

## Capítulo 13

### Aplicaciones de Python al procesamiento de señales

13.1 Introducción

13.2 Transformada de Fourier

13.3 Transformada Discreta de Fourier

13.4 Redes Neuronales en Python

13.5 Perceptrón Multicapa

13.6 Desarrollo de Filtros Digitales en Python

13.7 Controlador PID en Python

13.8 Instrucciones

13.9 Conclusiones

13.10 Ejercicios

*El hombre que se levanta es aún más grande que el que no ha caído.*

Concepción Arenal

### Objetivo

El objetivo de este capítulo es aprender a utilizar las herramientas que Python ofrece para resolver problemas de Procesamiento de Señales.

## 13.1 Introducción

Para mostrar la flexibilidad de Python, en este capítulo se realizarán diversas aplicaciones utilizadas en el ámbito de la investigación científica y tecnológica. Primero se presenta el manejo que Python hace de los números complejos aplicados a la transformada de Fourier. Posteriormente, el uso de redes neuronales. También se aplicará el manejo de arreglos para el procesado digital de señales, aplicando un filtro pasa bajas a una señal. Por último se desarrollará una aplicación tecnológica de control utilizada ampliamente en la industria, un control PID para ser evaluado en un sistema de segundo orden.

Se conoce como procesamiento digital de señales (Digital Signal Processing) al tratamiento matemático de una señal analógica utilizando circuitos digitales discretos. Los fundamentos matemáticos del Procesamiento de Señales fueron desarrollados por el genio de personajes tan importantes trascendentales en áreas de la física y las matemáticas como Leibnitz, Newton, Gauss, Euler, Bernoulli, Fourier, entre los más importantes[1].

## 13.2 Transformada de Fourier

Uno de los algoritmos más conocidos y estudiados dentro del procesamiento de Señales es la transformada de Fourier. Inicia su desarrollo a mediados del siglo XVI en los trabajos de Fourier sobre la conducción del calor en los cuerpos [2]. Fourier no sólo

encontró la ecuación que modelaba el fenómeno físico per se, también desarrolló el método para resolver la ecuación.<sup>1</sup>

Las ecuaciones (13.1-13.4) muestran el conjunto de lo que hoy se conoce como la serie trigonométrica de Fourier:

$$f(t) = \frac{1}{2}a_0 + \sum_{i=1}^n a_n \cos(n\omega_0 t) + b_n \text{sen}(n\omega_0 t) \quad (13.1)$$

$$a_0 = \frac{2}{T} \int_{-T/2}^{T/2} f(t) dt \quad (13.2)$$

$$a_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \cos(n\omega_0 t) dt \quad (13.3)$$

$$b_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \text{sen}(n\omega_0 t) dt \quad (13.4)$$

El resultado obtenido por Fourier encontrando que efectivamente una función periódica puede escribirse como una combinación lineal de funciones seno y coseno, dio pie a múltiples desarrollos posteriores realizados por las brillantes mentes de Dirichlet, Cantor, Riemman y Gibbs entre otros.

La representación de la serie de Fourier de forma exponencial, como aparece en las ecuaciones (13.5-13.6) se puede encontrar en la referencia 3.

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t} \quad (13.5)$$

$$c_n = \sum_{n=-\infty}^{\infty} f(t) e^{jn\omega_0 t} \quad (13.6)$$

La forma exponencial o compleja de la serie de Fourier, siempre aplicada a funciones periódicas, permitió extender su uso a otro tipo de funciones que no lo son. Utilizando una función no periódica como si tuviera un periodo infinito, con una

---

<sup>1</sup>El método de separación de variables para la resolución de ecuaciones diferenciales ya era utilizado por matemáticos de la época de Fourier pero es este último el que sistematiza el uso del método.

frecuencia infinitamente pequeña, es posible transformar las sumatorias de las expresiones 13.5 y 13.6 en integrales de funciones no periódicas.<sup>2</sup>

### 13.3 Transformada Discreta de Fourier

Con la llegada de la electrónica digital se hizo patente la necesidad de realizar el análisis de señales discretas, obtenidas y almacenadas en medios digitales. Para tal fin se usa la Transformada Discreta de Fourier (DFT por sus siglas en inglés), cuyo desarrollo se inició, según algunas fuentes, con Gauss en el Siglo XVI [5]. Esta transformada ha sido la herramienta más utilizada en el análisis de señales digitales en la última mitad del Siglo XX. La transformada se presenta en la siguiente ecuación:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-2\pi j\omega k/N} \quad (13.7)$$

En este punto cabe preguntarse. ¿Qué resultado se puede esperar de la Transformada de Fourier de una señal?

En primer lugar, lo que hace la transformada es descomponer una señal en sus elementos constituyentes en términos de la frecuencia. Adicionalmente, la DFT permite observar la energía de la señal para diversos componentes de frecuencia.

