Universidade Vigo

RED PERSONAL PARA ENTRENAMIENTO DEPORTIVO MEDIANTE INTELIGENCIA ARTIFICIAL

Autor: Pablo García Covelo
Tutor: Francisco Javier González Castaño

Escuela de Ingeniería de Telecomunicación

Grado en ingeniería de Tecnologías de Telecomunicación
Especialidad en Telemática

Curso: 2018-2019

Trabajo Fin de Grado

Índice

1. Introducción

Se plantea desarrollar un sistema que sea capaz de medir la fuerza con la que un boxeador impacta en un saco de boxeo, así como la altura a la que realiza los golpes para que los deportistas puedan ver los progresos que realizan durante sus entrenamientos

Estos datos podrían ayudar al boxeador a saber si está realizando los golpes con la técnica y la altura adecuadas, así como saber si también está golpeando con la fuerza deseada.

Para ello se adaptará el saco de boxeo para poder incorporarle nueve acelerómetros. Además, el propio boxeador también dispondrá de acelerómetros distribuidos por distintas partes de su cuerpo.

2. Objetivos

Por un lado, se dispone de una placa SeeedStudio Beaglebone Green Wireless para manejar los acelerómetros H3LIS331DL que irán incorporados en el saco de boxeo. Mediante esta placa se podrán realizar lecturas de dichos acelerómetros empleando el puerto I2c de la misma. También se emplearán cuatro *hubs* I2c para manejar los nueve acelerómetros que se emplearán en las mediciones.

Por otro lado, se dispone de un módulo WiFi iGS01 accesible en la dirección 192.168.1.10:8080, en la cual se podrá crear un *socket* por el que se recibirá la información de los acelerómetros. De esta manera se podrá recibir la información que aportan los acelerómetros iBS01G que llevará el propio boxeador incorporados en el cuerpo y que se comunicarán con el módulo WiFi empleando dicha dirección IP.

Además de leer la información que aportan cada uno de los acelerómetros a través de cada uno de los componentes anteriormente citados, también será necesario realizar una lógica de programación que permita detectar impactos en dichos acelerómetros a partir de la información que se obtiene de los mismos. Esta información debe ser enventanada en el momento que se detecta un impacto para capturar muestras en los instantes previos y posteriores a la detección del impacto.

Por último, para poder consultar y procesar las muestras obtenidas por los acelerómetros sobre los impactos detectados, se guardarán las muestras capturadas en una base de datos que posteriormente pueda ser accesible para su estudio.

3. SeeedStudio BeagleBone Green Wireless

"SeeedStudio BeagleBone Green Wireless es un esfuerzo conjunto de beaglebone.org y SeeedStdio. Se basa en el diseño de hardware de código abierto de BeagleBone Black y se desarrolló en esta versión diferenciada. SeeedStudio BeagleBone Green Wireless incluye una interfaz WiFi/Bluetooth flexible de alto rendimiento y dos conectores Grove, lo que facilita la conexión a la gran familia de sensores Grove."

3.1 Configuración

En lo referente a la configuración de la placa, se hará uso de una imagen del Sistema Operativo Debian adaptada para funcionar en la misma. Concretamente se empleará la imagen Stretch Snapshot iot con la versión 9.4 de Debian.

Una vez instalada la imagen del Sistema Operativo en la memoria física de la placa esta ya estaría lista para su uso.

En el **Anexo 2** se detalla más información sobre la configuración de la placa SeeedStudio BeagleBone Green Wireless.

3.2 Hardware

El acelerómetro empleado para este proyecto es el H3LIS331DL, el cuál realiza capturas tridimensionales y dispone de dos puertos para entregarlas. El puerto 18 está asociado con el bit 0 (señal baja) y el puerto 19 está asociado con el bit 1 (señal alta). Por defecto el sensor viene configurado de fábrica para utilizarlo únicamente con el puerto 18, estando puenteado el *bus* de la placa que gestiona los canales de escucha con una señal baja (conectado a masa para generar un 0) y (por tanto, el canal de escucha 19 asociado a una señal alta está inutilizado).

A la hora de realizar el proyecto, esto supone un problema al tener que elaborar una lógica de programación distinta para gestionar los nueve acelerómetros, puesto que simultáneamente solo se puede escuchar un acelerómetro por un mismo canal.

Como solución se opta por cortar el enlace que puentea el *bus* de gestión de canales de escucha con la señal baja. Al cortar esta unión el *bus* pasa a trabar por defecto en el canal 19, puesto que recibirá una señal alta, pero queda libre para gestionarlo mediante software. Empleando los puertos GPIO de los que dispone la placa, y soldando un cabe al *bus* que gestiona los canales de escucha del sensor, se puede enviar libremente señales bajas, bit 0, o señales altas, bit 1, a dicho *bus*, para seleccionar el canal en el que se quiere que trabaje el sensor.

Teniendo esto en cuenta se decide implementar una lógica de programación que gestione turnos de transmisión para cada acelerómetro, dejando el canal 19 para los sensores cuya información no va a ser tratada en ese instante de tiempo y el canal 18 para otorgar los turnos de transmisión a cada uno de los distintos acelerómetros. Solo puede haber un único sensor enviando información por el canal 18, sino no sería posible saber qué información corresponde a cada sensor. Cuando se le retira el turno de transmisión a un sensor se le envía una señal alta por el puerto GPIO asociado, para cambiar el canal en el que trabaja al 19.

3.3 Software

Para la elaboración del código que permitirá capturar la información que aportan los acelerómetros se ha optado por un desarrollo en Python. Las razones para escoger este lenguaje de programación y no otro han sido: elaborar un proyecto en un lenguaje de programación con el que nunca antes se había trabajado y su adecuación para el objetivo.

Se ha tratado de realizar un código lo más agnóstico posible respecto al tipo de sensor empleado para realizar las mediciones. De este modo la estructura del proyecto se ha segmentado en cuatro partes:

- Un fichero de configuración “config.ini” donde se recoge la configuración específica al funcionamiento de este sensor y que será tratada al principio de la ejecución del programa.

- Un fichero de especificaciones “spec.json” donde se recogen los puertos GPIO habilitados en la placa y que podrán ser usados por el programa para la gestión de los canales del acelerómetro H3LIS331DL.

- Una clase “Handler.py” que, usando la información del fichero de configuración, ofrecerá una serie de métodos específicos obtener y tratar la información que genera dicho modelo (H3LIS331DL).

- Por último, la clase principal “H3LIS331DL.py”, la cual recoge la estructura que sería común a cualquier tipo de acelerómetro. Crea los diccionarios que se encargarán de almacenar la información capturada por la clase “Handler.py” así como de gestionar los turnos de lectura de información de los acelerómetros; así como de, según la información que recibe, estimar si se ha producido o no un impacto para enventanar la información capturada.

