

# Detección de *pitch*

Antonio Bonafonte

## Resumen

En esta práctica:

- Se implementará el algoritmo básico de detección 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 los propios programas.
- Se evaluará el algoritmo comparando los resultados con unas *plantillas* de referencia.

## Detección de Pitch

En esta práctica se diseñará un sistema de detección de pitch basado en la autocorrelación.

El programa principal, `get_pitch.cpp` ya lee la señal de entrada, define el analizador de pitch, lo llama para cada trama, y escribe los resultados.

La clase `pitch_analyzer` (en los ficheros `.h`, `.cpp`) define el analizador de pitch, que deberá completar.

La función `wavfile_mono` encapsula las funciones de la librería `sndfile`, para leer ficheros de audio mono.

La clase `digital_filter` define un filtro digital (no el diseño). `FFTReal` implementa la transformada de Fourier para señales reales. El fichero `include/ffft/FFTReal_readme.txt` explica su uso. Y el programa `test_fft` es un ejemplo.

## 1. Instalación del entorno de trabajo *PAV*

En la práctica anterior se explicó como utilizar el comando `make` para facilitar la compilación separada. Aún así, si se cambian las *dependencias*, se debe modificar el fichero `makefile`. En esta se define un entorno, y se proporciona una utilidad, `make_wizard`,

para crear automáticamente los ficheros `makefile`, que a su vez crean las dependencias de forma automática.

El entorno que creemos se utilizará para el resto de las prácticas del curso. Básicamente, la organización que se propone, es la siguiente:

```
$ tree -d
.
├── bin
│   ├── debug
│   └── release
├── intermediate
├── lib
│   ├── debug
│   └── release
└── prj
    ├── get_pitch
    ├── html
    ├── include
    │   └── fft
    ├── make_wizard
    ├── pav
    ├── test_fft
    └── vad
```

- Todos los fuentes se guardan bajo la carpeta `prj`, donde habrá una carpeta (`prj/pav`) con una librería con las funciones accesibles desde cualquier *proyecto*. Además los *headers* de las funciones de esa librería estarán en la carpeta `prj/include`. Por otra parte, se creará una carpeta para cada *proyecto*, con los fuentes correspondientes a un (o varios) programa. En la carpeta de la librería o en la de los proyectos, el fichero `makefile` indica qué fuentes se han de compilar y cuál es el nombre del fichero ejecutable resultado de la compilación.
- Al compilar, las librerías se crean en la carpeta `lib` y los ejecutables en la carpeta `bin`, al mismo nivel que `prj`. Podrá crear dos versiones, la `release`, o versión optimizada por el compilador, o `debug` menos eficiente pero utilizable por *depuradores* como `ddd` o `gdb`.
- También se crea al mismo nivel la carpeta `intermediate` con los ficheros `.o` resultados de compilar y otros ficheros *intermedios*.
- La carpeta `prj` es la única carpeta que debe guardar al realizar copias de seguridad o para tener su entorno en otro ordenador. Su tamaño es pequeño y el resto de carpetas (ejecutables, etc.) se genera con el comando `make`.

Los ficheros compilados (`.o`, `.a`, y programas binarios), que estarán en las carpetas `bin` y `lib` pueden no ser compatibles en distintas máquinas (por ejemplo, `x86 (32b) → x64`), así que mejor no copiarlas (o borrar esas carpetas `bin`, `lib`) para que se recompilen.

Si cambiar la ruta del directorio `prj`, por ejemplo, si cambia de ordenador, **debe borrar la carpeta `intermediate`** pues se crean unos ficheros de texto con las *dependencias*, `.d`, que incluyen rutas absolutas.

1. En primer lugar copie y descomprima los ficheros proporcionados. Es importante que la ruta de la carpeta elegida no incluya el espacio. Verá que el software incluye una carpeta `prj`, que a su vez incluye varias carpetas.