#### Ejemplo 13.1

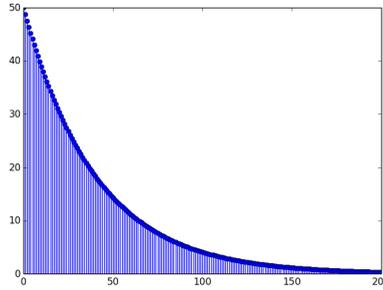
#### Cálculo directo de la DFT de una señal

Tomando como ejemplo una señal discreta consistente en un tren de pulsos con una amplitud de la forma definida por la expresión:

$$x[n] = \begin{cases} 50 \exp(-0.25t) & 0 \leq n \leq N - 1 \\ 0 & \text{Otros Casos} \end{cases} \quad (13.8)$$

La gráfica de la señal se observa en la Figura 13.1.

<sup>2</sup>El tratamiento matemático de la obtención de la Transformada de Fourier puede encontrarse en [10].



**Figura 13.1** Gráfica de un tren de pulsos exponenciales inversos.

Para ejemplificar el uso de la DFT, hagamos el cálculo directo de sus términos tal como aparece en la ecuación (13.7).

$$X[k] = \sum_{n=0}^{N-1} e^{-2\pi jnk/N} \quad (13.9)$$

¿Qué significa esta nueva ecuación? Antes de contestar necesitamos hacer una breve revisión de los números complejos.

El conjunto de los números complejos ha sido desde el siglo XIX una herramienta fundamental del álgebra, del cálculo, así como de variable compleja, ecuaciones diferenciales. En la física está presente en aerodinámica, hidrodinámica y electromagnetismo entre otras.

Los números complejos son una extensión del conjunto de los números reales, y forman, algebraicamente un conjunto cerrado para las operaciones suma y producto lo que significa que la suma y el producto de dos números complejos dan como resultado un número complejo.

Los números complejos están formados por un par ordenado de números reales. Una parte es conocida como la parte real y otra como la parte imaginaria. Se denota como un par ordenado (*real, imaginario*). La parte imaginaria va siempre acompañada del número imaginario  $i = \sqrt{-1}$ . Por ejemplo  $X = (5, -2i)$  es un número complejo con parte real 5 y parte imaginaria  $-2$ .

Existen otras formas de representar los números complejos. Si  $X = (x, y^*i)$ :

- binomial  $X = x - yi$

- polar  $X = |X|(\cos(x) + i \operatorname{sen}(y))$
- exponencial  $X = |X|\exp(i\theta)$

donde  $\theta$  es el argumento del número complejo que está dado por:

$$\theta = \tan^{-1} \frac{y}{x}$$

y

$$|X| = \sqrt{x^2 + y^2}$$

es la magnitud del número complejo  $X$ . En Python existe una clase que permite trabajar algebraicamente los números complejos, en la siguiente sección se hará mención a ella.

### 13.3.1 Clase `complex` en Python

Como puede observarse el cálculo anterior de la DFT utilizando la ecuación 13.7, implica el uso de números complejos. En Python existe una clase llamada `complex` que permite el manejo de números complejos y las operaciones que pueden utilizarse para hacer diversas operaciones aritméticas, y funciones que son sobrecargadas<sup>3</sup>. Las operaciones tradicionales para números flotantes se realizan también para números complejos.

Utilizando esta clase, los números complejos se definen y utilizan como un par ordenado ( *Parte Real*, *Parte Imaginaria* ) con lo cual la representación se hace más sencilla.

#### Ejemplo 13.2

### La aritmética de los números complejos

La aritmética de los números complejos y su aplicación puede observarse fácilmente en el siguiente listado:

```
a = complex(5, 8)
b = complex(4, -7)
print("\n Sean dos números complejos a: \n")
```

<sup>3</sup>La sobrecarga es la traducción de Overload que es el método en programación por el cual una función o un operador que está definido para un tipo de dato puede utilizarse para otro tipo de dato.

```
print(a)
print("\n y b: \n")
print(b)
print(" \n La suma de dos complejos es: \n")
print(a + b)
print("\n El producto de dos complejos es \n")
print(a*b)
print("\n La parte real del complejo a es: \n")
print(a.real)
print("\n La magnitud del complejo a es: \n")
print(abs(a))
print("\n El complejo conjugado del complejo a es: \n")
print( a.conjugate())
```

El resultado se muestra a continuación:

```
Sean dos números complejos a:
(5 + 8j)
y b:
(4 - 7j)
La suma de dos complejos es:
(9 + 1j)
El producto de dos complejos es
(76 - 3j)
La parte real del complejo a es:
5.0
La magnitud del complejo a es:
9.433981132056603
El complejo conjugado del complejo a es:
(5 - 8j)
```

---

Ahora, retomando el ejemplo del tren de impulsos, la DFT programada en Python puede realizarse con el siguiente código:

```

import cmath
imagina = complex(0, 1)

N = 201
a = [ ]
resultado = [ ]

i = 0
j = 0
for i in range(N):
    a.append(50*cmath.exp(-0.025*i))
    resultado.append(complex(0))

im = 1
for i in range(int(-N/2), int((N/2)), 1):
    resultado[im] = 0;
    k = 1
    for j in range(int(-N/2), int((N/2)), 1):
        resultado[im] = (resultado[im]
            + cmath.exp(imagina*cmath.pi*i*j/N)*a[k])
        k += 1
    im += 1

```

En este código, la señal de entrada está almacenada en la matriz **a**.

