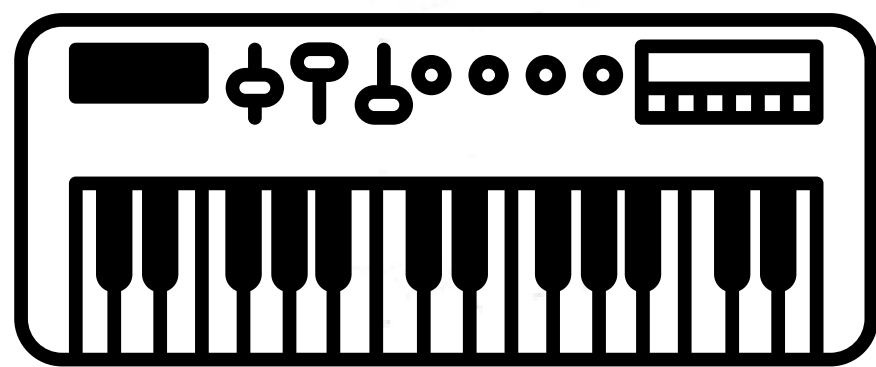


Trabajo Final

Pensamiento Computacional



Mateo Giacometti - 34573
mgiacometti@udesa.edu.ar

Lucio Luque Materazzi - 34608
lluquematerazzi@udesa.edu.ar

Ignacio Schuemer - 34575
ischuemer@udesa.edu.ar

Santiago Tomas Torres - 34580
storres@udesa.edu.ar

Tabla de contenido

Objetivos del trabajo	3
Sintetizador	3
Método compose:.....	4
Método read_partiture de Synthesizer:	6
Método attack_is_minor_than_duration:	8
Método read_instrument de Synthesizer:.....	8
Clase ReadInstrument:	9
Método get_max_duration de Synthesizer:.....	11
Método get_frequency de Synthesizer:.....	11
notes_mapping:.....	12
Método create_note de Synthesizer:.....	12
Método create_armonic_note de Synthesizer:.....	13
Método create_modulation de Synthesizer:	15
Metalófono	17
Synthesizer mediante la consola (argparser):.....	19
Bibliografía:	21

Objetivos del trabajo

Objetivos:

Los objetivos del presente trabajo se encuentran divididos en dos partes:

1. Realizar un programa que permita, a partir de una partitura, sintetizar las notas predefinidas.

2. A partir de una partitura, enviar las notas que se quieren tocar a un instrumento real. En este caso, se tratará de un metalófono.

Sintetizador

```
class Synthesizer:
    def __init__(self, filename_partiture, filename_instrument):

        self.filename_partiture = filename_partiture
        self.filename_instrument = filename_instrument
```

La clase *Synthesizer* toma como atributos *filename_partiture* y *filename_instrument*.

El primero debe ser un archivo de texto de la partitura de la canción, el segundo un archivo de texto del instrumento a usar.

```
def create_wav(self, song_name:str, song_frequency:int):
    """
    Creates a wav file of the song. Uses the filename_partiture as the name of the file.

    Parameters
    -----
    song_name : str
    | The name of the song
    song_frequency : int
    | The frequency of the song

    Returns
    -----
    A wav file of the song
    """
    song=self.compose(song_frequency)
    song_name+=".wav"
    return wavfile.write(song_name, song_frequency, song)
```

El método *create_wav* recibe dos parametros, *song_name*, el cual es el nombre de la canción que va a utilizar para crear el archivo .wav, *song_frequency*, que es la frecuencia de la canción. Dentro del método se crea una variable *song* con lo que devuelve el método [*compose*](#) tomando *song_frequency* como parámetro, el cual retorna todas las notas

compuestas en un array; luego genera el nombre de la canción añadiendo `.wav` a `song_name`, y devuelve el archivo.

