Práctica 2: detección de actividad vocal (VAD)

Antonio Bonafonte

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 `make`.

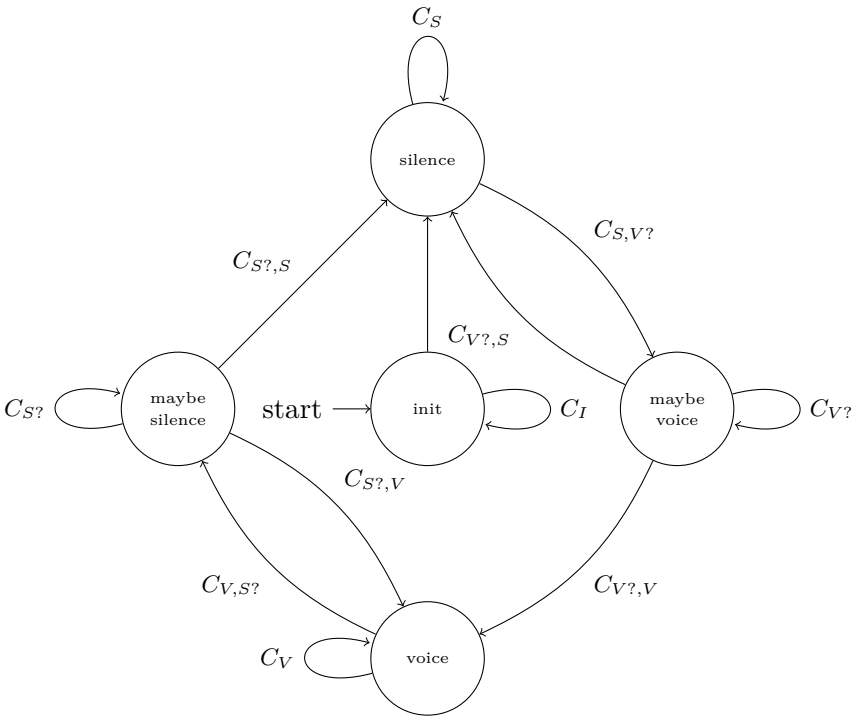
1. Introducción y fundamento teórico

En esta segunda práctica se utilizarán características de la señal de voz para realizar un detector de voz. Estos sistemas tienen varias aplicaciones, como por ejemplo en sistemas de teleconferencia, en sistemas codificación del habla, en el reconocimiento del habla, etc.

El sistema que se implementará compara la potencia de la señal de voz u otras características, con unos umbrales y puede ofrecer muy buenas prestaciones en señales sin ruido, recogidas en micrófonos cercanos (*close-talk*). Es conveniente imponer una duración mínima a la voz, ya que de lo contrario se consideraría voz cualquier ruido, o click. Análogamente, se impondrá una duración mínima al silencio para evitar *cortar* la señal en los pequeños silencios *intra-palabra* que preceden las plosivas.

En el diseño debe considerar que es más crítico perder voz que dejar pasar silencio.

El algoritmo puede definirse mediante el *autómata de estados finitos* ilustrado en la siguiente figura, donde se realiza una *transición* para cada tramo de voz (de $\approx$ 10ms). Primero hay que inicializar umbrales, en función del *nivel* del ruido, para que sea independiente del nivel de grabación. Después se permanece en el estado `silence` hasta que la potencia cumpla cierta condición, etc.

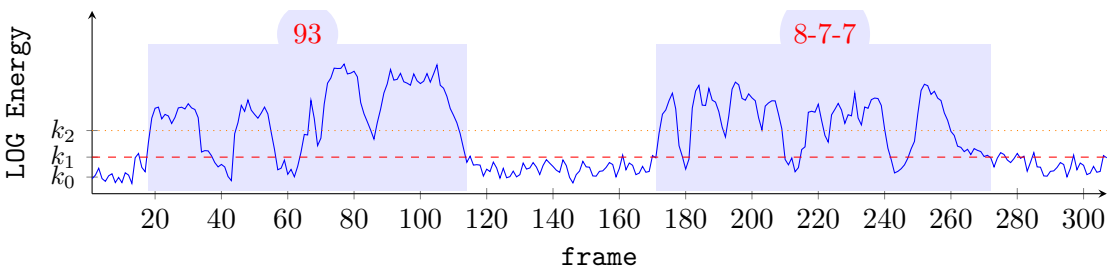


Además de los estados `silence` y `voice` hay unos estados `maybe` para los tramos en que no se puede decidir si hay o no actividad vocal, y debe retrasarse la decisión. Las condiciones de transición deben realizarse teniendo en cuenta tanto características de la voz (por ejemplo, la potencia), como restricciones en la duración del habla. Por ejemplo, la transición entre estado `maybe silence` y `silence` se realiza una vez que se supere cierto tiempo con la potencia por debajo de cierto umbral.