En este ejemplo se utilizó un tren de impulsos unitarios, por lo tanto hay que generarlo tomando un vector y llenándolo de unos para que sea el vector de entrada. Un ciclo **for** permite realizar esto de forma rápida:

```

import cmath # Biblioteca para números complejos
N = 1024
a = [ ]
i = 0
for i in range(N):
    a.append(1)
print((a))

```

Como se puede observar el código es sencillo de implementar, aunque de forma estricta computacionalmente no es el método óptimo. El número de operaciones de multiplicación compleja que debe realizar este algoritmo es proporcional a  $N^2$ , siendo  $N$  la longitud de la señal [3].

Al conjunto de algoritmos que permiten reducir el número de operaciones se conoce como Transformada Rápida de Fourier, FFT por sus siglas en inglés, la cantidad de operaciones se reduce utilizando las propiedades cíclicas de la DFT [4]. Volviendo al estudio del código, es necesario que el algoritmo diseñado permita guardar la información del resultado para analizarlo posteriormente, es por eso que se envía el resultado de la DFT a un archivo `csv` para poder graficarlo posteriormente si se desea. También se le pide al usuario que introduzca el nombre del archivo en el que desea que sean colocados los datos, eso se hace con el código que se lee a continuación:

```
# Abrir un archivo como escritura,
# solicitando el nombre al usuario
# Es necesario antes llamar una
# biblioteca que maneje archivos csv
import csv
print("\n Transformada Discreta de Fourier \n de una señal. ")
nombre = input("\nombre de archivo de salida (sin extension):\n")
f = open(nombre + ".csv", 'w')
```

El programa crea el archivo y lo abre para que los datos calculados sean guardados. Las diferencias entre los distintos tipos de archivos `csv` hace necesario configurar los delimitadores de campo entre otros elementos del archivo `csv` antes de proceder a escribir en el archivo.

```
obj = csv.writer(f, delimiter = ' ', quotechar = '|',
                 quoting = csv.QUOTE_MINIMAL)
obj.writerow (abs(resultado[im]))
```

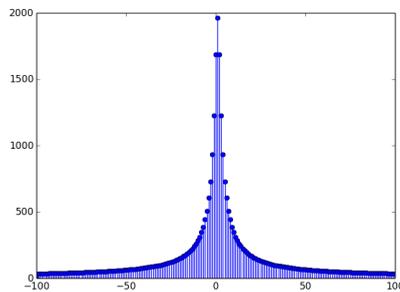
Para cerrar el archivo se usa la expresión `f.close()`. Una vez guardado el archivo `csv`, los datos se pueden graficar para observar la gráfica correspondiente. En el caso del tren de pulsos el resultado graficado se observa en la figura 13.2.

### Ejemplo 13.3

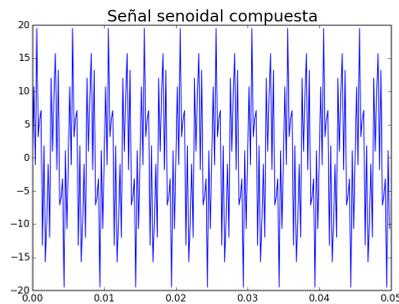
## DFT de la suma de dos señales senoidales

Para verificar la reusabilidad del código diseñado, éste puede utilizarse para realizar la DFT de una señal que sea la suma de dos senoidales.

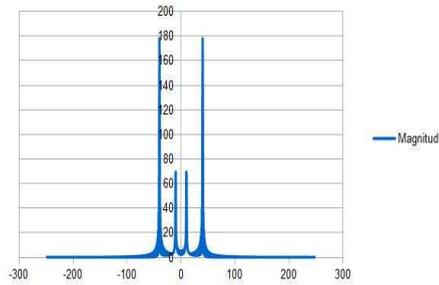
Nuevamente, la salida que muestra el resultado de la DFT al ejecutarlo puede ser abierta en una hoja de cálculo y gráficamente. En este caso se observa en el resultado como el espectro de frecuencias está concentrado en las frecuencias de ambas senoidales.



**Figura 13.2** Gráfica de la DFT de un tren de pulsos exponenciales.

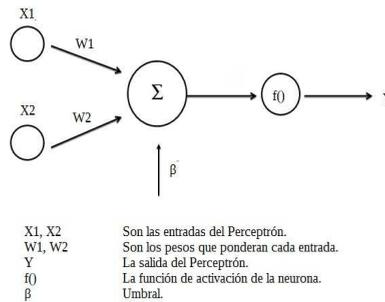


**Figura 13.3** Gráfica de la DFT de la suma de dos señales senoidales.



**Figura 13.4** Gráfica de la DFT de la suma de dos seales senoidales.

Con este sencillo ejemplo se demuestra la potencia de Python para el tratamiento digital de señales. Se deja al lector la implementación del algoritmo de la FFT.



