Práctica 2: detección de actividad vocal (VAD)

Antonio Bonafonte – profesores de la asignatura

marzo de 2020

Resumen

En esta práctica:

- Se implementará un detector de voz (VAD: *voice-activity detector*), sencillo y efectivo para situaciones de ruido moderado.
- Se verá como implementar en `C`, fácilmente, un sistema regido por un autómata de estados finitos (FSA), utilizando programación *orientada a objetos*.
- Se aprenderá a utilizar la librería `soundfile` que encapsula la lectura/escritura de ficheros de audio, independiente del formato de estos.
- Se realizará una evaluación del método desarrollado, participando en la producción de *datos de evaluación.*
- Se introducirá la programación en `bash` y la herramienta de compilación `Meson/Ninja`.
- Se introducirá el sistema de gestión de opciones y argumentos `docopt`.

Índice

1. Introducción y fundamento teórico.

1.1. Detección de voz usando umbrales.

En esta segunda práctica se utilizarán características de la señal de voz para realizar un detector de voz. La idea consiste en, dada una señal de voz, determinar en qué instantes tenemos realmente voz, y en cuáles sólo hay silencio (o ruido). Estos sistemas tienen varias aplicaciones, como por ejemplo en sistemas de teleconferencia, en sistemas codificación del habla, en el reconocimiento del habla, etc.

El criterio fundamental para discernir si un segmento es de voz o silencio es su potencia media. En situaciones de poco ruido, es de prever que la potencia de éste sea muy inferior a la de la voz. Por tanto, una primera aproximación al problema consistiría en calcular un umbral tal que, si la potencia supera este umbral, decidimos que el segmento contiene voz; y, si no lo supera, decidimos que contiene silencio.

Tareas:

Para analizar la capacidad de un detector de voz basado en medidas estadísticas de la señal, como las calculadas en la primera práctica, utilice una señal de voz (16kHz, mono) que contenga pausas internas (puede utilizar la señal grabada en la primera práctica o grabar una nueva al efecto).

Nombre al fichero `pav_ggLD.wav` (gg: grupo 11, 41, etc; L: id. del pc; D: índice de señal, si graba más de una: 1, 2, ...). Los ficheros grabados por todos los alumnos del curso se utilizarán para evaluar los distintos sistemas de detección de voz.

1. Tal y como hizo en la primera práctica, visualice en el programa `wavesurfer` la señal y las características potencia y tasa de cruces por cero. Cree un nuevo panel *transcription* y utilícelo para etiquetar los segmentos de silencio `(S)` y de voz `(V)`. Para ello, sitúese al final de cada segmento de voz o silencio e introduzca la etiqueta que corresponda. Sólo considere como silencio los segmentos de una cierta duración; no pausas cortas que no se perciban claramente.

   Guarde las transcripciones (fichero `.lab`) y suba los ficheros `.wav` y `.lab` a Atenea, para contribuir a la base de datos de evaluación.

2. Defina unas condiciones *razonables* para detectar la presencia de voz o silencio. Para ello, observe la señal y determine:

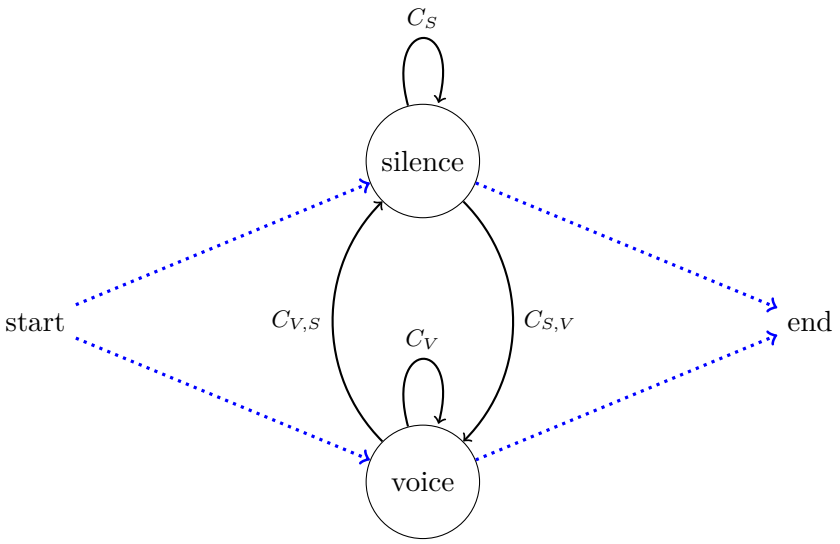
   - Nivel de potencia y tasa de cruces por cero en el silencio inicial de la señal.
   - Incremento aproximado del nivel de potencia, respecto al valor en el silencio inicial, que se tiene para distintos tipos de fonema: fricativas sordas, consonantes sonoras, vocales, etc.
     - ¿Es capaz de determinar un valor mínimo del incremento de nivel tal que, si se supera, podamos tener una cierta seguridad de que se trata de voz, pero, si no se hace, podamos considerar que se trata de silencio?
   - Duración mínima razonable de los segmentos de voz y de silencio.

3. Visualice con `less` o `cat` el fichero con la transcripción, e interprete el significado de su contenido.

1.2. Autómatas de estados finitos en la detección de voz.

La detección de voz basada en un umbral de potencia puede implementarse como un *autómata de estados finitos* (**FSA**, del inglés *Finite-State Automaton*). Un FSA es un modelo matemático en el que, para cada instante de tiempo, sólo podemos estar en un número finito de estados. En detección

de voz, estos estados son la voz (V) y el silencio (S). El FSA puede cambiar de un estado a otro en función de alguna variable externa, que, en nuestro caso, es la potencia media del segmento. El grafico siguiente muestra el FSA básico correspondiente a esta regla de decisión:



Los círculos representan los dos estados del FSA: *silence* y *voice*. Las flechas sólidas de color negro indican las posibles transiciones entre estos estados. Se realiza una transición por cada segmento de voz: $C_{S,V}$ indica que se pasa de silencio a voz; $C_{V,S}$, de voz a silencio; y C_S y C_V, que se permanece en el estado anterior. Las etiquetas *start* y *end*, y las flechas de rayas azules son, de hecho, externas al propio FSA, que se supone que se ejecuta por tiempo indefinido, y sirven para indicar los puntos de entrada y salida para secuencias de estados de longitud finita.

Este FSA, no obstante, presenta diversos problemas:

1. En principio, no se adapta el umbral de decisión a los niveles de la señal. Es decir, al menos en los primeros tramos de señal, deberemos tomar la decisión de si es voz o silencio en función de un umbral de potencia predeterminado. Sin embargo, el sistema debe funcionar con señales de cualquier nivel. Al fijar un umbral *a priori* corremos el riesgo de considerar a señales grabadas con un nivel muy bajo como completamente formadas por silencio, y a señales grabadas con un nivel elevado como completamente formadas por voz.

   Para evitarlo vamos a considerar que los primeros segmentos de la señal son siempre silencio y vamos a introducir un estado adicional, *init*, que usaremos para calcular el nivel de referencia del ruido de fondo. Como este nivel de referencia se verá afectado por el nivel de grabación en la misma medida que la señal de voz, podemos fijar el umbral de decisión en función del mismo; habitualmente, unos pocos decibelios por encima de él.

2. En esta primera aproximación, el sistema considerará silencio las breves pausas que se producen de manera natural en el habla normal. Por ejemplo, las consonantes oclusivas se caracterizan por la presencia de una breve interrupción en la señal de voz; pero esta pausa forma parte del propio fonema y no debería ser considerada como silencio.

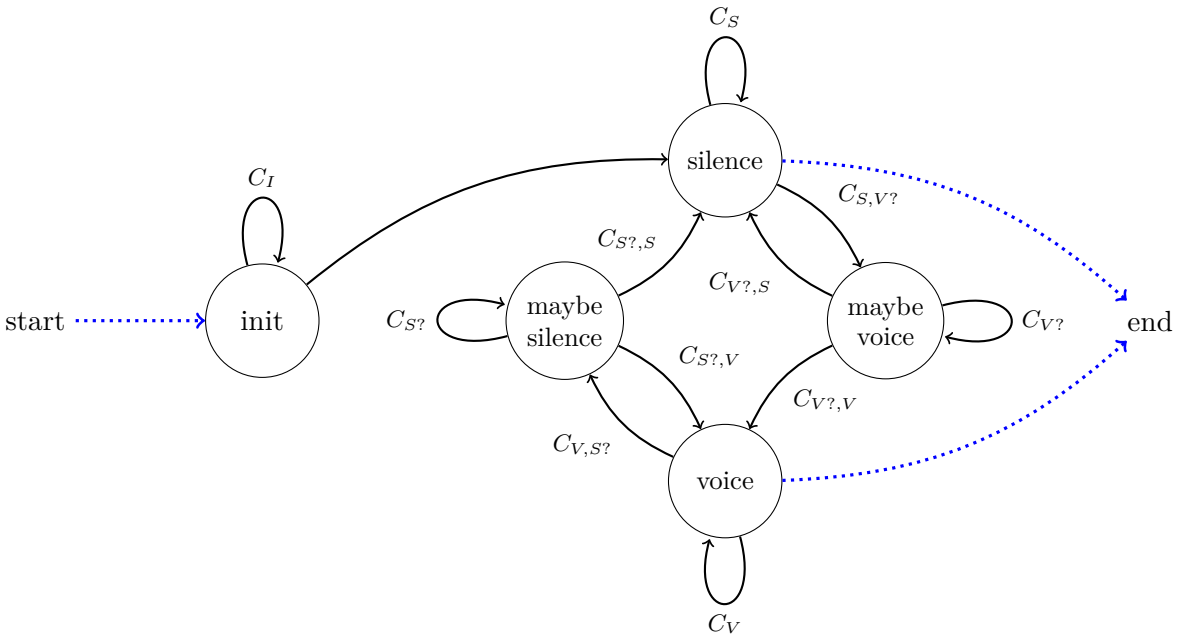
   Para evitar considerar las pausas breves como silencio creamos un nuevo estado: *maybe silence*. Este estado se alcanza cuando, estando en el estado V, el nivel de potencia de la señal baja por debajo del umbral de decisión. Sólo si nos mantenemos en esta situación un cierto tiempo mínimo consideraremos que se ha pasado de voz a silencio, y que éste empezó cuando se entró en el estado *maybe silence*.

3. El sistema también será sensible a los ruidos de corta duración que se producen de manera natural en una grabación: clicks, chasquidos de lengua, etc.

   Para evitar considerar estos ruidos como inicio de voz, creamos un nuevo estado, *maybe voice*. Este estado se alcanza cuando, estando en el estado S, el nivel de señal sube por encima del

umbral. Análogamente al caso del *maybe silence*, sólo consideraremos que realmente se trata de voz si se supera un cierto tiempo en él, y, en caso de hacerse, consideraremos que la voz empezó cuando se entró en *maybe voice*.

El esquema correspondiente a este nuevo FSA es:



1.3. Mejoras en la detección.

1.3.1. Nivel de referencia del ruido de fondo.

En principio, el sistema de detección de voz mostrado en la sección precedente se basa en determinar un nivel de referencia para el ruido de fondo, que se calcula utilizando los N_{init} primeros tramos de la señal. Este nivel de referencia, en adelante k_0, puede calcularse de diversas maneras. Siendo $P_{dB}[i]$ la potencia media en decibelios de la trama i-ésima, podemos utilizar:

Potencia media inicial:
Calculada como el valor en decibelios de la potencia media durante las N_{init} primeras tramas:

$$k_0 = 10 \log_{10} \left(\frac{1}{N_{init}} \sum_{i=0}^{N_{init}-1} 10^{P_{dB}[i]/10} \right) \tag{1}$$

Media de las potencias en decibelios:
Una alternativa más sencilla a la anterior consiste en utilizar, simplemente, el valor medio de $P_{dB}[i]$. Esta medida es equivalente al valor en decibelios de la media geométrica de las potencias medias. Esto puede resultar peligroso si alguna de las tramas tiene un nivel de potencia muy bajo, porque eso provocará una estimación igualmente muy baja del nivel de potencia del ruido.

