:~/PAV/P4$ make
Usage:
  make release       : create "bin-&-lib" of the release version
  make debug         : create "bin-&-lib" of the debug version
  make all           : make debug and release

  make clean_release : remove the "release" intermediate files
  make clean_debug   : remove the "debug"   intermediate files
  make clean         : make clean_debug and clean_release

  make doc           : generate the documentation of the project
```

Ejecutando `make release` o `make debug`, se invocan las reglas necesaria de `meson.build` para instalar los programas en el directorio `$HOME/PAV/bin` con las opciones correspondientes a:

`release:` Compilación con la máxima optimización del código. Los programas se ejecutarán más rápidamente (a menudo, mucho más), pero a costa de perderse la correspondencia uno a uno entre las instrucciones en C++ y las instrucciones en ensamblador. Además, el compilador hará de todo con las variables de C++: las eliminará, si son irrelevantes; las almacenará en los registros de la CPU, si son usadas a menudo; etc.

debug: Compilación en la que se respeta al máximo el código fuente en C++: cada instrucción C++ se traduce en un bloque compacto de instrucciones en ensamblador; y se respetan las variables. Además, junto con el código ensamblador se almacena el código fuente correspondiente, permitiendo, por ejemplo, ejecutar éste línea a línea.

El motivo de disponer de dos versiones es evidente a partir de su propio nombre: `release` se usará cuando queramos construir la versión definitiva de los programas, que querremos que se ejecuten tan rápido como sea posible. Pero el efecto de depurar con un *debugger* código optimizado por el compilador es simplemente demencial: la ejecución va dando tumbos entre líneas y muchas variables no son accesibles. Para ello se construye la versión `debug`, que facilita el depurado, a costa de una ejecución más lenta y de programas más grandes.

2.1.1. Instalación de los scripts.

Al ejecutar `make release` o `make debug`, también se instalan en el directorio `~/PAV/bin` los scripts de la práctica, cuyo código fuente está en el directorio `scripts`. De este modo, los scripts serán también accesibles desde cualquier directorio, sin necesidad de especificar su ruta completa (siempre que tenga `~/PAV/bin` en el `$PATH`). Por coherencia con el resto de casos, en los cuales los programas ejecutables pierden la extensión de su código fuente, los scripts instalados en `~/PAV/bin` también lo hacen.

El fichero de reglas `meson/ninja` para la gestión de los scripts del programa aparece descrito en el Apéndice I.D.

Tarea

Recompile los programas de la práctica con la orden `make release` y compruebe que no se produce ningún error:

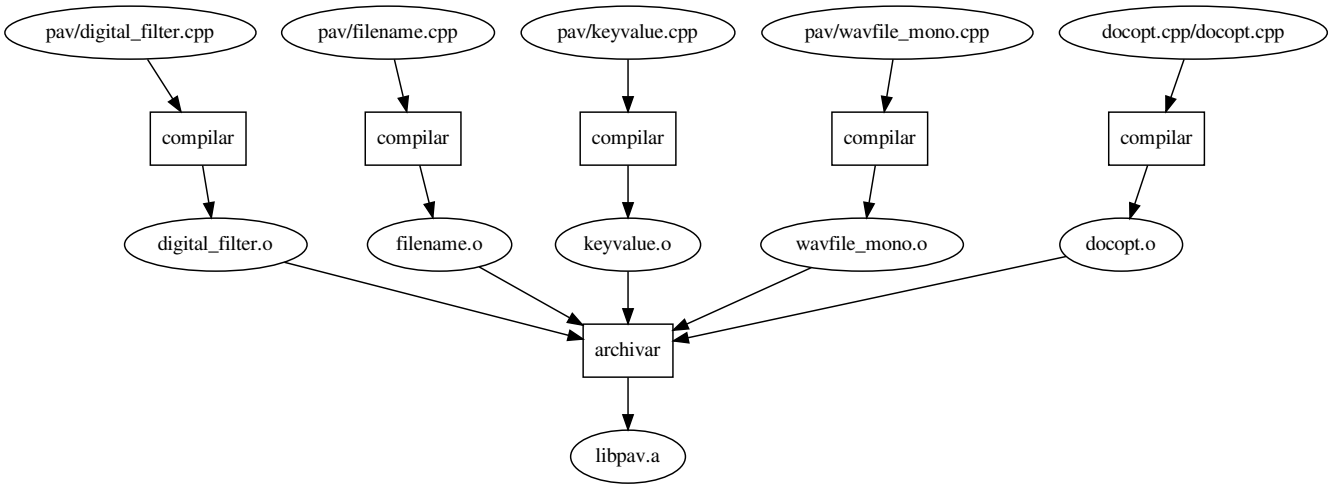
- Compruebe que se han generado los programas en el directorio `~/PAV/bin`.
- Compruebe que este directorio está incluido en su variable `$PATH`. En el caso de que no lo esté, edite el fichero `~/.profile` y añada la línea `PATH=$PATH:$HOME/PAV/bin`. Compruebe que los programas de la práctica pueden ser accedidos sin especificar su ruta. (Si ha tenido que modificar `~/.profile`, deberá ejecutar `source ~/.profile` para que los cambios tengan efecto en su sesión Bash actual).

2.2. Librería `libpav.a`

Es habitual que, en entornos de desarrollo, se construyan una serie de funciones y/o clases de propósito general que son compartidas por distintos proyectos. En lugar de tener copias de estas funciones para cada proyecto, lo más conveniente es agruparlas en un único archivo con el que se enlazan los distintos programas.

En el entorno de las prácticas de la asignatura PAV, vamos a almacenar estas funciones de propósito en la librería `libpav.a`, que se ubicará en el directorio `$HOME/PAV/lib`. En ella, por ahora, vamos a meter funciones que gestionan los nombres de los ficheros (`filename.cpp`), el paso de variables de configuración por fichero (`keyvalue.cpp`), etc. También vamos a meter aquí las clases y funciones asociadas al análisis de los parámetros y opciones en línea de comandos proporcionados por `docopt_cpp`.

Por ahora, la estructura de dependencias de esta librería es:

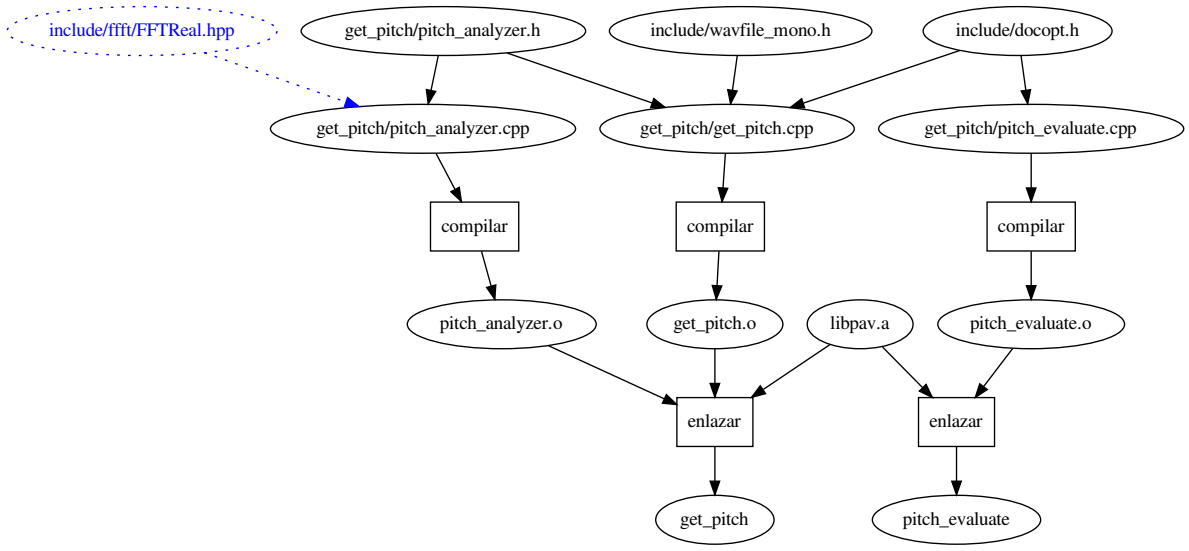


Que se corresponde con el fichero de reglas `src/pav/meson.build`, descrito en el Apéndice I.F.

2.3. Programas `get_pitch` y `pitch_evaluate`

Lo dos programas principales en esta práctica son `get_pitch` y `pitch_evaluate`. El primero realiza la estimación de pitch para una única señal de entrada, y el segundo evalúa las tasas de error para un conjunto de ficheros.

La estructura de dependencias para estos dos programas es:



Donde la cabecera `include/ffft/FFTReal.hpp` (en azul en el gráfico) no se incorpora en la versión inicial del proyecto, y está contemplada para el caso en que se desee realizar como ampliación la estimación de pitch usando métodos transformados; en concreto, *análisis cepstral* (ver el Anexo II.C).

El fichero `src/get_pitch/meson.build` dedicada al mantenimiento de estos dos programas, está descrito en el Apéndice I.C.

3. Herramientas empleadas en la pràctica.