**Figura 13.5** Arquitectura de un perceptrón con dos entradas y una salida.

## 13.4 Redes Neuronales en Python

Una red neuronal es un sistema que puede clasificar diversos patrones en conjuntos o clases a través del aprendizaje por ejemplos. El primer modelo de una red neuronal

fue ideado por los investigadores McCulloch y Pitts a principios de los años cuarenta del siglo XX. Este tipo de neurona era un autómata booleano, esto significa que sus entradas y su salida tomaban valores binarios.

A este autómata llegaban diversos potenciales (voltajes), los cuales eran sumados dentro del cuerpo de la neurona, si la suma rebasaba cierto nivel de umbral la neurona emitía una respuesta [11]. La expresión matemática que describe el comportamiento de esta neurona es:

$$y = f \left( \sum_{i=1}^n W_i e_i + \beta \right) \quad (13.10)$$

donde:

$y$  es la salida de la neurona

$e_i$  es la  $i$ -ésima entrada

$\beta$  es el umbral de entrada

$W_i$  es el peso de la  $i$ -ésima entrada

$f(\ )$  es la función umbral.

$n$  es el número de entradas de la neurona.

El primer modelo concreto de una red neuronal artificial fue propuesto por Rosenblatt [14] a finales de la década de 1950 y se conoce como el Perceptrón. Entre las características más interesantes de este modelo se puede destacar que es lo suficientemente complejo para tener un alto desempeño y lo suficientemente sencillo para poder ser objeto de análisis y permitir su estudio.

La arquitectura de ésta red es simple. Se trata de una estructura monocapa, en la que hay un conjunto de células de entrada, tantas como sea necesario según los términos del problema y una o varias células de salida. Cada una de estas células de entrada está conectada con todas las células de salida, siendo estas conexiones las que determinan el sistema. La figura 13.4 muestra la arquitectura del perceptrón.

### Ejemplo 13.4

## Perceptron programado en Python

Para crear un perceptrón en Python aplicaremos el concepto de Programación Orientada a Objetos y se diseñará una clase llamada neurona. Esta clase contendrá cuatro atributos, el valor que recibe cada entrada, el valor del peso ponderado de cada entrada, el umbral y la salida de la neurona. Por otra parte contendrá dos métodos, para actualizar los pesos y la información de cada entrada respectivamente.

```

# Neurona: Objeto de Cinco entradas, cada entrada
# con su matriz de pesos
# Cuenta con dos funciones, una para activar
# hacia adelante, feedforward
# la otra función actualiza los pesos cuando se está
# generando aprendizaje Backpropagation
classneurona():
    def__init__(self):
        self._entradas = [ ]
        self._pesos = [ ]
        self._salida = 0.0

```

La clase se define al principio del archivo como se muestra en el código anterior. Tomando como ejemplo el método `actualiza_pesos` se observa como se toma la información que se desea cambiar en el arreglo `pesos` de la neurona.

```

def actualiza_pesos(self, nvos_pesos):
    for i,elemento in enumerate(nvos_pesos):
        self._pesos.append(elemento)

```

La salida de la neurona se actualiza con el método `calcula`. Este método, para fines de demostración no implementa ninguna función, solamente realiza la combinación lineal de las entradas, se deja al lector implementar la función sigmoïdal clásica del perceptrón.

```

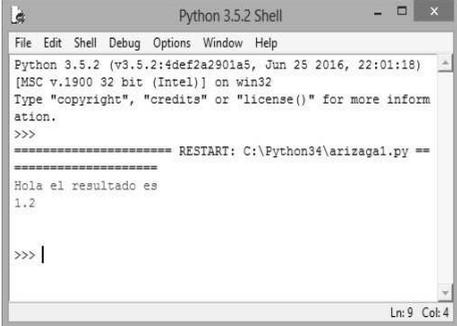
# *****
# calcula el resultado de la función
# y obtiene la salida como una
# combinación lineal de las entradas y los pesos.
# *****
def calcula(self):
    for a,b in zip(self._entradas,self._pesos):
        self._salida = self._salida + a*b
    self._salida = self._salida + self._bias
    return self._salida

```

Para realizar la prueba de este código, se incorporará esta clase a un programa nuevo donde sólo se actualicen pesos y se calcule la salida para una combinación de entradas:

```
mi_neurona = neurona( )
nuevos_pesos = [0.3, 0.4, 0.5, 0.1, 0.9]
mi_entrada = [1.0, 1.0, 1.0, 0.0, 0.0]
mi_neurona.actualiza_pesos(nuevos_pesos)
mi_neurona.actualiza_entradas(mi_entrada)
print("Hola el resultado es ")
print(mi_neurona.calcula())
```

El programa devuelve una salida donde aplicamos la ecuación 13.10. El resultado se observa en la figura 13.6.