Valor máximo de la potencia:
Si estamos seguros de que las N_{init} tramas iniciales contienen silencio, una posible elección para el valor de k_0 es el valor máximo de $P_{dB}[i]$ para todas ellas. Esta elección nos proporciona el valor de nivel de ruido alcanzado, pero no superado, en el segmento inicial. Es, por tanto, una elección *pesimista*.

1.3.2. Elección de los umbrales de decisión.

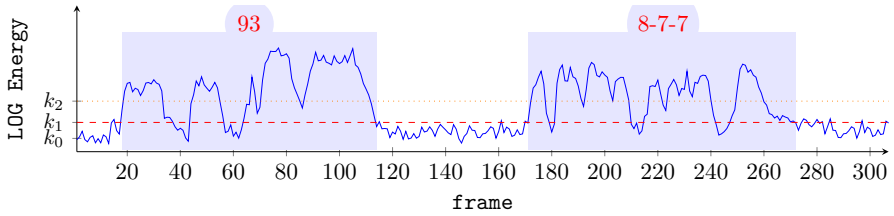
También en principio, la decisión de si la trama se corresponde a voz o silencio se basa en comparar el nivel de potencia de la misma con un umbral, en adelante k_1. Usando la potencia en decibelios, una elección razonable es hacer $k_1 = k_0 + \alpha_1$, con un margen α_1 adecuado. Esto nos lleva a reglas de decisión del tipo: si $P_{dB}[i] > k_1$, entonces la trama i-ésima es *voice*, si el estado anterior era voz; *maybe voice*, si el estado era silencio; y *voice* o *maybe voice*, si el estado era *maybe voice* (uno u otro en función de cuánto tiempo llevemos en ese estado).

Una mejora habitual en los sistemas de detección de voz consiste en introducir una cierta *histéresis* en la decisión. Esto se consigue definiendo un segundo umbral, $k_2 = k_1 + \alpha_2$ de tal modo que la regla de decisión para decidir si se trata de *maybe voice* continúa siendo $P_{dB}[i] > k_1$, pero, para confirmar que se trata de voz, exigimos que $P_{dB}[i]$ sea mayor que k_2 antes de un cierto tiempo. Si en ese tiempo no se alcanza un valor $P_{dB}[i] > k_2$, entonces consideramos que el intervalo era silencioso.

La justificación de esta modificación radica en que el nivel de potencia de algunos sonidos (particularmente las consonantes fricativas sordas como la <z> en 'caza' o la <f> en 'foca') es muy bajo. No obstante, ningún mensaje oral puede mantenerse mucho tiempo en este tipo de fonema: en poco tiempo hemos de pasar a otro sonido que, en general, será de nivel de potencia mucho mayor (como las vocales). El nivel k_1 debe ser, por tanto, algo menor que el nivel de estas fricativas sordas (para poderlas considerar como voz); y el tiempo máximo de espera antes de alcanzar el nivel k_2 debe ser su duración máxima esperada.

Esta modificación se hace especialmente necesaria en los extremos de segmentos de voz. Si el segmento viene precedido de silencio, pero empieza con un *maybe voice* de corta duración y seguido inmediatamente de un *voice*, es muy posible que el *maybe voice* deba ser considerado voz. Y lo mismo ocurre si el segmento de voz acaba con un *maybe silence* de corta duración y seguido de *silence*. Sin embargo, si la duración de estos segmentos *maybe silence* es elevada, la probabilidad de que realmente se trate de voz, y no de un ruido de fondo de intensidad elevada, es mucho menor.

La gráfica siguiente muestra esta idea aplicada a una señal concreta: el inicio de la pronunciación de un número de teléfono (93 8 7 7 ...) por un locutor real. En la gráfica, y de acuerdo con lo acabado de explicar, k_0 se corresponde con el nivel medio de potencia del segmento inicial de la señal, k_1 es el nivel de posibilidad de voz, y k_2 es el nivel de confirmación de voz.



En la gráfica podemos ver un caso en el que el uso de un nivel intermedio k_1 permite *rescatar* como voz un trozo de fricativa sorda, el final de la [s] de la palabra 'tres', que, de otro modo, se consideraría silencio. Sin embargo, en el caso de la [e] final del segundo 'siete', esta estrategia provoca que se considere como voz lo que claramente es silencio. Seguramente este error está debido a que el nivel de ruido de fondo parece aumentar durante el transcurso de la señal. Si no fuera así, es bastante probable que el sistema hubiera detectado correctamente el silencio final. Pero la situación creada en esta señal sí sirve para ilustrar lo delicado que será escoger los umbrales y duraciones de manera cuidadosa.

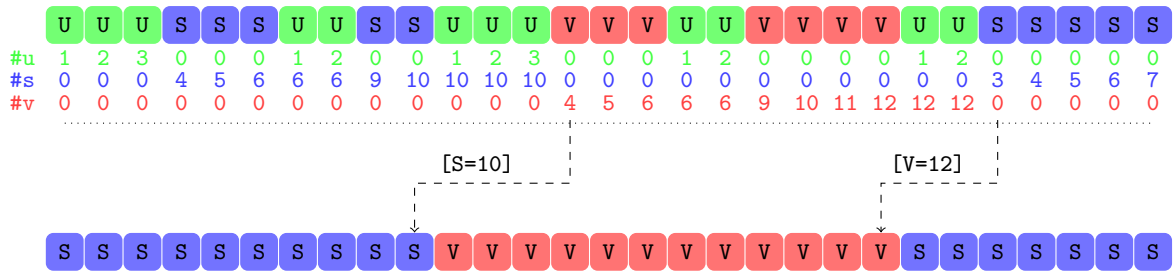
1.3.3. Decisión diferida.

Los criterios indicados en la sección precedente implican que hay situaciones en las que el sistema no es capaz de proporcionar una respuesta inmediata a la cuestión de si un segmento es de voz o silencio. Si el estado precedente es *voice* y la potencia del segmento actual es elevada, entonce podemos estar seguros de que el estado actual es *voice*. Análogamente, si el estado precedente es *silence* y la potencia

del segmento actual es baja, podemos estar seguros de que el estado actual sigue siendo *silence*. Pero, en el resto de las situaciones, o bien el estado queda indefinido (*maybe voice* o *maybe silence*), o bien el cumplimiento o incumplimiento de los criterios temporales sirve para resolver una indefinición previa.

Por ejemplo: si el estado anterior es *voice* y el nivel de potencia del segmento actual es bajo, el estado actual será *maybe silence*, pero no podremos estar seguros de si será silencio o no hasta que no sepamos cuánto tiempo permanecemos en ese estado. Por contra, si el estado actual es *maybe silence* y el nivel de potencia es bajo, pero el número de segmentos de baja potencia, incluyendo el actual, alcanza el número mínimo necesario para confirmar el silencio, tanto el segmento actual como los segmentos precedentes considerados *maybe silence* se clasifican como silencio.

La gráfica siguiente muestra el funcionamiento de esta *decisión diferida*. En ella, los dos estados provisionales, *maybe silence* y *maybe voice*, se representan con el mismo estado `U` (*undefined*).



1.4. Características de la señal apropiadas para la detección de voz.

Las características básicas que pueden utilizarse son:

- Potencia del tramo de señal, expresado habitualmente en dB.
- Amplitud media por tramo. Varios sistemas utilizan este valor; seguramente, porque es un poco más sencillo definir umbrales a partir del mismo. Pero hay que recordar que ésta es una medida lineal, y no logarítmica (como la potencia en dB). Por tanto, los umbrales se definen multiplicando el valor de referencia por los factores adecuados, en lugar de sumando estos factores.
- Tasa de cruces por cero (*zcr*). Esta característica puede ayudar a discernir silencio (ruido) de señales fricativas (`/s/, /f/`) de baja energía (con mayor contenido frecuencial, mayor *zrc*).
- Duración mínima de silencio, para evitar cortes en la mitad de palabras o separar palabras pronunciadas de forma continua.
- Duración mínima de la voz, para evitar identificar como voz ruidos de corta duración, como golpes en el micrófono, etc.
- Duración máxima de los estados *maybe voice* o *maybe silence* para considerar que podría tratarse de fonemas de baja potencia en los extremos de segmentos de voz.

Las funciones que calculan alguna de estas características, y otras que se puedan considerar, se encuentran en el fichero `pav_analysis.c`. Por tanto, será necesario incorporar este fichero al proyecto de esta práctica.

2. Implementación del detector de voz (VAD).

2.1. Programación orientada a objetos.

En la práctica vamos a desarrollar un programa que realice esta detección usando programación orientada a objeto (**OOP**). Este paradigma de programación se basa en la creación de estructuras de datos específicas (*objetos*), que son manejadas por funciones igualmente específicas (*métodos*). Desde el punto de vista del programador que usa los objetos, su estructura interna no es relevante; sólo los

interfaces con ellos, los métodos, lo son. La idea fundamental de la OOP es el *encapsulamiento* de los objetos y sus métodos en estructuras denominadas *clases*. Otras características relevantes son la herencia, el polimorfismo, etc.

Existen múltiples lenguajes de programación que proporcionan herramientas específicas para OOP: `C++`, `Java`, `Python`, etc. En `C` no se dispone de esas herramientas. No obstante, eso no impide utilizar buena parte de los conceptos básicos de la OOP utilizando `C`. Un ejemplo de ello es la librería `<stdio.h>`. En ella se define el *struct* `FILE`, que gestiona la lectura/escritura. Apenas los implementadores de la librería conocen los entresijos de `FILE`; al resto de usuarios le basta con conocer las funciones que los manejan: `fopen()`, `fclose()`, `fprintf()`, etc.

