FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Procesamiento de señales, fundamentos

Maestría en sistemas embebidos
Universidad de Buenos Aires
MSE 5Co2020

Clase 2 - CIAA<>Python

Ing. Pablo Slavkin
slavkin.pablo@gmail.com
wapp:011-62433453

Enuestas

Encuesta anónima clase a clase

Propiciamos este espacio para compartir sus sugerencias, criticas constructivas, oportunidades de mejora y cualquier tipo de comentario relacionado a la clase.

Encuesta anónima

https://forms.gle/1j5dDTQ7qjVfRwYo8

Link al material de la material

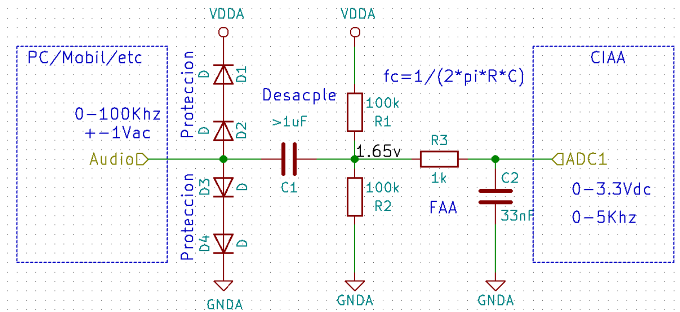
https://drive.google.com/drive/u/1/folders/1TlR2cgDPchL_4v7DxdpS7pZHtjKq38CK

Sampleo

Acondicionamiento de señal

Acondicionar la señal de salida del dispositivo de sonido (en PC ronda $\pm 1V$) al rango del ADC del hardware. En el caso de la CIAA sera de 0-3.3V.

Se propone el siguiente circuito, que minimiza los componentes sacrificando calidad y agrega en filtro anti alias de 1er orden.



Sampleo

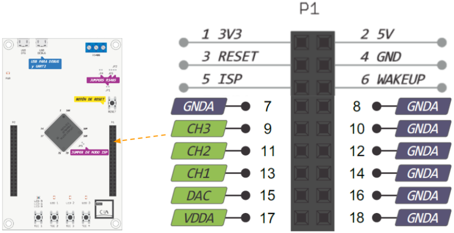
Acondicionamiento de señal

Pinout de la CIAA para conectar el ADC/DAC

ADC y DAC en la EDU-CIAA-NXP

Mapeo de ADC y DAC en la biblioteca sAPI:

- 3 entradas analógicas nombradas CH1, CH2 y CH3 (ADC).
- 1 salida analógica nombrada DAC.

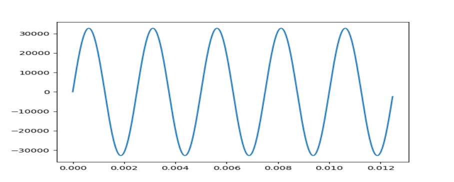


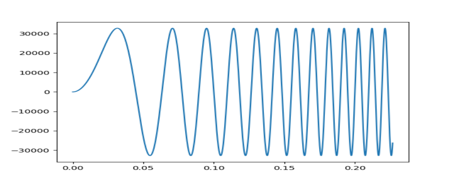
Generación de audio con Python

https://simpleaudio.readthedocs.io/en/latest/installation.html

Instalar el módulo simpleaudio (ver apéndice 6) para generar sonidos con python y utilizamos el siguiente código como base:

```
import numpy as np
import scipy.signal as sc
import simpleaudio as sa
import matplotlib.pyplot as plt

f   = 400
fs  = 44100
sec = 5
B   = 600
t   = np.arange ( 0,sec,1/fs )
#note = (2**15-1)*np.sin(2 * np.pi * f * t) #sin
#note = (2**15-1)*sc.sawtooth(2*np.pi*f*t,0) #saw
note = (2**15-1)*np.sin(2 * np.pi * B*t/sec * t)  # 
     sweept

audio = note.astype(np.int16)
for i in range(1000):
    play_obj = sa.play_buffer(audio, 1, 2, fs)
    play_obj.wait_done()
```