De esta manera se pretende que, si en un futuro se quisiese cambiar de sensor, solo sea necesario cambiar la información del fichero de configuración y modificar la clase “Handler.py” o crear una nueva, sin necesidad de modificar el código de la clase principal.

Las librerías de Python y Adafruit empleadas para la elaboración del código fuente del proyecto permiten realizar capturas de 18/19 muestras por segundo en cada uno de los nueve acelerómetros incorporados al saco.

En el **Anexo 3** se detalla más información sobre la elaboración del código.

4. INGICS Technology

Los dispositivos empleados en el proyecto son:

- BLE-WiFi Gateway iGS01
- Sensor Beacon iBS01G

4.1 BLE-WIFI Gateway iGS01

Este módulo permite conectar los dispositivos de INGICS Technology. Actúa a modo de gateway entre los sensores y nuestro PC. El dispositivo iGS01 crea un punto de acceso al que podremos conectar nuestro PC mediante WiFi, y al que los sensores se conectarán automáticamente una vez se enciendan. De esta manera se podrá capturar información de los mismos empleando el protocolo TCP.

Para poder proceder con la captura de información de los acelerómetros, además de conectar nuestro ordenador al punto de acceso generado por el gateway también será necesario que nuestro programa disponga de un *socket* escuchando en la dirección 192.168.2.1 y puerto 8080.

En el **Anexo 4** se detalla más información referente a la comunicación PC-gateway.

4.2 Sensor Beacon iBS01G

Este dispositivo es un acelerómetro que realiza capturas de información tridimensionales cada 100 milisegundos, hasta formar una trama con tres muestras, que posteriormente envía al gateway iGS01. Tarda 300 milisegundos en realizar una captura completa.

4.3 Software

Para la elaboración del código que permitirá capturar la información que aportan los acelerómetros, por las mismas razones citadas en el apartado **3.3**, se ha optado por un desarrollo empleando el lenguaje de programación Python.

Tratando de realizar un código que sea lo más agnóstico posible frente al tipo de sensor empleado, se ha estructurado en cuatro secciones:

- Un fichero de configuración “config.ini” donde se recoge la configuración específica al funcionamiento de este sensor y que será tratada al principio de la ejecución del programa.

- Un fichero de especificaciones “spec.json” donde se recogen los identificadores de los acelerómetros que van a ser usados para realizar las mediciones.

- Una clase “Handler.py” que, a partir de la información del fichero de configuración, ofrecerá una serie de métodos específicos para trabajar con este acelerómetro y obtener y tratar la información que genera.

- Por último, la clase principal “BLE_client.py”, que recoge la estructura que sería común a cualquier tipo de acelerómetro. Crea los diccionarios que se encargarán de almacenar la información capturada por la clase “Handler.py” así como de estimar si se ha producido o no un impacto para enventanar la información capturada.

Se pretende que solo sea necesario cambiar la información del fichero de configuración y modificar la clase “Handler.py” en caso de que se desee emplear un acelerómetro distinto.

En el **Anexo 5** se detalla más información sobre la elaboración del código.

5. Conclusión

Con la elaboración de este proyecto el alumno ha adquirido conocimientos sobre un lenguaje de programación nuevo, Python 2.7. También ha aprendido a utilizar algunas herramientas como los “virtualenviroment”, que permiten separar el entorno de desarrollo del núcleo de Python a la hora de instalar librerías, o el comando “pip”, que permite instalar las librerías de desarrollo necesarias.

En lo relacionado a la placa SeeedStudio BeagleBone Green Wireless, ha aprendido a trabajar con sistemas embebidos, así como a realizar su configuración desde cero. Desde instalar el Sistema Operativo desde una tarjeta micro SD, hasta manejar algunos de los puertos de los que dispone la placa, como los puertos I2C y GPIO para manejar módulos como los acelerómetros H3LIS331DL.

Anexo 1: Estado del arte

El uso de tecnología en el deporte cada vez está más extendido y existen diversos estudios sobre el análisis de rendimiento de deportistas mediante inteligencia artificial.

Por ejemplo, el estudio en **[1]** emplea acelerómetros para medir la repetitividad de la aceleración de las piernas de varios jugadores de futbol y relacionarla con su nivel de juego.

Otro estudio sobre nadadores **[2]** establece que "*El éxito de un nadador se determina por la capacidad de generar fuera propulsiva y, a la vez, de reducir la resistencia en su desplazamiento. Entre las causas que generan el desplazamiento del nadador (fuerzas) y resultado final de competición (tiempo) están los parámetros cinemáticos*" **[2].** Los nadadores llevan acelerómetros en sus muñecas durante sus entrenamientos para capturar datos que posteriormente se procesan.

Otros estudios persiguen supervisar la actividad deportiva general de las personas, también empleando acelerómetros para capturar la información, en vez de centrarse en un deporte concreto. La idea es proporcionar una asistencia similar a la de un entrenador personal **[3].**

Anexo 2: Configuración SeeedStudio BeagleBone Green Wireless

Cómo ya se ha dicho, para el funcionamiento de la placa SeeedStudio BeagleBone Green Wireless se empleará una imagen ISO basada en el sistema operativo Debian, que puede encontrarse en el siguiente enlace.

Para la instalación del Sistema Operativo que contiene la imagen descargada es necesario instalar primeramente dicha imagen en una tarjeta microSD. El programa Etcher permite realizar esta tarea dejando la tarjeta microSD lista para instalar el Sistema Operativo en la placa SeeedStudio BeagleBone Green Wireless.

El programa Etcher puede encontrarse en el siguiente enlace.

Imagen 2.1 *– Interfaz de usuario de Etcher*

Cuando el programa Etcher termina de instalar la imagen ISO en la tarjeta microSD, esta debe introducirse en el lector de tarjetas de la placa SeeedStudio BeagleBone Green Wireless, estando esta apagada. Una vez se encienda la placa, ella misma procederá por defecto con la instalación del Sistema Operativo en la memoria física que posee, tras detectar la tarjeta microSD en su lector.

Una vez finalizada la instalación del Sistema Operativo en la placa, dependiendo del Sistema Operativo que empleemos en nuestro PC, existen varios modos de conectarnos a la placa.

IP Address	Connection Type	Operating System(s)	Status
192.168.7.2	USB	Windows	Inactive
192.168.6.2	USB	Mac OS X, Linux	Inactive
192.168.8.1	WiFi	all	Inactive
beaglebone.local	all	mDNS enabled	Inactive
beaglebone-2.local	all	mDNS enabled	Inactive

Imagen 2.2 *- Modos de acceso a placa BeagleBone Green Wireless*

Empleando el comando “ssh” desde el terminal de nuestro PC a una de las direcciones mostradas en la ***Imagen 2.2***.