2.2. Estructura del proyecto.

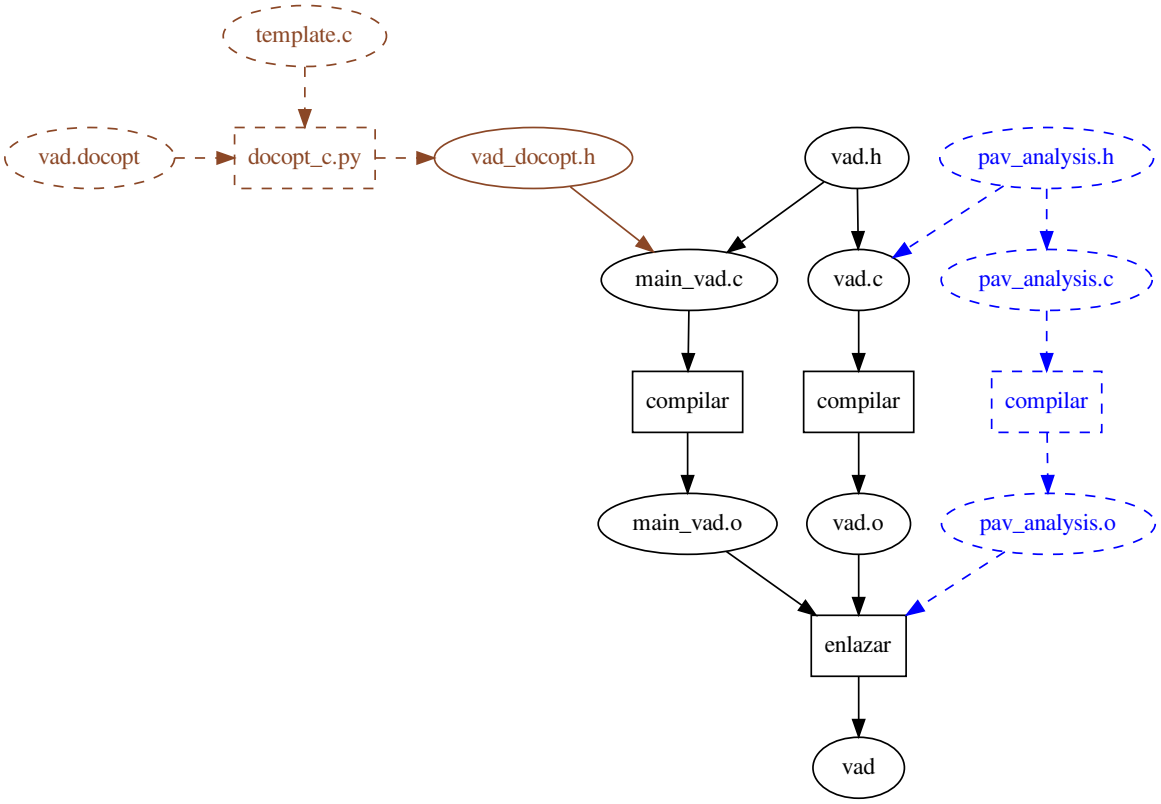
Siga las instrucciones del fichero README.md en el repositorio GitHub de la Práctica P2 para clonar los ficheros de la práctica en su directorio raíz de las prácticas `~/PAV`.

Después de clonar el repositorio en el directorio `PAV`, la estructura del directorio es la siguiente:

```
PAV
├── P1
│   ├── . . .
│    . . . . .
│   └── . . .
└── P2
    ├── MARKDOWN.md
    ├── README.md
    ├── p2_vad.pdf
    ├── db.v4
    │   ├── 2014
    │   │   ├── pav_4151.lab
    │   │   ├── pav_4151.wav
    │   │   ├── pav_4152.lab
    │   │   ├── pav_4152.wav
    │   │    . . .
    │   │   ├── pav_4341.lab
    │   │   └── pav_4341.wav
    │    . . .
    │   └── 2018-19q1
    │       └── . . .
    ├── docopt_c
    │   ├── docopt.py
    │   ├── docopt_c.py
    │   └── template.c
    ├── scripts
    │   ├── run_vad.sh
    │   └── vad_evaluation.pl
    └── src
        ├── main_vad.c
        ├── vad.c
        ├── vad.docopt
        ├── vad_docopt.h
        └── vad.h
```

El código fuente fundamental está ubicado en el directorio `PAV/P2/src`, y está formado por los ficheros `main_vad.c`, `vad.h` y `vad.c`. Puede encontrar el código de estos ficheros en el Anexo III.

La gráfica siguiente muestra la estructura del código fuente:



La parte de la izquierda, en color marrón, se encarga de la gestión las opciones y argumentos del programa. Usa el programa Python `docopt_c.py` para generar el fichero `vad_docopt.h`, que es incluido por el programa principal, `main_vad.c`. En la versión inicial, se proporciona directamente el fichero `vad_docopt.h`, por lo que, en principio, no hay que preocuparse más de esta parte. No obstante, puede ser muy interesante (pero mucho) modificar las opciones del programa para poder automatizar los experimentos con el VAD. En el Anexo II dispone de más información acerca de cómo hacerlo.

La parte de la derecha, dibujada en color azul, no se proporciona en la versión inicial del proyecto. En caso de utilizarse las funciones definidas en la primera práctica para el análisis de la señal, será necesario incorporar al proyecto los ficheros `pav_analysis.c` y `pav_analysis.h` de la primera práctica.

2.2.1. Programa principal: `main_vad.c`

Este fichero contiene la función `main()` del programa `vad`. Esta función, cuyo código puede consultarse en el Anexo III.A, realiza principalmente las siguientes tareas:

1. Analiza los argumentos de la invocación.
2. Gestiona los ficheros de entrada y salida, y la memoria necesaria para los vectores y estructuras utilizados por el programa.
3. Realiza el bucle principal del programa, determinando el estado que le corresponde a cada segmento de la señal de entrada, y escribiendo el resultado en el fichero (o ficheros) correspondiente.
4. Libera los recursos utilizados por el programa.

Librería `sndfile`: La gestión de la lectura y, en su caso, escritura de los ficheros de audio en formato WAVE se realiza usando funciones de la librería `sndfile`. En la práctica anterior, estas tareas se realizaron utilizando las funciones generales de lectura de ficheros binarios. Este procedimiento presenta varios inconvenientes:

- No se realizó una interpretación completa de los datos de la cabecera (número de canales, frecuencia de muestreo, formato de las muestras, etc.).
- Sólo se implementó la lectura de ficheros WAVE codificados con PCM lineal de 16 bits.
- La conversión de las muestras del formato entero de dos bytes (`short`) a real (`float`) se realizó en el cuerpo del programa principal.

En esta práctica se utiliza la librería `sndfile`, que encapsula todas estas funcionalidades y, además, permite leer y escribir audio en una gran variedad de formatos (incluyendo `WAVE`). `sndfile` permite manejar los ficheros de audio utilizando funciones con interfaces muy parecidos a los de las funciones usadas leer ficheros binarios:

- Fichero binario: `fopen(), fread(), fwrite(), fclose(), fseek()`
- Fichero de sonido: `sf_open(), sf_read_xxx(), sf_write_xxx(), sf_close(), sf_seek()`

Donde la partícula `xxx` en las funciones `sf_read_xxx()` y `sf_write_xxx()` indica el tipo de datos usado en el programa para representar las muestras de señal. Así, por ejemplo, la función `sf_write_short()` sería la utilizada para escribir las muestras cuando éstas se representan usando enteros de dos bytes. Nótese que esto es independiente de cómo se almacenan en disco, que viene indicado al abrir el fichero mediante el campo `format` en la estructura `SF_INFO`. Puede consultarse la documentación de estas funciones en:

http://www.mega-nerd.com/libsndfile/
http://www.mega-nerd.com/libsndfile/api.html

Si la librería `sndfile` no estuviera instalada en su sistema, deberá instalar el paquete `libsndfile1-dev`. Por ejemplo, en Ubuntu deberá ejecutar:

```
usuario:~/PAV/P2$ sudo apt-get install libsndfile1-dev
```

2.2.2. Definición del FSA: `vad.h`

En un programa escrito utilizando conceptos de programación orientada a objetos, el elemento principal es la definición de las estructuras de datos. En el caso de este proyecto, el tipo de datos fundamental es la estructura `VAD_DATA`, definida en el fichero `vad.h` (Anexo III.B).

En su versión inicial, el fichero `vad.h` sólo incluye una versión muy básica de la estructura `VAD_DATA`:

`frame_length`: Longitud de la trama en muestras.

`sampling_rate`: Frecuencia de muestreo en hercios, usada en combinación con `frame_length` para convertir el índice del segmento a segundos.

`state`: Estado del segmento anterior, usado para determinar el estado del actual.

`last_feature`: Variable usada para almacenar el valor de la característica, en principio la potencia, y hacerla disponible al programa principal; particularmente para efectos de depuración.

El programador del VAD deberá modificar esta estructura para incluir los campos necesarios para gestionar el FSA: umbrales (k_0, k_1...), duraciones ($lMín_{sil}$, $lMín_{voz}$...), etc.

En `vad.h` se incluye también los prototipos de las funciones de gestión del FSA y se define el tipo `VAD_STATE` con la enumeración de los posibles estados del FSA.

2.2.3. Implementación del FSA: `vad.c`

El fichero `vad.c` (Anexo III.B) incluye la función principal del proyecto: `vad()`, que implementa el FSA, devolviendo, para cada trama de señal de entrada, el estado que le corresponde. En su versión actual no hace nada sensato; es responsabilidad del alumno implementar un FSA que permita detectar la voz y el silencio con un mínimo de solvencia.

El fichero incluye, también, el resto de funciones usadas en la gestión del FSA: `compute_features()`, `vad_open()`, etc.

3. Mantenimiento del proyecto usando Meson/Ninja.

El programa `meson` se basa en ficheros de dependencias, semejantes a los `makefile` de `make`, denominados `meson.build`. Como en el caso de `makefile`, `meson.build` debe existir en el directorio raíz del código que deseamos mantener. Sin embargo, aparte de permitir muchas más funcionalidades que `make`, hay una serie de diferencias fundamentales que ya nos encontramos al mantener un proyecto tan sencillo como el presente:

- El fichero `meson.build` está escrito en un lenguaje propio semejante a Python.
- Existe el concepto de proyecto. Todo fichero `meson.build` debe iniciar con una línea con la descripción básica del proyecto. En principio, sólo es necesario indicar el nombre del proyecto y el lenguaje (o lenguajes) de programación usado, pero también es posible indicar la versión, el tipo de licencia bajo el que se distribuye, etc.
- Meson mantiene el código contenido en el directorio en el que se encuentra el fichero `meson.build`, que puede acceder, a su vez, a otros directorios. Sin embargo, la compilación y enlazado se realizan en un directorio distinto, que incluirá todos los ficheros temporales. De este modo, los directorios con el código original quedan *limpios*; pueden realizarse distintas versiones del proyecto (por ejemplo, cambiando las opciones de compilación); y puede eliminarse completamente una compilación con sólo eliminar el directorio que la contiene.
- No es necesario especificar los ficheros incluidos por otros (por ejemplo: las cabeceras incluidas en ficheros `.c`). Sólo es necesario especificar los códigos fuente. Al realizarse la primera compilación del proyecto, `meson`[1] identifica los ficheros incluidos y se encarga de comprobar si están al día. Si, al modificar el código fuente, los ficheros incluidos varían, `meson` es capaz de darse cuenta de ello y obrar en consecuencia.

Con esto en mente, construimos el `meson.build` del proyecto `vad` con los elementos siguientes:

1. La declaración del nombre del proyecto y del lenguaje usado para escribirlo, indicados en la directiva `project()` de `meson.build`.
2. Para facilitar la legibilidad y mantenimiento, definimos una serie de variables con los elementos que participan en la creación del programa:

`exe`	Nombre del programa a generar, que es el objetivo del proyecto.
`src`	Los códigos fuentes que forman el programa.
`lib`	Las librerías con las que queremos enlazarlo.

3. El comando de generación del objetivo. Meson reconoce tres tipos básicos de objetivo: librerías estáticas y dinámicas, y programas ejecutables.

[1] De hecho, no es `meson` quien lo hace, sino `ninja`.

- En nuestro caso, el objetivo es un programa ejecutable, que se indica con la directiva `executable()`.
- A `executable()` le pasamos el nombre del objetivo y los códigos fuente que hay que compilar para obtenerlo. Como se trata de un proyecto programado en C (tal y como se declara en la directiva `project`), `meson` reconocerá también opciones específicas de este tipo de proyecto. En concreto, reconoce la variable `link_args`, que podemos usar para pasarle las librerías con las que queremos enlazar el programa.