Captura de audio con la CIAA

CIAA->UART->picocom->log.bin

Utilizando picocom https://github.com/npat-efault/picocom o similar se graba en un archivo la salida de la UART para luego procesar como sigue

```
picocom /dev/ttyUSB1 -b 460800 –logfile=log.bin
```

```c
#include "sapi.h"

#define LENGTH 512
int16_t adc [ LENGTH ];
uint16_t sample = 0;

int main ( void ) {
   boardConfig        (                            );
   uartConfig         ( UART_USB, 460800           );
   adcConfig          ( ADC_ENABLE                 );
   cyclesCounterInit ( EDU_CIAA_NXP_CLOCK_SPEED );
   while(1) {
      cyclesCounterReset();
      uartWriteByteArray ( UART_USB ,(uint8_t* )&adc[sample] ,sizeof(adc[0])
         );
      adc[sample] = ((int16_t )adcRead(CH1)-512);
      if ( ++sample==LENGTH ) { //22.7hz para 512
         sample = 0;
         uartWriteByteArray ( UART_USB ,"header" ,6 );
         gpioToggle         ( LEDR                    );
      }
   while(cyclesCounterRead()< 20400)  //clk 204000000
      ;
   }
}
```

```
0000 c0ff 0300 80ff 0100 c0ff 0400 4000  a.............................@.
0500 0000 faff c0ff 0200 c0ff 0600 0000  ................................
0200 c0ff 1100 0000 ffff 80ff 0200 c0ff  ................................
0000 6865 6164 6572 2000 0000 f2ff 0000  ..............header .......
fcff 0000 0600 0000 0000 c0ff f6ff 0000  ................................
fcff 4000 f6ff 0000 0d00 4000 feff 0000  ..............@.......@.....
0400 c0ff 0200 0000 feff 0000 0800 c0ff  ................................
ffff 0000 0600 c0ff ffff 0000 fdff 0000  ................................
0000 6865 6164 6572 2000 0000 ecff c0ff  ..............header .......
e8ff 0000 0100 c0ff 0300 c0ff edff c0ff  ................................
0c00 0000 f0ff 0000 0100 c0ff 0500 c0ff  ................................
0200 c0ff 0500 c0ff f6ff 0000 feff 0000  ..@.............................
0600 4000 0900 0000 0a00 c0ff f8ff 0000  ......@.......@.................
0000 6865 6164 6572 2000 80ff f4ff 0000  ..............header .......
0200 c0ff f9ff c0ff 0400 c0ff 0500 0000  ................................
fcff 0000 f2ff 0000 0300 0000 0100 c0ff  ................................
f5ff c000 fcff 4000 0200 0000 fdff 0000  .................@..........
ffff 0000 f9ff 0000 0400 0000 f7ff 0000  ................................
0000 6865 6164 6572 2000 c0ff e0ff 0000  ..............header .......
fcff 0000 0a00 c0ff ffff 0000 f1ff c0ff
```