Método compose:

```
def compose(self, song_frequency: float) -> np.ndarray:
    """
    The main function to compose the song.
    Returns the song as a numpy array.
    Parameters
    -----
    song_frequency : float
        The frequency of the song
    Returns
    -----
    numpy.ndarray
        The song as a numpy array
    """
    armonics, modulations=self.read_instrument()
    decay= modulations[2][1]
    attack=modulations[0][1]
    list_of_notes=self.read_partiture(attack, decay)
    max_duration=self.get_max_duration(list_of_notes)

    song_duration=max_duration+decay+1 #+1 to not lose the last note
    song=np.empty(int(song_duration*song_frequency))
    for i in list_of_notes:
        starts, name, duration=i
        frequency=self.get_frequency(name)
        note=self.create_note(song_frequency, duration)
        armonic_note=self.create_armonic_note(frequency, duration, armonics, note)
        modulated_note=self.create_modulation( song_frequency,duration, modulations, armonic_note, note)

        start=int(starts*song_frequency)
        end=len(modulated_note) + start
        song[start:end]+=modulated_note #add the modulated note to the song

    song[song<-1]=-1 #set the song to -1 if it is less than -1
    song[song>1]=1 #set the song to 1 if it is greater than 1
    return song
```

El método *compose* primeramente crea un diccionario, *armonics*, y una lista *modulations* con el método [read_instrument](#)

Luego genera las variables *decay* y *attack*, los caules son el tiempo del decaimiento y del ataque respectivamente.

```
armonics, modulations=self.read_instrument()
decay= modulations[2][1]
attack=modulations[0][1]
list_of_notes=self.read_partiture(attack, decay)
max_duration=self.get_max_duration(list_of_notes)
```

A continuación, hace una lista de las notas con el método [read_partiture](#) que toma como parámetros *attack* y *decay*.

Luego genera *max_duration* con el método [get_max_duration](#) tomando como parámetro *list_of_notes*.

```
song_duration=max_duration+decay+1 #+1 to not lose the last note
song=np.empty(int(song_duration*song_frequency))
```

La variable *song_duration* es *max_duration* (es decir, cuánto dura la canción) más el *decay*, más uno, esto último se le agrega para no perder información al pasarlo a tipo *int*.

Posteriormente crea un array vacío de la longitud de la canción usando *song_duration* multiplicado por *song_frequency*. Este array tiene la dimensión que dura toda la canción para luego ir sumándole todas las notas de la partitura.

```
for i in list_of_notes:
    starts, name, duration=i
    frequency=self.get_frequency(name)
    note=self.create_note(song_frequency, duration)
    armonic_note=self.create_armonic_note(frequency, duration, armonics, note)
    modulated_note=self.create_modulation( song_frequency,duration, modulations, armonic_note, note)

    start=int(starts*song_frequency)
    end=len(modulated_note) + start
    song[start:end]+=modulated_note #add the modulated note to the song

song[song<-1]=-1 #set the song to -1 if it is less than -1
song[song>1]=1 #set the song to 1 if it is greater than 1
return song
```

Comienza un ciclo *for* de *list_of_notes*, con el objetivo de crear cada nota y luego sumarla a *song* en su debido tiempo para componer la canción.

De cada elemento de *list_of_note* consigue *starts*, *name*, *duration*. Genera *frequency* con el método [get_frequency](#) tomando *name* como parámetro. A continuación, se crea la nota con el método [create_note](#) que toma a *song_frequency* y *duration* como parámetros y crea *armonic_note* con el método [create_armonic_note](#) tomando como parámetros *frequency*, *duration*, *armonics* y *note*. Por último, crea *modulated_note* con el método [create_modulation](#) pasándole *song_frequency*, *duration*, *modulations*, *armonic_note* y *note*.

Luego se crea los índices para el *slicing* de *song*; a *start* lo genera mediante la multiplicación de *starts* de la nota por *song_frequency* y lo cambia a *int*. En cambio, a *end* lo crea con la longitud de la nota modulada sumándole el *start*. Posteriormente se realiza un *slicing* de *song* entre *start* y *end* y se le suma la nota modulada (esto ocurre para cada una de las notas de la partitura guardadas en *note_list*).

Una vez finalizado el ciclo *for*, a todo elemento mayor a 1 de *song* lo convierte a 1, y a todo elemento menor a -1 lo convierte a -1 (esto lo realiza para que el archivo *.wav* suene correctamente).