3.1. El lenguaje de programación C++.

En esta tercera práctica, y en todas las siguientes, se utilizará el lenguaje de programación C++. Este lenguaje se basa en los conceptos de programación orientada a objetos, desarrollados, sobre todo teóricamente, durante los años sesenta y setenta del siglo XX. En 1979, Bjarne Stroustrup diseñó C++ tomando como referencias la programación orientada a objeto y el lenguaje de programación C, con el que C++ mantiene compatibilidad hacia atrás (todo programa escrito en C es compatible con C++, en lo que se denomina C/C++; aunque un programa auténticamente C++ no es compatible con C). A partir de su primera versión pública, en 1983, la popularidad de C++ fue aumentando hasta convertirse en uno de los lenguajes más extendidos (y, por muchos, temidos).

A menudo se denomina a C++ como *C con clases*, aunque los elementos de programación orientada a objetos van mucho más allá de la definición de clases, introduciendo conceptos como la herencia (múltiple), plantillas, interfaces, etc.

En Atenea dispone de una Introducción a C++ en la que se exponen los conceptos básicos del lenguaje. Hay, además, toda una variedad de tutoriales, videos y demás disponibles en internet, entre los que se puede destacar el análisis del creador del núcleo de Linux y del sistema de gestión de versiones Git al respecto de su uso en el desarrollo este último, Linus Torvalds on C++.

3.2. Generación de la documentación: `doxygen`.

Documentar el código de un proyecto es una de las tareas más arduas para todo desarrollador. Entre otros motivos, porque se compagina malamente con su ciclo de trabajo. Si empieza escribiendo la documentación, o lo hace simultáneamente a la escritura del propio código, va tener que reescribirla tantas veces como veces tenga que modificar el código. Si no lo hace, cometerá el peor error que se puede cometer al documentar código: que no se refleje el comportamiento real de éste. Y es mejor no documentar nada, que hacerlo mal...

Si, por el contrario, decide esperar a completar el código para documentarlo, se encontrará con que, cuando finalmente ha conseguido que todo funcione, le queda la que, para muchos, es la parte más pesada del proceso: documentar lo que se ha hecho. Y es habitual que eso sea lo último que le apetezca hacer (piénsese en lo ameno y entretenido que resulta hacer la memoria de una práctica, que se ha tardado más de una semana en conseguir que funcione correctamente).

Doxygen es una herramienta que facilita enormemente la generación de documentación para distintos lenguajes de programación; entre los que se encuentran C, C++, Java y Python. Su nombre es el acrónimo de *docs generator* (generador de documentos), jugando con la casi homofonía con *oxygen*, porque sirve para dar oxígeno al programador que tiene que realizar la tarea.

Se basa en dos mecanismos:

- Interpreta la estructura del código para identificar los elementos que lo forman (funciones, clases, macros, etc.) y establecer la relación entre ellos.
- Admite un formato específico de comentarios del código que se utiliza para describir o comentar los elementos del punto anterior.

Con estos dos mecanismos, `doxygen` es capaz de generar documentos en HTML, LaTeX, `man`, y alguno más. Por sí sólo, es decir: sin que el programador escriba comentarios específicos, `doxygen` es capaz de generar una documentación del proyecto en la que es fácil ver todos los elementos que lo forman, cómo se relacionan entre ellos y saltar de uno a otro mediante enlaces dinámicos. Como apunte canalla, mencionar que, sin necesidad de que el programador le dedique ningún esfuerzo, `doxygen` generará un montón de páginas de documentación, que puede que algunos encuentren de poca utilidad, pero

que sin duda impresionan bastante. Por ejemplo, el PDF generado para un proyecto como el de esta práctica se va a más de 130 páginas (si no se incluye el código fuente, si sí se hace, supera las 170), con una pinta sumamente profesional.

Utilizando comentarios específicos es posible añadir información realmente valiosa, que sirve tanto para comentar el código en el que aparecen, como para complementar la documentación generada de modo automático. Estos comentarios pueden estar escritos en modo texto normal y corriente, del mismo modo que los comentarios habituales del código. Pero también permiten extensiones que harán que la documentación generada sea mucho más vistosa. Por ejemplo: el formato markdown, para escribir texto con formato y estructurado; fórmulas usando la sintaxis de LaTeX; representación gráfica de la jerarquía de clases usando el lenguaje `dot` de Graphviz; etc.

`doxygen` admite una variedad de maneras para indicar que debe interpretar un cierto comentario del código. Básicamente, se trata de modificar el indicador de inicio de comentario de manera que, sin dejar de ser un comentario válido en el lenguaje nativo, pueda ser localizado por `doxygen` para sus propios propósitos. En C++, los siguientes comentarios serán tratados por `doxygen` y adjuntados a la documentación de la función, clase o fichero en el que se encuentren:

```
/**
  * Un comentario iniciado por las cadenas /** o /*! es interpretado por doxygen.
  */

//! Un comentario iniciado por las cadenas /// o //! es interpretado por doxygen.
```

3.2.1. Ejecución de Doxygen.

El programa `doxygen` admite una serie de modos de funcionamiento distintos. El modo básico, que se encarga de generar la documentación, es:

```
doxygen [configName]
```

Donde, si no se indica `configName`, el programa utilizará el fichero `Doxyfile` que debe estar en el directorio. Podemos generar un fichero `Doxyfile` con las opciones por defecto usando la orden `doxygen -g`. Así pues, si no disponemos del fichero de configuración, las órdenes necesarias para obtener la documentación son:

```
usuario:~/PAV$ doxygen -g
usuario:~/PAV$ doxygen
```

Al ejecutar esta orden, se nos crearán dos directorios nuevos: `html`, con la documentación en formato de página web; y `latex`, donde, ejecutando el comando `make pdf`, obtendremos el fichero `refman.pdf` con la documentación en formato PDF.

Una vez generada, podemos visualizar la documentación abriendo el fichero `html/index.html` con un explorador. En esta primera versión, usando las opciones por defecto del fichero `Doxyfile`, la documentación generada es realmente espartana. Apenas se verá el contenido del fichero `README.md`, ya que, por defecto, sólo se utilizan los ficheros encontrados en el directorio actual. Para hacer que `doxygen` busque los ficheros en los subdirectorios de manera recursiva, hemos de editar `Doxyfile`, y, alrededor de la línea 867, modificar la que pone `RECURSIVE = NO` por `RECURSIVE = YES`.

Generación de la documentación usando `make` y `meson/ninja`.

En esta práctica, y las siguientes, usaremos el programa `make` para automatizar el proceso de generación de la documentación. El sistema se basa en ejecutar `make doc`, el cual se encarga de invocar el sistema

`meson/ninja` para, partiendo de un fichero de configuración inicial, `src/doxyfile/Doxyfile.in`, generar un nuevo `Doxyfile` que se adaptará mejor a nuestras necesidades.

Deberá tener instalado el paquete `graphviz` para poder generar los gráficos incluidos por `doxygen`. Para ello, ejecute el comando:

```
usuario:~/PAV$ sudo apt-get install graphviz
```

Una vez instalado `graphviz`, ejecute el comando siguiente cada vez que desee reconstruir la documentación:

```
usuario:~/PAV$ make doc
```

A partir de este momento, la documentación está disponible en el directorio `$HOME/PAV/html/P3`. Acceda al directorio con el navegador y, abriendo el fichero `index.html`, tendrá acceso a la página principal del proyecto.

El fichero `meson.build` dedicado a la generación y mantenimiento de la documentación es un poco más complicada que lo que se ha visto hasta el momento, y está descrito en el Apéndice I.E.

3.2.2. Empleo de Doxygen en las prácticas de PAV.

En esta práctica y las siguientes se usará Doxygen para dos objetivos distintos:

- Proporcionar la documentación del código del proyecto, facilitando la visualización de su estructura y la navegación por las distintas partes que lo componen.
- Proporcionar un mecanismo para realizar el seguimiento de las partes del proyecto a implementar y/o modificar, y registrar las modificaciones realizadas al respecto.

El primero de estos objetivos es el común de Doxygen, y la experiencia dice que, dependiendo del usuario, su utilidad puede variar entre la intrascendencia y la genialidad. El segundo echa mano de dos adaptaciones específicas de la asignatura para gestionar las partes del código que se deben modificar o completar:

`\TODO` Colocando la cadena `\TODO` en un comentario `doxygen`, éste se incorpora a la lista de tareas por realizar, que aparecerá en el árbol del proyecto con el nombre *Lista de TODOs*, y encabezada por la cadena '[TODO]:'.

`\FET` Al colocar la cadena `\FET`, el comentario se añade a la lista de tareas por realizar, *Lista de TODOs*, encabezado por la cadena '[FET]:'.

- También se pueden usar las cadenas `\DONE` o `\HECHO`.

El alumno deberá usar el primero de los comandos para localizar y acceder a las tareas pendientes en el código. Una vez realizada la tarea correspondiente, deberá escribir un comentario con el segundo de ellos para explicar los cambios realizados.

Tareas

- Genere la documentación del proyecto con la orden `make doc`.
- Acceda con el navegador al fichero `index.html` (o cualquier otro fichero `.html`) en el direc-

torio `~/PAV/html/P3`.

- Localice la posición de los comandos `\TODO` en la *lista de TODOs* y salte a la posición donde se encuentra en el fichero correspondiente.

3.3. `docopt_cpp`

La versión para C++ de la librería `docopt` para la gestión de opciones y argumentos en línea de comandos se llama `docopt_cpp`. Al contrario que la versión para C, `docopt_c`, la versión para C++ implementa el estándar POSIX, con las extensiones de GNU, de manera completa y sin *bugs*. Además, la sintaxis es ligeramente distinta y más parecida al original para Python:

- Se define una variable del tipo `const char USAGE[]` y se le asigna el mensaje con el modo de empleo del programa.
  - La cadena de texto ha de ser cruda (*raw*), lo cual se indica precediendo su definición con la letra `R` y encerrando su contenido entre comillas y paréntesis: `R"(contenido de texto)"`.
  - En el fichero `get_pitch.c`, definimos:

```
19 static const char USAGE[] = R"(
20 get_pitch - Pitch Estimator
21
22 Usage:
23     get_pitch [options] <input-wav> <output-txt>
24     get_pitch (-h | --help)
25     get_pitch --version
26
27 Options:
28     -h, --help  Show this screen
29     --version   Show the version of the project
30
31 Arguments:
32     input-wav   Wave file with the audio signal
33     output-txt  Output file, ASCII file with the result of the estimation
34                     - One line per frame with the estimated f0
35                     - If considered unvoiced, f0 must be set to f0 = 0
36 )";
```

- En el código de la función se incluye la llamada a la función `docopt()`.
  - Esta función toma por argumentos la cadena de texto con el modo de empleo; los argumentos usados en la invocación (exceptuando el primero, con el nombre del programa); un valor booleano, que indica si se desea que, al invocar el programa con las opciones `-h` o `--help`, el programa muestre el modo de empleo y termine la ejecución; y una cadena de texto con la versión.
  - Como salida, la función devuelve un diccionario (o *mapeo*), que usa como clave el nombre de la opción (dando preferencia a los nombres largos), y como contenido el valor del argumento correspondiente, de la clase `docopt::value`.