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18)
[MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python34\arizagal.py ==
=====
Hola el resultado es
1.2

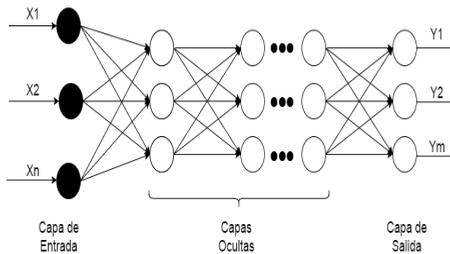
>>> |
```

**Figura 13.6** Salida del Programa Neuronal.

## 13.5 Perceptrón Multicapa

La arquitectura del perceptrón multicapa se caracteriza por que tiene sus neuronas agrupadas en capas de diferentes niveles. Distinguiéndose tres tipos de capas dife-

rentes: la capa de entrada, las capas ocultas y la capa de salida, como se observa en la figura 13.7.



**Figura 13.7** Red neuronal multicapa.

Las neuronas de la capa de entrada son encargadas de transmitir la información a la primera capa oculta. Estas a su vez procesan la información recibida aplicando la función de activación  $f$  a la suma de los productos de las activaciones que recibe multiplicados por la ponderación (pesos) de cada uno de ellos.

Las neuronas que conforman la capa de salida utilizan el mismo mecanismo para procesar la información recibida. Esta regla  $f$  es llamada *función de activación*. Para el perceptrón las funciones de activación más utilizadas son la función sigmoideal y la función tangente hiperbólica [12].

Estos modelos, que pueden llamarse clásicos, no son los únicos, actualmente existen diversos estudios sobre arquitecturas. Para una mayor comprensión del estudio de estas u otras arquitecturas de redes neuronales se recomienda consultar las referencias 11 y 13.

Para desarrollar una red neuronal es necesario establecer el número de capas y el número de neuronas por capa. Para realizar esto, la clase neurona que se ha desarrollado permite que se pueda generalizar su uso.

Uno de los ejemplo clásicos del uso del perceptrón multicapa es para generar una compuerta XOR. Los problemas de este tipo no pueden resolverse como una combinación lineal de las entradas, por lo que la solución de este y otros problemas de este tipo se resuelven utilizando redes neuronales. Se deja al lector intentar resolver este problema.

### 13.6 Desarrollo de Filtros digitales en Python

Las técnicas digitales de procesamiento de señales tuvieron a mediados de siglo XX un amplio desarrollo principalmente con el crecimiento de los sistemas digitales de cómputo. Antes de la década de los años 60 del siglo pasado, el procesamiento de señales de audio y video era en modo continuo en el tiempo (analógico). En la actualidad por el contrario, los sistemas de comunicaciones (civiles y militares), de entretenimiento (Radio y Televisión, música), análisis médicos (ECG, EEG), etc., se llevan a cabo usando electrónica digital a través de Procesadores de Señales Digitales.

El filtrado en el dominio de la frecuencia es quizá la forma más simple de manipulación matemática de una señal. La teoría de estas herramientas se encuentra bien establecida tanto en el dominio analógico como en el dominio digital.

La ecuación general de un filtro digital IIR (Infinite Impulse Response) se observa en la ecuación siguiente<sup>4</sup>:

$$y(n) = \sum_{k=0}^{M-1} b(k)x(n-k) - \sum_{k=1}^{N-1} a(k)y(n-k) \quad (13.11)$$

donde  $x$  es la señal de entrada que se desea filtrar y  $b(k)$  y  $a(k)$  son los coeficientes del filtro. El filtro es de orden  $N$  y la salida del mismo está representada con la letra  $y$ , de forma que para representar el filtro en el dominio de la frecuencia discreta  $z$  se tiene la siguiente expresión:

$$H(z) = \frac{\sum_{k=0}^{M-1} b(k)z^{-k}}{\sum_{k=1}^{N-1} a(k)z^{-k}} \quad (13.12)$$

donde  $H(z)$  es la función de transferencia del filtro [21].

Cada elemento de la entrada y la salida están muestreados con un periodo  $T$ . El inverso de este periodo se llama frecuencia de muestreo. El número de muestra se denota por la letra minúscula  $n$ .

---

<sup>4</sup>La ecuación 12 representa también la operación conocida como convolución, consultar la información de [13] para mayor referencia.

**Ejemplo 13.5****Diseño y aplicación de filtros digitales en Python**

El filtro IIR pertenece a los filtros que trabajan con las componentes de frecuencia de las señales. Dentro de estos filtros existen tres tipos elementales de filtrado: pasa-bajas, pasa-banda y pasa-altas.

El ejemplo que se propone para esta sección será un filtro pasa-bajas para una señal compuesta por la suma de dos señales senoidales de distintas frecuencias. Esta señal es modelada en Python usando el código siguiente:

```
# Creamos una señal que consta de la suma de dos senoidales
# y la graficamos.
fs = 5000.0
cutf = 500.0
T = 0.05
nsamples = T*fs
t = np.linspace(0, T, nsamples, endpoint = False)
x = (10*np.sin(2000*t*2*np.pi) + 10*np.sin(400*t*2*np.pi));
plt.figure(1)
plt.clf()
plt.plot(t, x, label = 'señal senoidal compuesta')
```

La última línea del código anterior genera la gráfica en el tiempo de la señal:

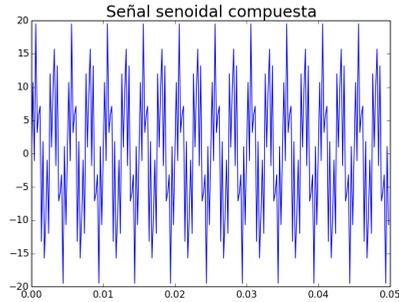
$$x = 10 \operatorname{sen}(2000\pi t) + 10 * \operatorname{sen}(400\pi t)$$

que se muestra en la figura 13.8.

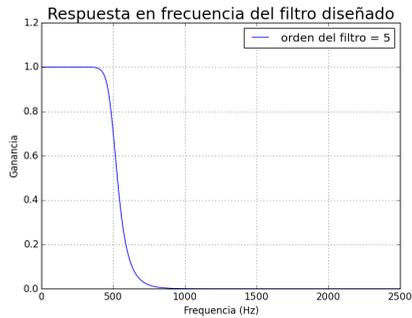
Se desea realizar el filtro de forma que la señal de 400 Hz de frecuencia sea atenuada y sólo la de 2000 Hz aparezca. El diseño del filtro, esto es, encontrar los coeficientes del mismo se realiza a través del código siguiente:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import freqz, butter, lfilter
# Se crea un filtro Butterworth con una frecuencia de corte de 500 Hz.
nyq = 0.5 * fs
```

```
low = cutf / nyq
b, a = butter(9, low, btype = 'low')
```



**Figura 13.8** Resultado de la suma de dos señales senoidales



**Figura 13.9** Respuesta en frecuencia del filtro pasabajos diseñado

Para realizar el filtro digital en Python se utilizan las bibliotecas `numpy` y `scipy` ya que cuentan con diversas herramientas matemáticas<sup>5</sup>, en este caso, la función `lfilter`, que realiza el filtrado de señales y la función `butter`, que permite crear los coeficientes de un filtro digital utilizando el desarrollo del filtro Butterworth [21].

```
y_filtrada = lfilter(b, a, x)
plt.figure(2)
plt.clf()
```

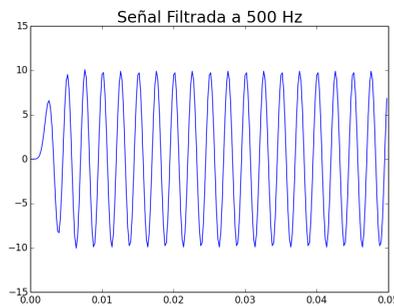
<sup>5</sup>Existen diferentes aproximaciones para diseñar filtros IIR. En las referencias al final del capítulo puede encontrarse más información sobre el tema.

```
plt.plot(t, y_filtrada)
```

Las líneas anteriores realizan el filtrado de la señal original. La señal filtrada está ahora contenida en la variable `y_filtrada` la cual es graficada a través del comando `plot`.

Como puede observarse la señal sólo presenta una sola senoidal, el filtro ha suprimido la senoidal de 400Hz.

Se deja al lector verificar que en el dominio de la frecuencia la señal solamente tiene la componente de 2000Hz utilizando el código para la DFT.



**Figura 13.10** Señal recuperada a la salida del filtro

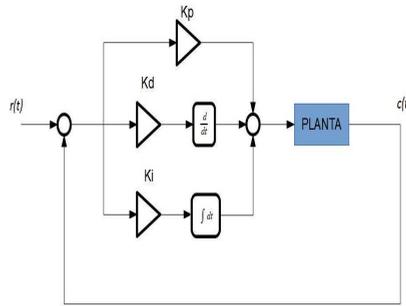
## 13.7 Controlador PID en Python

Los sistemas de control se han convertido, desde la última mitad del siglo XX, en una parte vital dentro del avance tecnológico y científico de la humanidad. Por ejemplo, los sistemas de control de procesos dentro de la industria automovilística, permitiendo la automatización de diversos elementos dentro del vehículo. Los sistemas de navegación marítima y aérea también están basados en sistemas de control de posición y velocidad. Incluso aparatos tan simples como lavadoras incluyen actualmente sistemas de control para realizar sus procesos.

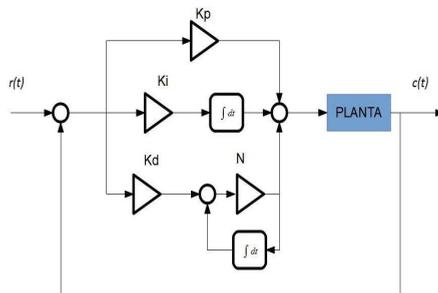
Un control Proporcional Integrativo y Derivativo (PID) es una estrategia basada en dispositivos electrónicos para mejorar la respuesta transitoria de sistemas físi-

cos a estímulos que los activan. Los controladores PID son diseñados para cumplir parámetros en el dominio del tiempo, ya sea tiempo de levantamiento, tiempo de asentamiento, sobrepaso, etc. [15]. El esquema a bloques de un Controlador PID se muestra en la figura 13.11.