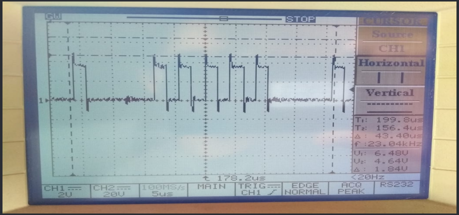
Ancho de banda

$$USB <> UART_{maxbps} = 460800bps$$

$$Eficacia = \frac{10b}{8b} = 0,8$$

$$bits_{muestra} = 16$$

$$Tasa_{efectiva} = \frac{460800_{bps} * 0,8}{16} = 23040$$



Máxima señal muestreable y reconstruible

11520hz

Sampleo

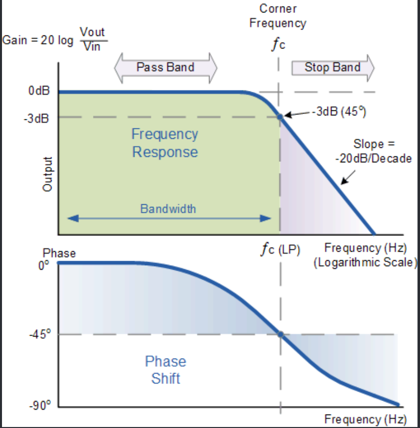
Calculo del filtro antialias 1er orden R-C

$$B = 10kbps$$

$$f_{corte} = \frac{1}{2 * \pi * R * C}$$

$$R = 1k\Omega$$

$$C = \frac{1}{f_{corte} * R * 2 * \pi} \approx 15nF$$



Máxima señal muestreable y reconstruible

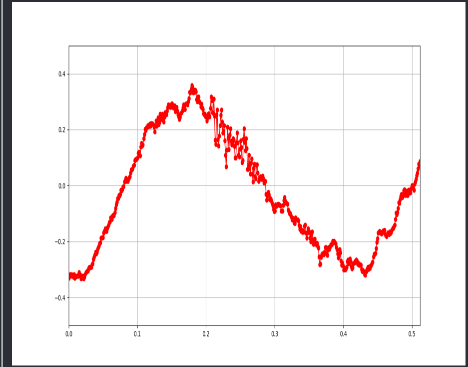
11520hz

Captura de audio con la CIAA

UART->Python

Lectura de un log y visualización en tiempo real de los datos

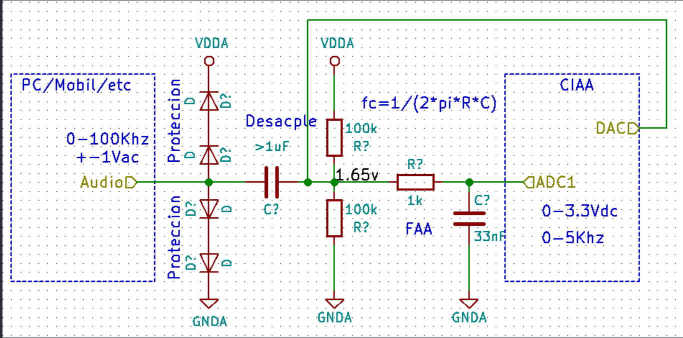
```
import numpy as np
import matplotlib.pyplot as plt
from   matplotlib.animation import
       FuncAnimation
import os

length = 512
fs     = 10000
header = b’header’
fig    = plt.figure (1 )
adcAxe = fig.add_subplot ( 1,1,1 )
time   = np.linspace(0,length/fs,length)
adcLn, = plt.plot ( [],[],’r’ )
adcAxe.grid ( True )
adcAxe.set_ylim ( -512 ,512 )
adcAxe.set_xlim ( 0 ,length/fs )

def findHeader(f):
    index = 0
    sync  = False
    while sync==False:
        data=b’’
        while len(data) <1:
            data = f.read(1)
        if data[0]==header[index]:
            index+=1
            if index>=len(header):
                sync=True
                print(sync)
        else:
            index=0
            print(sync)

def readInt4File(f):
    raw=b’’
    while len(raw)<2:
        raw += f.read(1)
    return (int.from_bytes(raw[0:2],"
          little",signed=True))

def update(t):
    findHeader ( logFile )
    adc = []
    for chunk in range(length):
        adc.append (readInt4File(logFile))
    adcLn.set_data ( time,adc )
    return adcLn,

logFile=open("log.bin","w+b")
ani=FuncAnimation(fig,update,10,None,blit=
      True,interval=10,repeat=True)
plt.draw()
plt.show()
```