Las características básicas que pueden utilizarse son:

- Potencia en tramo de señal, expresado habitualmente en dB.
- Amplitud media por tramo. Es similar a la anterior, varios sistemas utilizan este valor (seguramente es un poco más sencillo definir umbrales).
- Tasa de cruces por cero (*zcr*). Esta característica ayuda a discernir silencio (ruido) de señales fricativas (`/s/, /t/`) de baja energía (con mayor contenido frecuencial, mayor *zrc*).
- Duración mínima de silencio, para evitar cortes en la mitad de palabras o dividir palabras pronunciadas de forma continua.
- Duración mínima de la voz, para evitar detectar pequeños ruidos, como golpe en el micrófono, etc.

Para explicar cómo se realiza el cambio de estado, nos apoyaremos en la siguiente figura, que muestra la potencia de una señal con dos segmentos de voz, `93` y `8-7-7`.



Se ha definido el nivel de referencia de ruido, k_0, y por encima de éste, k_1 que indica la posibilidad que haya voz. Si la señal supera en un margen de tiempo un segundo umbral, k_2, se confirma que efectivamente hay voz desde el instante que se superó k_1. Para detectar el final puede utilizarse una estrategia similar, con umbrales específicos. Además, deben imponerse las condiciones de duración mínima en voz y en silencio.

Para que el sistema sea más robusto respecto al nivel de la señal, habitualmente el valor de referencia de ruido, k_0 (potencia o amplitud media), se estima a partir de los primeros tramos de la señal. Sobre este nivel de referencia se definen los distintos umbrales k_i. Si la potencia se expresa en dB, los umbrales serán aditivos respecto nivel de referencia. Si se trabaja con amplitud media, serán multiplicativos.

Relacionado las ideas anteriores con el diagrama de estados finitos (utilizando la potencia):

- En primer lugar, se estima el nivel de potencia de ruido, k_0. El estado `init` se encarga de la inicialización de los umbrales de referencia: en los primeros tramos de señal se guarda la información necesaria para calcular los umbrales al realizar la transición al estado `silence`.
- El umbral k_1 indica cuando parece que hay señal y se realiza una transición al estado `maybe voice`. k_1 debe ser *pequeño* para permitir sonidos de poca potencia, como fricativas al inicio de frase. Puede ajustarse en relación a la potencia de ruido ($k_1 = k_0 + \alpha_1$).
- En una señal de voz tiene que haber también sonidos de mayor potencia, por ejemplo las vocales. Por eso, para validar que la señal que supera k_1 es efectivamente voz, se exige que antes de una cierta duración se haya alcanzado el umbral k_2 (que puede definirse como $k_2 = k_0 + \alpha_2$). Nótese que en este momento, los tramos de voz desde que se superó k_1 pasan a ser considerados voz y se realiza transición al estado `voice`
- Puede definirse un mecanismo similar para el final de voz con unos umbrales y duraciones que pueden ser específicos para los finales.

2. Análisis y definición del sistema

Para definir las condiciones de las transiciones entre los estados, analice una señal de voz (16kHz, mono) que contenga pausas internas (puede utilizar la señal grabada en la primera práctica). Utilizaremos todas las señales grabadas para evaluar los sistema. 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, ...).

Tareas:

1. Tal y como hizo en la primera práctica,visualice en el programa `wavesurfer` la señal y las características potencia y *zcr*, generadas en la primera práctica: potencia. Utilice el panel de *transcription* de `wavesurfer` para etiquetar los segmentos de silencio `(S)` y de voz `(V)` para poder evaluar los resultados obtenidos. Considere silencios los segmentos con cierta entidad, no pausas que no se perciban claramente. Sitúese al final de cada segmento voz/silencio y introduzca la etiqueta que corresponda. 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 definir las transiciones entre estados. Para ello, observe la señal: ¿cuál es el nivel de potencia y cruces por cero en el silencio del inicio de la señal? ¿A cuánto sube la potencia en tramos de voz *débiles*? Y en las sílabas o palabras ¿que valor de potencia, relativamente alto, podemos exigir con cierta seguridad de que siempre se alcanza? ¿Cuánto es una duración mínima razonable para una palabra o sílaba? ¿Y para un silencio significativo? Puede validar las condiciones con algún otro fichero.