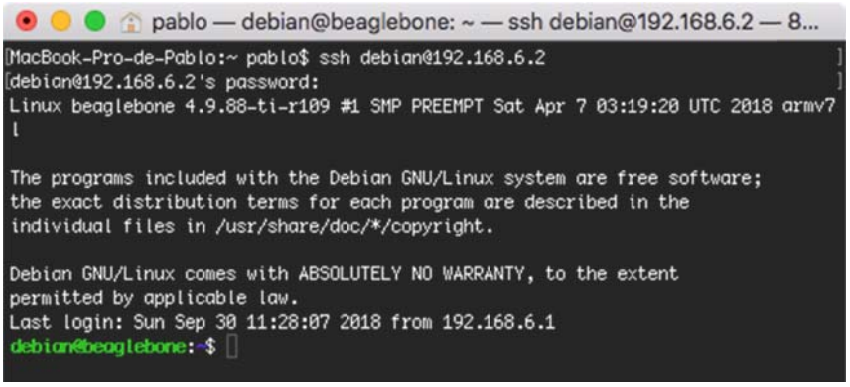


Imagen 2.3 – *Acceso remoto a placa BeagleBone*

Desde el navegador empleando una de las direcciones mostradas en la ***Imagen 2.2***, las cuáles cargan una nueva página donde se muestran dos IDEs para el desarrollo de código en la placa.

El primero es Node-RED, que permite realizar una programación basada en flujo empleando bloques ***(Imagen 2.4)***.

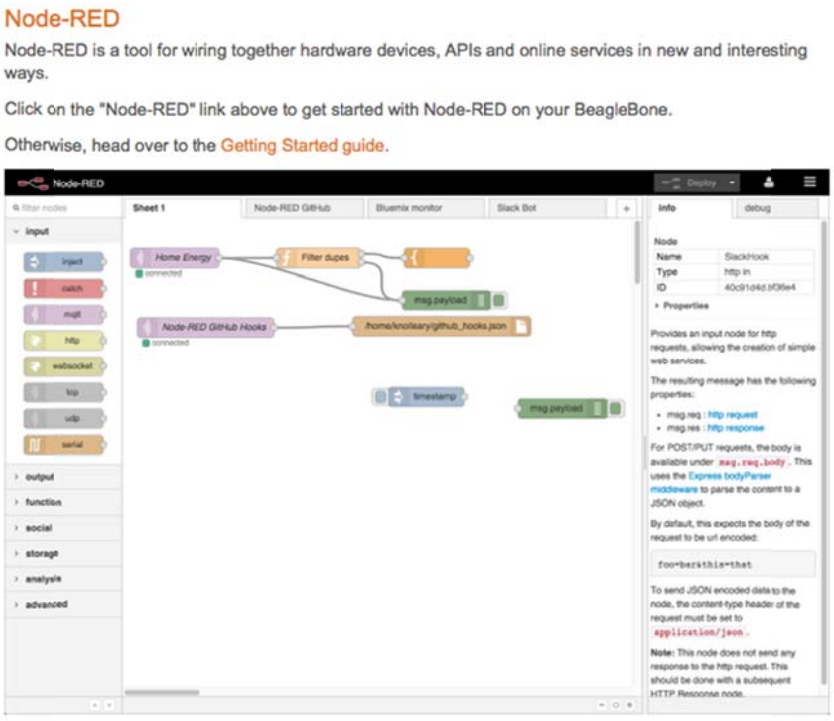


Imagen 2.4 – Interfaz de usuario de *Node-RED*

Por último, Cloud9 IDE, un editor de texto para el desarrollo de código ***(Imagen 2.5)***.

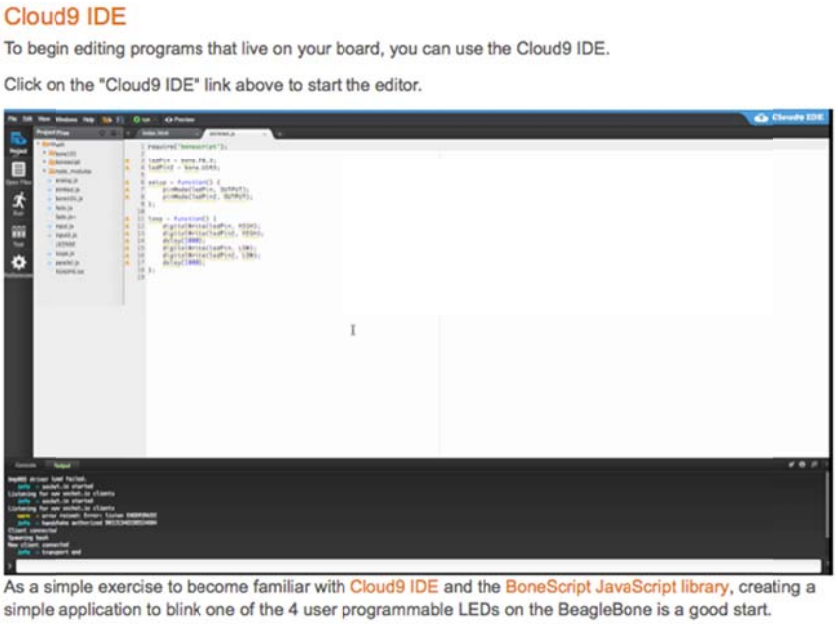


Imagen 2.5 *– Interfaz de usuario de Cloud9 IDE*

Para poder realizar la instalación de las distintas librerías Python y dependencias de Linux que harán falta para la correcta ejecución de código será necesario dotar a la placa de una conexión a Internet. Para ello la placa ya dispone de su propia tarjeta de red, así como dos antenas para amplificar el rango WiFi.

Comandos para la conexión a una red WiFi:

- sudo connmanctl
- enable wifi
- scan wifi
- services
- agent on
- connect wifi……
- (insertar la contraseña de la red wifi)
- quit

Comandos para la instalación de las dependencias necesarias:

- sudo apt-get update
- sudo apt-get install python
- sudo apt-get install python-virtualenv
- sudo apt-get install python-pip
- sudo apt-get install python-dev
- sudo apt-get install python-setuptools
- sudo apt-get install python-smbus
- sudo apt-get install build-essential
- sudo apt-get install libi2c-dev
- sudo apt-get install i2c-tools
- sudo apt-get install libffi-dev

Comandos para la instalación de las librerías necesarias para la correcta ejecución del código:

- sudo pip install cffi
- sudo pip install smbus-cffi
- sudo pip install Adafruit_BBIO

Anexo 3: Estructura de directorios y código para sensores H3LIS331DL

Como ya se ha dicho, se ha tratado de crear una estructura de proyecto que permita que el código de la clase principal del programa, H3LIS331DL.py, sea independiente del sensor que se esté empleando en cada momento, de forma que solo haya que modificar, en caso de que sea necesario, las clases o las partes de código que sean específicas de cada sensor.