Reconstrucción

Acondicionamiento de señal

Se realiza un loop del DAC al ADC permitiendo sumar a la señal de entrada ya existente



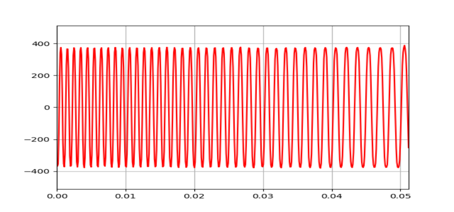
Generación de audio con el DAC de la CIAA

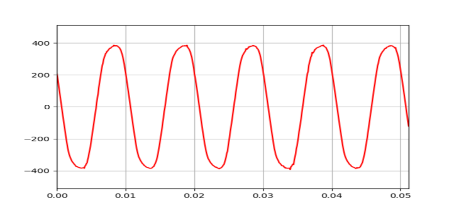
ARM CMSIS-DSP lib *https://www.keil.com/pack/doc/CMSIS/DSP/html/group__sin.html*

Con arm_sin_f32 se genera un tono y se convierte a analógico con el DAC

```c
#include "sapi.h"
#include "arm_math.h"

#define LENGTH 512
#define FS     10000
int16_t  adc[ LENGTH ];
uint16_t sample = 0   ;
uint32_t tick   = 0   ;
uint16_t f      = 100 ;
uint16_t B      = 5000;
uint16_t sweept = 1;
float t         = 0;

int main ( void ) {
   boardConfig ( );
   uartConfig ( UART_USB
         ,460800 );
   adcConfig (
         ADC_ENABLE );
   dacConfig (
         DAC_ENABLE );
```

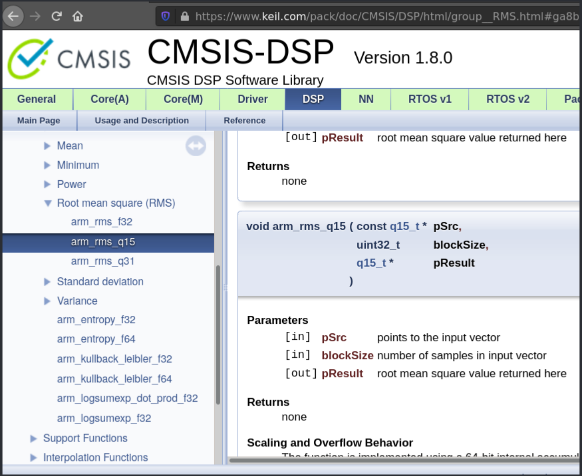
```c
   cyclesCounterInit ( EDU_CIAA_NXP_CLOCK_SPEED
         );
   while(1) {
      cyclesCounterReset();
      uartWriteByteArray ( UART_USB ,(uint8_t* )
            &adc[sample] ,sizeof(adc[0]) );
      adc[sample] = adcRead(CH1)-512;
      t=((tick%(sweept*FS))/(float)FS);
      //dacWrite( DAC, 512*arm_sin_f32 (t*B*(t/
            sweept)*2*PI)+512); //sweept
      dacWrite( DAC, 512*arm_sin_f32 (t*f*2*PI)
            +512); //tono
      if ( ++sample==LENGTH ) {
         sample = 0;
         uartWriteByteArray ( UART_USB ,"header"
               ,6 );
         gpioToggle ( LEDG );
      }
      tick++;
      while(cyclesCounterRead()<
            EDU_CIAA_NXP_CLOCK_SPEED/FS)  //clk
            204000000
         ;
   }
}
```