4. Fijémonos en que no es necesario pasarle el nombre de las cabeceras incluidas en los códigos fuente. La primera vez que compilemos el proyecto, `meson/ninja` se dará cuenta de cuáles son y, a partir de entonces, los incorporará automáticamente a las dependencias del programa.

meson.build

```
project('Práctica 2 de PAV - detección de actividad vocal', 'c')

exe = 'vad'
src = ['src/main_vad.c', 'src/vad.c']
lib = ['-lm', '-lsndfile']

executable(exe, sources: src, link_args: lib)
```

Hemos de ejecutar `meson` una única vez desde el directorio en el que se encuentra el fichero `meson.build` (en este caso, `PAV/P2`), indicando el directorio en el que queremos generar el proyecto. No es necesario que el directorio del proyecto exista antes de ejecutar `meson`, pero no está permitido ejecutarlo dos veces en el mismo directorio. En este caso, vamos a configurar el directorio `bin`:

```
usuario:~/PAV/P2$ meson bin
The Meson build system
Version: 0.45.1
Source dir: /mnt/d/usuario/UbuntuOnWindows/PAV/P2
Build dir: /mnt/d/usuario/UbuntuOnWindows/PAV/P2/bin
Build type: native build
Project name: Práctica 2 de PAV - detección de actividad vocal
Native C compiler: cc (gcc 7.4.0 "cc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0")
Build machine cpu family: x86_64
Build machine cpu: x86_64
Build targets in project: 1
Found ninja-1.8.2 at /usr/bin/ninja
```

Una vez configurado el directorio, hemos de ejecutar `ninja` para compilar el programa. Este comando puede ejecutarse en el propio directorio de la instalación (que, en nuestro caso, será `PAV/P2/bin`), o podemos indicárselo en línea de comandos con la opción `-C bin`:

```
usuario:~/PAV/P2$ ninja -C bin
ninja: Entering directory 'bin'
[3/3] Linking target vad.
```

Si queremos ver qué operaciones realiza en cada paso (en caso contrario, `ninja` es sumamente silencioso), hemos de usar la opción `-v`.

```
usuario:~/PAV/P2$ ninja -C bin -v
ninja: Entering directory `bin'
[1/3] cc  -Ivad@exe -I. -I.. -fdiagnostics-color=always -pipe                    ...
↪  -D_FILE_OFFSET_BITS=64 -Wall -Winvalid-pch -O0 -g -MD -MQ                     ...
↪  'vad@exe/src_main_vad.c.o' -MF 'vad@exe/src_main_vad.c.o.d' -o                ...
↪  'vad@exe/src_main_vad.c.o' -c ../src/main_vad.c
[2/3] cc  -Ivad@exe -I. -I.. -fdiagnostics-color=always -pipe                    ...
↪  -D_FILE_OFFSET_BITS=64 -Wall -Winvalid-pch -O0 -g -MD -MQ                     ...
↪  'vad@exe/src_vad.c.o' -MF 'vad@exe/src_vad.c.o.d' -o 'vad@exe/src_vad.c.o'    ...
↪  -c ../src/vad.c
[3/3] cc  -o vad 'vad@exe/src_main_vad.c.o' 'vad@exe/src_vad.c.o'                ...
↪  -Wl,--no-undefined -Wl,--as-needed -Wl,--start-group -lm -lsndfile            ...
↪  -Wl,--end-group
```

El programa ejecutable, además de un montón de ficheros temporales y de configuración, está en el directorio `PAV/P2/bin`. Si lo ejecutamos sin argumentos, las salida es [2]:

```
usuario:~/PAV/P2$ bin/vad
Usage:
   vad [options] -i input-wav -o output-vad [-w output-wav]
   vad (-h | --help )
   vad --version
```

El mensaje que nos proporciona `vad` es un mensaje típico conforme al estándar POSIX. En él se nos informa de que hay tres maneras distintas de invocar el programa: indicando las opciones y argumentos para realizar la detección de voz; solicitando ayuda acerca del modo de empleo (opciones `-h` o **`--help`**); o solicitando información de la versión del programa.

Si solicitamos el mensaje de ayuda, la salida que obtenemos es:

```
usuario:~/PAV/P2$ bin/vad --help
VAD - Voice Activity Detector

Usage:
   vad [options] -i input-wav -o output-vad [-w output-wav]
   vad (-h | --help)
   vad --version

Options:
   -i FILE, --input-wav=FILE   WAVE file for voice activity detection
   -o FILE, --output-vad=FILE  Label file with the result of VAD
   -w FILE, --output-wav=FILE  WAVE file with silences cleared
   -v, --verbose  Show debug information
   -h, --help     Show this screen
   --version      Show the version of the project
```

Vemos que, para ejecutar el programa, hemos de indicar un fichero WAVE de entrada y un fichero de salida donde se escribirá el resultado de la detección. Opcionalmente, podemos indicar un fichero

[2] Una convención típica en programación es hacer que el programa indique el modo de empleo cuando se ejecuta con un número erróneo de argumentos (típicamente, sin ningún argumento). Nosotros hemos seguido esa convención, pero no siempre es así. Además, hay programas que pueden ser invocados sin argumentos, sin que ello sea erróneo; en estos casos, las opciones `-h`, `--help` o `-?` suelen ser alternativas válidas.

WAVE de salida, donde se almacenará la señal de audio con los segmentos de ruido puestos a cero (aunque esta fucionalidad la tendrá que implementar el alumno).

Tareas:

- Escriba el fichero `meson.build`.
- Ejecute `meson` y `ninja` para generar el programa en el directorio `bin`.
  - Compruebe que `ninja` realiza correctamente el mantenimiento del proyecto, modificando alguno de los ficheros y comprobando que `ninja` sólo realiza las tareas necesarias para mantener el programa actualizado.
- Compruebe que `bin/vad` funciona conforme a lo esperado.

4. Utilización del VAD.

4.1. Detección de voz en el fichero de prueba.

Como se vio con anterioridad, podemos ver el modo de empleo del programa ejecutándolo con la opción `-help`:

```
usuario:~/PAV/P2$ bin/vad --help
VAD - Voice Activity Detector

Usage:
   vad [options] -i input-wav -o output-vad [-w output-wav]
   vad (-h | --help)
   vad --version

Options:
   -i FILE, --input-wav=FILE   WAVE file for voice activity detection
   -o FILE, --output-vad=FILE  Label file with the result of VAD
   -w FILE, --output-wav=FILE  WAVE file with silences cleared
   -v, --verbose  Show debug information
   -h, --help     Show this screen
   --version      Show the version of the project
```

Por tanto, debemos invocar el programa con dos argumentos obligatorios y uno opcional:

`input-wav`: Fichero WAVE con la señal a la cual se desea el VAD.

`output-lab`: Fichero de texto con la segmentación en estados resultante del VAD.

`output-vad`: Fichero WAVE que, en el caso de implementarse la correspondiente tarea de ampliación, almacenará la señal con los intervalos de silencio puesto a cero.

Volvemos a ejecutar `vad`, pero ahora dándole un fichero WAVE de entrada e indicándole el fichero en el que queremos la salida:

```
usuario:~/PAV/P2$ bin/vad -i hola.wav -o hola.vad
```

Ahora la orden se ejecuta silenciosamente, indicativo de que su ejecución ha finalizado con éxito. Podemos ver el resultado de la detección editando o visualizando el fichero de salida:

hola.vad

```
usuario:~/PAV/P2$ cat hola.vad
0.000000        0.140000        S
0.140000        1.720000        V
1.720000        1.890000        S
1.890000        2.350000        V
2.350000        2.800000        S
2.800000        2.810000        V
```

Conforme a lo programado en `main_vad.c`, en cada fila tenemos información sobre cada uno de los estados identificados por el FSA: instante en segundos del inicio del estado, instante del final y estado identificado.

Tareas:

- Ejecute `vad` con la señal que grabó y etiquetó al principio de la práctica.
- Abra la señal con `wavesurfer` y cree dos paneles de transcripción: uno con la segmentación manual (fichero `hola.lab`) y otro con la generada automáticamente (`hola.vad`).
- Compare el resultado obtenido con el resultado teórico.
  - Como el programa, en su estado actual, asigna un etiquetado aleatorio, no es de extrañar que no acierte ni una...

4.1.1. Evaluación del resultado del VAD.

Para poder compara las prestaciones de sistemas distintos, o de configuraciones distintas de un mismo sistema, es conveniente obtener una medida de la calidad del resultado obtenido por el VAD. El mecanismo básico para poder calcular una medida de este tipo es el alineado de la secuencia de estados obtenida por el sistema (**resultado**) con la secuencia de estados real (**referencia**), obtenida mediante segmentación manual. Una vez alineado el resultado con la referencia, podemos tomar distintas medidas. Por ejemplo, podemos calcular el tanto por ciento de las veces que se calcula el resultado correcto; conocido como *tasa de acierto*.

Sin embargo, la tasa de acierto no es una medida muy fiable de la calidad del clasificador. Pensemos, por ejemplo, una situación en la que un 70 % del tiempo tengamos voz. Un sistema que reconozca siempre voz (y que, por tanto, sería un desastre de clasificador), obtiene una tasa de acierto del 70 %. Lo cual, evidentemente, es una medida muy optimista. Además, la tasa de acierto es que pondera por igual las veces que un silencio se ha confundido con voz, y las veces en las que la confusión es al revés. Sin embargo, en un sistema VAD real, ambos errores tienen una importancia muy diferente: detectar silencio cuando hay voz implica que esos segmentos de voz van a perderse, mientras que, cuando el error es al revés, toda la voz se conserva.

Las tasas que vamos a utilizar en esta práctica son cinco: las tasas de **sensibilidad** (*recall*) y **precisión** (*precisión*) de cada clase, y la media geométrica de las **medidas-F** ponderadas de voz y silencio (F_β):

Sensibilidad:
Dada una clase, la sensibilidad o *recall* indica el tanto por ciento de veces que las realizaciones de la clase se detectan como tales.

Precisión:
Dada una clase, la precisión indica el tanto por ciento de veces que las realizaciones detectadas como pertenecientes a la clase realmente lo son.

Medida–F:

Por sí solas, ni la sensibilidad ni la precisión reflejan adecuadamente cómo se reconoce cada una de las clases. La medida–F permite combinar ambas medidas en una sola.

Sensibilidad y precisión. Sea `VV` el número de veces que se reconoce correctamente `V`; `SS`, el de las que se reconoce correctamente `S`; `VS`, el de realizaciones de `V` reconocidas como `S`; y `SV`, el de realizaciones de `S` reconcidas como `V`.

La sensibilidad de las clases `V` y `S`, $S(\mathrm{V})$ y $S(\mathrm{S})$, respectivamente, viene dada por:

$$S(\mathrm{V}) = \frac{\mathtt{VV}}{\mathtt{VV} + \mathtt{VS}} \qquad S(\mathrm{S}) = \frac{\mathtt{SS}}{\mathtt{SS} + \mathtt{SV}} \tag{2}$$

Fijémonos en que la sensibilidad es máxima cuando todas las realizaciones de una clase se reconocen como tales. Ahora bien, eso lo podemos conseguir si el sistema reconoce sistemáticamente esa clase para todos los segmentos; y, evidentemente, esto no quiere decir decir que la clasificación sea correcta.

Por otro lado, la precisión para las clases `V` y `S`, $P(\mathrm{V})$ y $P(\mathrm{S})$, respectivamente, viene dada por:

$$P(\mathrm{V}) = \frac{\mathtt{VV}}{\mathtt{VV} + \mathtt{SV}} \qquad P(\mathrm{S}) = \frac{\mathtt{SS}}{\mathtt{SS} + \mathtt{VS}} \tag{3}$$

La precisión es máxima cuando todas las veces en que el sistema detecta la clase, la realización es realmente de esa clase. Pero eso lo podemos conseguir si el sistema sólo reconoce la clase cuando está completamente seguro (en el límite, nunca), sin que esto tampoco quiera decir que la clasificación es correcta.