```
42 std::map<std::string, docopt::value> args = docopt::docopt(USAGE,
43     {argv + 1, argv + argc},    // arguments without program name
44     true,                       // show help if requested
45     "2.0");                     // version string
```

c

- Los argumentos pueden ser accedidos mediante el diccionario. Además, pueden ser convertidos al tipo deseado usando los métodos de la clase `docopt::value`:

```
47 std::string input_wav = args["<input-wav>"].asString();
48 std::string output_txt = args["<output-txt>"].asString();
```

  - En este caso, se convierte los dos argumentos a cadenas de texto con el método `asString()`.
  - También es posible realizar la conversión a booleano, con el método `asBool()`; a entero, con `asLong()`; o a lista de cadenas, con `asStringList`.
  - Si se desea obtener un valor real, hemos de realizar la conversión a partir de la cadena con la función `stof()`.

4. Construcción del estimador de pitch: `get_pitch`

El programa `get_pitch` incluye el esqueleto de un sistema de estimación de pìtch:

- Se lee la señal de entrada usando la función `wavfile_mono()`, que encapsula las funciones de la librería `sndfile` para la lectura de ficheros de audio en de un solo canal (mono).
- Se construye el analizador de pitch de la clase `pitch_analyzer` definida en los ficheros del directorio `src/get_pitch` `pitch_analyzer.cpp` y `pitch_analyzer.h`.
- Se invoca el analizador para cada trama de la señal de entrada y se escribe el resultado en el fichero de salida.

Una vez construido el programa `get_pitch`, con la orden `make release` o `make debug`, éste se ubica en el directorio `$HOME/PAV/bin`. Podemos ver su modo de empleo ejecutándolo sin argumentos:

```
usuario:~/PAV/P3$ $HOME/PAV/bin/get_pitch
Arguments did not match expected patterns

get_pitch - Pitch Estimator

Usage:
    get_pitch [options] <input-wav> <output-txt>
    get_pitch (-h | --help)
    get_pitch --version

Options:
    -h, --help  Show this screen
    --version   Show the version of the project

Arguments:
    input-wav   Wave file with the audio signal
    output-txt  Output file, ASCII file with the result of the estimation
                    - One line per frame with the estimated f0
                    - If considered unvoiced, f0 must be set to f0 = 0
```

Por tanto, `get_pitch` requiere que se le pase como primer argumento un fichero WAVE con la señal de voz, y como segundo argumento el nombre del fichero en el que queremos almacenar el resultado de la estimación.

Volvemos a ejecutar el programa, ahora con los dos argumentos obligatorios:

```
usuario:~/PAV/P3$ $HOME/PAV/bin/get_pitch pitch_db/test/rl001.wav pitch_db/test/rl001.f0
```

Si todo ha ido bien, ahora el programa se ejecuta silenciosamente, indicando que ha realizado la estimación con éxito. De hecho, en su versión actual, el estimador en sí no está implementado, así que, si miramos el resultado de la estimación, `pitch_db/test/rl001.f0`, lo que encontramos es un fichero de texto con tantas líneas como tramas tiene la señal (en esta práctica, en principio, se usa un desplazamiento de ventana de 15 ms), todas ellas con el valor 0. Este valor se usará para indicar que la trama es sorda. En adelante, cuando el programa esté operativo, las tramas sonoras se marcarán con un valor real positivo que indicará la frecuencia en hercios del pitch.

Incorporación de `$HOME/PAV/bin/` a la variable `$PATH`.

Para evitar tener que invocar el programa usando su ubicación, se recomienda editar el fichero de comandos de inicio de sesión (`~/.profile`) para añadir a la variable `$PATH` la ruta del directorio con el programa:

```
export PATH=$PATH:$HOME/PAV/bin
```

Tareas:

- Si ha colocado su directorio de trabajo para las prácticas de la asignatura en un directorio distinto de `~/PAV/`, deberá editar el fichero `Makefile` para que la variable `$PREFIX` apunte al directorio correcto.
- Modifique el fichero `~/.profile` (o el que le corresponda a su sistema operativo[a]), para que incluya el directorio correspondiente en la variable `$PATH`:

  ```
  export PATH=$PATH:$HOME/PAV/bin
  ```

  Recuerde que deberá ejecutar `source ~/.profile` para que los cambios realizados tengan efecto en el shell actual. Sin embargo, los shells que abra a partir de este momento ya tendrán la configuración correcta.

  Si ha decidido colocar sus archivos de la asignatura en un directorio distinto de `~/PAV`, deberá usar ese directorio al modificar la variable `$PATH` en el archivo `$HOME/.profile`
- Reconstruya el proyecto con la orden `make release` y compruebe que tanto `get_pitch` como `run_get_pitch` se ejecutan correctamente sin necesidad de indicar su directorio (ni la extensión, en el caso de `run_get_pitch`).

[a] En MacOS X, el fichero adecuado es `~/.bash_profile`, si usa `bash`; o `~/.zshrc`, si usa `zsh`.

5. Evaluación del resultado.

En un sistema de estimación de pitch podemos identificar cuatro grandes tipos de error:

Sonoro por sordo (VU):
Cuando una trama sorda es identificada como sonora.

Sordo por sonoro (UV):
Cuando una trama sonora es identificada como sordo.

Errores *groseros* (G):
Cuando una trama sonora es identificada como tal, pero el error cometido en la estimación del pitch es superior a un cierto umbral (fijado, en esta práctica, en el 20 %).

MSE de los errores *finos*:
Error cuadrático medio de los errores no considerados como groseros.

La importancia relativa de los cuatro tipos de error es diferente: es más grave confundir un segmento sonoro con uno sordo, o viceversa, que cometer un error grosero; y es más grave cometer un error grosero que uno fino. Teniendo esto en cuenta, construimos una puntuación global del modo siguiente:

- Aumentamos el número de confusiones **VU** con la mitad de los errores groseros, disminuyendo los aciertos en la estimación de segmentos sonoros en la misma cantidad.
- Calculamos la medida-F de sonoro y sordo usando los valores modificados por los errores groseros. Como las medida-F de sonoro y sordo son diferentes, y en aras a usar una medida simétrica, tomamos como puntuación-F global, $F(S/S)$, la media geométrica de ambas.
- Multiplicamos la medida-F global por uno menos el error cuadrático medio de los errores finos.

$$
\begin{aligned}
VU' &= VU + G/2 \\
VV' &= VV - G/2 \\
F(V) &= \sqrt{\frac{VV'}{VV' + VU'} \frac{VV'}{VV' + UV}} \\
F(U) &= \sqrt{\frac{UU}{UU + UV} \frac{UU}{UU + VU'}} \\
F(S/S) &= \sqrt{F(V)\,F(U)} \\
score &= F(S/S)\,(1 - \mathrm{MSE})
\end{aligned}
$$

El programa C++ `pitch_evaluate` toma como argumento uno o más ficheros (extensión `.f0ref`) de referencia y calcula estas medidas:

```
usuario:~/PAV/P3$ pitch_evaluate pitch_db/test/rl001.f0ref
### Compare pitch_db/test/rl001.f0ref and pitch_db/test/rl001.f0
Num. frames:    547 = 302 unvoiced + 245 voiced
Unvoiced frames as voiced:      6/302 (1.99 %)
Voiced frames as unvoiced:      23/245 (9.39 %)
Gross voiced errors (+20.00 %): 1/222 (0.45 %)
MSE of fine errors:     2.93 %

===>    pitch_db/test/rl001.f0:        91.74 %
```

5.1. Evaluación en la base de datos `pitch_db`.

Como en el caso de la estimación de actividad vocal, realizar las pruebas de estimación de pitch usando un único fichero es poco significativo de las prestaciones del sistema por dos motivos: por un lado, los resultados obtenidos en un fichero concreto no tienen por qué corresponderse con los que se obtendrían con uno distinto; por otro, al ajustar los parámetros del sistema usando el mismo material que se usará

en la evaluación estamos, en cierto modo, *haciendo trampa*, ya que estamos adaptándonos a esa señal en concreto.

Igual que se hizo en el caso de la estimación de actividad vocal, la solución pasa por dos cuestiones: en primer lugar, en vez de un único fichero se usará un conjunto de ficheros, que llamaremos *base de datos*; en segundo lugar, se usará dos bases de datos distintas para el ajuste de los algoritmos y parámetros del sistema (`train`) y para la evaluación del sistema (`test`).

La base de datos que se va a utilizar es la Fundamental Frequency Determination Algorithm (FDA) Evaluation Databa desarrollada por Paul Bagshaw en el curso de su Tesis Doctoral.

Esta base de datos se compone de 100 señales, 50 de ellas por un locutor masculino (ficheros `rl001` a `rl050`), y 50 más por un locutor femenino (ficheros `sb001` a `sb050`). La mitad de estos ficheros se usarán para *entrenar* el sistema `pitch_db/train`, y la otra mitad para evaluar sus prestaciones `pitch_db/test`. Esta segunda parte permanecerá oculta para el desarrollador, que sólo accederá a los resultados evaluados por una mano imparcial.

El fichero `run_get_pitch.sh` del directorio `scripts` puede ser usado para automatizar la estimación de pitch en la base de datos de entrenamiento:

```
usuario:~/PAV/P3$ run_get_pitch
pitch_db/train/rl002.wav ----
pitch_db/train/rl004.wav ----

         .........

pitch_db/train/rl050.wav ----
pitch_db/train/sb002.wav ----

         .........

pitch_db/train/sb050.wav ----
```

A continuación, podemos realizar la evaluación de los resultados usando el programa `pitch_evaluate` con los ficheros de referencia a evaluar:

```
usuario:~/PAV/P3$ pitch_evaluate pitch_db/train/*f0ref
### Compare pitch_db/train/rl002.f0ref and pitch_db/train/rl002.f0
Num. frames:    134 = 83 unvoiced + 51 voiced
Unvoiced frames as voiced:      2/83 (2.41 %)
Voiced frames as unvoiced:      7/51 (13.73 %)
Gross voiced errors (+20.00 %): 1/44 (2.27 %)
MSE of fine errors:     2.06 %

===>    pitch_db/train/rl002.f0:        90.38 %
------------------------------------------------

                .........

------------------------------------------------

### Compare pitch_db/train/sb050.f0ref and pitch_db/train/sb050.f0
Num. frames:    267 = 134 unvoiced + 133 voiced
Unvoiced frames as voiced:      7/134 (5.22 %)
Voiced frames as unvoiced:      3/133 (2.26 %)
Gross voiced errors (+20.00 %): 3/130 (2.31 %)
MSE of fine errors:     1.52 %

===>    pitch_db/train/sb050.f0:        94.24 %
------------------------------------------------

### Summary
```

```
Num. frames:    11200 = 7045 unvoiced + 4155 voiced
Unvoiced frames as voiced:      190/7045 (2.70 %)
Voiced frames as unvoiced:      376/4155 (9.05 %)
Gross voiced errors (+20.00 %): 34/3779 (0.90 %)
MSE of fine errors:     2.47 %

===>    TOTAL:  92.02 %
```

6. Ejercicios y entrega.

El objetivo principal de esta práctica es completar la implementación del estimador de pitch esbozado en el programa `get_pitch`. Para ello, deberá localizar los comentarios Doxygen marcados con la palabra clave `\TODO`, indicando las tareas realizadas en el propio comentario usando la palabra clave `\FET` (o, si se prefiere, `\DONE` o `\HECHO`). Estos comentarios quedarán reflejados en la *Lista de TODOs* de la documentación Doxygen.

Conforme realice la práctica, vaya completando los ejercicios del fichero `README.md`. Al finalizar la práctica, deberá realizar un *pull request* al repositorio GitHub del enunciado. La rama objeto del pull request deberá tener por nombre los primeros apellidos de los integrantes del grupo de prácticas, separados por guion.

En el pull request sólo deberán figurar los archivos necesarios para evaluar el trabajo realizado. En concreto, **no deben** los directorios con los ficheros intermedios de compilación (por ejemplo, `build`). **Sí deben** adjuntarse todos los ficheros necesarios para recompilar los programas usando el comando `make release`.

Una parte importante de la evaluación se basará en el resultado obtenido en la estimación de pitch de la base de datos `pitch_db/test`, desconocida para el alumno. Más arriba dispone de resultados reales, obtenidos con la base de datos de entrenamiento (`pitch_db/train`, que le pueden servir de referencia en su desarrollo (pero tampoco hay que desanimarse, en el sistema usado para obtenerlos se hace de todo...).

ANEXOS.

I. Mantenimiento de la práctica usando `meson` y `make`.

La estrategia seguida para mantener las prácticas precedentes, consistente en un único fichero `makefile` o `meson.build` que construye todos los elementos cada vez que invocamos el comando correspondiente, es válida para proyectos pequeños como las dos primeras prácticas. Sin embargo, en proyectos de mayor envergadura, como esta P3 o las dos siguientes, resulta ineficiente y difícil de mantener.

Para simplificar el mantenimiento del proyecto, así como para aumentar su eficiencia, se han introducido una serie de modificaciones que se detallan en las secciones siguientes.

I.A. Empleo de `make`.

Aunque la herramienta fundamental que se usará en el mantenimiento de la práctica seguirá siendo Meson/Ninja, se va a emplear `make` debido a que esta herramienta permite o simplifica toda una serie de cuestiones:

- Desde `make` es posible acceder a las variables de entorno del sistema, algo que Meson no permite, al menos de un modo sencillo.

  Al poder acceder, por ejemplo, a la variable de entorno `$HOME`, seremos capaces de instalar los programas, librerías y demás en el directorio `$HOME/PAV`, sin necesidad de editar ningún fichero para que el usuario lo especifique explícitamente.

- `make` simplifica mucho la construcción parcial del proyecto. Esto es: seleccionar qué partes se compilan y con qué opciones. Esto es así por dos motivos: por un lado, si no se especifica ningún objetivo concreto, `make` sólo construye el primero que encuentra en el fichero `Makefile`; por el contrario, Meson/Ninja construye todos los objetivos presentes en el fichero `meson.build`. Por otro lado, en `make` es muy sencillo definir objetivos que siempre se actualizarán (siempre están desfasados) con el comando `.PHONY`.

- `make` permite simplificar mucho la invocación de comandos Meson/Ninja complejos, que, usando Meson/Ninja directamente, deberían recordarse y escribir sin error.

  Por ejemplo, usando `make`, la orden fundamental para construir los programas del proyecto es `make release`; mientras que, usándolos directamente, la misma tarea, utilizando Meson/Ninja, se debe invocar como:

```
usuario:~/PAV/P4$ meson --buildtype=release --prefix=$HOME/PAV --libdir=lib bin/release
usuario:~/PAV/P4$ ninja install bin/release
```

Así pues, usaremos `make` como *front-end* para Meson/Ninja con el siguiente fichero de reglas:

Makefile

```
# PREFIX sets the 'prefix' option of Meson's project() on 'ninja install'.
PREFIX = ${HOME}/PAV

BUILD_RELEASE = bin/release
BUILD_DEBUG   = bin/debug
FILE_RELEASE  = ${BUILD_RELEASE}/build.ninja
FILE_DEBUG    = ${BUILD_DEBUG}/build.ninja

.PHONY: help release debug all clean_release clean_debug clean doc

```

```
help:
    @echo '------------------------------------------------------------------'
    @echo 'Usage:'
    @echo '  make release       : create "bin-&-lib" of the release version'
    @echo '  make debug         : create "bin-&-lib" of the debug version '
    @echo '  make all           : make debug and release'
    @echo ' '
    @echo '  make clean_release : remove the "release" intermediate files'
    @echo '  make clean_debug   : remove the "debug"   intermediate files'
    @echo '  make clean         : make clean_debug and clean_release'
    @echo ' '
    @echo '  make doc           : generate the documentation of the project'
    @echo '------------------------------------------------------------------'

release: ${FILE_RELEASE}
        ninja install -C ${BUILD_RELEASE}

${FILE_RELEASE}:
        meson --buildtype=release --prefix=${PREFIX} --libdir=lib ${BUILD_RELEASE}

clean_release:
        \rm -rf ${BUILD_RELEASE}

debug: ${FILE_DEBUG}
        ninja install -C ${BUILD_DEBUG}

${FILE_DEBUG}:
        meson --buildtype=debug --prefix=${PREFIX} --libdir=lib ${BUILD_DEBUG}

clean_debug:
        \rm -rf ${BUILD_DEBUG}

all: release debug

clean: clean_release clean_debug

doc:
        ninja doc -C ${BUILD_RELEASE}
```

I.B. Estructura de subdirectorios.

En este proyecto, Meson/Ninja se usa de manera distribuida entre los subdirectorios que forman el código fuente. Un fichero `meson.build` en el directorio principal del proyecto se encarga de importar, mediante el comando `subdir()`, los ficheros de dependencias correspondientes a:

- Librería `libpav.a`, con las funciones y clases genéricas desarrolladas en la asignatura y que son usadas por múltiples proyectos.
- Programas de la práctica, repartidos entre los subdirectorios `src/gmm`, `src/pearson` y `src/fmatrix`.
- Scripts de la práctica, almacenados en el directorio `src/scripts`.
- Documentación del proyecto, a partir de los ficheros de configuración en `src/doxyfile`.

meson.build

```
project(
    'Práctica 3 de PAV - detección de pitch', 'cpp',
    default_options: 'cpp_std=c++11',
    version: 'v2.0'
)
```

```

inc = include_directories(['src/include', 'src/pav/docopt_cpp'])

# Librería PAV
subdir('src/pav')

# Programas del proyecto
subdir('src/get_pitch')

# Scripts de la práctica
subdir('scripts')

# Documentación del proyecto
subdir('src/doxyfile')
```

La función `subdir(`*`directorio`*`)` es, a casi todos los efectos, equivalente a insertar el contenido del fichero *`directorio`*`/meson.build` en el punto de su invocación por el `meson.build` principal. Por este motivo, los diferentes ficheros *`directorio`*`/meson.build` no deben incluir definición de proyecto (usarán la definición del principal). Por otro lado, todas la variables y dependencias definidas previamente por el fichero `meson.build` son accesibles desde los ficheros `directorio/meson.build` subordinados; y todas las variables y dependencias definidas en éstos son accesibles por el `meson.build` principal, y sus subordinados, a partir del punto de la invocación de `subdir()`.

La principal diferencia entre invocar `subdir(`*`directorio`*`)` o incluir el contenido de *`directorio`*`/meson.build` directamente en el fichero `meson.build` es que `ninja` será consciente de que el fichero `meson.build` está en un subdirectorio de `src`, en lugar de en el directorio principal del proyecto, y generará los programas y librerías en `bin/src` en lugar de en `bin`.

I.C. Programas de la práctica.

El fichero de reglas para realizar el mantenimiento de los dos programas se encuentra en el directorio `src/get_pitch`, y su contenido es el siguiente:

src/get_pitch/meson.build

```
# Programas del proyecto

# Programa get_pitch
exe = 'get_pitch'
src = ['get_pitch.cpp',  'pitch_analyzer.cpp']

get_pitch = executable(
    exe,
    sources: src,
    include_directories: inc,
    link_args: ['-lm', '-lsndfile'],
    link_with: lib_pav,
    install: true,
)

# Programa pitch_evaluate
exe = 'pitch_evaluate'
src = 'pitch_evaluate.cpp'

pitch_evaluate = executable(
    exe,
    sources: src,
    include_directories: inc,
    link_with: lib_pav,
    install: true,
)
```

I.D. Scripts de la práctica.

En el caso de los scripts de la práctica, al ejecutar `make release` o `make debug`, los scripts se enlazan simbólicamente en `~/PAV/bin` con el mismo nombre que el script original, pero sin su extensión.

Como son enlaces simbólicos, cualquier cambio en los códigos fuente en el directorio `scripts` se reflejan automáticamente en los ejecutables en `~/PAV/bin`, así que no hace falta volver a ejecutar `make`.

scripts/meson.build