Documentación

ARM CMSIS-DSP lib *https://www.keil.com/pack/doc/CMSIS/DSP/html/group__sin.html*



Sistemas de números

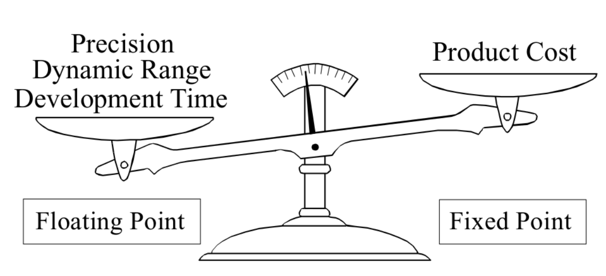
Punto fijo vs punto flotante

Punto fijo:

- Cantidad de patrones de bits= 65536
- Gap entre números constante
- Rango dinámico 32767, −32768
- Gap 10 mil veces mas chico que el numero

Punto flotante:

- Cantidad de patrones de bits= 4,294,967,296
- Gap entre números variable
- Rango dinámico $\pm 3{,}4e10^{38}$, $\pm 1{,}2e10^{-38}$
- Gap 10 millones de veces mas chico que el numero



Sistemas de números

Sistema Q

Qm.n:

- m: cantidad de bits para la parte entera
- n: cantidad de bits para la parte decimal

Q1.15:

1000 0000 0000 0000 = -1

0111 1111 1111 1111 = 1/2 + 1/4 + 1/8 + .. + $1/2^{15} = 0{,}99$

Q2.14:

1010 0000 0000 0000 = -1.5

0101 0000 0000 0000 = 1.25

Sistemas de números

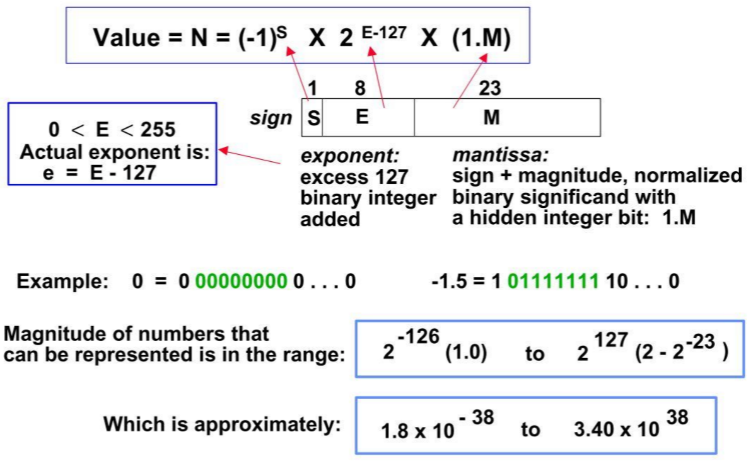
Tabla de ejemplos Q1.2 y Q2.1 signado y no signado

UQ3.0	UQ2.1	UQ1.2
011 = 3	01.1 = 1+1/2= 1.5	0.11 = 0+1/2+1/4= 0.75
010 = 2	01.0 = 1+0/2= 1.0	0.10 = 0+1/2+0/4= 0.5
001 = 1	00.1 = 0+1/2= 0.5	0.01 = 0+0/2+1/4= 0.25
000 = 0	00.0 = 0+0/2= 0.0	0.00 = 0+0/2+0/4= 0.0
111 = 7	11.1 = 3+1/2= 3.5	1.11 = 1+1/2+1/4= 1.75
110 = 6	11.0 = 3+0/2= 3.0	1.10 = 1+1/2+0/4= 1.5
101 = 5	10.1 = 2+1/2= 2.5	1.01 = 1+0/2+1/4= 1.25
100 = 4	10.0 = 2+0/2= 2.0	1.00 = 1+0/2+0/4= 1.0