Basado en la información de varias fuentes, el PID es utilizado en el 97% de los sistemas de regulación y control en la industria. Existen dispositivos digitales que son utilizados para sistemas embebidos los cuales controlan sistemas físicos, un ejemplo sería la computadora que determina la cantidad de gasolina que se inyecta a un motor, el control se hace basado en el movimiento del pedal. Otro ejemplo es el control de dirección de piloto automático en un avión. En ambos casos es necesario asegurar la respuesta del sistema.



**Figura 13.11** Diagrama a bloques de controlador PID clásico.



**Figura 13.12** Controlador PID con filtro en la parte diferencial.

Para asegurar que el sistema responderá de manera eficiente es necesario también realizar una etapa de control PID, este proceso del control puede ser programado en Python. La función de transferencia de un Control PID (ver ec. 13.13) se desprende de la imagen de la figura 13.11:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau) d\tau + \frac{de(t)}{dt} \quad (13.13)$$

### Ejemplo 13.6

#### Diseño de un control PID en Python

Esta ecuación es llamada la forma ideal. En la literatura se encuentran otras formas distintas, una que resulta interesante por generar una etapa de filtrado que estabiliza el proceso de control se observa en la figura 13.12. La función de transferencia del Control PID con el filtro se muestra en la ecuación 13.14. En su libro Del Vecchio y Murray [22] explican de forma muy detallada cual es la labor del filtro pasa bajas en la respuesta del control.

$$\frac{U(s)}{E(s)} = K_p + \frac{K_i}{s} + \frac{K_d N}{1 + N_s} \quad (13.14)$$

Primero construimos una clase llamada `pid` donde se definen las variables que guardarán los valores de las ganancias, el tiempo de muestreo y se define la salida del control:

```
class pid():
    def __init__(self):
        self._entrada = [ ]
        self._salida = 0.0
        self._Kp = [ ]
        self._Ki = [ ]
        self._Kd = [ ]
        self._N = [ ]
        self._Ts = 0.001
        self._Fs = 1/Ts
```

```

self._integra = 0.0
self._error_anterior = 0.0
self._deriva = 0.0

```

Esta clase cuenta con tres métodos. El primero sirve para para inicializar (guardar) en las variables los valores de  $K_d$ ,  $K_i$ ,  $K_p$  y  $N$ .

```

def inicializa(Kp,Ki,Kd,N):
    self._Kp = Kp;
    self._Ki = Ki;
    self._Kd = Kd;
    self._N = N;

```

El segundo permite cambiar el tiempo de muestreo del control.

```

def set_Ts(Ts):
    self._Ts = Ts;
    self._Fs = 1/Ts;

```

Por otro lado el valor de salida del control debe de calcularse y actualizarse para lo cual se dispone de un método que realice esa tarea. El periodo de tiempo en el cual se debe de actualizar la salida de control depende de la planta que se desea controlar. El método `calcula` realiza el cálculo del control utilizando la ecuación 13.13.

```

def calcula(error):
    self._integra += error*self._Ts;
    #La integral realizada con la forma rectangular
    deriva = ((self._Kd*error - self._deriva)*self._N);
    #haciendo la derivada con una aproximación sencilla usando Filtro
    #Con el Coeficiente N
    self._deriva += self._Ts* deriva;
    #En este punto se actualiza el valor del error anterior
    self._error_anterior = error
    return(self._kp*error + self._integra + deriva)

```

Este tipo de control PID es muy utilizado en plataformas de desarrollo con microcon-

troladores de 8, 16 y 32 bits. Como ejemplo se tiene Arduino [18] y mbed [19] los cuales cuentan con bibliotecas que permiten el control de motores de CD controlándolos con estrategia PID.

### 13.8 Instrucciones de Python del Capítulo 13

Instrucción	Descripción
<code>complex(real, img)</code>	Crea una variable compleja.
<code>dato.real</code>	Obtiene la parte real de un número complejo.
<code>abs(dato)</code>	Obtiene la norma de un número complejo.
<code>dato.conjugate</code>	Obtiene el complejo conjugado.
<code>open(nombre.ext, 'o')</code>	Se utiliza para crear un archivo de nombre como solo lectura 'r' o escritura 'w'.
<code>dato.append(elemento)</code>	Permite agregar un dato extra.
<code>butter( )</code>	Función de la biblioteca <code>scipy.signal</code> para diseñar filtros Butterworth.
<code>freqz(b, a)</code>	Función de la biblioteca <code>scipy.signal</code> la cual, dados los coeficientes de un filtro digital obtiene la respuesta en frecuencia.
<code>lfilter(b, a, señal)</code>	Función de la biblioteca <code>scipy.signal</code> la cual, dados los coeficientes (b, a) de un filtro obtiene la señal filtrada.
<code>figure(x)</code>	Función de la biblioteca <code>matplotlib.pyplot</code> que despliega la gráfica de una señal.
<code>plot(x, y)</code>	Función de la biblioteca <code>matplotlib.pyplot</code> que despliega una gráfica bidimensional.
<code>clf</code>	Función que limpia el área de la gráfica para desplegar una nueva.
<code>linspace(x<sub>i</sub>, x<sub>f</sub>, x)</code>	Función de la biblioteca <code>numpy</code> que retorna valores espaciados en todo el intervalo.