Tanto la sensibilidad como la precisión se expresan en tanto por ciento y están acotados entre el cero y el cien por cien:

$$0\,\% \leq S(\mathrm{V}), S(\mathrm{S}), P(\mathrm{V}), P(\mathrm{S}) \leq 100\,\% \tag{4}$$

Medida–F ponderada. Hemos visto que tanto la sensibilidad como la precisión proporcionan medidas de calidad de la clasificación incompletas: cada una de ellas, por separado, puede alcanzar valores arbitrariamente altos sin que ello quiera decir que el clasificador funciona correctamente. Sin embargo, si ambas medidas son elevadas, entonces sí podemos estar seguros de la calidad del clasificador. La medida–F se calcula, para cada unidad `X`, como la media armónica (inverso de la media de los inversos) de su sensibilidad y precisión:

$$F_1(X) = \frac{1}{\dfrac{S^{-1}(\mathrm{X}) + P^{-1}(\mathrm{X})}{2}} = 2 \cdot \frac{S(\mathrm{X}) \cdot P(\mathrm{X})}{S(\mathrm{X}) + P(\mathrm{X})} \tag{5}$$

Al igual que la sensibilidad y la precisión, la medida–F se expresa en tanto por ciento y está acotada $0\,\% \leq F_1(X) \leq 100\,\%$. Una característica de la media armónica es que su valor está más cerca del del operando de valor más bajo. Es, por lo tanto, una medida *pesimista*. Además, es simétrica: si intercambiamos los valores de sensibilidad y precisión se obtiene el mismo resultado. Finalmente, en general, es distinta para cada una de las clases: $F_1(V) \neq F_1(S)$.

El hecho de que sea simétrica no siempre es interesante. Por ejemplo, en la detección de voz suele ser mucho más grave reconocer la voz como silencio que al revés. Cuando un segmento de voz se identifica como silencio, el segmento se suele perder. Sin embargo, si lo que hacemos es identificar un segmento de silencio como voz, el único problema será que tendremos que procesar un trozo más grande de señal, pero no se perderá nada del mensaje oral. Por este motivo, es habitual emplear la medida–F ponderada, $F_\beta(X)$, definida como:

$$F_\beta(X) = (1+\beta^2)\frac{S(\mathrm{X})\cdot P(\mathrm{X})}{S(\mathrm{X})+\beta^2 P(\mathrm{X})} \tag{6}$$

Esta medida es igual a $F_1(X)$ cuando $\beta = 1$; para $\beta > 1$ se enfatiza la importancia de la sensibilidad; mientras que para $\beta < 1$ se enfatiza la de la precisión.

En nuestra aplicación, interesará usar una $\beta < 1$ en la medida del silencio, porque de este modo se da más relevancia a que todo lo que se reconoce como silencio lo sea (precisión). Por contra, interesará una $\beta > 1$ en la medida de la voz, por que así se le da relevancia a que todo lo que es voz se reconozca como tal (sensibilidad).

Igualmente, tenemos el problema de que la medida–F es distinta para cada una de las clases, por lo que, finalmente, el valor que tomaremos como referencia para evaluar los sistemas será la media geométrica de $F_2(\mathrm{V})$ y $F_{1/2}(\mathrm{S})$:

$$F_{mix} = \sqrt{F_2(\mathrm{V})\cdot F_{1/2}(\mathrm{S})} \tag{7}$$

`vad_evaluation.pl`. Para realizar el alineado entre el resultado y la referencia, y calcular estas tasas, disponemos del script Perl `scripts/vad_evaluation.pl`. Este programa se invoca con el nombre del fichero, o ficheros, de referencia (extensión `.lab`). El fichero de resultado del VAD (extensión `.vad`) debe tener el mismo nombre y estar en el mismo directorio que el fichero de referencia:

```
usuario:~/PAV/P2$ scripts/vad_evaluation.pl hola.lab
**************** hola.lab ****************
Recall V:  3.80/4.74   80.17%   Precision V:  3.80/6.27   60.61%   F-score V (2)  : 75.31%
Recall S:  0.99/3.46   28.61%   Precision S:  0.99/1.93   51.30%   F-score S (1/2): 44.28%
===> hola.lab: 57.743%
```

Como era de prever, dado el carácter aleatorio de la implementación actual del VAD, el resultado es cercano al de lanzar una moneda al aire.

4.2. Detección de voz en la base de datos de evaluación.

Una sola señal de prueba puede ser útil para analizar el funcionamiento de un sistema, pero resulta poco significativa para sacar conclusiones de sus prestaciones. Por ejemplo, podemos ajustar los parámetros del FSA de manera que se obtenga una detección de voz/silencio óptima para esa señal; pero, para una señal diferente, las condiciones de grabación y/o ruido ambiente pueden ser tan distintas que el resultado sea un desastre.

Por este motivo, es habitual en tareas del estilo del VAD (como puedan ser las de reconocimiento del habla o del locutor) evaluar las prestaciones del sistema utilizando bases de datos con un número elevado de señales. En nuestro caso, se dispone de una base de datos de unas cien señales grabadas, voluntaria y desinteresadamente, por alumnos de la ETSETB. De ellas, 81 están disponibles para optimizar los algoritmos del VAD y ajustar sus parámetros (`db.v4`), y las 21 señales restantes, desconocidas para el desarrollador, serán utilizadas para realizar la evaluación final.

Para aplicar el VAD a todas las señales de la base de datos, se ha desarrollado el script Bash `scripts/run_vad.sh`. Este script debe ser editado para indicar la ruta y nombre exactos del programa a utilizar (en principio, `bin/vad`, y admite como argumentos el nombre de las señales para las que se quiere realizar la detección:

```
usuario:~/PAV/P2$ scripts/run_vad.sh db.v4/*/*wav
**************** db.v4/2014/pav_4151.wav ****************
**************** db.v4/2014/pav_4152.wav ****************
```

```
                              ...

**************** db.v4/2018-19q1/pav_4385.wav ****************
```

Y obtenemos el resultado de la evaluación invocando a `scripts/vad_evaluation.pl` con el nombre de todos los ficheros de referencia que queremos evaluar:

```
usuario:~/PAV/P2$ scripts/vad_evaluation.pl db.v4/*/*lab
**************** db.v4/2014/pav_4151.lab ****************
Recall V:  4.04/6.94   58.25%   Precision V:  4.04/6.60   61.22%   F-score V (2)  : 58.82%
Recall S:  0.57/3.13   18.28%   Precision S:  0.57/3.47   16.51%   F-score S (1/2): 16.83%
===> db.v4/2014/pav_4151.lab: 31.466%

                              ...

**************** db.v4/2018-19q1/pav_4385.lab ****************
Recall V:  1.62/3.24   49.99%   Precision V:  1.62/3.81   42.54%   F-score V (2)  : 48.30%
Recall S:  1.48/3.67   40.32%   Precision S:  1.48/3.10   47.71%   F-score S (1/2): 46.02%
===> db.v4/2018-19q1/pav_4385.lab: 47.148%

**************** Summary ****************
Recall V:240.79/384.10 62.69%   Precision V:240.79/410.35 58.68%   F-score V (2)  : 61.84%
Recall S:103.02/272.58 37.80%   Precision S:103.02/246.33 41.82%   F-score S (1/2): 40.95%
===> TOTAL: 50.324%
```

Como vemos, un VAD perfectamente aleatorio, como el desarrollado hasta ahora, alcanza una puntuación entorno al `TOTAL: 50%`. A modo de orientación, el resultado obtenido por los mejores sistemas del curso 2018/19 rondaba el `TOTAL: 95%`.

Tarea:

Edite el fichero `scripts/run_vad.sh` para que el script llame al programa `vad`, y ejecute `scripts/run_vad.sh` y `scripts/vad_evaluation.pl` con los ficheros de la base de datos `db.v4` para comprender su funcionamiento.

5. Ejercicios y entrega.

Ejercicios básicos:

1. Complete el código de los ficheros `main_vad.c` y `vad.c` para que el programa realice la detección de actividad vocal. Escriba las funciones de análisis o incorpore al proyecto los ficheros de la primera práctica `pav_analysis.c` y `pav_analysis.h`. Recuerde incorporar las cabeceras necesarias en los ficheros correspondientes.

   Tiene completa libertad para implementar el algoritmo del modo que considere más oportuno, pero el código proporcionado puede ser un buen punto de partida para hacerlo usando un autómata de estados finitos (FSA). Encontrará en los ficheros sugerencias para ello, marcadas con la palabra `TODO` (del inglés *to do*, a realizar).

   Ayúdese de la visualización de la señal y sus transcripciones usando `wavesurfer`, de la opción `-verbose` del programa `vad`, de la colocación de *chivatos* y de cualquier otra técnica que se le pueda ocurrir para conseguir que el programa funcione correctamente. Recuerde que el fichero de salida sólo debe incluir las etiquetas `V` y `S`.

2. Optimice los algoritmos y sus parámetros de manera que se maximice la puntuación de la detección de la base de datos de desarrollo (`db.v4`).

Ejercicios de ampliación:

1. Complete el código del fichero `main_vad.c` para que el programa, en el caso de que se indique un fichero `.wav` de salida (opción `--output-wav`), escriba en él la señal de la entrada, sustituyendo por cero los valores de los intervalos identificados como silencio.

2. Modifique los ficheros necesarios para permitir que los parámetros del FSA (nivel de los umbrales, α_0, α_1...; duraciones mínimas y máximas; etc.) sean accesibles desde la línea de comandos (ver el Anexo II).

   Esta ampliación, de hecho, le puede resultar muy útil para optimizar el resultado de la detección.

Con el código proporcionado, los profesores de la asignatura realizarán una evaluación usando una base de datos distinta de la utilizada durante el desarrollo. Una parte sustancial de la nota de la práctica estará vinculada al resultado obtenido en ella.

- El código fuente debe estar en tal estado que, ejecutando `meson bin; ninja -C bin` se obtenga un fichero ejecutable correcto, que debe proporcionar los mismos resultados que los publicados, y que será el usado en la evaluación final del proyecto.

ANEXOS.

I. Plataforma de desarrollo colaborativo GitHub.

En este proyecto se va a introducir la plataforma para desarrollo colaborativo **GitHub**, tanto para distribuir la versión inicial del proyecto, como para realizar la entrega de la memoria y el software correspondiente.

Aunque la filosofía de Git es la de gestionar proyectos distribuidos, suele ser interesante disponer de un repositorio que ejerza de referencia, lo que se suele designar como *repositorio central*. De este modo, los diferentes participantes de un proyecto sólo necesitan sincronizarse con este repositorio central, en lugar de tener que hacerlo con todos los demás.

Para ilustrar este concepto, vamos a considerar un supuesto: diez desarrolladores trabajando en un mismo proyecto. Cada uno de ellos recibe la misma copia del repositorio inicial e implementa sus modificaciones en ella. Cuando los diez han concluido su trabajo, tenemos dos opciones para integrarlo:

- Siguiendo una lógica completamente distribuida, cada uno de los diez desarrolladores debe intercambiar sus aportaciones con cada uno de los nueve restantes.
    - Han de realizarse $10 \times (10-1) = 90$ intercambios bidireccionales. En cada intercambio, uno de los desarrolladores integra el trabajo del otro (*merge*), y, a continuación, el segundo se sincroniza con la versión del primero.
    - Lo normal es que en cada intercambio aparezcan conflictos que deberán resolverse. Aunque todos los participantes resuelvan los conflictos de una manera lógica y correcta, será difícil que todos lo hagan con exactamente los mismos criterios.
    - Incluso si no hay conflictos, el orden en el que se realizan los merges puede alterar el resultado final. Por ejemplo, intercambiando el orden de bloques distintos de código de un mismo fichero, pero independientes funcionalmente entre sí.
    - El resultado final es la aparición de múltiples versiones diferentes del mismo software. De hecho, lo más probable es que se acabe con tantas versiones diferentes como participantes.
- Como alternativa a la integración distribuida, tenemos la integración centralizada: uno de los participantes, o uno externo, actúa de repositorio central e integra las aportaciones de cada uno de los otros nueve (diez, si se usa un participante externo). Una vez integrado el trabajo de todos los participantes, éstos se sincronizan con la versión final.
    - Como mucho hay diez intercambios bidireccionales.
    - El resultado final es la misma versión para todos los participantes.