SQ3.0	SQ2.1	SQ1.2
011 =+3	01.1 = 1+1/2=+1.5	0.11 = 0+1/2+1/4=+0.75
010 =+2	01.0 = 1+0/2=+1.0	0.10 = 0+1/2+0/4=+0.5
001 =+1	00.1 = 0+1/2=+0.5	0.01 = 0+0/2+1/4=+0.25
000 =+0	00.0 = 0+0/2=+0.0	0.00 = 0+0/2+0/4=+0.0
111 =-1	11.1 =-1+1/2=-0.5	1.11 =-1+1/2+1/4=-0.25
110 =-2	11.0 =-1+0/2=-1.0	1.10 =-1+1/2+0/4=-0.5
101 =-3	10.1 =-2+1/2=-1.5	1.01 =-1+0/2+1/4=-0.75
100 =-4	10.0 =-2+0/2=-2.0	1.00 =-1+0/2+0/4=-1.0

Sistemas de números

Float32 IEEE 754



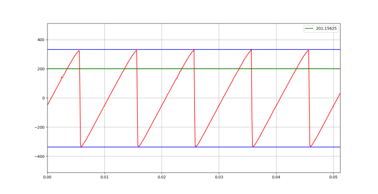
Calculo de propiedades temporales

ARM CMSIS-DSP lib

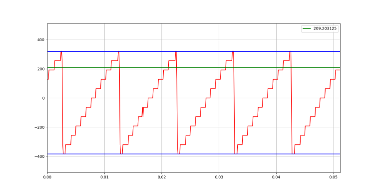
Calculamos max, min y rms con la CIAA

```c
#include "sapi.h"
#include "arm_math.h"

#define LENGTH 512
#define FS     10000
int16_t  adc[ LENGTH ];
uint16_t sample             = 0;
uint32_t maxIndex,minIndex  = 0;
q15_t maxValue,minValue,rms = 0;

int main ( void ) {
   boardConfig ( );
   uartConfig ( UART_USB ,460800 );
   adcConfig ( ADC_ENABLE );
   dacConfig ( DAC_ENABLE );
   cyclesCounterInit ( EDU_CIAA_NXP_CLOCK_SPEED );
   while(1) {
      cyclesCounterReset();
      uartWriteByteArray ( UART_USB ,(uint8_t* )&adc[sample] ,sizeof(adc[0]) );
      adc[sample] = ((adcRead(CH1)-512)>>6)<<12;
      if ( ++sample==LENGTH ) {
         arm_max_q15 ( adc, LENGTH, &maxValue,&maxIndex );
         arm_min_q15 ( adc, LENGTH, &minValue,&minIndex );
         arm_rms_q15 ( adc, LENGTH, &rms                );
         uartWriteByteArray ( UART_USB ,(uint8_t* )&maxValue ,2 );
         uartWriteByteArray ( UART_USB ,(uint8_t* )&minValue ,2 );
         uartWriteByteArray ( UART_USB ,(uint8_t* )&rms      ,2 );
         sample = 0;
         uartWriteByteArray ( UART_USB ,"header" ,6 );
         gpioToggle ( LEDG );
      }
      while(cyclesCounterRead()< EDU_CIAA_NXP_CLOCK_SPEED/FS)  //clk 204000000
         ;
   }
}
```