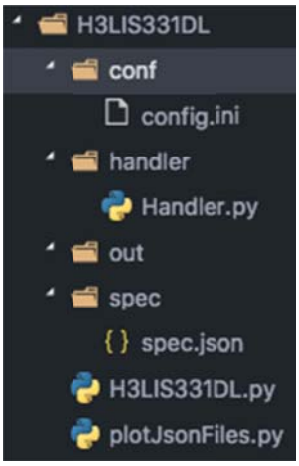


Imagen 3.1 *– Estructura de directorios para sensor H3LIS331DL*

- **H3LIS331DL:** es el directorio raíz que contiene todos los subdirectorios y los ficheros necesarios para la correcta ejecución de todo el código desarrollado.
- **conf:** en este directorio se encuentra un fichero de configuración "config.ini" que contiene la configuración específica para poder trabajar con los sensores H3LIS331DL.
- **handler:** este directorio contiene la clase "Handler.py", la cual se encarga de detectar los sensores que haya conectados en la placa, así como, cada vez que se produce un impacto, de realizar la captura de los datos y el posterior enventanado de los mismos, generando los ficheros de salida correspondientes a las mediciones realizadas por cada sensor.
- **out:** en este directorio se almacenarán los ficheros de salida que contendrán las medidas realizadas por cada sensor cada vez que se detecte un impacto, anidando las medidas y los instantes temporales un fichero JSON ***(Imagen 3.2)***. Cada acelerómetro dispondrá de su propio fichero.
- **spec:** este directorio contiene un fichero JSON "spec.json" en el que se listan todos los puertos GPIO activos donde pueden conectarse los distintos acelerómetros.
- **H3LIS331DL.py:** este fichero contiene el código principal para la ejecución del programa. Lee y procesa tanto el fichero de configuración como el fichero de especificaciones. El código de este fichero está permanentemente procesando los datos que generan los acelerómetros para identificar cuándo se produce un impacto, y así decidir cuándo realizar una captura y el posterior almacenamiento de los datos.
- **plotJsonFiles.py:** este fichero permite visualizar de manera gráfica los datos guardados en los ficheros de salida para determinar si el código implementado captura correctamente los impactos.

Como puede observarse en la ***Imagen 3.2*** cada muestra capturada por los acelerómetros va asociada a un instante temporal. Cada vez que placa recibe y procesa una medida, le asigna el instante temporal de su reloj interno expresado en segundos, en UTC y que se almacena en el fichero de salida correspondiente en el directorio "out" junto con la muestra. Siguiendo el mismo proceso con los sensores del cuerpo **Anexo 5** ***(Imagen 5.2)*** se consigue una base de tiempos común entre todos los acelerómetros.

***Imagen 3.2** – Ejemplo de JSON generado por acelerómetro*

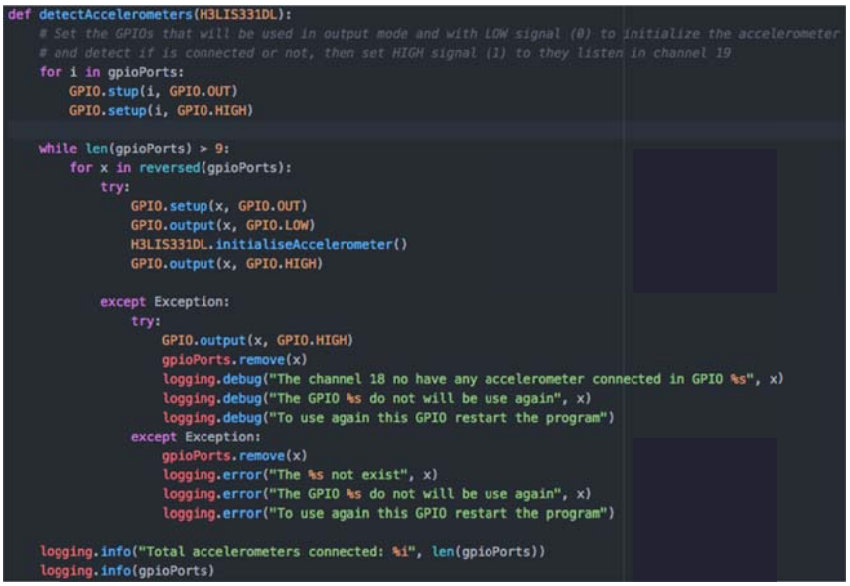
La función "readConf" mostrada en la ***Imagen 3.3*** se encarga de leer la configuración guardada en el fichero "config.ini". Primeramente, se indica la ruta donde se encuentra el fichero de configuración dentro del directorio de trabajo H3LIS331DL. A continuación, se indica la cabecera del fichero de configuración que se quiere leer, en este caso H3LIS331DL, y por último en un bucle se leen las distintas líneas del fichero y se van guardando en un diccionario. De esta manera solo es necesario acceder al fichero de configuración una única vez, durante la primera ejecución del programa, y después estos datos pueden usarse en cualquier momento accediendo al diccionario.

```
def readConf():
    # Read and load de configuration data in a dictionary
    config = ConfigParser.ConfigParser()
    config.read("./conf/config.ini")
    H3LIS331DL_options = config.options("H3LIS331DL")
    conf = dict()
    for op in H3LIS331DL_options:
        conf[op] = config.get("H3LIS331DL", op)
    logging.debug("[CONFIGURATION]: %s", conf)
    return conf
```

***Imagen 3.3** – H3LIS331DL.py, función readConf*

Cada vez que se ejecuta el código es necesario detectar los acelerómetros que hay conectados en la placa y determinar a qué puertos GPIO están conectados. Para ello se emplea la función detallada en la ***Imagen 3.4***. Con la librería para Python Adafruit_BBIO, se puede seleccionar la señal a enviar, HIGH o LOW, a cada puerto GPIO especificado en el fichero "spec.json". De esta

manera se pueden gestionar los turnos de escucha de los acelerómetros. Posteriormente se intenta configurar el acelerómetro conectado a dicho puerto GPIO sabiendo que, si la respuesta es una excepción, solo caben tres posibilidades; el acelerómetro del puerto esta estropeado, el cable de dicho puerto tiene una mala conexión, o no existe ningún acelerómetro conectado a dicho puerto. Independiente del factor que produzca la excepción, se estima que no será necesario volver a trabajar con ese puerto GPIO durante la sesión puesto que no será posible capturar ningún evento para dicho puerto.
Así se puede detectar a qué puertos GPIO están conectados los acelerómetros, y saber si están funcionando todos los esperados, de forma que se trabajará durante el resto de la ejecución únicamente con dichos puertos GPIO. Esto permite disminuir la carga computacional y mejorar los tiempos de acceso a cada uno de los distintos acelerómetros para la realización de las lecturas.



```
def detectAccelerometers(H3LIS331DL):
    # Set the GPIOs that will be used in output mode and with LOW signal (0) to initialize the accelerometer
    # and detect if is connected or not, then set HIGH signal (1) to they listen in channel 19
    for i in gpioPorts:
        GPIO.stup(i, GPIO.OUT)
        GPIO.setup(i, GPIO.HIGH)

    while len(gpioPorts) > 9:
        for x in reversed(gpioPorts):
            try:
                GPIO.setup(x, GPIO.OUT)
                GPIO.output(x, GPIO.LOW)
                H3LIS331DL.initialiseAccelerometer()
                GPIO.output(x, GPIO.HIGH)

            except Exception:
                try:
                    GPIO.output(x, GPIO.HIGH)
                    gpioPorts.remove(x)
                    logging.debug("The channel 18 no have any accelerometer connected in GPIO %s", x)
                    logging.debug("The GPIO %s do not will be use again", x)
                    logging.debug("To use again this GPIO restart the program")
                except Exception:
                    gpioPorts.remove(x)
                    logging.error("The %s not exist", x)
                    logging.error("The GPIO %s do not will be use again", x)
                    logging.error("To use again this GPIO restart the program")

    logging.info("Total accelerometers connected: %i", len(gpioPorts))
    logging.info(gpioPorts)
```