3. Implementación del detector de voz (VAD)

En la práctica vamos a desarrollar unas funciones para esta detección. A pesar que `C` no es en sí un lenguaje orientado a objetos (como lo es `java` o `C++`), sí permite una programación estructurada, similar.

La cabecera `vad.h` (véase el anexo 8.1) indica las funciones que se ofrecen al usuario de este sistema. Básicamente, es una única función, llamada `vad()`, que se le pasan las muestras de un tramo de voz, e indica si es silencio o voz. En algunos casos, la *respuesta* es indefinida, correspondiente a los estados `maybe`, pues hay que esperar para confirmar la decisión. La función necesita variables de estado (memoria) que se guardar en la estructura `VAD_DATA`. Hay una función de inicialización de variables (`vad_open()`) y otra de liberación al finalizar (`vad_close()`).

El programa principal, `main_vad.c` (incluido en 8.1) ilustra cómo se utilizarán estas funciones.

Tareas:
Complete el fichero `vad.h` y el fichero `vad.c` para implementar el detector de voz, llamando a las funciones de cálculo de características (recuerde incluir el *header*, `.h`) e implementando el automata de estados finitos con las condiciones definidas. En el anexo 8.2 puede ver como implementar en `C` un sistema regido por un autómata de estados finitos.

4. Utilización del VAD

Para utilizar el detector implementado se completará el programa escrito en `main_vad()`. A dicho programa se le han de pasar dos argumentos, con el nombre del fichero `.wav` de entrada y el nombre del fichero de texto, con extensión `.vad`, con el resultado de la detección. Además, en el siguiente apartado, deberemos extender el programa para que opcionalmente generare un fichero de audio que copiará el de entrada pero escribiendo valores nulos en los segmentos de silencio. Ésta es la sintaxis de uso del programa:

```
vad input_file.wav output_file.vad [output_file.wav]
```

En el fichero `.vad`, que podrá visualizarse en `wavesurfer`, se escribirá para cada segmento de *silencio* o *audio*, una lína con el formato `"t_beg t_end label"`, por ejemplo:

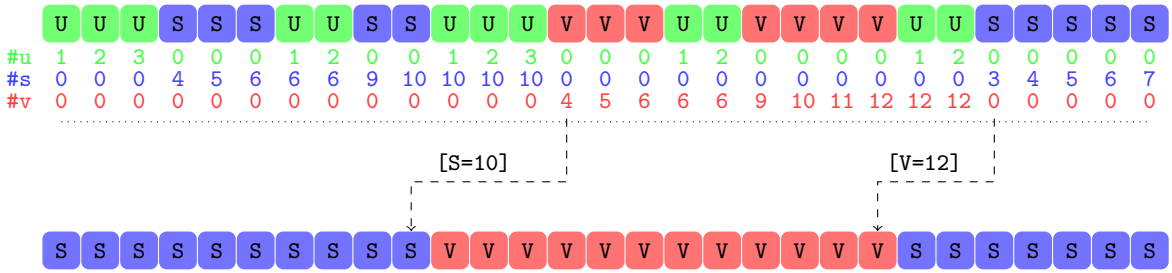
```
0.000 0.980 S
0.980 1.230 V
1.230 2.500 S
```

Tareas:

1. Compile y ejecute el programa. Para compilar, es recomendable que escriba un fichero `makefile` indicando las *reglas* de compilación (véase anexo 8.3). Compile y ejecute el programa. Visualice el resultado cargando el resultado en un panel de transcripción de `wavesurfer`. Compruebe que el detector que ha implementado funciona como esperaba.
   Para poder comprobar mejor el resultado, puede ver los estados *internos* y alguna de las características calculadas. Para ello defina la variable `verbose = DEBUG_VAD`. Ejecute y guarde los resultados en un fichero. Puede visualizarlo en un panel `data` de `wavesurfer`.

2. Si el resultado de la ejecución con una o varias grabaciones no es satisfactorio, redefina las condiciones que regula el algoritmo.