```
# Scripts del proyecto

scripts = [
    'run_spkid.sh',
    'plot_gmm_feat.py',
    'spk_verif_score.pl',
    'wav2lp.sh',
]

foreach script: scripts
    source = join_paths(meson.source_root(), 'scripts', script)     # Ruta completa
    prog = script.split('.')[0]                                     # Nombre sin extensión
    dest = join_paths(get_option('prefix'), get_option('bindir'), prog)
    custom_target(script,
        input : source,
        output : prog,
        command : ['ln', '-sf', source, dest],
        build_by_default: true,
    )
endforeach
```

I.E. Generación de la documentación con Doxygen en `~/PAV/html/P3`.

El fichero de reglas para generar la documentación usando `doxygen` es bastante más complicado que los usados para el mantenimiento de los programas ya que es necesario configurar la llamada a un programa externo, el propio `doxygen`.

src/doxyfile/meson.build

```
# Documentación usando Doxygen

items = [
    'README.md',
        'src',
    'scripts',
]

inputs = ''
foreach item : items
    inputs += '"' + join_paths(meson.source_root(), item) + '" '
endforeach

doxygen = find_program('doxygen')
practica = meson.source_root().split('/')[-1]
html = join_paths(get_option('prefix'), 'html')

config_DG = configuration_data()
config_DG.set('project_name', meson.project_name())
config_DG.set('project_version', meson.project_version())
config_DG.set('inputs', inputs)
config_DG.set('install_dir', html)
config_DG.set('dir_html', practica)
config_DG.set('style_sheet', join_paths(meson.source_root(), 'src/doxyfile', 'style.css'))
```

```
26
27 doxyfile = configure_file(
28     input: 'Doxyfile.in',
29     output: 'Doxyfile',
30     configuration: config_DG,
31 )
32
33 doc = run_target('doc', command: [doxygen, doxyfile])
```

Este fichero se muestra, sobre todo, a modo de información y como ejemplo de las capacidades extendidas de `meson/ninja`. Si desea conocer más detalles del mismo, póngase en contacto con los profesores de la asignatura.

I.F. Librería `libpav.a`.

Finalmente, el `meson.build` para reconstruir la librería `libpav.a` es idéntico al de la práctica P3 con el añadido de los dos nuevos códigos fuente introducidos en esta práctica: `gmm.cpp` y `gmm_vq.cpp`.

src/pav/meson.build