***Imagen 3.4** – H3LIS331DL.py función detectAccelerometers*

El fichero “config.ini” ***(Imagen 3.5)***, además de contener la configuración principal para el correcto funcionamiento de los sensores, también pueden ajustarse otros parámetros como:

- **SENSITIVITY:** para las pruebas realizadas, se estima que la magnitud mínima de fuerza/aceleración capturada por los acelerómetros para estimar que se ha producido un impacto debe ser ±1680 sobre ±32768, que es el rango máximo de lectura de los acelerómetros especificado por el fabricante.
- **DEVIBRATE:** ciclos de captura que los acelerómetros deben superar antes de poder realizar una nueva lectura tras detectar un impacto, para comprobar si las capturas siguen por encima del umbral de impacto. En dicho caso se reiniciará el bucle DEVIBRATE hasta que las capturas estén por debajo del umbral de impacto, que es cuando se podrá realizar una nueva captura sin interferencias de impactos pasados.
- **BUFFER_MAX_LENGTH:** especifica el tamaño máximo de los *buffers* donde se guardarán los datos que captura cada acelerómetro antes de volcarlos al correspondiente fichero JSON de salida.
- **ACCL_NUMBER:** señala el número de sensores involucrados en cada ejecución.

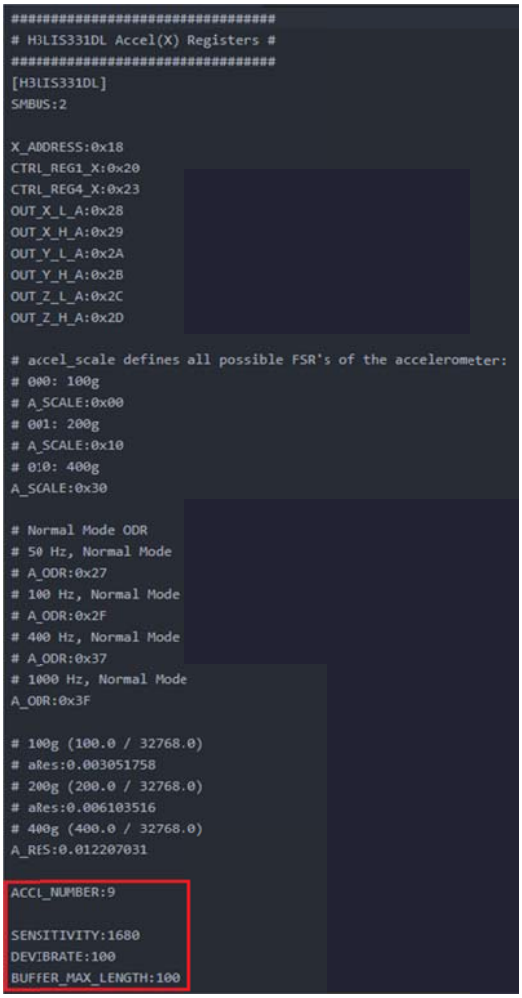


***Imagen 3.5** – config.ini, contenido del fichero de configuración*

La función "detectImpact" mostrada en la ***Imagen 3.6*** procesa la información entregada por los acelerómetros y calcula la fuerza/aceleración que los mismos capturan en cada instante de tiempo. Compara los datos capturados con la medida estimada de impacto y estudia, si para los datos recogidos, se ha producido o no un impacto, disparando el evento de impacto en caso afirmativo.

```python
def detectImpact(H3LIS331DL, sensor, sensitivity, vibration):
    Acclx, Accly, Acclz = H3LIS331DL.readAcclValues(sensor)

    magnitude = 0
    try:
        magnitude = math.sqrt(Acclx * Acclx + Accly * Accly + Acclz * Acclz)
    except Exception:
        logging.error("An error are ocurred during the magnitude operations")

    if magnitude >= sensitivity and vibration == 0:
        return True
    else:
        return False
```

***Imagen 3.6** – H3LIS331DL.py, función detectImpact*

Para realizar una lectura completa es necesario realizar capturas en los tres ejes x, y, z. Por ello, tal como se muestra en la **Imagen 3.7** se dispone de una función para cada uno de los ejes, "readAcclx, readAccly, readAcclz".

```python
# Read the accelerometer output registers.
# This will read all six accelerometer output registers.
# Reading the  Accelerometer X-Axis Values from the Register
def readAcclx(self):
    Accl_l = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_x_l_a"], 0))
    Accl_h = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_x_h_a"], 0))
    Accl_total = (Accl_l | Accl_h << 8)
    return Accl_total if Accl_total < 32768 else Accl_total - 65536

# Reading the  Accelerometer Y-Axis Values from the Register
def readAccly(self):
    Accl_l = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_y_l_a"], 0))
    Accl_h = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_y_h_a"], 0))
    Accl_total = (Accl_l | Accl_h << 8)
    return Accl_total if Accl_total < 32768 else Accl_total - 65536

# Reading the  Accelerometer Z-Axis Values from the Register
def readAcclz(self):
    Accl_l = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_z_l_a"], 0))
    Accl_h = self.bus.read_byte_data(int(self.conf["x_address"], 0),
                                     int(self.conf["out_z_h_a"], 0))
    Accl_total = (Accl_l | Accl_h << 8)
    return Accl_total if Accl_total < 32768 else Accl_total - 65536

def AcclDataTotal(self):
    atotal = (((self.readAcclx() ** 2) + (self.readAccly() ** 2) + (self.readAcclz() ** 2)) ** 0.5)
    return atotal

def readAcclValues(self, gpio):
    # Read our Accelerometer values
    Acclx = self.readAcclx()
    self.ring_buffer_x[gpio]["values"].append(Acclx)
    self.ring_buffer_x[gpio]["times"].append(time.time())
    Accly = self.readAccly()
    self.ring_buffer_y[gpio]["values"].append(Accly)
    self.ring_buffer_y[gpio]["times"].append(time.time())
    Acclz = self.readAcclz()
    self.ring_buffer_z[gpio]["values"].append(Acclz)
    self.ring_buffer_z[gpio]["times"].append(time.time())

    return Acclx, Accly, Acclz
```

Imagen 3.7 – *Handler.py, capturando datos de los acelerómetros*

En la ***Imagen 3.8*** pueden observarse las capturas que ha realizado un acelerómetro para varios impactos. En el eje x puede apreciarse claramente un pico de impacto, en el eje y dos y en el eje z, aunque no muy claramente, un único pico.