Método `read_partiture` de `Synthesizer`:

```
def read_partiture(self, attack:float, decay:float) -> list:
    """
    Returns a list of notes.
    Each note is a tuple of the form (start:float, name:str, duration:float).

    Parameters
    -----
    attack : float
    |     The attack of the instrument
    decay : float
    |     The decay of the instrument
    Returns
    -----
    list
    |     The list of notes
    """
    return ReadPartiture(self.filename_partiture).read_partiture(attack,decay)
```

Este método crea un objeto utilizando la clase *ReadPartiture* y el método de esa clase, es decir, *read_partiture*. el cual toma como parametros *attack* y *decay*.

```
class ReadPartiture:
    def __init__(self, filename_partiture):
        """
        Parameters
        -----
        filename_partiture : str
        |     The name of the file containing the partiture
        """

        if type(filename_partiture) != str:
            raise TypeError
        self.filename_partiture = filename_partiture
```

```
def read_partiture(self, attack, decay):
    """
    Reads the partiture file.
    It will check if the duration is greater than the attack. See attack_is_minor_than_duration.
    Will add the note to the list if the duration is greater than the attack.
    Will add the decay to the duration of the note.
    Each note is a tuple of the form (start:float, name:str, duration:float).
    Parameters
    -----
    attack : float
        The attack of the instrument
    decay : float
        The decay of the instrument
    Returns
    -----
    list
        The list of notes
    """
    list_of_notes = []
    failed_notes=0

    with open (self.filename_partiture, 'r') as f:
        for line in f:
            line=line.strip().split(' ')
            starts=float(line[0])
            type=line[1]
            duration=float(line[2])
            if self.attack_is_minor_than_duration(duration,attack):
                list_of_notes.append((starts, type, duration+decay)) # add decay to the duration
            else:
                failed_notes+=1
        if failed_notes>0:
            print(f"Warning: There are {failed_notes} notes that are lower than the attack:{attack}, and there are not going to reproduce")
        if len(list_of_notes)==0:
            raise ValueError(f"All the notes are lower than the attack: {attack}")
    return list_of_notes
```

El método `read_partiture` de la clase `ReadPartiture` crea una lista vacía denominada `list_of_notes`, y `failed_notes` que le asigna el valor cero. Abre el archivo de texto (en modo de lectura) y recorre línea por línea. A cada una de ellas las separa y se le asigna a `starts` el primer elemento (en tipo `float`), a `type` se le asigna el segundo elemento y a `duration` el tercer elemento (también en tipo `float`).

En cada línea se fija si la duración del ataque es menor que la duración de la nota con el método [`attack is minor than duration`](#) que toma como argumentos `duration` y `attack`.

Si lo que devuelve es `True`, se adjunta a la lista `list_of_notes`, una tupla obtenida por `starts`, `type`, `duration`, este último se le suma `decay`.

Si el resultado es `False`, va a sumarle a `failed_notes` uno, con el objetivo de saber cuántas notas tienen una duración menor que el ataque y no se van a reproducir. Cuando termina el `for` retorna la cantidad de notas que no se reproducen y devuelve la lista `list_of_notes`.

Método `attack_is_minor_than_duration`:

```
def attack_is_minor_than_duration(self, duration, attack):  
    """  
    Checks if the duration of each note is greater than the attack.  
    Prints a warning if the duration is smaller than the attack. It won't be added to the list of notes.  
    Returns True if the duration is greater than the attack.  
  
    Parameters  
    -----  
    duration : float  
        The duration of the note  
    attack : float  
        The attack of the instrument  
  
    Returns  
    -----  
    bool  
        True if the duration is greater than the attack. False otherwise.  
  
    """  
    if (duration) <= attack:  
        return False  
    else:  
        return True
```

El método recibe *duration* y *attack* como parámetros, con el objetivo de que si la duración de la nota es menor o igual devuelve *False* y de lo contrario devuelve *True*.

Método `read_instrument` de *Synthesizer*:

```
def read_instrument(self):  
    """  
    Returns a dictionary and a list.  
    The dictionary contains the armonics of the instrument.  
    The list contains the modulations of the instrument.  
    """  
    return ReadInstrument(self.filename_instrument).read()
```

Este método creará un objeto con la clase [ReadInstrument](#) y va a llamar al método *read*. A continuación, se muestra un ejemplo de un archivo de texto de un instrumento:

```
4  
1 1  
2 0.72727272  
3 0.31818181  
4 0.090909  
LINEAR 0.02  
CONSTANT  
INVLINEAR 0.06
```


La primera línea será la cantidad de armónicos. Luego línea por línea se muestra el número de armónico hasta llegar a esa cantidad acompañado cada uno por sus amplitudes y sus frecuencias; y para finalizar las últimas tres líneas van a estar ordenadas en ataque, sostenido y decaimiento con los segundos que dura esa modulación.