3. El programa, tal y como se proporciona, escribe en el fichero de resultados no sólo el estado `silence` y `voice`, sino también el estado `undef`, para las tramas que aún no pueden decidirse. Modifique el programa `main_vad.c` para que tenga el comportamiento deseado de etiquetar el fichero únicamente en zona de voz o zona de silencio. La siguiente figura puede ayudarle a entender el problema. En la parte superior se muestra los estados que devuelve el `vad` y en la inferior la interpretación que debe realizarse de las tramas asociadas a estados `undef`.



5. Evaluación

Una vez finalizado, se realizará una evaluación formal, comparando los resultados de su algoritmo con las etiquetas manuales. Para que los resultados sean estadísticamente significativos se deben comparar varios ficheros. Para cada fichero de audio debe generar un fichero de texto, con extensión `.vad`, como las que utiliza `wavesurfer` para las transcripciones, con una línea por etiquetay e en cada línea instante de inicio y final, y etiqueta. Por ejemplo:

```
0.00 0.57 S
0.57 1.15 V
1.15 1.48 S
```

Tareas:
Evalué el sistema con la base de datos de evaluación, escribiendo tanto para la detección de voz como la de silencio, el porcentaje de tiempo que su decisión es correcta.

En el resto de esta sección se explica como puede automatizar dicho cálculo.

En Linux las tareas repetitivas pueden automatizarse fácilmente mediante *scripts*. Utilizaremos *scripts* en *bash*. Vamos a realizar primero una breve introducción al `bash`, a continuación veremos como ejecutar su programa con un conjunto de ficheros de audio, y finalmente comentaremos como calcular las métricas de evaluación.

5.1. Introducción: *scripts* en *bash*

El *shell* más utilizado actualmente en Linux es el `bash`, que es el interprete que seguramente utiliza en su terminal (puede comprobarlo escribiendo `echo $SHELL`). En el pueden definirse variables, funciones, bucles, condiciones, etc. Por ejemplo, suponga que quieren mostrarse las dos primeras líneas de todos los ficheros `.txt` del directorio, usando el comando `head`. Puede escribir desde el terminal:

```
for name in *.txt; do echo $name; head -2 $name; done
```

Por otra parte, puede guardar estas instrucciones en un fichero de texto y podremos ejecutarlo como un programa. La primera línea debe indicar quien interpretará el fichero (en este caso, *bash*):

`Fichero` **`showheads`**

```
#!/bin/bash
for name in *.txt; do echo $name; head -2 $name; done
```

Ahora debemos cambiar los permisos del fichero, para que sea ejecutable, bien desde el entorno gráfico, o con el comando `chmod +x showheads`. Disponemos ahora un nuevo comando para mostrar las primeras líneas de los ficheros de texto.

```
./showheads
```

Aunque aquí no se ha utilizado, se pueden pasar argumentos, como hacemos en programas en `C`.

La sintaxis de la programación en `bash` es relativamente similar a `C` u otros lenguajes, por lo que no requiere demasiado tiempo aprenderla. Encontrará muchos tutoriales o introducción a la programación en `bash`. Por ejemplo, en `tldp.org` encontrará varias guías, sobre la programación `bash`, como *Advanced Bash-Scripting Guide*.

5.2. Detección de voz en un conjunto de ficheros.

El *script* `run_vad.sh` le permite ejecutar un programa con un conjunto de ficheros. Copie dicho fichero en su directorio de trabajo (o en un directorio incluído en su `PATH`), y asegurese que tiene permiso de ejecución (por ejemplo, `chmod +x run_vad.sh`). Edite el *script*, entienda qué hace, y modifíquelo para indicar su propio comando `vad`. Asegúrese que le invoca con los argumentos adecuados para su programa.

Para ejecutar su programa `vad` con varios ficheros puede escribir comandos como los siguientes:

```
./run_vad.sh f1.wav f2.wav
./run_vad.sh *.wav
./run_vad.sh f02*.wav
```

5.3. Métrica de evaluación

Una vez creados los ficheros `.vad`, puede calcular los resultados utilizando un segundo *script*, `vad_evaluation.pl`. Este es un *script* escrito en `perl`, un lenguaje de programación interpretado, con multitud de paquetes para realizar con facilidad muchas tareas. En particular, es muy adecuado para tratar datos de texto y generar informes automatizados.