```
2  # Librería PAV
3
4  lib = 'pav'
5  src = [
6     'digital_filter.cpp',
7     'filename.cpp',
8     'wavfile_mono.cpp',
9     'keyvalue.cpp',
10    'gmm.cpp',
11    'gmm_vq.cpp',
12    'docopt_cpp/docopt.cpp',
13 ]
14
15 lib_pav = static_library(
16     lib,
17     sources: src,
18     include_directories: inc,
19     install: true,
20 )
```

II. Mejoras en la estimación de pitch.

II.A. Preprocesado: *center clipping* y filtrado/diezmado.

II.A.A. *Center clipping.*

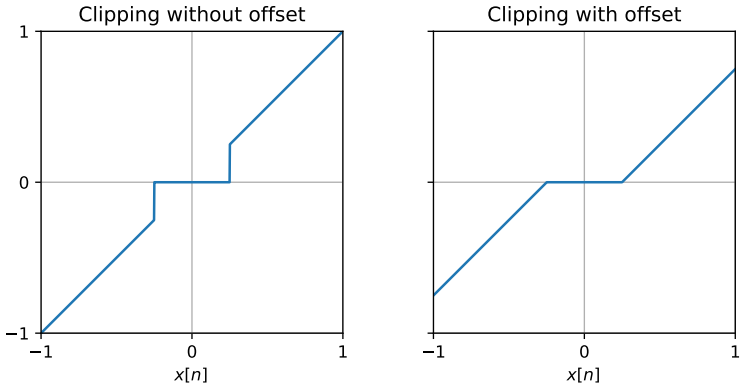
Una técnica sencilla que suele mejorar las prestaciones de los sistemas de estimación de pitch es el *center clipping*. Éste consiste, básicamente, en anular los valores de $x[n]$ de magnitud pequeña. Con ello se pretende un doble objetivo: al introducir una distorsión no lineal, se aumenta la intensidad de los armónicos de orden elevado; y, al poner a cero los instantes de tiempo en los que la señal tiene una amplitud menor, se aumenta la robustez frente al ruido.

Existen dos variantes del clipping: con y sin offset. En el caso de aplicar offset, los segmentos de amplitud mayor que el umbral se desplazan hacia cero de manera que la función de transformación es continua:

$$x_c[n] = \begin{cases} x[n] - x_{th} & \text{si} \quad x[n] > x_{th} \\ x[n] + x_{th} & \text{si} \quad x[n] < -x_{th} \\ 0 & \text{resto} \end{cases}$$

En el caso de no aplicarse offset, sólo se modifican los valores cuya amplitud es menor que el umbral, que se ponen a cero, con lo que aparece una discontinuidad en la función de transformación:

$$x_c[n] = \begin{cases} x[n] & \text{si} \quad |x[n]| > x_{th} \\ 0 & \text{resto} \end{cases}$$



Puede aplicar esta técnica tanto a nivel global de la señal, el programa principal en `get_pitch.cpp`, o a nivel de trama, en la función `compute_pitch()` de `pitch_analyzer.cpp`.

II.A.B. Filtrado y diezmado de la señal.

Otra técnica sencilla que puede mejorar las prestaciones del sistema de estimación de pitch es el filtrado, tanto paso alto como paso bajo. En ambos casos lo que se pretende es eliminar de la señal tanto ruido como sea posible, sin afectar a la periodicidad de la señal útil.

El filtrado paso alto, con frecuencia de corte $f_{HP} \approx 50\,Hz$ suele resultar en una ligera mejoría en la estimación, sobre todo en presencia de ruido. Del mismo modo, un filtrado paso bajo con $f_{LP} > 2\,kHz$ es también útil ya que la frecuencia de corte es lo suficientemente alta como para no afectar en demasía a los armónicos de la señal, con lo que su periodicidad se mantiene intacta. Hay que señalar, no obstante, que el filtrado paso bajo no siempre es conveniente, sobre todo en el caso de trabajar con señal limpia.

El hecho de poder filtrar paso bajo la señal sin afectar demasiado a las prestaciones del estimador tiene la ventaja (a menudo, importante) de que permite diezmar la señal sin introducir *aliasing*. Este diezmado es fundamental para reducir los requisitos computacionales del sistema.

Tanto los posibles filtrados como el diezmado pueden implementarse de un modo sencillo y seguro usando la herramienta de procesado de audio SoX que ya se vio en la primera práctica.

II.B. Postprocesado.

II.B.A. Filtro de mediana

Un problema frecuente en los estimador de pitch es la presencia de errores *groseros* en la estimación de la frecuencia del pitch. Entre otros, hay dos tipos de error grosero que son muy frecuentes: la estimación de un múltiplo o de un submúltiplo de la frecuencia real.

En el primer caso, en función de la posición y amplitud de los formantes, el segundo o tercer armónico puede presentar una potencia muy grande, superior incluso a la del primer armónico. Aunque esto no es óbice para que, en condiciones ideales, el máximo de la autocorrelación siga correspondiendo a la frecuencia fundamental, en condiciones reales puede provocar el error en la estimación.

Por otro lado, en condiciones de alto ruido, es fácil que la señal sea más parecida a la señal desplazada dos o tres periodos de pitch que a la desplazada sólo uno, con lo que el máximo de la autocorrelación se puede encontrar a las frecuencias correspondientes a la mitad o un tercio de la real.

La técnica más habitual para reducir el ruido en una señal, el filtrado paso bajo, resulta contraproducente en estos casos, ya que, en lugar de eliminar el problema, lo que hacemos es desparramarlo en un intervalo de mayor duración. También es contraproducente el filtrado paso bajo en las transiciones entre sordo y sonoro, y viceversa, ya que se produce un efecto de suavizado de las mismas.

El filtro de mediana es un filtro no lineal que evita este desparrame. Se basa en calcular el valor mediano en una ventana centrada en cada instante de tiempo. La tabla siguiente muestra el comportamiento de ambos filtros (de longitud tres) en una situación con errores de este tipo, demostrándose la utilidad del filtro de mediana.

cuidado de no hacer un filtro recursivo!!!

Pitch real	0	0	100	100	100	100	100	100	100	100	100	0	0	0	0
$\hat{p}[i]$	0	0	100	100	200	100	100	100	50	100	100	0	0	68	0
Filtro LP	0	25	75	125	150	125	100	87.5	75	87.5	75	25	17	34	17
Mediana	0	0	100	100	100	100	100	100	100	100	100	0	0	0	0

Como puede verse, en este ejemplo (de lo más tramposo), el filtro de mediana consigue corregir los tres errores cometidos por el estimador de autocorrelación (marcados en rojo), mientras que el filtrado lineal paso bajo sólo consigue aumentar el número de tramas mal estimadas. Debe señalarse, no obstante, que su aplicación, cuando no hay errores groseros, puede incrementar el error fino de la estimación.

II.C. Métodos alternativos de estimación de pitch.

El análisis cepstral es una técnica ampliamente usada en procesado de audio y voz. Consiste, básicamente, en utilizar el espectro de potencia en forma logarítmica. Si aplicamos transformada de Fourier inversa a este espectro logarítmico, obtenemos el denominado *cepstrum real*:

$$x[n] \xrightarrow{\mathcal{F}\{\cdot\}} X[k] \xrightarrow{|\cdot|} |X[k]| \xrightarrow{\log(\cdot)} \log|X[k]| \xrightarrow{\mathcal{F}^{-1}\{\cdot\}} c[n]$$

Una de las características del cepstrum es que, al igual que en el caso de la autocorrelación, la posición del primer máximo secundario indica el valor del periodo de una señal periódica. También como en el caso de la autocorrelación, la relación entre el valor del cepstrum en su máximo secundario y en el origen puede ser usado para determinar si la señal es periódica o no.

Puede utilizar la librería `FFTReal` para implementar un estimador basado en el cepstrum. Se proporciona el programa `test_fft` para ilustrar el cálculo de la FFT con ella. Añada el código de la transformación en los ficheros `cepstrum.cpp` y `cepstrum.h` en las carpetas `pav` e `include`, e incluya el código fuente en `meson.build` para que se incluya en la librería `libpav.a`.

II.C.A. *Dynamic time warping* (DTW).

El ajuste temporal usando *Dynamic time warping* (DTW) es una metodología dirigida, en principio, al alineado de dos secuencias que se desarrollan paralelamente pero con velocidades que varían en cada punto. En estimación de pitch se puede usar una variante de DTW para transformar el problema de qué pitch tenemos en cada instante en un problema de qué secuencia de pitches justifica mejor los valores medidos. Se menciona, sobre todo, a nivel informativo, ya que su implementación práctica sobrepasa los objetivos de este trabajo.

II.C.B. *Average Magnitude Difference Function* (AMDF).

La AMDF es una alternativa al uso de la autocorrelación que presenta dos grandes ventajas: en primer lugar, a menudo conduce a resultados más precisos en la estimación del pitch, reduciendo algo la tasa

de errores no groseros; en segundo, resulta computacionalmente ventajoso. Se basa en calcular la función siguiente:

$$\text{AMDF}[k] = \sum_i \big|x[i] - x[i+k]\big|$$

Esta suma es mínima cuando $x[i] = x[i+k]$, por tanto, el objetivo de determinar la frecuencia de pitch se reduce a localizar el valor mínimo de la función. Este mínimo resulta ser, habitualmente, mucho más agudo que el máximo de la autocorrelación, que presenta un comportamiento más bien suave. Por este motivo, la AMDF suele proporcionar resultados más precisos. No está tan claro, por contra, cómo determinar si el sonido es sonoro o sordo, así que se suele acudir a los mismos criterios que se usan con la autocorrelación, pero sin la necesidad de calcular ésta de manera completa, ya que sólo necesitamos los valores de $r_{xx}[0]$, $\rho_{xx}[1]$ y $\rho_{xx}[k_{\text{mín}}]$.

La gran ventaja de la AMDF en estimación de pitch es que su cálculo no precisa multiplicaciones, como ocurre con la autocorrelación, con lo que puede realizarse más rápidamente. Además, en todo momento podemos conocer el valor mínimo de la AMDF hasta entonces, con lo que, en cuanto la función supera este valor, podemos interrumpir el cálculo.

II.D. Inclusión VAD.

Podría, y en ocasiones se hace, combinarse la estimación de pitch con el estimador de voz de la práctica anterior, de forma que clasifique como sordo allá donde el VAD detecta silencio. No obstante, hay que tener en cuenta que, a los errores cometidos por nuestro estimador de pitch, habrá que añadir los cometidos por el VAD. Así pues, es difícil que esta técnica resulte en una mejora importante.

Cabe remarcar, no obstante, que este sería un caso en el que queda de manifiesto la diferente importancia relativa que tiene en un VAD la confusión de voz con silencio que la de silencio con voz. Así, podemos diseñar un sistema que tenga una alta sensibilidad en la estimación de la voz, de manera que el máximo de voz sea clasificado como tal; aunque la precisión no sea tan elevada, porque los segmentos de silencio clasificados como voz aún deben pasar al estimador de pitch que puede clasificarlos correctamente como sordos. Sin embargo, los segmentos de voz clasificados como silencio se perderán irremediablemente.