Clase ReadInstrument:

```
class ReadInstrument:
    def __init__(self, filename):
        """
        Parameters
        -----
        filename : str
            The name of the file containing the instrument
        """
        if type(filename) != str:
            raise TypeError

        self.filename=filename
```

```
def read(self):
    """
    Reads the instrument file and returns a dictionary and a list.
    The dictionary contains the armonics of the instrument.
    The list contains the modulations of the instrument.
    """
    armonics={}
    modulations=[]
    with open (self.filename, 'r') as f:
        amount_armonics= f.readline().strip()
        if amount_armonics.isnumeric() == False:
            raise TypeError
        amount_armonics= int(amount_armonics)
        for line in range(0,amount_armonics):#read the armonics
            line=f.readline().strip().split(" ")
            if len(line) < 2:
                raise ValueError
            else:
                if (isfloat(line[0]) or isfloat(line[1])) ==False:
                    raise TypeError
            armonics[int(line[0])]= float(line[1])
```

El método *read* de esta clase crea un diccionario y una lista vacía, los cuales denominaremos *armonics* y *modulations*. A continuación, abre el archivo de texto nuevamente en formato de lectura.

Se le asigna a la variable *amount_armonics* la primera línea (del archivo) en forma *int*, ya que como se explicó en la anterior captura esa primera línea posee la cantidad de armónicos que tiene el [instrumento](#).

A ello se le sigue un bucle *for* desde 0 hasta el número guardado en la variable *amount_armonics*; con este separaremos cada línea por espacios. Luego se le asigna al diccionario armónicos con la *key* obtenida del primer elemento de la línea en tipo *int* y le asigna el *value* (valor) del segundo elemento de la línea esta vez en tipo *float*.

Una vez terminado el *for*, es decir que ya se recorrió la sección de armónicos del archivo de texto, se pasará a la siguiente parte de la función.

```
module_lines=(f.readlines())#read the modulations
if len(module_lines) < 3:
    raise ValueError

for line in module_lines:
    l=[]
    line=line.strip().split(" ")
    for i in range(0,len(line)):
        if i!=0:
            if isfloat(line[i]) == False:
                raise ValueError
            l.append(float(line[i]))
        else:
            if line[i].isalpha() == False:
                raise ValueError
            l.append((line[i]))
    modulations.append(l)
return armonics,modulations
```

A continuación, se explicará la siguiente parte del método

Comienza asignando a la variable *module_lines* las líneas restantes del archivo.

Con un ciclo *for* recorre línea por línea y crea una lista vacía llamada *l*. divide y separa la cada línea y dentro de cada línea recorre con otro ciclo *for* desde cero hasta el largo de la línea. Si el elemento es cero, adjunta a la lista *l* el elemento cero de la línea y si no es el cero lo adjunta a la lista *l* pero en forma *float*. Luego de recorrer este último *for*, adjunta la lista *l* a *modulations*, y así con cada línea hasta que termina y devuelve *armonics* y *modulations*.

Método get_max_duration de Synthesizer:

```
def get_max_duration(self, list_of_notes: list) -> float:
    """
    Searches the list of notes for the longest duration, with the sum of the start times and the duration.
    Returns the maximum duration of the sum in the list_of_notes.
    That will be the duration of the song.

    Parameters
    -----
    list_of_notes : list
        The list of notes

    Returns
    -----
    float
        The maximum duration of the notes"""

    max_duration = max(list_of_notes, key=lambda x: x[0] + x[2])
    return max_duration[0] + max_duration[2]
```

La función de este método es buscar la tupla con el máximo valor de la suma de los elementos cero y dos de la lista de list_of_notes (el elemento cero es donde empieza y el elemento dos es la duración de la nota), con el objetivo de así saber cuánto dura la canción, incluso sabiendo que la partitura puede llegar desordenada. Finalmente, realiza la suma del elemento cero y el elemento dos (de la tupla) y así devuelve el máximo valor de la suma de los elementos cero y dos.

Método get_frequency de Synthesizer:

```
def get_frequency(self, name: str):
    """
    Get the frequency of a note using the name of the note as a key of the notes_mapping dictionary.
    See notes.py for the notes_mapping dictionary.

    Parameters
    -----
    name : str
        The name of the note

    Returns
    -----
    float
        The frequency of the note
    """

    if type(name) != str:
        raise TypeError(f"{name} must be str")
    if name not in notes_mapping:
        raise KeyError(f"{name} is not a valid note")
    else:
        return notes_mapping[name]
```

Este método tiene como parámetro a *name* que se define como el nombre de la nota, el cual lo utiliza como *key* de un diccionario llamado *notes_mapping* y devuelve la frecuencia de esa nota.

notes_mapping:

```
notes_mapping={"C8": 4186.01,  
              "B7": 3951.07,  
              "Bb7": 3729.31,  
              "A7": 3520.00,  
              "Ab7": 3322.44,  
              "G7": 3135.96,
```

Este es el ejemplo de una parte del diccionario *notes_mapping*, el cual se encuentra en el módulo *notes*.