Incluya en esta organización los ficheros de la práctica de detección de detección

de voz, VAD:

- Los ficheros que implementan las funciones de detección, `vad.c`, `pav_analysis.c`, añádalo en la carpeta `prj/pav`.
  - Los ficheros con las declaraciones de las funciones, `vad.h`, `pav_analysis.h`, en la carpeta `prj/include`.
  - Para el fichero con el programa principal, cree una carpeta, por ejemplo, `prj/vad`.
2. Como vimos en la práctica anterior, el fichero `makefile` permite simplificar tareas como la compilación. Allí se especifican las *dependencias* de cada tarea minimizando las tareas de compilación a sólo las necesarias, según los cambios realizados.

Ejecute `make` desde la carpeta `prj/pav` para crear la librería. Las opciones de `debug` o `release` cambian las opciones de optimización del compilador. El `makefile` de ese directorio indica que se han de compilar todos los ficheros fuentes que se encuentren, por lo que si quiere añadir nuevas funciones a la librería, debe añadir aquí el programa en C o C++ (extensiones `.c` y `.cpp`), los correspondientes *headers* en la carpeta `include` y ejecutar `make`

3. Para compilar todos los directorios, puede situarse en cada uno de ellos y ejecutar `make`, tal y como hemos hecho con la librería de `pav`. Alternativamente, sitúese en la carpeta `prj` y ejecute `make`. El `makefile` de este directorio básicamente consiste en una lista de directorios. `make` entiende que debe situarse en cada uno de ellos (por ejemplo `pav`) y ejecutar `make`.

Cada vez que cree un directorio que deba compilarse (por ejemplo el directorio `vad`), edite el `makefile` del directorio `prj` para añadir ese directorio.

Desde `prj` también puede ejecutar `make doc`: el programa `doxygen` analiza los ficheros `.h`, `.cpp`, y `.c`, y crea documentación de funciones, variables, etc.

4. Para compilar el programa `vad` necesitamos un fichero `makefile` en la carpeta del programa. La elaboración de estos ficheros `makefile` es algo laboriosa, por lo que los entornos de desarrollo lo hacen automáticamente, analizando las cabeceras que incluye cada fuente. En esta práctica se incluye una herramienta, `make_wizard` para *generar* automáticamente `makefiles`.

Para que `make_wizard` sea accesible desde cualquier directorio, vamos a crear un *enlace simbólico* en la carpeta `bin/release`, que es donde tendremos todos los ejecutables del curso. Aunque la instrucción directa es sencilla (`ln -s`), para automatizarla se ha indicado en el fichero `makefile` del directorio de `make_wizard`. Para ejecutarlo, vaya a la carpeta `prj/make_wizard` y escriba `make`. Elija la opción `release`. Compruebe que en la carpeta `bin/release` tiene un *enlace* a `make_wizard`<sup>1</sup>

5. Para ejecutar este y otros programas sin tener que escribir la ruta, se puede cambiar la variable `PATH`. Típicamente, esto se realiza en el fichero `.bashrc`<sup>2</sup> de su directorio `$HOME`, ya que este fichero se ejecuta cada vez que abrimos un terminal. Edítelo (o créelo) y añada:

<sup>1</sup>Como `make_wizard.pl` es un script en `perl`, que no necesita compilarse, sólo se copia en la carpeta `bin` para tener allí todos los ejecutables y facilitar su acceso.

<sup>2</sup>En Linux los ficheros que empiezan por punto se consideran *ocultos*. Puede verlos mediante `ls -a`, o seleccionando la opción de mostrar ficheros ocultos en el navegador de ficheros

```
export PATH=".:directorio_donde_esta_bin_release:$PATH"
```

Después de hacerlo abra un nuevo terminal<sup>3</sup>, y escriba `make_wizard` para comprobar que la variable `PATH` es correcta y encuentra nuestra carpeta de ejecutables.

6. Ahora que ya podemos acceder a la herramienta `make_wizard` vamos a usarla para crear el `makefile` para el programa de detección de voz. Vaya a la carpeta donde se encuentra su programa principal. Ejecute `make_wizard` para crear un `makefile` adecuado (opción `-p1`, para crear 1 programa. Puede editar el fichero `makefile`, aunque por defecto, compilará todos los fuentes del directorio y le da al ejecutable el nombre de la carpeta. Ejecute `make`. Compruebe que puede ejecutar el programa de detección de voz, `vad`.

