Práctica 1 (b): análisis por bloques

Antonio Bonafonte – profesores de la asignatura

septiembre de 2021

Resumen

En esta práctica:

- Pondremos en marcha un entorno de desarrollo en C y Linux.
  - Edición, compilación, enlazado y ejecución.
  - Mantenimiento del software: `make`.
- Repasaremos algunos conceptos básicos de C.
  - Estructuras, ficheros, punteros, memoria dinámica...
  - Paso de argumentos por línea de comandos.
- Construiremos un programa sencillo de procesado por bloques.
  - Procesado por bloques en tiempo real.
  - Características temporales sencillas:
    - Contorno de potencia.
    - Tasa de cruces por cero.
    - Amplitud media.
- Integraremos las características calculadas en `wavesurfer`.
- Presentaremos dos (o tres) herramientas de desarrollo.
  - Mantenimiento de software usando `Meson`/`Ninja`.
  - Gestión de versiones con `Git` (y `GitHub`).

Índice

1. Procesado por bloques: programa `p1`.

En muchas aplicaciones de procesado de audio es necesario realizar el procesado en tiempo real. El procesado en tiempo real significa que el sistema ha de proporcionar su salida tan pronto como dispone de la entrada, tal vez con un cierto retraso. Por ejemplo, si queremos ecualizar un concierto de música, no podemos esperar a que el concierto haya acabado para empezar a realizar la ecualización. Tampoco podemos permitirnos que la salida del ecualizador llegue al público segundos después de producirse la señal, aunque retardos de unos pocos milisegundos sí serán perfectamente asumibles.

El procesado en tiempo real puede realizarse de dos modos, fundamentalmente:

- Procesado *muestra a muestra* que, como su nombre indica, proporciona un valor de la salida para cada nuevo valor de la entrada. Ejemplos serían la amplificación de la señal, o su filtrado con estructuras de filtrado en el tiempo.
- En general, no habrá retardo entre la entrada y la salida, aunque hay que tener en cuenta la respuesta de fase del sistema de procesado.
- Procesado *por bloques*. En este caso, primero se espera a disponer de un cierto número de muestras de entrada (bloque) y, cuando el bloque está completo, se procesa y se envía el procesado a la salida.
  - Necesario cuando el procesado implica medidas estadísticas y/o frecuenciales de la señal.
  - Forzosamente existirá un retardo, como mínimo igual a la duración del bloque, entre la entrada y la salida.

La estructura básica de un sistema de procesado por bloques es:

```
mientras haya bloques de entrada disponibles:
        procesar el bloque
        enviar el bloque procesado a la salida
```

1.1. Funciones de análisis.

En esta práctica, utilizaremos bloques de 10 ms que se leerán de un fichero WAVE. El procesado del bloque consistirá en la extracción de tres medidas estadísticas de la señal de entrada:

- Potencia media de la señal en decibelios, definida como:

$$P_{dB} = 10 \log_{10} \left(\frac{1}{N} \sum_{n=0}^{N-1} x^2[n] \right) \tag{1}$$

- Amplitud media, entendida como el valor medio del valor absoluto:

$$A = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]| \tag{2}$$

- Tasa de cruces por cero: es la fracción de muestras cuyo signo es diferente al de la muestra anterior, y se normaliza con respecto al inverso de la frecuencia de Nyquist:

$$ZCR = \frac{fm}{2} \frac{1}{N-1} \sum_{n=1}^{N-1} \delta \left(\operatorname{sgn}\left(x[n]\right) \neq \operatorname{sgn}\left(x[n-1]\right) \right) \tag{3}$$

$\delta(\cdot)$ Es la función indicador, que vale uno si su argumento es cierto, y cero si es falso.

$\mathrm{sgn}(\cdot)$ Es la función signo, que vale uno si su argumento es positivo; menos uno si su argumento es negativo; y cuyo comportamiento si el argumento es cero requiere una profunda reflexión.

Se obtendrá cada una de las medidas de la señal utilizando una función C. Las tres funciones estarán ubicadas en el fichero `pav_analysis.c` e, inicialmente, su definición será:

pav_analysis.c

```c
#include <math.h>
#include "pav_analysis.h"

float compute_power(const float *x, unsigned int N) {
    return 0;
}

float compute_am(const float *x, unsigned int N) {
    return 0;
}

float compute_zcr(const float *x, unsigned int N, float fm) {
    return 0;
}
```

Donde N es el número de muestras del bloque, de modo que que, si la frecuencia de muestreo es f_m y el bloque tiene una duración T, entonces $\mathrm{N} = f_m \mathrm{T}$.

Como el prototipo[1] de estas funciones debe conocerse en el programa principal, escribimos también la cabecera (*header* en inglés) `pav_analysis.h` con sus declaraciones:

pav_analysis.h

```c
#ifndef PAV_ANALYSIS_H
#define PAV_ANALYSIS_H

float compute_power(const float *x, unsigned int N);
float compute_am(const float *x, unsigned int N);
float compute_zcr(const float *x, unsigned int N, float fm);

#endif        /* PAV_ANALYSIS_H        */
```

Las directivas de preprocesado referentes a la variable `PAV_ANALYSIS_H` garantizan que el fichero no sea incluido recursivamente. Ésta es una estrategia habitual al construir cabeceras ya que, si ésta incluye otras cabeceras, es posible que acabemos incluyendo una que, a su vez, la incluya a ella. Si no nos protegiéramos frente a esta recursividad, el sistema acabaría reventando debido al agotamiento de la memoria y después de haberse quedado temblando un buen rato.

La cabecera deberá ser incluida en el código del programa principal y, como garantía de seguridad frente a incoherencias, también se incluirá en el fichero `pav_analysis.c`.

1.2. Obtención de la señal de entrada.

Para simular el comportamiento de un sistema que accede a una fuente de señal externa, construiremos tres funciones que se incluirán en el fichero `fic_wave.c`:

[1] El prototipo de una función es la declaración de los tipos de sus argumentos y del valor retornado.

abre_wave(): Función que accede al dispositivo y obtiene un descriptor de fichero en el que podemos leer las muestras de entrada. En esta práctica el dispositivo es un fichero WAVE, pero con pocos cambios podría tratarse del micrófono, un puerto de internet, etc.

lee_wave(): Función que lee las muestras de señal del dispositivo de manera análoga a como la función de la librería estándar `fread()` lee el contenido de un fichero. Se puede obtener una descripción completa de esta función ejecutando `man 3 fread`.

cierra_wave(): Función que cierra el dispositivo, liberándolo para otros programas o funciones.