Para utilizarlo deberá asegurarse que tenga permisos para ejecución y escribir comandos como los siguientes:

```
./vad_evaluation.pl file1.lab file2.lab
./vad_evaluation.pl *.lab
```

Deberá disponer en el mismo directorio:

- Ficheros `.lab`: referencias manuales, con el formato `wavesurfer`
- Ficheros `.vad`: estimación de su VAD, con el mismo formato

El *script* calcula, para cada etiqueta en los ficheros `.vad` que intervalo temporal coincide con la referencia, y muestra el porcentaje de tiempo que las etiquetas `.vad` coinciden con las referencias proporcionadas mediante los ficheros `.lab`.

6. Ampliación: eliminar partes sin audio

Finalmente, se propone utilizar el VAD para escribir un fichero de audio, sustituyendo los silencios por *ceros*. En la práctica anterior, para leer ficheros de audio, utilizamos las funciones generales de lectura de ficheros binarios. Sin embargo, no interpretamos los datos de la cabecera (número de canales, frecuencia de muestreo, etc.) y tuvimos que

realizar una conversión entre el formato de las muestras en el fichero (16 bits) al formato deseado (`float`). En esta práctica se ha utilizado la librería `sndfile` que encapsula estas funcionalidades. Permite leer audio en muchos formatos (incluyendo `wav`), con funciones muy parecidas a las que utilizamos para 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`

Existen varias funciones, `sf_read, sf_write` según los datos que se utilicen en el programa como `short, int, float, double`. Nótese que esto es independiente de como se almacenan en disco, que viene indicado al abrir el fichero mediante el campo `format` en la estructura `SF_INFO` Consúltese la documentación de estas funciones en:

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

Tareas:
Complete el programa principal para escribir el audio de entrada en el fichero opcional de audio de la salida, pero escribiendo ceros en los segmentos de silencio.

7. Instrucciones sobre las memorias de las prácticas.

En ésta y en el resto de prácticas de la asignatura se plantean problemas para los que no hay un método perfecto. Aunque se dan ciertas pautas, quedan aspectos abiertos para la experimentación, y de hecho puede aplicar cualquier método que considere. Si es posible, lea documentación sobre otras técnicas que suelen utilizarse para realizar la tarea planteada. Las memorias de las prácticas deben plantearse la explicación de un proceso de diseño y no como un registro de las actividades indicadas en el enunciado de la práctica.

Orientativamente, la memoria debe describir el sistema incluyendo los apartados:

- Resumen: en pocas palabras, cuál es el objetivo, en que se basa el método utilizado, qué resultados se han obtenido.
- Introducción: explica con más detalle el objetivo, otros métodos que se utilizan en la bibliografía, introduce en que consistirá el método de la práctica, indica los apartados de los que consta la memoria.
- Descripción del diseño, de forma teórica.
- Aspectos de implementación.
- Evaluación: describe el marco experimental, los datos utilizados y presenta los resultados obtenidos.
- Discusión de los resultados.
- Conclusiones del trabajo.
- Si ha utilizado bibliografía general (o si es concreta, incluya citas en el texto).

En general, suele ser preferible no incluir *código*, sino explicaciones, expresiones matemáticas, etc.

Recuerde que siempre debe citar las fuentes, tanto en ideas, como en resultados, gráficas, etc.

Escriba las memorias de la prácticas siguiendo la plantillas para *papers* en conferencias proporcionadas por IEEE (http://www.ieee.org/conferences_events/conferences/publishing/templates.html), con un máximo de 4 páginas.

Suba a atenea tanto la memoria (en formato `.pdf`), como los programas (no es necesario ficheros compilados) y los ficheros propios de audio o etiquetas que haya utilizado.

8. Anexo

8.1. Programas proporcionados

Cabecera del VAD (`vad.h`), con las funciones que deben definirse.

```
#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
```

vad.h

Programa `main_vad.c`, para ilustrar el uso del VAD y la librería `soundfile`.