10 bits



4 bits



Cálculo de propiedades temporales

ARM CMSIS-DSP lib

Visualizamos max, min y rms con Python

```python
import numpy as np
import matplotlib.pyplot as plt
from   matplotlib.animation import FuncAnimation
import os

N = 512
fs     = 10000
header = b’header’
fig    = plt.figure (1 )
adcAxe = fig.add_subplot ( 1,1,1 )
time   = np.linspace(0,N/fs,N)
adcLn, maxLn, minLn, rmsLn, \
= plt.plot ([],[],’r’,[],[],’b’,[],[],’b’,[],[],’g’)
adcAxe.grid ( True )
adcAxe.set_ylim ( -512 ,512 )
adcAxe.set_xlim ( 0 ,N/fs )

def findHeader(f):
    index = 0
    sync  = False
    while sync==False:
        data=b’’
        while len(data) <1:
            data = f.read(1)
        if data[0]==header[index]:
            index+=1
            if index>=len(header):
                sync=True
                print(sync)
        else:
            index=0
            print(sync)

def readInt4File(f):
    raw=b’’
    while len(raw)<2:
        raw += f.read(1)
    return (int.from_bytes(raw[0:2],"little",signed=True))

def update(t):
    findHeader ( logFile )
    adc = []
    for chunk in range(N):
        adc.append (readInt4File(logFile)/2**6)
    adcLn.set_data ( time,adc )
    maxLn.set_data ( time,np.full(N,readInt4File(logFile)/2**6))
    minLn.set_data ( time,np.full(N,readInt4File(logFile)/2**6))
    rmsValue=readInt4File(logFile)/2**6
    rmsLn.set_data ( time,np.full(N,rmsValue))
    rmsLn.set_label(rmsValue)
    rmsLg    = adcAxe.legend()
    return adcLn, maxLn, minLn, rmsLn,rmsLg

logFile=open("log.bin","w+b")
ani=FuncAnimation(fig,update,10,None,blit=True,interval=10,repeat=True)
plt.draw()
plt.show()
```

Bibliografía

Libros, links y otro material

[1] *Numeracion Q*.
https://en.wikipedia.org/wiki/q_(number_format)

[2] *Calculador float on-line*.
https://www.binaryconvert.com/result_float.html?decimal=048046053

[3] *Calculando numeros Q*.
https://www.rfwireless-world.com/calculators/floating-vs-fixed-point-converter.html

Apéndice

Instrucciones para usar simpleaudio

Docs » Installation

Installation

Make sure you have pip installed. For Linux, this is usually done with the distro package manager (example: `sudo apt-get install python3-pip`). For Windows and macOS, have a look at the pip documentation.

> **Note**
>
> The actual `pip` command may vary between platforms and Python versions. Substitute the correct one for usages in these examples. Some common variants for Python 3.x are: `pip3`, `pip3.x`, and `pip-3.x`.
>
> Also, in some cases you may need to manually add pip's location to the 'path' environment variable.

Upgrade pip and setuptools:

```
pip install --upgrade pip setuptools
```

Install with:

```
pip install simpleaudio
```

Linux Dependencies

The Python 3 and ALSA development packages are required for pip to build the extension. For Debian variants (including Raspbian), this will usually get the job done:

```
sudo apt-get install -y python3-dev libasound2-dev
```

Docs » Short Tutorial

Short Tutorial

Import

Import the `simpleaudio` module:

```
import simpleaudio as sa
```

Playing audio directly

The simplest way to play audio is with `play_buffer()`. The `audio_data` parameter must be an object which supports the buffer interface. (`bytes` objects, Python arrays, and **Numpy** arrays all qualify.):

```
play_obj = sa.play_buffer(audio_data, 2, 2, 44100)
```

The *play_obj* object is an instance of `PlayObject` which could be viewed as a 'handle' to the audio playback initiated by the `play_buffer()` call. This can be used to stop playback of the audio clip:

```
play_obj.stop()
```

It can used to check whether a sound clip is still playing:

```
if play_obj.is_playing():
    print("still playing")
```

It can also be used to wait for the audio playback to finish. This is espcially useful when a script or program would otherwise exit before playback is done (stopping the playback thread and consequently the audio):

```
play_obj.wait_done()
# script exit
```

WaveObject's

In order to facilitate cleaner code, the `WaveObject` class is provided which stores a reference to the object containing the audio as well as a copy of the playback parameters. These can be instantiated like so:

```
wave_obj = sa.WaveObject(audio_data, 2, 2, 44100)
```

Playback is started with `play()` and a `PlayObject` is returned as with `play_buffer()`:

```
play_obj = wave_obj.play()
```

A class method exists in order to conveniently create WaveObject instances directly from WAV files on disk:

```
wave_obj = sa.WaveObject.from_wave_file(path_to_file)
```