Método create_note de Synthesizer:

```
def create_note(self, song_frequency:int,duration:float) -> np.ndarray:  
    """  
    Returns a note of the given duration.  
  
    Parameters  
    -----  
    duration : float  
        The duration of the note  
  
    Returns  
    -----  
    numpy.ndarray  
        The note as a numpy array  
    """  
  
    if type(duration) != float:  
        raise TypeError  
    return CreateArrayNote(song_frequency,duration).array_of_note()
```

La función de este método es crear un objeto utilizando la clase *CreateArrayNote* y el método *array_of_note*.


```
class CreateArrayNote:
    def __init__(self, song_frequency:int, duration:float):
        """
        Parameters
        -----
        duration : float
            The duration of the note
        """
        if type(song_frequency) != int or type(duration) != float:
            raise TypeError

        self.song_frequency= song_frequency
        self.duration = duration

    def array_of_note(self):
        """
        Returns the array of the note.

        Returns
        -----
        numpy.ndarray
            The array of the note as a numpy array
        """
        return np.linspace(0, self.duration,int(self.song_frequency*self.duration))
```

En esta clase lo que se realiza es crear un array de la nota con sus parámetros correspondientes.

Método create_armonic_note de Synthesizer:

```
def create_armonic_note(self, frequency:float, duration:float, armonics: dict, note: np.ndarray) -> np.ndarray:
    """
    Returns the armonic note of the given frequency and duration.

    Parameters
    -----
    frequency : float
        The frequency of the note
    duration : float
        The duration of the note
    armonics : dict
        The armonics of the instrument
    note : numpy.ndarray
        The note as a numpy array

    Returns
    -----
    numpy.ndarray
        The armonic note as a numpy array
    """
    return ArmonicNote(frequency, duration, armonics).get_armonic(note)
```

Este método crea un objeto a través de la clase *ArmonicNote* y utiliza su método *get_armonic*.

```
class ArmonicNote:
    def __init__(self, frequency:int, duration:float, armonics:dict):
        """
        Parameters
        -----
        frequency : float
            The frequency of the note
        duration : float
            The duration of the note
        armonics : dict
            The armonics of the instrument. See the read_files.py module.
        """
        if ((type(frequency) != float)
            or (type(duration) != float)
            or (type(armonics) != dict)):
            raise TypeError

        self.frequency= frequency
        self.duration= duration
        self.armonics= armonics
```

Esta clase tiene los parámetros propios de la nota para luego poder crear cada armónico de ella los cuales varían según el instrumento que se esté tocando.

```
def get_armonic(self, note:np.ndarray)->np.ndarray:
    """
    Returns the armonic note.

    Parameters
    -----
    note : numpy.ndarray
        The note as a numpy array
    Returns
    -----
    numpy.ndarray
        The armonic note as a numpy array
    """

    if type(note) != np.ndarray:
        raise TypeError

    d=self.armonics
    armonics=np.zeros(len(note))
    for i in d:
        armonics+=(d[i] * np.sin((2*np.pi*i*self.frequency * note))))
    return armonics
```

En este método se recorre un *for* que contiene los índices del diccionario que se creó anteriormente a través de la lectura del archivo del instrumento. En este bucle se suman todos los armónicos de la nota.

i es el coeficiente que indica qué armónico es (es decir el primero (1), segundo (2), ..., enésimo(*n*)).

Y por último, *d[i]*, que contiene las amplitudes de cada armónico.