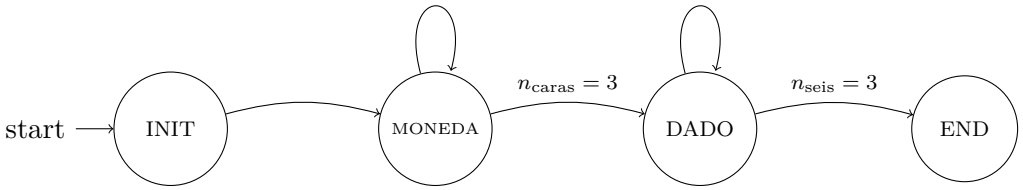
```
#include <stdio.h>
#include <stdlib.h>
#include <sndfile.h>
#include "vad.h"

#define DEBUG_VAD 0x1

int main(int argc, const 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, n_write, 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 */

  if (argc != 3 && argc != 4) {
    fprintf(stderr, "Usage: %s input_file.wav output.vad [output_file.wav]\
   n",
            argv[0]);
    return -1;
  }

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

  if (sf_info.channels != 1) {
    fprintf(stderr, "Error: the input file has to be mono: %s\n", argv[1]);
    return -2;
  }

  /* Open vad file */
  vadfile = fopen(argv[2], "wt");
  if (vadfile == 0) {
    fprintf(stderr, "Error opening output vad file: %s\n", argv[2]);
    return -1;
  }

  /* Open output sound file, with same format, channels, etc. than input */
  if (argc == 4) {
    sndfile_out = sf_open(argv[3], SFM_WRITE, &sf_info);
    if (sndfile_out == 0) {
      fprintf(stderr, "Error opening output wav file: %s\n", argv[3]);
      return -1;
    }
  }

  vad_data = vad_open(sf_info.samplerate);
  /* Allocate memory for buffer */
```

```
  frame_size   = vad_frame_size(vad_data);
  buffer       = (float *) malloc(frame_size * sizeof(float));
  buffer_zeros = (float *) malloc(frame_size * sizeof(float));
  for (i=0; i< frame_size; ++i) buffer_zeros[i] = 0.0F;

  frame_duration = (float) frame_size/ (float) sf_info.samplerate;
  t = last_t = 0;
  last_state = ST_UNDEF;

  while(1) { /* For each frame ... */
    n_read = sf_read_float(sndfile_in, buffer, frame_size);

    /* End loop when file has finished (or there is an error) */
    if  (n_read != frame_size)
      break;

    if (sndfile_out != 0) {
      /* TODO: copy all the samples into sndfile_out */
    }

    state = vad(vad_data, buffer);
    if (verbose & DEBUG_VAD)
      vad_show_state(vad_data, stdout);


    /* TODO: print only SILENCE and VOICE labels */
    /* As it is, it prints UNDEF segments but is should be merge to the
    proper value */
    if (state != last_state) {
      if (t != last_t)
        fprintf(vadfile, "%f\t%f\t%s\n", last_t * frame_duration, t *
    frame_duration, state2str(last_state));
      last_state = state;
      last_t = t;
    }

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

    t++;
  }

  state = vad_close(vad_data);
  /* TODO: what do you want to print, for last frames? */
  if (t != last_t)
    fprintf(vadfile, "%f\t%f\t%s\n", last_t * frame_duration, t *
    frame_duration, 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;
}
```

main_vad.c

8.2. Autómata de estados finitos en C

El siguiente ejemplo muestra como puede implementarse un autómata de estados finitos en el lenguaje C. Se modela un proceso de lanzar una moneda hasta que salgan tres caras y a continuación lanzar un dado hasta que salgan tres seises.




```c
#include <stdio.h>
#include <stdlib.h>
/* Create a random int i,  min_value <= i <= max_value */
int DiscreteRand(int min_value, int max_value) {
  float v = (float) rand()/ (float) RAND_MAX;
  return (int) (min_value + v * (max_value-min_value+1));
}

/* Define possible states */
typedef enum {ST_INI, ST_COIN, ST_DICE, ST_END} STATE;

int main(int argc, const char *argv[]) {
  STATE state = ST_INI;
  int count=0, n_heads, n_six;
  while (state != ST_END) {
    ++count;
    switch (state) {
    case ST_INI:
      count =  n_heads = n_six = 0;
      state = ST_COIN;
      break;
    case ST_COIN:
      if (DiscreteRand(0,1) == 0)
        if (++n_heads == 3) state = ST_DICE;
      break;
    case ST_DICE:
      if (DiscreteRand(1,6) == 6)
        if (++n_six == 3) state = ST_END;
      break;
    case ST_END:
      break;
    }
    fprintf(stdout, "State: %d; cnt=%d, nHeads=%d, nSixs=%d\n",
      state, count, n_heads, n_six);
  }
  return 0;
}
```

fsa_example.c

8.3. Compilación: librería sndfile y programa make

El programa vad necesita compilar el programa principal (main_vad.c), el ficheros con la definición del VAD (vad.cp c) y las funciones de análsis de la práctica 1 (pav_analysis.c). Además, deberá incluir la librería matemática de C (libm), para el logaritmo, y la librería libsndfile, con la funciones para lectura/escritura de ficheros de audio.