También puede añadir el directorio `vad` en el `makefile` de `prj`, para que se compile, si ha habido cambios, al ejecutar `make` desde el directorio global `prj`.

## 2. Tareas

1. Mientras realizar las primeras tareas de la práctica, instale el programa `doxygen`, que se puede utilizar para documentar programas: `sudo apt-get install doxygen`.
2. Visualize el fichero `pitch_analyzer.h`. Entienda el funcionamiento de la clase que define. Consulte también el fichero `wavfile_mono.h`.
3. Una de las herramientas para documentar código fuente es `doxygen`. Se escriben comentarios en el código, con cierta sintaxis, y el programa `doxygen` documenta clases y funciones, puede definir diagramas con jerarquías, etc. Ejecute `make doc` para generar la documentación. Mire como se documentan los ficheros fuente con `doxygen` y visualice el fichero `index.html` generado. Puede editar el fichero de configuración, `Doxygen`, para tener otros formatos de salida, por ejemplo `latex`, que después deberá compilar con su propio `makefile`.
4. Analice el programa principal para entender su funcionamiento.
5. Hay varias funciones, en el fichero `pitch_analyzer.cpp` que están por finalizar: ventana `hamming`, cálculo de la correlación, búsqueda del máximo, etc. Edite este fichero y complete el código.
6. Una de las funciones debe determinar si un sonido es sonoro o si se trata de un tramo sordo, o de ruido. La detección de sonoridad puede basarse en la energía ( $R[0]$ ), así como en la relación entre  $R[1]/R[0]$  o  $R[Npitch]/R[0]$ . Imprima estos valores, visualícelos con `wavesurfer`<sup>4</sup> e implemente una regla de detección de sonoridad. Otra característica sencilla de calcular que puede ser útil, son los cruces por cero.
7. Visualice el resultado de su análisis de pitch en `wavesurfer`. Compare con la estimación del propio programa, panel pitch contour.
8. Una vez finalizado su analizador de pitch, la carpeta `evaluation` incluirá las instrucciones para realizar la evaluación.

<sup>3</sup>Alternativamente, ejecute `source $HOME/.bashrc`

<sup>4</sup>Recuerde configurar el panel de datos para que cada dato corresponda a un tramo, que en esta práctica son 15 ms.

## Ampliaciones

Puede mejorar el algoritmo de detección de pitch, utilizando técnicas de *pre-procesado* y/o *postprocesado*.

### Post-procesado

El *post-procesado* es muy necesario. Una técnica muy sencilla y efectiva es el filtro de mediana, que puede utilizarse tanto para eliminar errores de la detección de sonoridad, como para corregir los valores de F0 de los tramos sonoros.

### Preprocesado

Una técnica sencilla de *pre-procesado* es diezmar los ficheros a una frecuencia de muestreo de aprox. 2kHz. La señal es más sencilla y contiene varios armónicos que por lo que tiene la misma periodicidad. Puede utilizar el programa `sox`, antes de ejecutar su programa.

Otra técnica sencilla es el *center clipping*, que puede incluirlo en la función `compute_pitch()`. Puede añadir una variable de configuración (local del analizador, `bCenterClipping`), que se inicializa en el constructor, o añadir una función de control: `center_clipping(bool b)`

### Avanzado: detector cepstral.

Utilizar la rutina `FFTReal` para implementar un detector basado en el cepstrum: calcule el cepstrum real,

$$x[n] \xrightarrow{\mathcal{F}} X[k] \rightarrow |X[k]| \xrightarrow{\mathcal{F}^{-1}} c[n]$$

Se proporciona el programa `test_fft` para ilustrar el cálculo de la FFT con la librería `FFTReal`.

Puede añadir un programa `cepstrum.cpp` y `cepstrum.h` en las carpetas `pav` e `include`

A partir del cepstrum, busque el máximo de forma similar a como se hace con el método de la autocorrelación.

Para determinar la sonoridad puede comparar la relación entre el valor del pico y el cepstrum en el origen (que es el logaritmo de la energía).

Compare los resultados de ambos algoritmos.

**Inclusión VAD.** Puede combinar la detección de pitch con el detector de voz de la práctica anterior, de forma que donde hay silencio se imprima F0=0.