### 13.9 Conclusiones

En este capítulo se cubrieron algunas aplicaciones que tiene Python en el procesamiento de señales. Se revisaron funciones de números complejos para el análisis en frecuencia de señales así como dos de las bibliotecas de análisis matemático `numpy` y `scipy` más utilizadas en la actualidad, las cuales hacen de Python un lenguaje ideal para el ámbito científico y tecnológico.

### 13.10 Ejercicios

1. Realizar una red neuronal con dos capas que sea capaz de resolver el problema de la función XOR.
2. Crear un filtro pasa altas que elimine la señal de baja frecuencia del ejemplo 13.6. Verifique que esto suceda obteniendo la transformada de Fourier de la señal filtrada usando el programa que viene como ejemplo de la transformada de Fourier.
3. Un motor de corriente directa tiene una función de transferencia de la forma

$$\frac{C(s)}{R(s)} = \frac{185}{s + 10.22}$$

Diseñe un control PID para realizar el control en lazo cerrado.

4. Para la señal

$$f(t) = 10\text{sen}(50 * 2\pi * t) + 5\text{sen}(100 * 2\pi * t) + 3\text{sen}(200 * 2\pi * t)$$

diseñe un filtro pasabandas Butterworth de orden 11 que elimine las componentes de 50 y 100Hz.

5. Grafique utilizando `plot()` la gráfica de la señal del ejercicio anterior.
6. Utilice `freqz(b, a)` para obtener la respuesta en frecuencia del filtro diseñado del ejercicio anterior. Gráfique la respuesta.

### 13.11 Referencias

1. A. V. Oppenheim, R. W. Schaffer "Discrete-time signal processing", 3era edición, Cap. 8, Upper Saddle River, NJ: Prentice-Hall, Inc., 2010.
2. <http://fftw.org/>
3. J. Duoandikoetxea, "Análisis de Fourier: historia y aplicaciones recientes", American Mathematical Society, 2001.
4. <http://www.lextrait.com/vincent/implementations.html>
5. A. V. Oppenheim, R. W. Schaffer "Discrete-time signal processing", 3era edición, Cap. 9, Upper Saddle River, NJ: Prentice-Hall, Inc., 2010.
6. J.B. Joseph Fourier, "Théorie analytique de la chaleur", Firmin Didot Père et Fils, Paris, 1822.
7. R. Rodríguez y E. Zuazua, "Series de Fourier y fenómeno de Gibbs", Departamento de Matemática Aplicada, Universidad Complutense de Madrid, Madrid, España, 2003.
8. E. Zuazua, "Temas relevantes de la Matemática actual: el reto de la Enseñanza Secundaria", Centro de publicaciones del Ministerio de Educación, Cultura y Deportes/UIMP, Madrid, 2000.
9. <http://www.phys.ufl.edu/~coldwell/Fourier/DFT%20code.htm>
10. H.P. Zhu, Análisis de Fourier, Prentice Hall, México, 1998.
11. E. Davalo y P. Naïm, *Neural Networks*, 1ra Ed., Macmillan Press Ltd., Londres UK, 1991.
12. S. Haykin, *Neural Networks a Comprehensive Foundations*, Ed. Macmillan USA, 1994.
13. P. Isasi, "*Redes neuronales artificiales, Un enfoque práctico*", Pearson Educación, Madrid, 2004.
14. F. Rosenblatt, *The Perceptron: a probabilistic Model for Information Storage and Organization in the Brain*, Psychological Review, Vol. 65, pp. 386 - 408.
15. K. Ogata, "Ingeniería de Control Moderna", Pearson Educación, 2003.
16. D. DelVecchio y R. M. Murray, "Biomolecular Feedback Systems", Princeton University Press, Princeton, NJ., 2015.
17. [http://es.wikis.org/wiki/Programacion\\_en\\_C++](http://es.wikis.org/wiki/Programacion_en_C++)

18. <http://www.arduino.cc/>
19. <https://mbed.org/>
20. S. K. Mitra, “Digital Signal Processing. A Computer-Based approach”, McGraw-Hill Book Co., NY., 2004.
21. S. Butterworth. “Experimental Wireless and the Radio Engineer”, Wireless Engineer , vol. 7, 1930, pp. 536-541.
22. D. Del Vecchio y R. M. Murray “Biomolecular Feedback Systems”, Princeton University Press, Princeton, NJ., 2014.