GitHub es una plataforma web para alojar proyectos de software usando Git, especialmente adecuado para servir de repositorio central. Cada participante del proyecto baja su versión inicial del repositorio GitHub, realiza sus modificaciones localmente, y solicita del administrador del repositorio central su integración en él (*pull request*). El administrador resuelve los eventuales conflictos y actualiza la versión del repositorio. Una vez se han integrado las aportaciones de todos los participantes (o en cualquier otro momento durante el proceso de desarrollo), éstos pueden sincronizarse con la versión *central*.

Aparte de proporcionar funcionalidad Git casi completa, GitHub también proporciona un interfaz de gestión de versiones mucho más razonable e intuitivo que el de Git, Wiki de proyecto y otras funcionalidades añadidas. Además, al utilizarse un servidor externo dotado de las máximas medidas de seguridad, proporciona un mecanismo magnífico para gestionar copias de seguridad. Todo ello hace que sean muchos los desarrolladores que tienden a minimizar la interacción con el siempre áspero Git, realizando buena parte de su trabajo con el mucho más amigable GitHub.

GitHub permite crear cuentas de usuario gratuitas siempre que el código subido a ella sea accesible para todo el mundo, es decir: apenas se pueden poner restricciones de acceso al código, aunque no se

exige que éste sea de dominio público. En concreto, los términos de servicio establecen: *"by setting your repositories to be viewed publicly, you agree to allow others to view and fork your repositories"*. También existe la posibilidad de crear cuentas privadas en las que el usuario puede limitar el acceso al público, pero estas cuentas son de pago.

Fundada en 2008, cuenta en la actualidad con unos cuarenta millones de cuentas de usuario, que albergan unos cien millones de repositorios. Hoy en día (otoño de 2019) puede decirse que GitHub es la opción de referencia cuando se desea crear un proyecto de código abierto accesible por internet, y a un precio razonable (es decir, gratis). También puede decirse que tener unos cuantos proyectos personales colgados de GitHub es, en la actualidad, uno de los mejores apuntes en el currículum de un desarrollador; y que no tenerlos será, en poco tiempo, una de sus mayores carencias.

Clonado de las prácticas

Siguiendo las instrucciones del fichero `README.md`:

- Abra una cuenta GitHub para gestionar esta y el resto de prácticas del curso.
- Cree un repositorio GitHub con el contenido inicial de la práctica (sólo debe hacerlo uno de los integrantes del grupo de laboratorio, cuya página GitHub actuará de repositorio central):
  - Acceda la página de la Práctica P2.
  - En la parte superior derecha encontrará el botón Fork. Apriételo y, después de unos segundos, se creará en su cuenta GitHub un proyecto con el mismo nombre (`P2`). Si ya tuviera uno con ese nombre, se utilizará el nombre `P2-1`, y así sucesivamente.
- Habilite al resto de miembros del grupo como *colaboradores* del proyecto; de este modo, podrán subir sus modificaciones al repositorio central:
  - En la página principal del repositorio, en la pestaña Settings, escoja la opción Collaborators y añada a su compañero de prácticas.
  - Éste recibirá un email solicitándole confirmación. Una vez confirmado, tanto él como el propietario podrán gestionar el repositorio, por ejemplo: crear ramas en él o subir las modificaciones de su directorio local de trabajo al repositorio GitHub.
- En la página principal del repositorio, localice el botón Branch: **master** y úselo para crear una rama nueva con los primeros apellidos de los integrantes del equipo de prácticas separados por guion (**fulano-mengano**).
- Todos los miembros del grupo deben realizar su copia local en su ordenador personal:
  - Copie la dirección de su copia del repositorio de la práctica apretando en el botón Clone or download.
    - Asegúrese de usar `Clone with HTTPS`.
  - Abra una sesión de Bash en su ordenador personal y vaya al directorio `PAV`. Desde ahí, ejecute:

```
usuario:~/PAV$ git clone dirección-del-fork-de-la-práctica
```

  - Vaya al directorio de la práctica `cd P2`.
  - Cambie a la rama *fulano-mengano* con la orden:

```
usuario:~/PAV/P2$ git checkout fulano-mengano
```

- A partir de este momento, todos los miembros del grupo de prácticas pueden trabajar en su directorio local del modo habitual.
  - También puede utilizar el repositorio remoto en GitHub como repositorio central para el trabajo colaborativo de los distintos miembros del grupo de prácticas; o puede serle útil usarlo como copia de seguridad.
  - Cada vez que quiera subir sus cambios locales al repositorio GitHub deberá confirmar los cambios en su directorio local:

    ```
    usuario:~/PAV/P2$ git add .
    usuario:~/PAV/P2$ git commit -m "Mensaje del commit"
    ```

    Y, a continuación, subirlos con la orden

    ```
    usuario:~/PAV/P2$ git push origin fulano-mengano
    ```

- Al final de la práctica, la rama *fulano-menganos* del repositorio GitHub servirá para remitir la práctica para su evaluación utilizando el mecanismo *pull request*.
  - Vaya a la página principal de la copia del repositorio y asegúrese de estar en la rama *fulano-mengano*.
  - Pulse en el botón **New pull request**, y siga las instrucciones de GitHub.

I.A. Fichero `~/.gitignore` y primer *commit* de la práctica.

De cara a gestionar las versiones del software con Git, sólo estamos interesados en los ficheros y directorios que realmente forman el código. No tiene sentido gestionar las versiones de los directorios de compilación, porque son directorios que se pueden regenerar automáticamente a partir del código en `src`. Tampoco tiene sentido gestionar la base de datos de evaluación, porque siempre puede recuperarse la original.

Lo que queremos que gestione Git es:

- Los directorios `src`, `scripts` y `docopt_c`, y todo su contenido.
- El fichero `meson.build` y, en general, cualquier fichero que coloquemos en el directorio `PAV/P2`. Por ejemplo: la memoria de la práctica.
- Los propios ficheros de gestión de Git (a continuación crearemos el fichero `.gitignore`, que debería acompañar siempre al paquete de software).

Y, al menos por ahora, lo que queremos que se excluya de la gestión de versiones es cualquier subdirectorio de `PAV/P2` distinto de `src` o `scripts`. Para ello, vamos a construir un fichero `.gitignore` con el contenido siguiente:

.gitignore

```
/*/
!/docopt_c/
!/scripts/
!/src/
```

La primera línea de `.gitignore` (`/*/`) indica que no debe realizarse el seguimiento de ningún subdirectorio del proyecto (esto lo indicamos con las barra al inicio y al final); la segunda línea (`!/docopt_c/`)

indica que sí debemos seguir el directorio `docopt_c` (lo indicamos con la exclamación inicial; de hecho, esa exclamación indica que un directorio llamado `docopt_c` es una excepción a la regla anterior que excluye todos los directorios). Hacemos lo mismo con los directorios `src` y `scripts`.

Puede consultar la sintaxis completa en el enlace .gitignore.

A partir de este momento, el proyecto está bajo el control de versiones de Git. Cada vez que queramos almacenar una nueva versión, todo lo que tenemos que hacer es repetir los comandos `git add .` y `git commit` con un mensaje explicativo de los cambios incorporados en ella.

Tareas:

- Escriba el fichero `.gitignore` de manera que Git sólo gestione los directorios `P2/src`, `P2/docopt_c` y `P2/scripts`, y los ficheros del directorio `P2`.
- Realice la primera confirmación (*commit*) con los ficheros originales de la práctica.

II. Gestión de opciones y argumentos usando `docopt`.

II.A. El interfaz en línea de comandos según POSIX y GNU.

El paso de parámetros y argumentos a los programas es una cuestión de suma importancia. Existen, básicamente, dos alternativas: pasarlos en tiempo de ejecución o por línea de comandos. La primera suele requerir que el usuario introduzca texto o seleccione con el ratón o los dedos las opciones deseadas. Por lo tanto, es difícil, por no decir imposible, de automatizar. La alternativa, el uso de un *interfaz en línea de comandos* (**CLI**, por sus iniciales en inglés), es la preferida en sistemas del estilo de UNIX, y es especialmente útil para lanzar programas con la mínima interacción posible (por ejemplo, si se desea lanzar una tanda de experimentos ejecutando un mismo programa con parámetros distintos).

Desde los inicios de los sistemas informáticos han aparecido infinidad de estrategias para el paso de argumentos por línea de comandos. Siendo N el número de programadores, el número de criterios es, seguramente, no inferior a $2N$, ya que casi cada programador tiene su estilo preferido, que no le gusta a casi nadie más, y casi nunca es el primero que desarrolló.

Para evitar esta dispersión, y muchas otras que existen en el ámbito informático[3], el IEEE lanzó en los años 80 del siglo XX una iniciativa para unificar criterios, el llamado POSIX.

II.A.A. Opciones y argumentos según POSIX.

Según POSIX, los argumentos de un programa se pueden dividir en dos tipos: opciones y operandos, también conocidos como argumentos propiamente dichos. Los aspectos más relevantes de opciones y operandos son:

- Una opción es un argumento que comienza por guion (`-`) seguido de un único carácter alfanumérico, denominado *nombre*. En general, los operandos serán cualquier otro argumento, que empiece por un carácter distinto de guion y que no sea el argumento de una opción.
  - Todas las opciones empiezan por guion, pero hay casos especiales en los que un argumento que empieza por guion no es una opción, sino operando.
    - El argumento especial doble guion (`--`) sirve para dar por finalizada las opciones del programa. A partir de su posición, todo argumento será tratado como un operando, con independencia de cómo empiece.

[3] El estándar ocupa 6900 páginas, de las que sólo 8 se refieren estrictamente al interfaz en línea de comandos

    - El argumento especial guion (`-`) suele usarse para indicar que un fichero de entrada debe ser sustituido por la entrada estándar (`stdin`). Si no hay ambigüedad, también puede usarse para indicar que un fichero de salida debe ser sustituido por la salida estándar (`stdout`).
  - Aunque POSIX sólo permite las denominadas *opciones cortas*, cuyo nombre es de un solo carácter, GNU introdujo las *opciones largas*, formadas por palabras de más de un carácter y que se han incorporado al estándar *de facto* usado por la mayoría de programadores.
  - En principio, las opciones pueden aparecer en cualquier orden, aunque hay situaciones concretas en las que el orden puede ser relevante.
- También en general, las opciones deben anteceder al resto de argumentos (operandos).
- Las opciones se dividen en dos tipos:
  - Sin argumento de la opción, también conocidas como *flags*. Por ejemplo, en el programa `vad`, la opción `-v` activa el modo verboso.
  - Con argumento de la opción, en cuyo caso el argumento será una sola palabra que puede estar unida o separada del nombre de la opción. Por ejemplo: `-ihola.wav` o `-i hola.wav`.
  - Las opciones sin argumentos pueden agruparse entre sí y con, como máximo, una opción con argumento, que debe aparecer en último lugar. Por ejemplo: `-vihola.wav` es equivalente a `-v -i hola.wav`

Todo programa deberá contar con una *sinopsis* que mostrará su modo de empleo. Aunque POSIX no lo especifica, la opción `-?` suele ser indicativa de que el usuario quiere ver esta sinopsis en pantalla. También en este caso, GNU ha introducido su propio criterio, que se ha extendido como un estándar entre los programadores: el uso de las opciones `-h` o `--help` como solicitud de visualización de la sinopsis.