Por ahora, la función `abre_wave()` simplemente saltará la cabecera del fichero, es decir, los 44 primeros bytes del fichero, y fijará la frecuencia de muestreo a $f_m = 16\,\text{kHz}$.

fic_wave.c

```c
#include <stdio.h>
#include "fic_wave.h"

FILE    *abre_wave(const char *ficWave, float *fm) {
    FILE    *fpWave;

    if ((fpWave = fopen(ficWave, "r")) == NULL) return NULL;
    if (fseek(fpWave, 44, SEEK_SET) < 0) return NULL;

    *fm = 16000;

    return fpWave;
}

size_t   lee_wave(void *x, size_t size, size_t nmemb, FILE *fpWave) {
    return fread(x, size, nmemb, fpWave);
}

void    cierra_wave(FILE *fpWave) {
    fclose(fpWave);
}
```

Algunas observaciones acerca de la gestión de errores:

- En `abre_wave()`, se comprueban todas las condiciones de error posibles. En caso de producirse uno, se indica devolviendo el puntero a cero (`NULL`).
- La función `lee_wave()` es simplemente la invocación de una función de librería, así que dejamos en manos de la función superior la gestión de eventuales errores.
- La función `cierra_wave()` no realiza ningún tipo de comprobación de errores, ya que es una llamada a la función de librería `fclose()` que tampoco la realiza.

Como en el caso de `pav_analysis.c`, este fichero de código debe ir acompañado de su cabecera, que se incluirá tanto en el programa principal, para que éste *conozca* los prototipos de las funciones usadas, como en el propio `pav_analysis.c`, para que el compilador nos avise de posibles incoherencias.

fic_wave.h

```c
#ifndef FIC_WAVE_H
#define FIC_WAVE_H

```

```
FILE   *abre_wave(const char *ficWave, float *fm);
size_t lee_wave(void *x, size_t size, size_t nmemb, FILE *fpWave);
void   cierra_wave(FILE *fpWave);

#endif        /* FIC_WAV_H        */
```

1.3. Programa principal.

Inicialmente, para cada bloque se escribirá en pantalla una línea de texto con el resultado del procesado expresado como cuatro valores separados por tabulador: el índice del bloque, seguido de las tres medidas estadísticas. Más adelante, deberá modificarse el programa para que esta salida se escriba en un fichero y no en la pantalla.

El programa principal deberá realizar las siguientes funciones básicas:

- Abrir el dispositivo de entrada para acceder a los bloques de señal.
- Iterar mientras haya bloques disponibles a la entrada.
  - Convertir el bloque de entero a real acotado entre menos uno y uno.
  - Procesar cada bloque, obteniendo el valor de sus tres medidas estadísticas.
  - Escribir la salida en pantalla.
- Liberar los recursos utilizados (ficheros abiertos, memoria dinámica...).

p1.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include "pav_analysis.h"
#include "fic_wave.h"

int main(int argc, char *argv[]) {
    float durTrm = 0.010;
    float fm;
    int   N;
    int   trm;
    float *x;
    short *buffer;
    FILE  *fpWave;

    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Empleo: %s inputfile [outputfile]\n", argv[0]);
        return -1;
    }

    if ((fpWave = abre_wave(argv[1], &fm)) == NULL) {
        fprintf(stderr, "Error al abrir el fichero WAVE de entrada %s (%s)\n",
        ↪  argv[1], strerror(errno));
        return -1;
    }
```

```
26
27      N = durTrm * fm;
28      if ((buffer = malloc(N * sizeof(*buffer))) == 0 ||
29          (x = malloc(N * sizeof(*x))) == 0) {
30          fprintf(stderr, "Error al ubicar los vectores (%s)\n", strerror(errno));
31          return -1;
32      }
33
34      trm = 0;
35      while (lee_wave(buffer, sizeof(*buffer), N, fpWave) == N) {
36          for (int n = 0; n < N; n++) x[n] = buffer[n] / (float) (1 << 15);
37
38          printf("%d\t%f\t%f\t%f\n", trm, compute_power(x, N),
39                                          compute_am(x, N),
40                                          compute_zcr(x, N, fm));
41          trm += 1;
42      }
43
44      cierra_wave(fpWave);
45      free(buffer);
46      free(x);
47
48      return 0;
49  }
```

En el programa se comprueban todas las condiciones predecibles de error y, en caso de producirse, se escribe un mensaje explicativo del mismo y se interrumpe la ejecución. Cuando se produce un error al ejecutar una función de la librería estándar, como `malloc()`, ésta ésta da a la variable global `errno` un valor entero distinto de cero. Utilizamos este valor para obtener un mensaje explicativo de cuál ha sido el problema usando la función `strerror()`. Por ejemplo, si intentamos ejecutar el programa con un fichero que no existe, el mensaje que obtendremos será:

```
usuario:~/PAV/P1$ ./p1 fichero_que_no_existe.wav
Error al abrir fichero_que_no_existe.wav (No such file or directory)
```

Mientras que, si no tenemos permiso de lectura en el fichero, el mensaje es:

```
usuario:~/PAV/P1$ ./p1 fichero_sin_permirso_de_letura.wav
Error al abrir fichero_sin_permirso_de_lectura.wav (Permission denied)
```

2. Compilado, enlazado y ejecución de p1.

2.1. Compilado y enlazado de programas.

El sistema de desarrollo de software por excelencia en Linux es el paquete *GNU Compiler Collection*. Este paquete incluye compiladores para múltiples lenguajes de programación, como C (`gcc`), C++ (`g++`), Java (`gcj`), etc., y está disponible para múltiples arquitecturas y sistemas operativos, tanto en compilación nativa como cruzada (compilación de un programa en un sistema, por ejemplo: Linux, para ser ejecutado en otro distinto, por ejemplo: Windows).

La construcción de un programa a partir de su código fuente comprende dos fases:

Compilado: Es el proceso de convertir el código fuente en un código intermedio, denominado *código objeto* en el que cada instrucción se corresponde con una, y sólo una, instrucción de código máquina del procesador.

- El código objeto, no obstante, no es ejecutable directamente por el procesador por dos motivos: es frecuente que el programa incluya más códigos objeto correspondientes a otros códigos fuente o a funciones de librería; y, hasta el momento de unir todas las piezas, es imposible determinar en qué posiciones de memoria se ubicarán las variables y funciones del programa.

Enlazado: El enlazado (en inglés, *link*, a menudo "traducido" como *lincado*) es el proceso de unir todas las funciones y variables del programa, asignarles una posición concreta en memoria y convertir el resultado a código máquina ejecutable por el procesador.

El compilador y enlazador de GNU para código C es `gcc`. Este es un programa con muchas opciones (que se pueden consultar ejecutando `gcc --help`, aunque `man gcc` proporciona mucha más información). Sin embargo, el proceso para generar un programa a partir de su código fuente puede ser relativamente sencillo. Así, la orden siguiente sería bastante para obtener el programa `p1`:

```
usuario:~/PAV/P1$ gcc -o p1 p1.c pav_analysis.c fic_wave.c
```

La opción `-o p1` es necesaria para que el programa resultante se llame `p1`. En caso contrario, el programa toma el nombre por defecto `a.out`, lo cual no es muy conveniente si queremos tener más de un programa en nuestro ordenador...

Alternativamente, podemos compilar cada uno de los códigos fuente con la orden `gcc -c fichero.c`. Esta orden generará un fichero objeto, `fichero.o`, para cada uno de ellos, que podremos enlazar de manera semejante al caso anterior:

```
usuario:~/PAV/P1$ gcc -c p1.c
usuario:~/PAV/P1$ gcc -c pav_analysis.c
usuario:~/PAV/P1$ gcc -c fic_wave.c
usuario:~/PAV/P1$ gcc -o p1 p1.o pav_analysis.o fic_wave.o
```

Cuando el programa esté completo, seguramente llame a funciones de librería, en concreto, y como mínimo, a la función `log10()` de la librería matemática `math.h`. Para poder enlazar el programa será necesario indicar dónde está la librería correspondiente que, en este caso, es `/usr/lib/x86_64-linux-gnu/libm.so`. Esto lo podemos hacer de dos maneras:

- Indicando la librería, con su ruta completa en la orden de lincado:

```
usuario:~/PAV/P1$ gcc -o p1 p1.c pav_analysis.c fic_wave.c                ···
↪  /usr/lib/x86_64-linux-gnu/libm.so
```

- Si la librería está en una de las ubicaciones estándar (como es el caso) y su nombre responde a la forma `libXXX.so`, donde `XXX` es el nombre clave de la librería, podemos indicar a `gcc` que enlace con ella con la opción `-lXXX`:

```
usuario:~/PAV/P1$ gcc -o p1 p1.c pav_analysis.c fic_wave.c -lm
```

Otras opciones de interés en `gcc` son la opción `-O`, que indica a gcc que optimice el programa generado para que ocupe menos y se ejecute más rápido, y `-g`, que le indica que se incluya información del código fuente en el ejecutable para facilitar la depuración usando debuggers como `gdb` o `ddd`.

2.2. Ejecución de programas.

Una vez compilado y enlazado el programa, lo cual quiere decir que también se habrán resuelto todos los errores sintácticos y de enlazado, podemos ejecutar el programa para comprobar su funcionamiento. En principio, en Linux y, en general, todos los sistemas operativos semejantes a Unix, es necesario indicar la ruta completa del programa para poderlo ejecutar. Así pues, la orden para ejecutar `p1` sería:

```
usuario:~/PAV/P1$ /home/usuario/PAV/P1/p1
Empleo: /home/usuario/PAV/P1/p1 inputfile [outputfile]
```

Por suerte, tenemos varios atajos para abreviar esta invocación:

- El directorio raíz del usuario se indica con `~`. Análogamente, el directorio raíz de un usuario distinto es `~usuario`. Así pues, podemos ejecutar:

  ```
  usuario:~/PAV/P1$ ~/PAV/P1/p1
  Empleo: /home/usuario/PAV/P1/p1 inputfile [outputfile]
  ```

- Si el programa está en el directorio actual (`.`), lo podemos indicar explícitamente:

  ```
  usuario:~/PAV/P1$ ./p1
  Empleo: ./p1 inputfile [outputfile]
  ```

- Si no se indica una ruta explícitamente, Linux busca el programa en una serie de directorios por defecto. La lista de directorios se denomina *path* y está indicada por el contenido de la variable de entorno `PATH`. Esta variable debe contener todos los directorios en los que queremos buscar los programas, separados por dos puntos y en el orden en el que los queremos buscar.

  Por motivos históricos (en teoría, de seguridad), Linux nunca incluye el directorio actual en el path por defecto. pero nosotros sí podemos hacerlo. La mejor manera de hacerlo es añadir una orden al final del fichero de comandos de inicio de sesión (típicamente `~/.bashrc` o `~/.profile`), añadiendo `.` al path:

  ```
  export PATH+=:.
  ```

  Al incluir esta orden, el path incluirá el directorio actual en las nuevas sesiones que se inicien. Par que también tenga efecto en la sesión en ejecución, hay que ejecutar este script explícitamente, pero sin que esto se haga en un nuevo shell porque, si lo hiciera, la variable recuperaría su valor original al terminar el script. La ejecución de scripts en Bash en el shell de la llamada se consigue con la orden `source` (también se puede usar un punto (`.`) con el mismo efecto). A continuación, el programa se podrá ejecutar simplemente invocando el nombre del programa:

  ```
  usuario:~/PAV/P1$ source ~/.profile
  usuario:~/PAV/P1$ p1
  Empleo: p1 inputfile [outputfile]
  ```

Tareas:

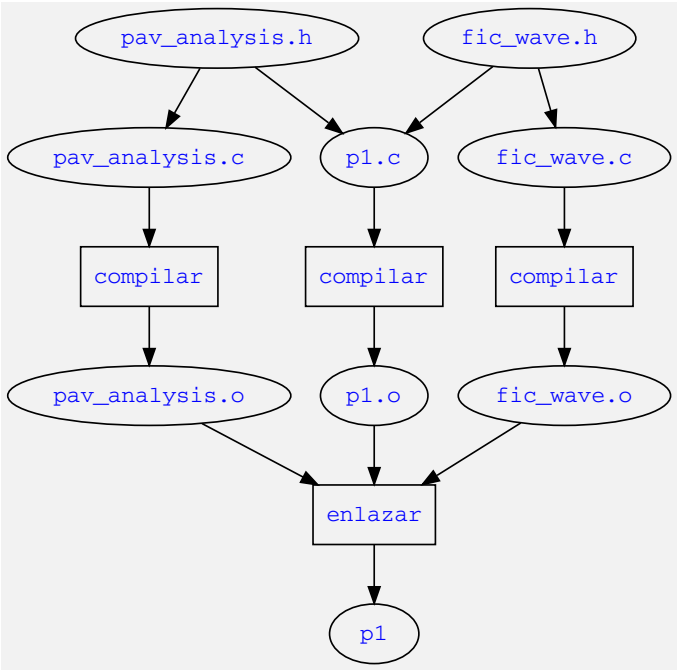
- Escriba los códigos fuente de `pav_analysis.c`, `pav_analysis.h`, `fic_wave.c`, `fic_wave.h` y `p1.c`.
- Compile y enlace el programa. Corrija los errores de sintaxis y enlazado hasta conseguirlo.
- Modifique `~/.bashrc` (o `~/.profile`, o el que corresponda a su distribución de Linux), para incluir el directorio actual en el path.
- Compruebe que el programa se ejecuta correctamente.

3. Mantenimiento de programas: `make`.

El código que hemos diseñado para construir el programa `p1` está formado por cinco ficheros: tres son el código fuente con las funciones que necesitamos; los otros dos son cabeceras con la declaración de las funciones, pero que podrían tener otros datos relevantes para la ejecución del programa. Por ejemplo, `pav_analysis.h` podría definir la duración del bloque de señal, o `fic_wave.h` podría definir el tamaño en bytes de la cabecera del fichero WAVE.

Por tanto, si cualquiera de estos cinco ficheros cambia, será necesario reconstruir el programa. Pero eso no quiere decir que sea necesario volver a compilar los tres códigos fuente. Por ejemplo, si sólo modificamos `pav_analysis.c`, sólo necesitaremos volver a compilar este fichero y enlazar el resultado con los códigos objeto `fic_wave.o` y `p1.o` para obtener el programa. No es necesario que volvamos a compilar `fic_wave.c` y `p1.c` porque no se han modificado. Sin embargo, si lo que modificamos es `pav_analysis.h`, además de volver a compilar `pav_analysis.c`, deberemos compilar `p1.c`, porque éste contiene la cabecera modificada y eso podría afectar a su comportamiento.

La gráfica siguiente, construida usando `graphviz`, muestra las dependencias del programa `p1`. Si hay cualquier cambio en alguno de los ficheros, indicados con elipses, deberemos seguir el gráfico hacia abajo desde el nodo correspondiente, realizando todas las acciones que nos encontremos por el camino, indicadas con rectángulos.



Un sistema de gestión de dependencias realiza esta tarea de manera automática, decidiendo qué partes de un proyecto se deben reconstruir cada vez que un grupo de ficheros es modificado. El gestor de dependencias clásico es el programa `make`, inicialmente creado para los primeros sistemas Unix, pero que, dada su utilidad, pronto se extendió a todo tipo de sistema.

3.1. `Makefile`

El programa `make` se basa en resolver las reglas de dependencia expresadas en un fichero de texto cuyo nombre puede ser `GNUmakefile`, `Makefile` o `makefile`. El primero de estos nombres está desaconsejado, y sólo debería usarse si el fichero contiene extensiones específicas de la versión de GNU de `make` (`gmake`, aunque, en sistemas Linux, `make` suele ser un alias de éste). Además, suele ser una buena idea llamar al fichero `Makefile`. Esto es así porque los ficheros *normales* suelen estar escritos en minúscula. Es una buena práctica nombrar a los ficheros más importantes, como `README.txt`, `TODO.txt`, `LICENSE.txt`, etc. con mayúsculas, de manera que al listar el contenido del directorio aparezca agrupados y antes del resto de ficheros, de manera que ganen visibilidad.

En este fichero se escriben las reglas que queremos resolver con la sintaxis siguiente:

```
objetivo: dependencias
        comandos
```

`objetivo:` Fichero que queremos mantener actualizado.

- En general será un programa, pero puede ser cualquier otra cosa.

`dependencias:` Lista de ficheros de los que depende el objetivo.

- La lista de dependencias se separa del objetivo por un carácter dos puntos (:); los ficheros que forman la lista se separan entre sí mediante espacios en blanco.
- Un objetivo está obsoleto cuando alguno de los ficheros indicados en la lista de dependencias es más moderno que éste.
- Un caso particular es cuando el objetivo no existe. En este caso, independientemente del estado de las dependencias, se considerá que el objetivo está obsoleto y debe reconstruirse.

`comandos:` Sucesión de comandos que han de ejecutarse cuando el objetivo esté obsoleto.

- Cada línea debe comenzar por tabulador; en caso contrario, `make` no la reconocerá como tal.

Un `makefile` puede tener más de una regla. Es este caso, `make` intentará actualizar el primer objetivo que encuentre. Si alguna de las dependencias aparece como objetivo de otra regla, entonces `make` intentará primero actualizar esta dependencia ejecutando la regla correspondiente.

Por ejemplo: en nuestro proyecto, `p1` depende de los objetos `p1.o`, `pav_analysis.o` y `fic_wave.o`; a su vez, cada uno de estas dependencias depende de sus respectivos códigos fuente y cabeceras. Así pues, un `makefile` posible sería:

Makefile

```
1 p1: p1.o pav_analysis.o fic_wave.o
2     gcc -o p1 p1.o pav_analysis.o fic_wave.o -lm
3
4 p1.o: p1.c pav_analysis.h fic_wave.h
5     gcc -c p1.c
```

```
6
7  pav_analysis.o: pav_analysis.c pav_analysis.h
8      gcc -c pav_analysis.c
9
10 fic_wave.o: fic_wave.c fic_wave.h
11     gcc -c fic_wave.c
```

Al ejecutar `make` por primera vez (y suponiendo que no hay errores), el resultado es:

```
usuario:~/PAV/P1$ make
gcc -c p1.c
gcc -c pav_analysis.c
gcc -c fic_wave.c
gcc -o p1 p1.o pav_analysis.o fic_wave.o -lm
```

Con lo que el programa ha sido reconstruido. Si, ahora, volvemos a ejecutar `make`:

```
usuario:~/PAV/P1$ make
make: 'p1' is up to date.
```

Indicando que el objetivo ya está al día y no es necesario realizar nada. Sin embargo, si modificamos `pav_analysis.c`:

```
usuario:~/PAV/P1$ make
gcc -c pav_analysis.c
gcc -o p1 p1.o pav_analysis.o fic_wave.o -lm
```

Es decir, `make` recompilará `pav_analysis.c` y enlazará los tres objetos para crear `p1`, pero no recompilará `p1.c` ni `fic_wave.c`, porque éstos ya estaban al día.

3.1.1. Reglas implícitas.

`make` es un programa muy listo, que sabe hacer muchas cosas por sí solo. Una de las cosas que sabe es cuáles son las reglas más habituales dados un objetivo y unas dependencias. Por ejemplo, si el objetivo es un código objeto, y la lista de dependencias incluye un código fuente con el mismo nombre que el objeto, pero con una extensión adecuada, `make` es capaz de intuir que lo que queremos hacer es compilar el código fuente para obtener el objeto. Y lo mismo pasa si el objetivo es un nombre de fichero sin extensión. En este caso, `make` intuye que lo que queremos es enlazar las dependencias para obtener el programa. Son las *reglas implícitas* que, además, pueden ser escritas o adaptadas por el usuario.

Dejando a `make` escoger las reglas a usar, el fichero `Makefile` queda:

Makefile