Método create_modulation de Synthesizer:

```
def create_modulation(self, song_frequency:int,duration:float, modulations:list, armonic_note:np.ndarray, note:np.ndarray) -> np.ndarray:
    """
    Returns the modulated note of the given duration.

    Parameters
    -----
    duration : float
        The duration of the note
    modulations : list
        The modulations of the instrument
    armonic_note : numpy.ndarray
        The armonic note as a numpy array
    note : numpy.ndarray
        The note as a numpy array

    Returns
    -----
    numpy.ndarray
        The modulated note as a numpy array
    """
    return ModulatedNote(song_frequency, duration, modulations).modulation(armonic_note, note)
```

Lo que hace este método es llamar a la clase *ModulatedNote* que crea un objeto que representa la nota modulada, para ello se requieren todos los parámetros de la propia nota. Lo hace para llamar al método *modulations*.

```
class ModulatedNote:
    def __init__(self, song_frequency: int, duration: float, modulations:list):
        """
        Parameters
        -----
        duration : float
            The duration of the note
        modulations : dict
            The modulations of the instrument. See the read_files.py module.
        """

        if ((type(song_frequency) != int) or (type(duration) != float) or (type(modulations) != list)):
            print(song_frequency, duration, modulations)
            raise TypeError

        self.song_frequency= song_frequency
        self.duration= duration
        self.modulations= modulations
```

El método de esta clase que modula la nota según el momento de la misma (variando según, ataque, sostenido y decaimiento) es *modulation*.

```
def modulation(self, armonic_note, array_of_note):  
    """  
    Returns the modulated note.  
  
    Parameters  
    -----  
    armonic_note : numpy.ndarray  
        The armonic note as a numpy array  
    array_of_note : numpy.ndarray  
        The note as a numpy array  
  
    Returns  
    -----  
    numpy.ndarray  
        The modulated note as a numpy array  
    """  
    modulation, first_time, second_time= self.divide_modulation()  
    keys= [modulation[0][0], modulation[1][0], modulation[2][0]]  
  
    m= np.empty(int(self.song_frequency*(self.duration)))  
    slice1=int(self.song_frequency*first_time[0])  
    slice2=int(self.song_frequency*second_time[0])  
  
    m[:slice1]=dic_funcs[keys[0]](array_of_note[:slice1], first_time)  
    if modulation[1][0]=="PULSES":  
        arg=[modulation[1][1],modulation[1][2],modulation[1][3]]  
        m[slice1:slice2]=dic_funcs[keys[1]](array_of_note[slice1:slice2]-first_time[0],arg)  
    else:  
        m[slice1:slice2]=dic_funcs[keys[1]](array_of_note[slice1:slice2]-first_time[0],second_time)  
    m[slice2:]=m[slice2-1]*dic_funcs[keys[2]](array_of_note[slice2:]-second_time[0], [self.duration-second_time[0]])  
    modulated_note=0.02*m*armonic_note  
  
    return modulated_note
```

Primero se establecen los tiempos de cada parte de la nota (serán 3), luego se aplica la modulación correspondiente a cada una de estas partes (mediante 3 slicings del array). Observación: cuando la modulación del instrumento es "PULSES", los parámetros que se necesitan varían en cantidad del resto de las funciones de sostenido, y por este motivo se separó este caso mediante el uso de un condicional if.

Metalófono

Para la implementación de esta parte del trabajo práctico, utilizamos el programa provisto por los profesores obtenido en el siguiente repositorio:

<https://github.com/udesai-ai/xylophone>

A partir de la información provista dentro del repositorio, creamos un cliente con el cual se conectará a la red generada por el instrumento y nos permitirá comunicarnos de forma inalámbrica con el instrumento. Este archivo se encargará de leer una partitura (con el mismo formato que el utilizado en la primera parte del trabajo práctico) y extraer los 2 datos fundamentales que necesitamos pasarle al instrumento: es decir qué tipo de nota es la que se envía y en qué tiempo debe comenzar a sonar. Como existen muchas notas las cuales no son compatibles con el metalófono que vamos a utilizar para realizar las pruebas, decidimos además implementar un sistema de corrección de notas. Este se encargará de subir o bajar las octavas de las diferentes notas que no son compatibles, para que de esta forma puedan ser reproducidas por el instrumento. Como estas acciones pueden propiciar que la canción a tocar se escuche de una manera “extraña”, es recomendable utilizar una canción cuyas notas se encuentren dentro del rango de compatibilidad que tiene el programa (todas las especificaciones sobre las notas compatibles se pueden encontrar en el README del repositorio anterior).