Los elementos más relevantes de la sinopsis son:

- Si un programa tiene distintos modos de operación, con opciones diferentes, cada modo se escribe en una línea separada. Por ejemplo, `vad` tiene tres modos de operación: modo VAD, modo ayuda y modo versión.
- Los argumentos opcionales se escriben encerrados entre corchetes (`[argumento-opcional]`). Por ejemplo, `[-w output-wav]` es un argumento opcional, que puede especificarse o no.
- Los argumentos obligatorios pueden aparecer tal cual o encerrados entre paréntesis. Por ejemplo, `(argumento-obligatorio)`. El uso de los paréntesis sólo suele ser conveniente para agrupar argumentos alternativos. Así, `-i input-wav` es un argumento obligatorio.
- Los argumentos alternativos, es decir, mutuamente excluyentes, se escriben separados por barra vertical (`|`) y, a menudo, encerrados entre corchetes si son opcionales, o entre paréntesis si son obligatorios.
- Cuando un argumento puede ser repetido un número indefinido de veces, se indica con tres puntos a continuación del argumento correspondiente.

II.A.B. Extensiones y convenios de GNU.

A los criterios marcados por POSIX, GNU ha añadido una serie de extensiones y convenios que se han convertido en un estándar *de facto*. Puede encontrarse información más completa en GNU Argument Syntax y en GNU: Standards for Command Line Interfaces. Estas extensiones no están contempladas por POSIX, pero no impiden el cumplimiento del estándar en tanto en cuanto se mantenga la posibilidad de ejecutar el programa usando un CLI estrictamente POSIX. Por ejemplo, en el programa `vad`

se usan nombres de opción largos, pero en casi todos los casos estas opciones largas pueden sustituirse por otras cortas. Sólo la opción `--version`, que no tiene una alternativa corta, rompe el cumplimiento del estándar POSIX.

Algunas de las extensiones más importantes apostadas por GNU son:

- Nombres de opción largos: indicados por dos guiones seguidos del nombre de la opción, que será de más de un carácter.
  - El nombre largo deberá ser descriptivo del uso o comportamiento de la opción y, habitualmente, estará formado por dos o tres palabras separadas por guion (`-`). Estos guiones, en el código del programa, son sustituidos por subrayados (`_`).
  - No es necesario escribir el nombre completo de la opción, sólo los caracteres necesarios para identificarla unívocamente.
  - Si la opción tiene un argumento, éste se escribirá separado de aquélla mediante espacios o un signo igual (`=`).
- Las opciones `-h`, `--help` y `--version` tienen significado especial: las dos primeras muestran la sinopsis del programa, la última su versión.
- Los comandos y argumentos de las opciones escritos en minúscula son considerados literales y equivalentes a *flags*. En el programa, estos literales toman el valor *cierto*, si aparecen en la invocación del programa, o *falso*, si no lo hacen.
- Los comandos y argumentos que representan valores variables deben escribirse, o bien en mayúsculas, o bien encerrados entre los símbolos menor que (`<`) y mayor que (`>`). En este último caso, el nombre del argumento puede estar formado por distintas palabras separadas por guion o espacio. En ambos casos, el carácter delimitador se cambia por subrayado en la variable del código del programa que lo almacena.

Ejemplo de sinopsis: el recuadro siguiente muestra una sinopsis típica siguiendo las recomendaciones de POSIX con las extensiones de GNU. Es un ejemplo clásico, útil para ilustrar (casi) todas las posibilidades comentadas en las dos secciones precedentes:

```
Naval Fate.

Usage:
  naval_fate ship new <name>...
  naval_fate ship <name> move <x> <y> [--speed=<kn>]
  naval_fate ship shoot <x> <y>
  naval_fate mine (set|remove) <x> <y> [--moored|--drifting]
  naval_fate -h | --help
  naval_fate --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Moored (anchored) mine.
  --drifting    Drifting mine.
```

II.B. `docopt` y `docopt_c`.

Para facilitar la gestión del CLI se han desarrollado toda una serie de aplicaciones y funciones de librería. Durante mucho tiempo, la única opción en programas C era la función `getopt()` (ejecutar `man 3 getopt` para ver su funcionamiento). El problema con esta función es que todo el trabajo de gestión de las opciones y del mensaje con la sintaxis recae sobre el programador.

En el lenguaje Python, se introdujo la función `optparse()`, luego sustituida por `argparse()`. Además de ser notoriamente más versátiles y potentes que el `getopt()` de C, la gran ventaja de `optparse()` es que, a partir de la definición algorítmica de las opciones del programa, el mensaje de sinopsis se genera automáticamente. No obstante, la gestión de las opciones continuaba siendo bastante farragosa (aunque no tanto como con `getopt()`).

En 2012, Vladimir Keleshev ideó un sistema de gestión del CLI para programas escritos en Python muy original: dado que todo programa debería tener un mensaje de sinopsis que, a su vez, sirve de documentación del código, construyamos un mecanismo que permita realizar el análisis de los argumentos y opciones a partir del mismo. Esta estrategia tiene dos ventajas: permite mayor flexibilidad en la construcción del mensaje de sinopsis y, sobre todo, la gestión algorítmica de las opciones se simplifica al extremo. En su versión inicial, esta idea se implementó como el módulo de Python `docopt`.

II.B.A. `docopt_c`.

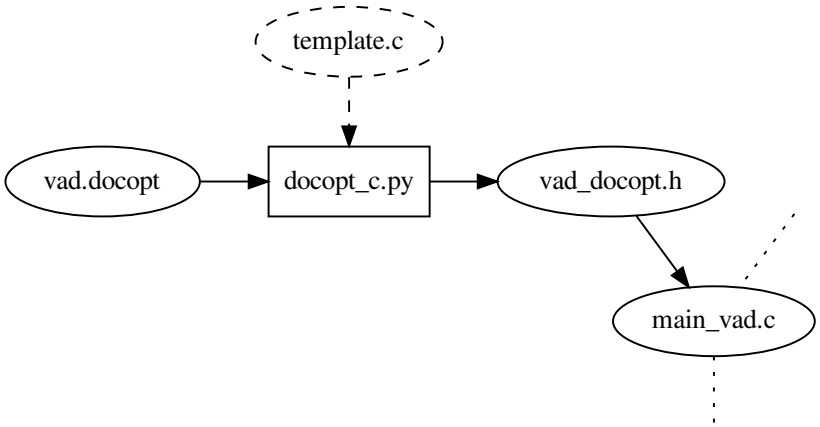
El éxito de `docopt` en el *pythoniverso* (la comunidad de los *pythonisos*) fue abrumador. Realmente, se cumple el lema del programa: *pythonic argument parser, that will make you smile*. Por este motivo, `docopt` a sido portado a multitud de lenguajes de programación: C, C++, Java, Bash, etc. En el caso de C, el port se denomina `docopt_c`. Desgraciadamente, a fecha de hoy, otoño de 2019, `docopt_c` sigue estando incompleto y presenta errores importantes. Lo peor del caso es que es un proyecto aparentemente inactivo: hace más de tres años de la última actualización. El motivo de este fracaso cabe buscarlo en las propias características del lenguaje C: `docopt` se basa en gran medida en programación orientada a objeto y funcional, herramientas poco disponibles en C.

Otros proyectos paralelos han intentado evitar los problemas de `docopt_c`; por ejemplo, el paquete `rouming/docopt.c`. No obstante, en esta práctica usaremos la *defectuosa* `docopt_c` por varios motivos:

- El proyecto `docopt` está mucho más establecido y tiene mucha más proyección que cualquiera de las otras alternativas.
- Varias de las versiones para otros lenguajes son plenamente operativas; no es descartar que, tarde o temprano, se corrijan sus errores y carencias.
    - En concreto, la versión para C++, `docopt_cpp` está completa y, supuestamente, en perfecto funcionamiento. Como el resto de prácticas se realizarán en C++, parece conveniente empezar con `docopt_c`.
- Para los propósitos del programa `vad`, `docopt_c` tiene algunas limitaciones y provoca algunos errores incomprensibles, pero, ciñéndonos al modo de empleo que se documenta en este enunciado, su funcionamiento es el correcto.

II.B.B. Gestión de los argumentos y opciones usando `docopt_c`.

`docopt_c.py` es un *script* de Python3 que lee un fichero de texto, con la sinopsis del programa (`vad.docopt`), y una plantilla, con el esqueleto de una función C (`template.c`), para generar una cabecera (`vad_docopt.h`) que debe ser incluido por el fichero con el programa principal (`main_vad.c`):



La cabecera generada proporciona la definición de una estructura, `DocoptArgs`, que incluye tantos campos como argumentos y opciones tiene la sinopsis del programa. El nombre de los campos es el mismo que el de los respectivos argumentos, con la salvedad de que, si el nombre incluye espacios o guiones, éstos son sustituidos por subrayados (`_`). Si el argumento es del tipo booleano, su tipo será `int`, en cualquier otro caso, el tipo será cadena de caracteres (`char *`), que recibirá, literalmente, el valor introducido al ejecutar el programa. Finalmente, también incluye la sinopsis y el mensaje de ayuda:

```
typedef struct {
    /* options without arguments */
    int help;
    int verbose;
    int version;
    /* options with arguments */
    char *input_wav;
    char *output_vad;
    char *output_wav;
    /* special */
    const char *usage_pattern;
    const char *help_message;
} DocoptArgs;
```

La cabecera también proporciona una función llamada `docopt()`, que toma como argumentos el número de argumentos del programa (`argc`), sus valores (`argv`), un indicador de si se quiere que las opciones `-h` y `-help` muestren la sinopsis del programa y una cadena de texto descriptiva de la versión.

```
DocoptArgs docopt(int argc, char *argv[], bool help, const char *version);
```

Lo único que debemos hacer para generar un nuevo conjunto de opciones es:

1. Editar el fichero `vad.docopt` para que refleje la sinopsis que deseamos. Como era de esperar, el fichero inicial incluye exactamente la misma información que la sinopsis mostrada por el programa cuando se ejecuta con la opción `--help`:

   vad.docopt

   ```
   VAD - Voice Activity Detector

   ```

```
Usage:
   vad [options] -i <input-wav> -o <output-vad> [-w <output-wav>]
   vad (-h | --help)
   vad --version

Options:
   -i FILE, --input-wav=FILE   WAVE file for voice activity detection
   -o FILE, --output-vad=FILE  Label file with the result of VAD
   -w FILE, --output-wav=FILE  WAVE file with silences cleared
   -v, --verbose  Show debug information
   -h, --help     Show this screen
   --version      Show the version of the project
```

2. Ejecutar `docopt_c/docopt_c.py` para generar la nueva cabecera `src/vad_docopt.h`:

```
usuario:~/PAV$ docopt_c/docopt_c.py src/vad.docopt -o src/vad_docopt.h
```

3. Modificar el código de `main_vad.c` para que llame a la función `docopt()` y tenga en cuenta los nuevos argumentos. La parte de `main_vad.c` que se encarga actualmente de esto es:

main_vad.c

```
DocoptArgs args = docopt(argc, argv, /* help */ 1, /* version */ "2.0");

verbose    = args.verbose ? DEBUG_VAD : 0;
input_wav  = args.input_wav;
output_vad = args.output_vad;
output_wav = args.output_wav;

if (input_wav == 0 || output_vad == 0) {
  fprintf(stderr, "%s\n", args.usage_pattern);
  return -1;
```

II.B.C. Mantenimiento de `vad_docopt.h` usando `meson`.