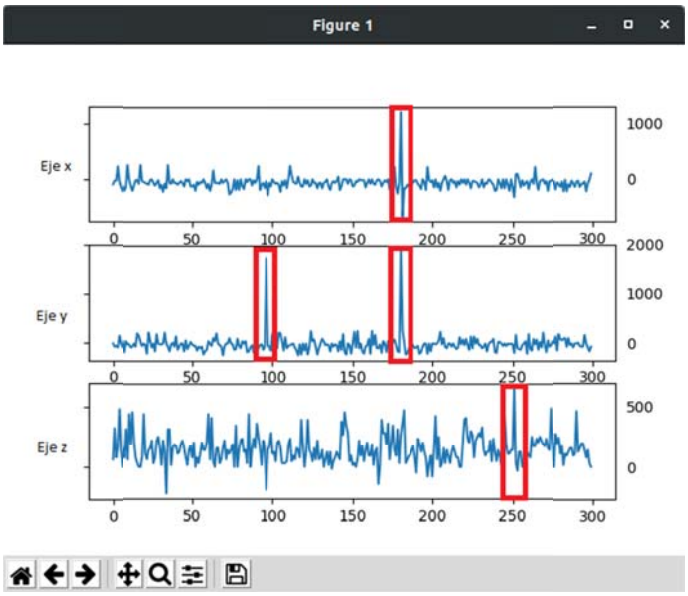


Imagen 3.8 – Ejemplo gráfico de captura de impacto

Anexo 4: Configuración y comunicación con *gateway* iS01

Primeramente, se debe conectar el módulo BLE-WiFi Gateway iS01 a nuestro PC o a otra fuente de alimentación. De este modo, el módulo generará de manera automática un nuevo punto de acceso WiFi "BLE_WiFi_7E_B9" ***(Imagen 4.1)*** al que será posible conectarse. Cuando se establezca la conexión se debe introducir la contraseña para dicha WiFi, la cual será "12345678".

Imagen 4.1 – *punto de acceso generado por módulo BLE-WiFi gateway iS01*

Este módulo puede funcionar como servidor TCP, cliente TCP, cliente HTTP o cliente MQTT. Para el desarrollo del proyecto se usará como servidor TCP, empleando un *socket* en la dirección 192.168.10.1:8080 para capturar los datos que los sensores envían al módulo BLE-WiFi Gateway iS01. En la ***Imagen 4.2*** se muestra un esquema de la comunicación entre los sensores, el gateway y el PC.

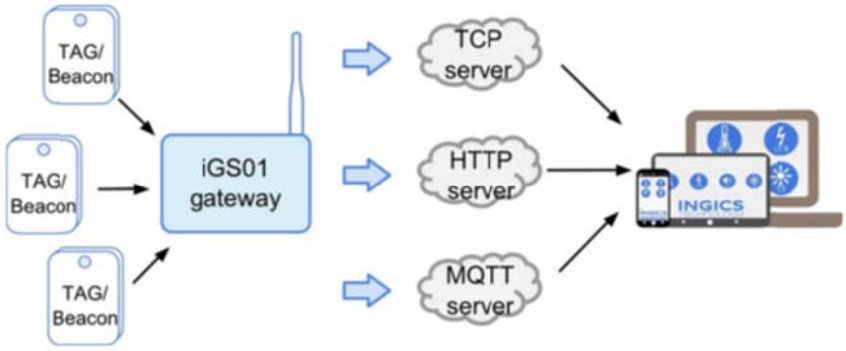


Imagen 4.2 – *Ejemplo comunicación sensores -> PC*

Accediendo a la dirección 192.168.10.1 mediante un navegador, se podrá configurar el módulo WiFi para que se trabaje en cualquiera de los anteriores modos citados: servidor TCP, cliente TCP, cliente HTTP o cliente MQTT. El usuario/contraseña que se debe introducir en la página para configurar el módulo será admin/admin.

Tras ello, los acelerómetros comenzarán a enviar tramas al Gateway con el formato que se muestra en la ***Imagen 4.3***, en donde los últimos 18 *bytes* corresponden a tres capturas diferentes de los acelerómetros sobre los ejes x, y, z.

```
$GPRP,EAC653D3AA8D,CB412F0C8EDC,-57,02010619FF590081BC4B01F5FFFEFFE800F4FFFCFFE700F5FFFBFFE800
```

Imagen 4.3 *- Ejemplo de trama*

Anexo 5: Estructura de directorios y código para sensores BLE_IBS01RG

Al igual que en el Anexo 3, se ha tratado de crear una estructura de proyecto que permita que el código de la clase principal del programa, BLE_Client.py, sea independiente del sensor que se esté empleando en cada momento, de forma que solo haya que cambiar, en caso de que sea necesario, las clases o las partes de código que sean específicas de cada sensor.

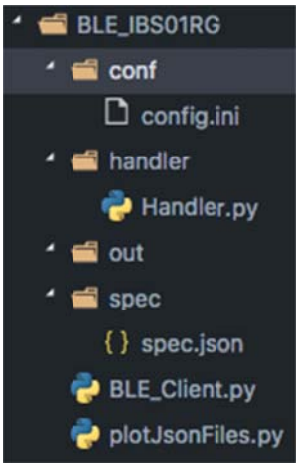


Imagen 5.1 – *Estructura de directorios para sensores IBS01RG*

- **BLE_IB01RG:** es el directorio raíz que contiene todos los subdirectorios y ficheros necesarios para la correcta ejecución de todo el código desarrollado.
- **conf:** en este directorio se encuentra un fichero de configuración "config.ini" que contiene la configuración específica para trabajar con los sensores IBS01RG.
- **handler:** este directorio contiene la clase "Handler.py" que detecta los sensores encendidos al inicio de la ejecución del programa, y que, cada vez que se produce un impacto, realiza la captura de los datos y el posterior enventanado de los mismos, generando los ficheros de salida con las mediciones realizadas por cada sensor.
- **out:** en este directorio se almacenarán los ficheros de salida que contendrán las medidas realizadas por cada sensor cada vez que se detecte un impacto, anidando las medidas y los instantes temporales un fichero JSON ***(Imagen 5.2)***. Cada acelerómetro dispondrá de su propio fichero.
- **spec:** este directorio contiene un fichero JSON "spec.json" en el que se listan los identificadores de los acelerómetros de que disponemos. Así se puede saber a qué acelerómetro pertenece cada una de las tramas que recibe el gateway, puesto que cada trama viene acompañada del identificador del acelerómetro que la genera.
- **BLE_Client.py:** este fichero contiene el código principal para la ejecución del programa. Lee y procesa tanto el fichero de configuración como el fichero de especificaciones. El código de este fichero procesa continuamente los datos que generan los acelerómetros para identificar cuando se produce un impacto, y así decidir cuándo realizar una captura y el posterior almacenamiento de los datos.
- **plotJsonFiles.py:** este fichero permite visualizar de manera gráfica los datos guardados en los ficheros de salida para poder determinar si el código implementado funciona correctamente.