A continuación, se muestra el archivo “metallophono.py”:

```
from xylophone.client import XyloClient
from xylophone.xylo import XyloNote
import argparse

parser = argparse.ArgumentParser(description='Sending musical notes to an instrument')
parser.add_argument('-i', type=str, help='Music Sheet name')
parser.add_argument('-o', type=str, help='Instrument IP')
args = parser.parse_args()

def main():
    """
    This function is responsible for sending the different notes corresponding to the instrument "Metallophone".
    """
    notes = []
    with open (args.i, 'r') as f:
        for line in f:
            line=line.strip().split(' ')
            start_time=float(line[0])
            value = ''
            #Notes are raised and lowered in octaves to be compatible with the instrument
            if len(line[1]) == 2:
                if line[1][0] == 'F' or line[1][0] == 'E' or line[1][0] == 'D' or line[1][0] == 'C':
                    value = line[1][0] + '5'
                elif line[1] == 'C8':
                    value = 'C7'
                elif int(line[1][1]) < 4 and line[1][0] != 'C':
                    value = line[1][0] + '4'
                elif int(line[1][1]) > 6 and line[1][0] != 'C':
                    value = line[1][0] + '6'
                else:
                    value = line[1]
```

```

elif len(line[1]) == 3:
    if line[1][0:2] == 'Eb' or line[1][0:2] == 'Db':
        value = line[1][0] + line[1][1] + '5'
    elif line[1][0:2] == 'Fb':
        if int(line[1][2]) <= 4:
            value = line[1][0] + '4'
        elif int(line[1][2]) >= 6:
            value = line[1][0] + '6'
    elif line[1][0:3] == 'Cb4' or line[1][0:3] == 'Cb5':
        value = 'Cb6'
    elif line[1][0:2] == 'Gb':
        value = 'Gb6'
    elif int(line[1][2]) < 4:
        value = line[1][0] + line[1][1] + '4'
    elif int(line[1][2]) > 6:
        value = line[1][0] + line[1][1] + '6'
    else:
        value = line[1]
    notes.append(XyloNote(value, start_time, 90))
#Send the information to the instrument
client = XyloClient(host=args.o, port=8080)
client.load(notes)
client.play()

if __name__ == '__main__':
    main()

```

Si se desea correr el programa, se lo puede llamar mediante la consola, pasándole los siguientes argumentos:

\$ python3 metallophone.py -i <Music sheet name> -o <Instrument IP>

Los parámetros son los siguientes:

- i es el archivo de texto que contiene la partitura de la canción a tocar.
- o es la IP de la red generada por el instrumento.

A continuación, se presenta un ejemplo de una invocación:

```
PS C:\Users\elmat\xylophone\xylophone\xylophone-pruebas> python3 metallophone.py -i BlackBird.txt -o 10.42.1.0
```

En el caso de que se desee realizar una prueba del envío de datos del cliente al servidor, existe la posibilidad de crear un “mockserver”, con el cual nuestro cliente se puede conectar y enviar información. Para hacerlo, se debe crear un archivo de este estilo:

```

from xylophone.server.server import MockXyloServer

server = MockXyloServer(host='localhost', port=8080)

server.start()

```

Una vez generado el servidor, solo es necesario correrlo y, al mismo tiempo, correr el archivo de cliente, pasándole como parámetro -o la palabra "localhost" o cualquier str que se asigne como host. Se debería invocar al cliente de la siguiente manera:

```
PS C:\Users\elmat\xylophone\xylophone\xylophone-pruebas> python3 metallophono.py -i BlackBird.txt -o localhost
```

Si la conexión se estableció de manera correcta, se deberían visualizar mensajes de este estilo en la terminal de programa que esté corriendo el código:

```
INFO: 2022-07-05 17:03:33,103 - Message was sent: 18090
INFO: 2022-07-05 17:03:33,263 - Message was sent: 16090
INFO: 2022-07-05 17:03:33,263 - Message was sent: 18090
INFO: 2022-07-05 17:03:33,576 - Message was sent: 16090
INFO: 2022-07-05 17:03:33,576 - Message was sent: 18090
INFO: 2022-07-05 17:03:33,888 - Message was sent: 18090
INFO: 2022-07-05 17:03:33,888 - Message was sent: 18090
INFO: 2022-07-05 17:03:33,888 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,196 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,196 - Message was sent: 15090
INFO: 2022-07-05 17:03:34,357 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,510 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,510 - Message was sent: 17090
INFO: 2022-07-05 17:03:34,671 - Message was sent: 12090
INFO: 2022-07-05 17:03:34,823 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,823 - Message was sent: 18090
INFO: 2022-07-05 17:03:34,823 - Message was sent: 16090
INFO: 2022-07-05 17:03:34,978 - Message was sent: 14090
INFO: 2022-07-05 17:03:35,453 - Message was sent: 16090
INFO: 2022-07-05 17:03:35,758 - Message was sent: 16090
INFO: 2022-07-05 17:03:35,758 - Message was sent: 18090
INFO: 2022-07-05 17:03:35,919 - Message was sent: 16090
INFO: 2022-07-05 17:03:36,227 - Message was sent: 18090
INFO: 2022-07-05 17:03:36,383 - Message was sent: 16090
INFO: 2022-07-05 17:03:36,383 - Message was sent: 18090
Time: 9.452243999999999
```