```
3  p1: p1.o pav_analysis.o fic_wave.o
4
5  p1.o:            p1.c pav_analysis.h fic_wave.h
6  pav_analysis.o:  pav_analysis.c pav_analysis.h
7  fic_wave.o:      fic_wave.c fic_wave.h
```

Al ejecutar `make` ahora, el resultado es:

```
usuario:~/PAV/P1$ make
cc    -c -o p1.o p1.c
cc    -c -o pav_analysis.o pav_analysis.c
cc    -c -o fic_wave.o fic_wave.c
cc   p1.o pav_analysis.o fic_wave.o   -o p1
pav_analysis.o: In function `compute_zcr':
pav_analysis.c:(.text+0x43): undefined reference to `log10'
collect2: error: ld returned 1 exit status
<builtin>: recipe for target 'p1' failed
make: *** [p1] Error 1
```

¡Vaya, ha habido errores!

Esto ha sucedido porque `make` es listo, pero no tanto como para saber que una de las funciones llama a otra que no está en la librería estándar, sino en la matemática, `log10()`. Por suerte, `make` permite configurar las reglas implícitas a partir de, entre otras cosas, variables de entorno. En concreto, la variable `LDLIBS` indica las librerías con las que se tienen que enlazar los programas del `makefile`. Estas variables tienen unos valores por defecto que, habitualmente, no deseamos eliminar. En lugar de ello, lo que habitualmente se quiere es **añadir** nuestros valores a los que ya están definidos. Esto se hace con la orden `LDLIBS += -lm`, con lo que `Makefile` queda:

Makefile

```
1 LDLIBS += -lm
2
3 p1: p1.o pav_analysis.o fic_wave.o
4
5 p1.o:            p1.c pav_analysis.h fic_wave.h
6 pav_analysis.o:  pav_analysis.c pav_analysis.h
7 fic_wave.o:      fic_wave.c fic_wave.h
```

Puede encontrarse más información de las reglas implícitas y, en general, de todas las características de `make` en la página web de GNU Make.

Tarea:

Edite el fichero `Makefile` y compruebe el funcionamiento de `make`. En concreto, compruebe que si alguno de los códigos fuentes de `p1` o de los ficheros de cabecera cambia, se ejecutan las órdenes justas y necesarias para reconstruir el programa; y, si ninguno de ellos cambia, `make` simplemente no hace nada.

4. Control de versiones con Git.

Un sistema de control de versiones (**CVS**) permite gestionar los cambios sufridos por un conjunto de ficheros a lo largo de su historia. Aunque pueden usarse para mantener cualquier tipo de fichero, por ejemplo: las distintas versiones de un tema grabado en un estudio, suelen estar asociados a la gestión de paquetes de software.

Entre sus capacidades más básicas (aunque de las más útiles), está la de permitir regresar a un estado del paquete anterior en el tiempo. Un caso típico sería el siguiente: construimos un programa y conseguimos que funcione correctamente, pero lo modificamos para que funcione mejor... y el resultado es un desastre. Si el paquete estaba (bien) gestionado con CVS, podremos restaurar el paquete al punto en que sí funcionaba, o podremos ver y analizar los cambios que provocaron el error.

Las principales ventajas de casi todos los sistemas CVS son:

- Proporciona un mecanismo inteligente de almacenado de los ficheros.
  - Aunque hagamos alguna trastada, como borrar un fichero importante, en un paquete gobernado por VCS es posible regresar en el tiempo al instante previo a ella.
- Permite hacer un seguimiento de los cambios realizados en cada uno de los ficheros del paquete: quién y cuándo lo hizo, y, si se dedica un poco de tiempo a ello, por qué se hizo.
- Permite que distintos programadores contribuyan al proyecto de manera independiente.
  - En la mayoría de los casos, las modificaciones hechas por cada uno de ellos se pueden integrar en una nueva versión.
  - Si existe conflicto, porque distintos programadores han *metido la mano* en exactamente el mismo sitio, el VCS proporciona información precisa para intentar resolverlo.
- Permite mantener versiones distintas de un mismo paquete, o integrarlas en una nueva.

4.1. Git.

Git, cuyo nombre no quiere decir nada (aunque, según su autor puede querer decir "*goddamn idiotic truckload of sh*t*"), es el VCS más extendido en la actualidad, en parte debido a que es, seguramente, el más potente y, además, está disponible en software libre. Lo desarrolló el *padre* de Linux, Linus Torvalds, cuando los propietarios del VCS que se utilizaba para desarrollar el núcleo de Linux, BitKeeper, decidieron convertirlo en software de pago. Torvalds buscó una alternativa de código libre que cumpliera una serie de condiciones, pero, como no encontró ninguna que le gustara, desarrolló Git.

La idea inicial de Torvalds era construir un VCS de bajo nivel que fuera manejado desde aplicaciones de alto nivel *user-friendly*. Pero, como el resultado le encantó, a él y a sus colaboradores, lo dejaron tal cual. El resultado es absolutamente farragoso, http://stevelosh.com/blog/2013/04/git-koans/.

Entre las características de Git, destaca el hecho de que es completamente distribuido. Esto quiere decir que si varios usuarios obtienen una copia de una versión concreta del proyecto, todos reciben, no sólo esa versión, sino todas las versiones anteriores también. De este modo, aunque podamos declarar un repositorio concreto como el oficial o central, todos los usuarios tienen el mismo contenido que éste, y cualquiera de ellos puede asumir el papel de repositorio oficial si el anterior sufre una catástrofe. Por otro lado, cada usuario puede trabajar en su copia local del proyecto sin necesidad de estar conectado a ningún repositorio externo (trabajo *off-line*). Sólo cuando dispone de una versión definitiva de su copia local será necesario conectarse al repositorio central para actualizar su contenido, que estará más *limpio* porque no será necesario que el usuario suba todos los estados intermedios.

4.2. Gestión de la práctica P1 usando Git.

En la gestión de un repositorio, Git utiliza distintos datos del usuario. Entre ellos, dos son imprescindibles y Git no permitirá el trabajo con el repositorio si no están definidos previamente: el nombre del usuario y su dirección de correo electrónico. Estos datos pueden proporcionarse de manera independiente para cada proyecto o de manera global para todos los proyectos gestionados por un usuario Linux.

Por ejemplo, para definir el nombre y correo electrónico por defecto de un usuario Linux, ejecutaríamos las órdenes siguientes:

```
usuario:~/PAV/P1$ git config --global user.email "usuario.pav@upc.edu"
usuario:~/PAV/P1$ git config --global user.name "Usuario PAV ETSETB"
```

La información por defecto se almacena en el fichero oculto del directorio raíz del usuario `~/.gitconfig`. Es posible editar este fichero directamente, sin necesidad de ejecutar los comandos anteriores. En mi caso, el contenido del fichero es el siguiente:

~/.gitconfig

```
[user]
        email = usuario.pav@upc.edu
        name = Usuario PAV ETSETB
[core]
        editor = vim
```

Una vez inicializado el usuario, hemos de inicializar el proyecto. Para ello, vamos al directorio `PAV/P1` y ejecutamos el comando `git init`. Podemos ver cuál es el estado inicial con `git status`:

```
usuario:~/PAV/P1$ git init
Initialized empty Git repository in /mnt/d/usuario/UbuntuOnWindows/PAV/P1/.git/
usuario:~/PAV/P1$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Makefile
        fic_wave.c
        fic_wave.h
        fic_wave.o
        hola.wav
        hola8.au
        p1
        p1.c
        p1.o
        pav_analysis.c
        pav_analysis.h
        pav_analysis.o

nothing added to commit but untracked files present (use "git add" to track)
```

Git nos informa de que hay una serie de ficheros que están en el directorio del proyecto, pero que no se han incorporado al repositorio (no se están siguiendo por Git). Estos ficheros aparecen en rojo para remarcar el hecho de que no están incorporados, pero entre ellos nos encontramos ficheros que queremos gestionar, y otros para los que no tiene mucho sentido hacerlo.

De cara a gestionar las versiones del software con Git, sólo estamos interesados en los ficheros y directorios que pueden variar y no se pueden generar automáticamente. No tiene sentido gestionar los ficheros ejecutables u objeto (`.o`), porque son ficheros que se pueden crear fácil y automáticamente a partir del código. Tampoco tiene mucho sentido, en este caso, gestionar las señales de audio `hola.wav` y `hola8.au`. Lo que sí queremos que gestione Git es el código fuente, incluyendo el archivo `Makefile`.

En Git, el mecanismo para seleccionar qué archivos (y directorios) se gestionan, y cuáles no, es el fichero de texto `.gitignore`. Todos los ficheros y directorios que respondan a las máscaras incluidas en el fichero serán excluidos de la gestión por Git, salvo que vayan precedidos por el carácter exclamación, en cuyo caso serán incluidos aunque cumplan alguna máscara precedente. Puede consultarse la sintaxis completa del fichero en el enlace .gitignore.

Construimos un fichero `.gitignore` de manera que Git sólo gestione lo que realmente nos interesa:

.gitignore

```
1 *
2 */
3
4 !*akefile
5 !*.[ch]
6 !.gitignore
```

La primera línea de `.gitignore` (`*`) indica que no debe realizarse el seguimiento de ningún fichero, y la segunda, de ningún directorio (esto lo indicamos con la barra final). Por tanto, en principio, Git no gestionará nada de nada. A continuación, le indicamos que sí queremos gestionar cualquier fichero cuyo nombre acabe en `akefile` (eso incluirá `Makefile`, `makefile` y `GNUmakefile`). Esto lo conseguimos anteponiendo la exclamación a la máscara; de hecho, esa exclamación indica que los ficheros que verifiquen la máscara son excepciones a las reglas anteriores que excluyen todos los ficheros y directorios. Finalmente, hacemos lo mismo para todos los ficheros de código fuente C (extensiones `.c` y `.h`), y el propio fichero `.gitignore`.

Ahora podemos añadir todo el directorio al proyecto mantenido por Git; será la versión inicial del proyecto, con los ficheros originales. Primero usamos el comando `git add .` para añadir el directorio actual completo (salvo lo excluido por `.gitignore`); a continuación usamos el comando `git commit` para realizar la primera confirmación del proyecto; y, finalmente, ejecutamos `git log` para ver el historial de versiones del proyecto:

```
usuario:~/PAV/P1$ git add .
usuario:~/PAV/P1$ git commit -a -m 'Ficheros originales de la práctica P1'
7 files changed, 113 insertions(+)
create mode 100644 .gitignore
create mode 100644 Makefile
create mode 100644 fic_wave.c
create mode 100644 fic_wave.h
create mode 100644 p1.c
create mode 100644 pav_analysis.c
create mode 100644 pav_analysis.h

usuario:~/PAV/P2$ git log
commit a704ce98804413ab81a592c4cd334fa17c3a0f89 (HEAD -> master)
Author: Usuario PAV ETSETB <usuario.pav@upc.edu>
Date:   Sat Sep 28 02:29:36 2019 +0200