Cada vez que se cambie el fichero con la sinopsis del programa, `src/vad.docopt`, hemos de volver a generar el fichero `src/vad_docopt.h` usando el programa `docopt_c/docopt_c.py`, y hemos de regenerar el proyecto usando `ninja`.

Lo ideal sería que, cada vez que modificamos `src/vad.docopt`, con ejecutar `ninja` bastara para que éste se encargue de regenerar `src/vad_docopt.h` y el programa `vad`. Pero eso no es tan sencillo como parece, porque representa crear una nueva cabecera que no podrá estar en el directorio `src`, porque `meson` sólo permite trabajar *fuera del directorio*. Así que acabamos teniendo dos ficheros `vad_docopt.h`: uno en el directorio donde construimos el programa (`bin`) y otro en el del código fuente (`src`). Al incluir la cabecera desde `main_vad.c`, el segundo tiene preferencia sobre el primero, con lo que `ninja` no hará caso del fichero que acaba de generar... Para evitarlo:

Hay que borrar el fichero `src/vad_docopt.h`.

Hecho esto, añadimos las líneas siguientes al fichero `meson.build` para que realice el mantenimiento de las opciones y argumentos del programa, regenerando la cabecera `vad_docopt.h`, cada vez que modificamos la sinopsis contenida en `src/vad.docopt`, y recompilando el programa:

meson.build

```
7  inputDO  = files('src/vad.docopt')
8  progDO   = files('docopt_c/docopt_c.py')
9  outputDO = 'vad_docopt.h'
10
11 headerDO = custom_target('Construye vad_docopt.h',
12         output: outputDO, command: [progDO, inputDO, '-o', outputDO])
13
14 executable(exe, sources: [src, headerDO], link_args: lib)
```

- Las modificaciones al `meson.build` original se reflejan en las variables cuyo nombre finaliza en `DO` (por **d**oc**o**pt).
  - En las líneas `7` a `9` se definen las variables necesarias.
  - En las líneas `11` y `12` se construye la cabecera `vad_docopt.h` ...
  - ... que se incluye entre los códigos fuente del programa en la línea `14`.
- Como la compilación se realiza, forzosamente, fuera del directorio, no tenemos control de dónde se hace. Esto nos impide hacer referencia directa a los ficheros utilizados para generar la cabecera.
  - El objeto retornado por `files()` incorpora la información necesaria para localizar un fichero del árbol del código fuente con independencia de desde dónde se construya el proyecto.
  - Si usáramos el nombre del fichero directamente, `ninja` no sería capaz de encontrarlo.
- `custom_target()` es una función que permite construir un objetivo a partir de un comando arbitrario. En este caso, `vad_docopt.h` a partir de `docopt.py` y del fichero `vad.docopt`.
  - El objeto devuelto por `custom_target()` puede ser usado como código fuente o como dependencia de otros objetivos.
- Usamos la salida de `custom_target()` como parte del código necesario para construir el programa `pav` con la función `executable()`.

III. Código de los ficheros proporcionados.

III.A. `main_vad.c`

main_vad.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sndfile.h>

#include "vad.h"
#include "vad_docopt.h"

#define DEBUG_VAD 0x1

int main(int argc, char *argv[]) {
  int verbose = 0; /* To show internal state of vad: verbose = DEBUG_VAD; */

  SNDFILE *sndfile_in, *sndfile_out = 0;
  SF_INFO sf_info;
  FILE *vadfile;
  int n_read = 0, i;

  VAD_DATA *vad_data;
  VAD_STATE state, last_state;

  float *buffer, *buffer_zeros;
  int frame_size;         /* in samples */
  float frame_duration;   /* in seconds */
  unsigned int t, last_t; /* in frames */

  char        *input_wav, *output_vad, *output_wav;

  DocoptArgs args = docopt(argc, argv, /* help */ 1, /* version */ "2.0");

  verbose    = args.verbose ? DEBUG_VAD : 0;
  input_wav  = args.input_wav;
  output_vad = args.output_vad;
  output_wav = args.output_wav;

  if (input_wav == 0 || output_vad == 0) {
    fprintf(stderr, "%s\n", args.usage_pattern);
    return -1;
  }

  /* Open input sound file */
  if ((sndfile_in = sf_open(input_wav, SFM_READ, &sf_info)) == 0) {
    fprintf(stderr, "Error opening input file %s (%s)\n", input_wav,
    ↪  strerror(errno));
    return -1;
  }

  if (sf_info.channels != 1) {
```

```
48      fprintf(stderr, "Error: the input file has to be mono: %s\n", input_wav);
49      return -2;
50    }
51
52    /* Open vad file */
53    if ((vadfile = fopen(output_vad, "wt")) == 0) {
54      fprintf(stderr, "Error opening output vad file %s (%s)\n", output_vad,
        ↪  strerror(errno));
55      return -1;
56    }
57
58    /* Open output sound file, with same format, channels, etc. than input */
59    if (output_wav) {
60      if ((sndfile_out = sf_open(output_wav, SFM_WRITE, &sf_info)) == 0) {
61        fprintf(stderr, "Error opening output wav file %s (%s)\n", output_wav,
          ↪  strerror(errno));
62        return -1;
63      }
64    }
65
66    vad_data = vad_open(sf_info.samplerate);
67    /* Allocate memory for buffers */
68    frame_size   = vad_frame_size(vad_data);
69    buffer       = (float *) malloc(frame_size * sizeof(float));
70    buffer_zeros = (float *) malloc(frame_size * sizeof(float));
71    for (i=0; i< frame_size; ++i) buffer_zeros[i] = 0.0F;
72
73    frame_duration = (float) frame_size/ (float) sf_info.samplerate;
74    last_state = ST_UNDEF;
75
76    for (t = last_t = 0; ; t++) { /* For each frame ... */
77      /* End loop when file has finished (or there is an error) */
78      if  ((n_read = sf_read_float(sndfile_in, buffer, frame_size)) !=
        ↪  frame_size) break;
79
80      if (sndfile_out != 0) {
81        /* TODO: copy all the samples into sndfile_out */
82      }
83
84      state = vad(vad_data, buffer);
85      if (verbose & DEBUG_VAD) vad_show_state(vad_data, stdout);
86
87      /* TODO: print only SILENCE and VOICE labels */
88      /* As it is, it prints UNDEF segments but is should be merge to the proper
        ↪  value */
89      if (state != last_state) {
90        if (t != last_t)
91          fprintf(vadfile, "%.5f\t%.5f\t%s\n", last_t * frame_duration, t *
            ↪  frame_duration, state2str(last_state));
92        last_state = state;
93        last_t = t;
94      }
95
```

```
    if (sndfile_out != 0) {
      /* TODO: go back and write zeros in silence segments */
    }
  }

  state = vad_close(vad_data);
  /* TODO: what do you want to print, for last frames? */
  if (t != last_t)
    fprintf(vadfile, "%.5f\t%.5f\t%s\n", last_t * frame_duration, t *
    ↪  frame_duration + n_read / (float) sf_info.samplerate,
    ↪  state2str(state));

  /* clean up: free memory, close open files */
  free(buffer);
  free(buffer_zeros);
  sf_close(sndfile_in);
  fclose(vadfile);
  if (sndfile_out) sf_close(sndfile_out);
  return 0;
}
```

III.B. vad.h

vad.h

```
#ifndef _VAD_H
#define _VAD_H
#include <stdio.h>

/* TODO: add the needed states */
typedef enum {ST_UNDEF=0, ST_SILENCE, ST_VOICE, ST_INIT} VAD_STATE;

/* Return a string label associated to each state */
const char *state2str(VAD_STATE st);

/* TODO: add the variables needed to control the VAD
   (counts, thresholds, etc.) */

typedef struct {
  VAD_STATE state;
  float sampling_rate;
  unsigned int frame_length;
  float last_feature; /* for debuggin purposes */
} VAD_DATA;

/* Call this function before using VAD:
   It should return allocated and initialized values of vad_data

   sampling_rate: ... the sampling rate */
VAD_DATA *vad_open(float sampling_rate);

```

```
/* vad works frame by frame.
   This function returns the frame size so that the program knows how
   many samples have to be provided */
unsigned int vad_frame_size(VAD_DATA *);

/* Main function. For each 'time', compute the new state
   It returns:
    ST_UNDEF   (0) : undefined; it needs more frames to take decission
    ST_SILENCE (1) : silence
    ST_VOICE   (2) : voice

    x: input frame
       It is assumed the length is frame_length */
VAD_STATE vad(VAD_DATA *vad_data, float *x);

/* Free memory
   Returns the state of the last (undecided) states. */
VAD_STATE vad_close(VAD_DATA *vad_data);

/* Print actual state of vad, for debug purposes */
void vad_show_state(const VAD_DATA *, FILE *);

#endif
```

III.C. vad.c

vad.c

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "vad.h"

const float FRAME_TIME = 10.0F; /* in ms. */

/*
 * As the output state is only ST_VOICE, ST_SILENCE, or ST_UNDEF,
 * only this labels are needed. You need to add all labels, in case
 * you want to print the internal state in string format
 */

const char *state_str[] = {
  "UNDEF", "S", "V", "INIT"
};

const char *state2str(VAD_STATE st) {
  return state_str[st];
}

/* Define a datatype with interesting features */
```

```c
typedef struct {
  float zcr;
  float p;
  float am;
} Features;

/*
 * TODO: Delete and use your own features!
 */

Features compute_features(const float *x, int N) {
  /*
   * Input: x[i] : i=0 .... N-1
   * Ouput: computed features
   */
  /*
   * DELETE and include a call to your own functions
   *
   * For the moment, compute random value between 0 and 1
   */
  Features feat;
  feat.zcr = feat.p = feat.am = (float) rand()/RAND_MAX;
  return feat;
}

/*
 * TODO: Init the values of vad_data
 */

VAD_DATA * vad_open(float rate) {
  VAD_DATA *vad_data = malloc(sizeof(VAD_DATA));
  vad_data->state = ST_INIT;
  vad_data->sampling_rate = rate;
  vad_data->frame_length = rate * FRAME_TIME * 1e-3;
  return vad_data;
}

VAD_STATE vad_close(VAD_DATA *vad_data) {
  /*
   * TODO: decide what to do with the last undecided frames
   */
  VAD_STATE state = vad_data->state;

  free(vad_data);
  return state;
}

unsigned int vad_frame_size(VAD_DATA *vad_data) {
  return vad_data->frame_length;
}

/*
 * TODO: Implement the Voice Activity Detection
```

```
 * using a Finite State Automata
 */

VAD_STATE vad(VAD_DATA *vad_data, float *x) {

  /*
   * TODO: You can change this, using your own features,
   * program finite state automaton, define conditions, etc.
   */

  Features f = compute_features(x, vad_data->frame_length);
  vad_data->last_feature = f.p; /* save feature, in case you want to show */

  switch (vad_data->state) {
  case ST_INIT:
    vad_data->state = ST_SILENCE;
    break;

  case ST_SILENCE:
    if (f.p > 0.95)
      vad_data->state = ST_VOICE;
    break;

  case ST_VOICE:
    if (f.p < 0.01)
      vad_data->state = ST_SILENCE;
    break;

  case ST_UNDEF:
    break;
  }

  if (vad_data->state == ST_SILENCE ||
      vad_data->state == ST_VOICE)
    return vad_data->state;
  else
    return ST_UNDEF;
}

void vad_show_state(const VAD_DATA *vad_data, FILE *out) {
  fprintf(out, "%d\t%f\n", vad_data->state, vad_data->last_feature);
}
```