Correr el synthesizer mediante la consola

Con el archivo denominado *argparse_synthesizer.py* se podrá utilizar el sintetizador para generar el archivo .wav mediante la consola.

```
1 import argparse
2 from synthesizer import Synthesizer
3 parser = argparse.ArgumentParser(description="Create a .wav file using a partiture and an instrument")
4 parser.add_argument("-p", type=str, help="The Partiture")
5 parser.add_argument("-i", type=str, help="The Instrument")
6 parser.add_argument("-o", type=str, help="The name of the output file")
7 parser.add_argument("-f", type=int, help="The sample rate")
8
9 args = parser.parse_args()
10
11 Synthesizer(args.p, args.i).create_wav(args.o,args.f)
```

Para ello utilizamos el paquete de python llamado *argparse*.

Para ejecutarse primero se debe poseer un archivo de texto con la partitura (parámetro -p del código), otro archivo con el instrumento (parámetro -i) que tocará la partitura y luego el nombre del archivo wav (parámetro -o) que se va a generar, y por último la frecuencia o simple rate (parámetro -f). Por ejemplo:

```
$ python3 argparse_synthesizer.py -h
usage: argparse_synthesizer.py [-h] [-p P] [-i I] [-o O] [-f F]


Create a .wav file using a partiture and an instrument

options:
  -h, --help  show this help message and exit
  -p P        The Partiture
  -i I        The Instrument
  -o O        The name of the output file
  -f F        The sample rate
```

Esta sintaxis que se ejecutó “*python3 argparse_synthesizer.py -h*” muestra la “ayuda” para conocer acerca del programa próximo a ejecutar para conocer cómo debería escribirse. Una vez conociendo aquello, procedemos a ejecutar el programa en la consola.

```
$ python3 argparse_synthesizer.py -p queen.txt -i piano.txt -o Queen_proof -f 44100
Warning: There are 59 notes that are lower than the attack:0.02, and there are not going to reproduce
```

Observemos que introducimos como partitura al archivo queen.txt, como instrumento al archivo piano.txt, Queen_proof como nombre del archivo .wav (que se generará en la carpeta donde esté el archivo argparse_synthesizer.py), y por último la frecuencia (en este caso 44100).

 Queen_proof	5/7/2022 19:59	WAV Audio File (V...	113.350 KB
---	----------------	----------------------	------------

Podremos observar que (como se explicó anteriormente) hay una advertencia ya que existen algunas notas que no son posibles de reproducir.

Algo importante a tener en cuenta es que si, por ejemplo, asignamos un valor de distinto tipo al asignado a cada parámetro no podrá ejecutarse el programa y se retornará un error, como se puede ver en la siguiente captura:

```
$ python3 argparse_synthesizer.py -p queen.txt -i piano.txt -o Queen_proof -f "valor_erroneo"
usage: argparse_synthesizer.py [-h] [-p P] [-i I] [-o O] [-f F]
argparse_synthesizer.py: error: argument -f: invalid int value: 'valor_erroneo'
```


Bibliografía:

Procesamiento de audio con python: <https://www.youtube.com/watch?v=iNmtSnb7TIQ>

Trabajo con archivos .wav: <https://docs.python.org/es/3/library/wave.html>

Información sobre testing:

- <https://docs.python.org/3/tutorial/errors.html>
- <https://docs.python.org/3/library/unittest.html>
- <https://docs.pytest.org/en/6.2.x/assert.html>
- <https://cientificaserbias.github.io/blog/lo%20cotidiano%20es%20ciencia/FisicayMusicaEnArmonia/>
- <https://quanam.com/analizando-series-temporales-a-partir-de-la-musica/>
- <https://www.musiquiatra.com/index.php?/forums/topic/164701-frecuencias-fundamentales-y-arm%C3%B3nicos-en-los-instrumentos/>
- <http://www.fgsaja.com/?p=8537>
- <https://www.programiz.com/python-programming/examples/check-string-number>