En la ***Imagen 5.2*** puede observarse que cada muestra capturada por los acelerómetros tiene asociada una medida temporal. Esta medida corresponde al instante en el que la placa procesa la trama, referida a su reloj interno expresado en segundos, en UTC. Siguiendo el mismo proceso que en el Anexo 3 ***(Imagen 3.2)*** se consigue una base de tiempos común a todos los acelerómetros.

```
{
  "axis_x": {
    "times": [
      1535799186.307107,
      1535799186.35372
    ],
    "values": [
      -112,
      -192
    ]
  },
  "axis_y": {
    "times": [
      1535799186.307107,
      1535799186.35372
    ],
    "values": [
      208,
      -224
    ]
  },
  "axis_z": {
    "times": [
      1535799186.307107,
      1535799186.35372
    ],
    "values": [
      256,
      80
    ]
  }
}
```


Imagen 5.2 – Ejemplo de JSON generado por acelerómetro

La función "readConf" mostrada en la ***Imagen 5.3*** lee la configuración guardada en el fichero "config.ini". Primeramente, se indica la ruta donde se encuentra el fichero de configuración dentro del directorio de trabajo BLE_IBS01RG. A continuación, se indica la cabecera del fichero de configuración que se quiere leer, en este caso IBS01RG, y por último en un bucle se leen las distintas líneas del fichero y se van guardando en un diccionario. De esta manera solo es necesario acceder al fichero de configuración una única vez, durante la primera ejecución del programa, y después sus datos pueden usarse en cualquier momento accediendo al diccionario. Puede observarse que esta función es exactamente la misma que la del **Anexo 3 *(Imagen 3.3)***.

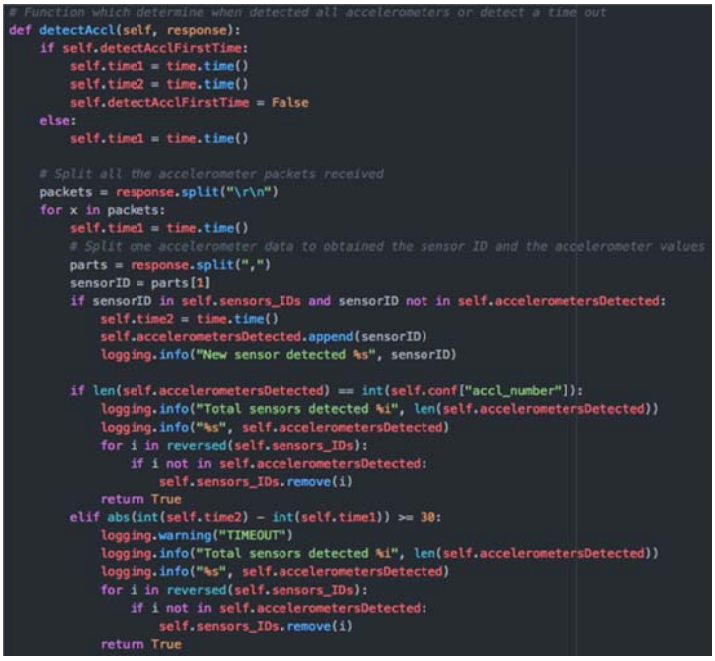
```python
def readConf():
    # Read and load de configuration data in a dictionary
    config = ConfigParser.ConfigParser()
    config.read("./conf/config.ini")
    H3LIS331DL_options = config.options("IBS01RG")
    conf = dict()
    for op in H3LIS331DL_options:
        conf[op] = config.get("IBS01RG", op)
    logging.debug("[CONFIGURATION]: %s", conf)
    return conf
```

Imagen 5.3 – *BLE_Client.py, función readConf*

Cada vez que se ejecuta el código es necesario detectar los acelerómetros que están encendidos y sus respectivos identificadores. Para ello se emplea la función "detectAccl"

detallada en la ***Imagen 5.4***. Al comienzo de la ejecución del código se capturan las tramas recibidas en el gateway y se obtienen los identificadores asociados a las mismas. Estos identificadores se comparan con los especificados en el fichero "spec.json", para saber si las tramas recibidas se corresponden con los acelerómetros disponibles.

Para evitar posibles bucles infinitos en la ejecución de esta función se ha preestablecido un tiempo de 30 segundos, tras el cual, si no se recibe ninguna trama con algún identificador especificado en el fichero "spec.json" finaliza la función.



```
# Function which determine when detected all accelerometers or detect a time out
def detectAccl(self, response):
    if self.detectAcclFirstTime:
        self.time1 = time.time()
        self.time2 = time.time()
        self.detectAcclFirstTime = False
    else:
        self.time1 = time.time()

    # Split all the accelerometer packets received
    packets = response.split("\r\n")
    for x in packets:
        self.time1 = time.time()
        # Split one accelerometer data to obtained the sensor ID and the accelerometer values
        parts = response.split(",")
        sensorID = parts[1]
        if sensorID in self.sensors_IDs and sensorID not in self.accelerometersDetected:
            self.time2 = time.time()
            self.accelerometersDetected.append(sensorID)
            logging.info("New sensor detected %s", sensorID)

        if len(self.accelerometersDetected) == int(self.conf["accl_number"]):
            logging.info("Total sensors detected %i", len(self.accelerometersDetected))
            logging.info("%s", self.accelerometersDetected)
            for i in reversed(self.sensors_IDs):
                if i not in self.accelerometersDetected:
                    self.sensors_IDs.remove(i)
            return True
        elif abs(int(self.time2) - int(self.time1)) >= 30:
            logging.warning("TIMEOUT")
            logging.info("Total sensors detected %i", len(self.accelerometersDetected))
            logging.info("%s", self.accelerometersDetected)
            for i in reversed(self.sensors_IDs):
                if i not in self.accelerometersDetected:
                    self.sensors_IDs.remove(i)
            return True
```

Imagen 5.4 – *Handler.py, función detectAccl*

El fichero "config.ini" ***(Imagen 5.5)***, además de contener la configuración principal para el correcto funcionamiento del programa, también permite ajustar otros parámetros como:

- **SENSITIVITY:** para las pruebas realizadas, se estima que la magnitud mínima de fuerza/aceleración capturada por los acelerómetros para estimar que se ha producido un impacto debe ser ±1680 sobre ±32768, que es el rango máximo de lectura de los acelerómetros especificado por el fabricante.
- **DEVIBRATE:** ciclos de captura que los acelerómetros deben superar antes de poder realizar una nueva lectura tras detectar un impacto para comprobar si las capturas que realizan siguen estando por encima del umbral de impacto. En dicho caso se reiniciará el bucle DEVIBRATE hasta que las capturas estén por debajo del umbral de impacto, que es cuando se podrá realizar una nueva captura sin interferencias de impactos pasados.
- **BUFFER_MAX_LENGTH:** especifica el tamaño máximo de los *buffers* donde se guardarán los datos capturados por cada acelerómetro antes de volcarlos al correspondiente fichero JSON de salida.
- **ACCL_NUMBER:** señala el número de sensores que habrá involucrados en cada ejecución.