Similarly, instances can be created from Wave_read objects returned from `wave.open()` from the Python standard library:

```
wave_read = wave.open(path_to_file, 'rb')
wave_obj = sa.WaveObject.from_wave_read(wave_read)
```

Using Numpy

Numpy arrays can be used to store audio but there are a few crucial requirements. If they are to store stereo audio, the array must have two columns since each column contains one channel of audio data. They must also have a signed 16-bit integer dtype and the sample amplitude values must consequently fall in the range of -32768 to 32767. Here is an example of a simple way to 'normalize' the audio (making it cover the whole amplitude rage but not exceeding it):

```
audio_array *= 32767 / max(abs(audio_array))
```

And here is an example of converting it to the proper data type (note that this should always

Apéndice

Instrucciones para usar simpleaudio

be done *after* normalization or other amplitude changes):

```
audio_array = audio_array.astype(np.int16)
```

Here is a full example that plays a few sinewave notes in succession:

```
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.25
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave notes
A_note = np.sin(A_freq * t * 2 * np.pi)
Csh_note = np.sin(Csh_freq * t * 2 * np.pi)
E_note = np.sin(E_freq * t * 2 * np.pi)

# concatenate notes
audio = np.hstack((A_note, Csh_note, E_note))
# normalize to 16-bit range
audio *= 32767 / np.max(np.abs(audio))
# convert to 16-bit data
audio = audio.astype(np.int16)

# start playback
play_obj = sa.play_buffer(audio, 1, 2, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```

In order to play stereo audio, the **Numpy** array should have 2 columns. For example, one second of (silent) stereo audio could be produced with:

```
silence = np.zeros((44100, 2))
```

We can then use addition to layer additional audio onto it - in other words, 'mixing' it together. If a signal/audio clip is added to both channels (array columns) equally, then the audio will be perfectly centered and sound just as if it were played in mono. If the proportions vary between the two channels, then the sound will be stronger in one speaker than the other, 'panning' it to one side or the other. The full example below demonstrates this:

```
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.5
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave notes
A_note = np.sin(A_freq * t * 2 * np.pi)
Csh_note = np.sin(Csh_freq * t * 2 * np.pi)
E_note = np.sin(E_freq * t * 2 * np.pi)

# mix audio together
audio = np.zeros((44100, 2))
n = len(t)
offset = 0
audio[0 + offset: n + offset, 0] += A_note
audio[0 + offset: n + offset, 1] += 0.125 * A_note
offset = 5500
audio[0 + offset: n + offset, 0] += 0.5 * Csh_note
audio[0 + offset: n + offset, 1] += 0.5 * Csh_note
offset = 11000
audio[0 + offset: n + offset, 0] += 0.125 * E_note
audio[0 + offset: n + offset, 1] += E_note

# normalize to 16-bit range
audio *= 32767 / np.max(np.abs(audio))
# convert to 16-bit data
audio = audio.astype(np.int16)

# start playback
play_obj = sa.play_buffer(audio, 2, 2, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```

24-bit audio can be also be created using **Numpy** but since **Numpy** doesn't have a 24-bit integer dtype, a conversion step is needed. Note also that the max sample value is different for 24-bit audio. A simple (if inefficient) conversion algorithm is demonstrated below, converting an array of 32-bit integers into a `bytes` object which contains the packed 24-bit audio to be played:

```
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.5
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave tone
tone = np.sin(440 * t * 2 * np.pi)

# normalize to 24-bit range
tone *= 8388607 / np.max(np.abs(tone))

# convert to 32-bit data
tone = tone.astype(np.int32)

# convert from 32-bit to 24-bit by building a new byte buffer, skipping every fourth
bit
# note: this also works for 2-channel audio
i = 0
byte_array = []
for b in tone.tobytes():
    if i % 4 != 3:
        byte_array.append(b)
    i += 1
audio = bytearray(byte_array)

# start playback
play_obj = sa.play_buffer(audio, 1, 3, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```