Básicamente las instrucciones son:

```
gcc -c main_vad.c
gcc -c vad.c
gcc -c pav_analysis.c

gcc main_vad.o vad-o pav_analysis.o -lm -lsndfile -o vad
```

Si el sistema no encuentra la librería sndfile deberá instalar el paquete libsndfile1-dev

```
apt-get install libsndfile1-dev
```

Si cambia un fichero con código C, deberá recompilar ese fichero y *lincar*. Si cambia un fichero de cabecera (.h), deberá recompilar los ficheros .c que lo incluyan (se dice, que *dependen* de él) y *lincar*. Para automatizar este proceso, suele utilizase el programa make, que lee un fichero makefile donde se le indican las dependencias. El fichero makefile está compuesto por *reglas*, formadas por el *nombre de la regla*, las *dependencias*, y los comandos a ejecutar si la dependencia ha cambiado. Puede escribir este fichero makefile para compilar esta práctica. Una vez escrito, sólo debe ejecutar make cada vez que haga un cambio en un .h o en un .c

```
1
2
3  vad:  main_vad.o vad.o pav_analysis.o
4    gcc main_vad.o vad.o pav_analysis.o -lm -lsndfile -o vad
5
6  main_vad.o: main_vad.c vad.h
7    gcc -c main_vad.c
8
9  vad.o: vad.c vad.h pav_analysis.h
10   gcc -c vad.c
11
12 pav_analysis.o: pav_analysis.c pav_analysis.h
13   gcc -c pav_analysis.c
```

Makefile

La primera regla (que es la que se utiliza si se invoca a make sin argumentos) en este caso es la regla vad. Indica que se ha lincar si sus dependencias (los .o) han cambiado. Para cada .o, hay una regla que indica que se ha de compilar si cambia el fichero fuente .c o los headers que incluyen.

Por ejemplo, si ejecuta make tras cambiar el fichero main_vad.c, la regla vad (línea 3) comprueba si se han de aplicar las reglas de las dependencias. La primera de ellas, main_vad.o, sí se ha de aplicar, pues depende de main_vad.c (línea 6) que, al haberse editado, es más reciente que el fichero main_vad.o. Se aplica por tanto esta regla (compilación, línea 7), no se aplican las reglas de las otras dependencias (ni línea 10 ni 13), y al haberse aplicado alguna dependencia en la regla vad, se aplican la acción de esta regla, *lincar* (línea 4).

Hay reglas muy comunes que están definidas de forma ímplicita. Por ejemplo, hay una regla implícita que indica que se si un fichero con código objeto (`.o`) depende de un código fuente (por ejemplo, `.c`), se ha de compilar cada vez que cambian las dependencias. Por ello, el `makefile` anterior puede simplificarse:

```
CFLAGS=-I. -g

vad:  main_vad.o vad.o pav_analysis.o
  gcc main_vad.o vad.o pav_analysis.o -lm -lsndfile -o vad

main_vad.o: main_vad.c vad.h
vad.o: vad.c vad.h pav_analysis.h
pav_analysis.o: pav_analysis.c pav_analysis.h
```

Makefile

La variable `CFLAGS` la utiliza la regla implícita, así que pueden añadirse los flags para el compilador.

Es importante mencionar que las *acciones* en los ficheros `makefile` (líneas 4, 7, 10 y 13 del primer ejemplo) van precedidas por tabulador, NO por espacios en blanco.

Esta estrategia de compilación es la que también utilizan los entornos integrados para desarrolladores (IDE), frecuentemente generando fichermos `makefile`. Cada vez que se cambia un fichero fuente, se actualizan las dependencias. De hecho, el compilador `gcc` también puede utilizarse para automatizar la generación de dependencias, usando el flag `-MM`. Pruebe a escribir:

```
for f in *.c; do gcc -MM $f; done
```