    Ficheros originales de la práctica P1
```

A partir de este momento, al 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 el comando `git commit` con un mensaje explicativo de los cambios incorporados en ella.

4.3. Más información acerca de Git.

El comando `git help` permite obtener información rápida acerca de los comandos, funciones y argumentos de git, desde la línea de comandos. Su modo de empleo, para obtener información acerca de *algo* es `git help algo`. Por ejemplo, `git help commit` muestra el modo de empleo, con todas sus opciones, del comando commit (hay más de 400 líneas). Un caso de especial interés para principiantes es `git help everyday`, que muestra los comandos básicos de `git` en función de su modo de interacción con el mismo: usuario independiente, participante colaborativo, integrador de proyecto o administrador del repositorio.

En el libro online (y en castellano) Pro Git puede encontrar información muy completa de Git y (en la medida de lo posible) fácil de entender. También puede usar el documento de Atenea Flujos de trabajo en Git, que no es ninguna maravilla (y, además, está incompleto), pero explica un poco más en detalle cómo empezar a gestionar un proyecto usando Git.

Tareas:

- Inicialice, de modo global, la información del usuario de Git.
- Inicialice la gestión de versiones para el proyecto de la primera práctica.
- Escriba el fichero `.gitignore` de manera que Git sólo gestione el código fuente de `PAV/P1`.
- Realice la primera confirmación (*commit*) con los ficheros originales de la práctica.

5. Ejercicios, memoria y entrega.

Ejercicios básicos:

1. Complete el código de `pav_analysis.c` para que las funciones de análisis calculen los parámetros estadísticos correspondientes.
2. Modifique `p1.c` para que el programa acepte un argumento, que será el fichero de audio de entrada, o dos, en cuyo caso el segundo será un fichero de texto en el que se escribirá el resultado del análisis.
3. Modifique `fic_wave.c` para que `abre_wave()` extraiga la información de la cabecera del fichero WAVE, en concreto, de la frecuencia de muestreo.
4. Use su programa `p1` para obtener los parámetros estadísticos de la señal grabada en el laboratorio y visualícelos con `wavesurfer`.
   - Compare el resultado obtenido para la potencia media con el calculado por `wavesurfer`. ¿A qué se deben las diferencias?

     Es posible que deba cambiar los parámetros de `wavesurfer` para poder realizar esta comparación.

Ejercicios de ampliación:

1. Modifique los ficheros necesarios para realizar una interpretación completa de la cabecera de los ficheros WAVE, adaptando el comportamiento de `p1` a ellos.
   - Como mínimo, deberá comprobarse que la señal es de un solo canal (mono) y codificada con PCM Lineal de 16 bits. En caso de no serlo, el programa deberá dar un mensaje de error y finalizar la ejecución.

- Como máximo, deberá adaptar el funcionamiento del programa a alguna característica de la señal. Por ejemplo, si la señal es estéreo...

2. Es habitual utilizar enventanado para evitar el ruido debido a los bordes del bloque en el cálculo de la potencia. Éste consiste en tomar bloques de duración T_{Long} tomados con un desplazamiento de T_{Desp}, con lo cual aparece solapamiento si $T_{Desp} < T_{Long}$, y multiplicar la señal por una ventana adecuada. Típicamente, la ventana utilizada es la de Hamming y se toma un desplazamiento $T_{Desp} = T_{Long}/2$.

   Siendo $x_i[n] = x[iM + n]p_N[n]$ una señal con las muestras del i-ésimo bloque de la señal $x[n]$ para una longitud $N = f_m T_{Long}$ y un desplazamiento de $M = f_m T_{Desp}$, el cálculo de la potencia usando la ventana $w[n]$ sigue la fórmula:

$$P_{dB}[i] = 10 \log_{10} \left(\frac{\sum_{n=0}^{N-1} (x_i[n]w[n])^2}{\sum_{n=0}^{N-1} w^2[n]} \right) \tag{4}$$

   Modifique los ficheros necesarios para calcular la potencia de la señal utilizando enventanado con ventana de Hamming, una duración del bloque $T_{Long} = 20ms$ y un desplazamiento $T_{Desp} = 10ms$. Compare gráficamente el resultado con el cálculo de la potencia sin enventanado.

5.1. Entrega.

Como resultado de esta práctica debe subirse a Atenea un fichero comprimido en uno de los formatos siguientes: `zip`, `tgz` (o `tar.gz`) o `xz` (o `tar.xz`). Se ruega encarecidamente evitar otros formatos de archivo, particularmente `rar` y `arj`. El nombre del fichero responderá a la forma siguiente:

```
pX_YYZ_apellidoA_apellidoB
```

`X:` Número de la práctica. En este caso, X = 1.

`YY:` Grupo de prácticas, YY = 41 o YY = 43.

`Z:` Número del puesto de trabajo en el laboratorio, $1 \leq Z \leq 9$.

`apellido?:` Primer apellido del alumno o alumnos.

El fichero deberá incluir los tres elementos siguientes, y sólo ellos:

- La memoria del proyecto en formato `PDF`. Se ruega, también, evitar cualquier otro tipo de formato, particularmente los de Microsoft Word o OpenOffice. El nombre del documento será el mismo que indicado más arriba para el fichero comprimido.
- La señal WAVE grabada durante las sesiones de laboratorio (o en cualquier otro momento).
- El directorio con el código fuente de la práctica. El directorio no debe incluir el programa compilado ni los objetos; sólo el código fuente. Sí deberá incluir un fichero `Makfile` tal que, al ejecutar `make` se genere el programa `p1`, que deberá ser plenamente funcional, sin errores de compilación ni ejecución, y el mismo que se haya usado para obtener los resultados publicados en la memoria.

5.2. Memoria.

La memoria debe incluir en su primera página el nombre y apellidos del autor o autores de la misma.

El primer apartado de la práctica debe ser una **introducción** en la que se explique cuál es el objetivo de la misma, se enuncien los métodos usados en su resolución y se esbocen los resultados obtenidos. Si hay aportaciones propias (por ejemplo, se ha implementado un método alternativo a los propuestos en el enunciado), es importante remarcarlo aquí (aquí, en el cuerpo de la memoria y en las conclusiones). La introducción finalizará con la mención de los apartados de los que consta la memoria.

El **cuerpo de la memoria** deberá incluir:

- Descripción teórica del proyecto realizado, incluyendo las fórmulas y referencias bibliográficas oportunas.
- Los aspectos más significativos de la implementación, incluyendo las partes del código más relevantes[2]. En general, no será necesario presentar todo el código fuente, sino que bastará con las líneas escritas por el alumno, acompañadas del contexto necesario para facilitar su comprensión.
- Evaluación, con la descripción del marco experimental, los datos usados y los resultados obtenidos.

La memoria deberá finalizar con un apartado de **conclusiones** del trabajo con una discusión, cualitativa y cuantitativa, de los resultados obtenidos, y la consideración de las posibles mejoras o ampliaciones que se podrían introducir. Es importante, de nuevo, destacar en este apartado las aportaciones propias que se hayan podido hacer.

Se valorará la inclusión de todos tipo de material que ayude a la comprensión del trabajo realizado: fórmulas, tablas, gráficas, referencias bibliográficas, enlaces a páginas web, etc. Se prestará especial atención a la inclusión del material generado durante la realización de las tareas de laboratorio. Se recuerda que toda inclusión de material ajeno, sea del tipo que sea, debe mencionar sus fuentes.

Aunque esta primera práctica está bastante guiada en sus objetivos, en las sucesivas se incentivará la aportación propia de los autores, considerando las propuestas realizadas en el enunciado como un punto de partida válido, pero abierto a la ampliación y experimentación. Como regla general, la memoria deberá dejar constancia de las tareas y ejercicios planteados en el enunciasdo (señalados mediante recuadros de color verde).

[2] Dado que es muy posible que la evaluación se realice a partir de la impresión en papel de la memoria, se ruega que el código incluido tenga fondo claro, ya que, si no, se dificulta mucho su lectura.

ANEXOS.

I. Alternativas a `make`: `ninja`/`meson`.

`make` es una herramienta fantástica y enormemente popular que todo desarrollador debe conocer, entender y saber usar (al menos, a un nivel básico), pero su ámbito de utilización es, sobre todo, el que se ha visto en los apartados anteriores: pequeños proyectos contenidos en un único directorio y con dependencias sencillas. Y, en estas situaciones, muchas veces cabe preguntarse si no acabamos antes compilando todo el código cada vez que sospechemos que alguna parte se ha quedado obsoleta...

Cuando se abordan situaciones más complejas y/o proyectos de mayor tamaño, `make` presenta toda una serie de desventajas que han hecho que hayan aparecido toda una serie de alternativas más adecuadas. Y, si bien es cierto que es posible llegar a realizar casi cualquier cosa con `make` (y aliados suyos, como las `autotools`), el precio a pagar suele ser estructuras de `makefiles` anidados, a veces recursivos, que hacen que su compresión y mantenimiento sea demencial.

Todo junto, puede decirse que, para proyectos de una cierta envergadura o complejidad, `make` y sus aliados han quedado obsoletos. Entre los motivos para afirmar esto se pueden nombrar:

- Su funcionamiento se basa en generar todos los ficheros, intermedios y finales, en el propio directorio. Si se observa el estado del directorio de trabajo después de ejecutar `make` se observará que en él ha aparecido, además del programa `p1`, los tres ficheros objeto. Esto hace que el directorio de trabajo vaya creciendo con elementos irrelevantes (porque pueden generarse de nuevo automáticamente).

  Es perfectamente posible modificar este comportamiento, colocando cada pieza en directorios específicos (por ejemplo: los ejecutables en `~/bin`, los objetos en `~/obj`, etc.), pero entonces las reglas implícitas dejan de funcionar. Claro que siempre podemos modificar a nuestro antojo las reglas implícitas, pero entonces acabaremos con un mejunje que ni nosotros mismos entenderemos al cabo de un tiempo.

  Y un problema parecido aparece si queremos distribuir el código fuente en directorios separados.

- `make` no está diseñado para soportar la portabilidad del software a distintas arquitecturas. Por ejemplo, si construimos un paquete para trabajar en Linux, deberemos hacer adaptaciones específicas para que todo compile correctamente en Windows. De hecho, pueden aparecer incompatibilidades entre distintos sistemas de desarrollo de un mismo sistema operativo, o distintas distribuciones de un mismo sistema (de desarrollo u operativo). Mención especial merece la llamada *compilación cruzada* en la que se pretende desarrollar programas que se ejecutarán en máquinas distintas a la que usamos en el desarrollo. Por ejemplo, si queremos desarrollar en Linux un programa que se ejecute en Arduino.

  Para ayudar a resolver esta problemática, GNU desarrolló el *GNU Build System*, también conocido como `autotools`. Ésta es una colección de herramientas que solucionan buena parte de los problemas de portabilidad. Podemos darnos cuenta de que un paquete se ha desarrollado usando `autotools` si las órdenes necesarias para instalarlo son del estilo `./configure.sh; make; sudo make install`. Pero son muchos los desarrolladores que consideran que las `autotools` han quedado aún más obsoletas que `make`:

  - Se basan en usar ficheros de configuración, alguno generado automáticamente, que sirven para generar los ficheros de configuración que se acaban usando para generar el proyecto. Al final es necesario generar media docena larga de ficheros (extensiones `am`, `ac`, `m4`, `in`...) con sintaxis que sólo tienen en común su cripticidad.
  - Es innecesariamente lento porque ejecuta un montón de veces distintos programas, como el compilador de C, sólo para saber qué características están instaladas y cuáles no.

- Aunque la *inteligencia* de las reglas implícitas de `make` es una característica muy potente, también es muy lenta. Cada vez que se invoca una regla implícita, el programa ha de comprobar las características del objetivo y las dependencias para decidir qué es lo que tiene que hacer.
- Sólo es sensible a la fecha de modificación de los ficheros. Si, por ejemplo, cambiamos las opciones de compilación, `make` no se dará cuenta y puede acabar generando un programa inservible debido a que unas partes están compiladas con unas opciones, y otras con otras distintas.
- Es propenso a errores. Por ejemplo, es imprescindible indicar de qué cabeceras depende cada fichero de código, olvidar incluir una puede ser desastroso. Si bien es cierto que existen mecanismos para resolver esta situación (por ejemplo, el comando `gcc -MM` proporciona la lista de dependencias de un fichero de código fuente en C), lo cierto es que, una vez creado el primer `makefile` del proyecto (que suele hacerse con sumo cuidado), modificarlo después para reflejar los cambios en las dependencias requiere mucha sangre fría y buen entendimiento. Nuevamente, es posible construir los `makefiles` de manera que esto se realice automáticamente, pero entonces el precio a pagar es `makefiles` cada vez más difíciles de entender y mantener.

Entre las opciones que han aparecido modernamente a `make` (y `autotools`) podemos mencionar, entre las más populares, a `CMake` y `Meson/Ninja`. Siendo ésta última la opción que se usará en este curso a partir de la segunda práctica.

I.A. Ninja

Ninja es un remplazo de `make` explícitamente diseñado para ser rápido y silencioso (de ahí el nombre: *rápido y silencioso como una tortuga*). Fue diseñado para reemplazar `make` en el desarrollo del navegador Google Chrome. Este software comprende más de 30.000 ficheros, y millones de líneas de código. Ejecutar `make` sólo para comprobar si había alguna pieza obsoleta requería, en su momento, unos diez minutos.

El principal, y casi único, objetivo de Ninja es ser rápido. Para conseguirlo, Ninja apenas hace nada de lo que es capaz de hacer `make`. Básicamente, lo único que hace es leer un fichero de reglas semejante a `makefile`, pero en el que sólo se indica el objetivo, la dependencia y la acción a realizar. Todo el resto de parafernalia de `make` es eliminada. Este fichero de dependencias es llamado `.ninja_build`, está en formato binario y, por tanto, no está pensado para ser editado por el usuario de ningún modo. En lugar de ello, Ninja está pensado para que sea un programa externo el que genere este fichero.

I.B. Meson

Entre los programas diseñados para construir `.ninja_build`, uno de los más interesantes es Meson, llamado así porque los nombres que les gustaban a los desarrolladores ya estaban ocupados. Este programa está escrito en Python 3, por lo que su disponibilidad es casi la misma que la de este lenguaje de programación. Su base de funcionamiento consiste en construir un fichero de reglas escrito en un lenguaje de alto nivel semejante, pero distinto, a Python.

Meson siempre se ejecuta *fuera del árbol*. Esto quiere decir que toma una estructura de directorios con el código fuente y genera todos los ficheros intermedios y finales en otro directorio que tiene que ser, forzosamente, distinto. De este modo, tenemos la garantía de que el directorio con el código fuente siempre está limpio de ficheros temporales. Además, es sensible a los cambios de opciones de compilación, enlazado u otras; genera dependencias automáticamente durante el propio proceso de compilación; y permite compilaciones cruzadas o en arquitecturas diferentes con un lenguaje razonable.

> A partir de la segunda práctica, se usará `Meson/Ninja` o una combinación de éstos con `make` para realizar el mantenimiento de los programas.