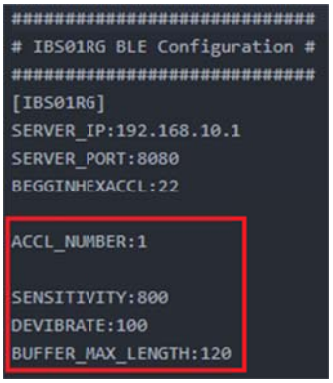


Imagen 5.5 *– Config.ini, contenido del fichero de configuración*

La función "detectImpact" mostrada en la ***Imagen 5.6*** procesa la información entregada por los acelerómetros y calcula la fuerza/aceleración que los mismos capturan en cada instante de tiempo. Compara los datos capturados con la medida estimada de impacto y decide, si para los datos recogidos se ha producido o no un impacto, disparando el evento de impacto en caso afirmativo.

```
def detectImpact(BLE, sensorID, data, sensitivity, vibration):
    xDecAccl1, yDecAccl1, zDecAccl1, xDecAccl2, yDecAccl2, zDecAccl2, \
        xDecAccl3, yDecAccl3, zDecAccl3 = BLE.obtainAccelerometerValues(sensorID, data)

    magnitude1 = 0
    magnitude2 = 0
    magnitude3 = 0
    try:
        magnitude1 = math.sqrt(xDecAccl1 * xDecAccl1 + yDecAccl1 * yDecAccl1 + zDecAccl1 * zDecAccl1)
        magnitude2 = math.sqrt(xDecAccl2 * xDecAccl2 + yDecAccl2 * yDecAccl2 + zDecAccl2 * zDecAccl2)
        magnitude3 = math.sqrt(xDecAccl3 * xDecAccl3 + yDecAccl3 * yDecAccl3 + zDecAccl3 * zDecAccl3)
    except Exception:
        logging.error("An error are ocurred during the magnitude operations")

    if magnitude1 >= sensitivity or magnitude2 >= sensitivity or magnitude3 >= sensitivity and vibration == 0:
        return True
    else:
        return False
```

Imagen 5.6 – *BLE_Client.py, función detectImpact*

Al recoger la información que entregan los acelerómetros es necesario tener en cuenta que devuelven en cada trama, ***(Anexo 4, Imagen 4.3)***, tres capturas, en total 18 *bytes*, los últimos de cada trama. La función "getAcclData" detallada en la ***Imagen 5.7***, procesa las tramas capturadas y separa los últimos 18 *bytes* de cada trama.

```
def getAcclData(self, response):
    toretID = []
    toretData = []

    # Split all the accelerometer packets received
    packets = response.split("\r\n")
    for x in packets:
        # Split one accelerometer data to obtained the sensor ID and the accelerometer values
        parts = response.split(",")
        sensorID = parts[1]
        if sensorID in self.sensors_IDs:
            # Get the accelerometer raw data string
            rawData = parts[len(parts) - 1]
            acclData = rawData[int(self.conf["begginhexaccl"]):]
            toretID.append(sensorID)
            toretData.append(acclData)

    return toretID, toretData
```

Imagen 5.7 – *Handler.py, función getAcclData*

Una vez separados los últimos 18 *bytes* de cada trama del resto, es necesario separarlos en grupos de 6, ya que cada uno de dichos grupos corresponderá a las tres muestras entregadas por el acelerómetro. Por último, para obtener los valores de cada eje, es necesario volver a

dividir los grupos de 6 *bytes* en grupos de 2. Cada grupo de 2 *bytes* corresponde a uno de los ejes x, y, z ***(Imagen 5.8)***.

```
def obtainAccelerometerValues(self, sensorID, data):
    # Obtain the 3 accelerometer samples
    accl1 = data[0:12]
    xHexAccl1 = accl1[2:4] + accl1[0:2]
    yHexAccl1 = accl1[6:8] + accl1[4:6]
    zHexAccl1 = accl1[10:12] + accl1[8:10]

    accl2 = data[12:24]
    xHexAccl2 = accl2[2:4] + accl2[0:2]
    yHexAccl2 = accl2[6:8] + accl2[4:6]
    zHexAccl2 = accl2[10:12] + accl2[8:10]

    accl3 = data[24:36]
    xHexAccl3 = accl3[2:4] + accl3[0:2]
    yHexAccl3 = accl3[6:8] + accl3[4:6]
    zHexAccl3 = accl3[10:12] + accl3[8:10]

    # Convert Hexadecimal values to decimal values
    xDecAccl1 = self.Hex2Dec(xHexAccl1)
    yDecAccl1 = self.Hex2Dec(yHexAccl1)
    zDecAccl1 = self.Hex2Dec(zHexAccl1)

    xDecAccl2 = self.Hex2Dec(xHexAccl2)
    yDecAccl2 = self.Hex2Dec(yHexAccl2)
    zDecAccl2 = self.Hex2Dec(zHexAccl2)

    xDecAccl3 = self.Hex2Dec(xHexAccl3)
    yDecAccl3 = self.Hex2Dec(yHexAccl3)
    zDecAccl3 = self.Hex2Dec(zHexAccl3)

    # Add decimal values to the ring buffers
    self.ring_buffer_x[sensorID]["values"].append(xDecAccl1)
    self.ring_buffer_x[sensorID]["times"].append(time.time())
    self.ring_buffer_y[sensorID]["values"].append(yDecAccl1)
    self.ring_buffer_y[sensorID]["times"].append(time.time())
    self.ring_buffer_z[sensorID]["values"].append(zDecAccl1)
    self.ring_buffer_z[sensorID]["times"].append(time.time())

    self.ring_buffer_x[sensorID]["values"].append(xDecAccl2)
    self.ring_buffer_x[sensorID]["times"].append(time.time())
    self.ring_buffer_y[sensorID]["values"].append(yDecAccl2)
    self.ring_buffer_y[sensorID]["times"].append(time.time())
    self.ring_buffer_z[sensorID]["values"].append(zDecAccl2)
    self.ring_buffer_z[sensorID]["times"].append(time.time())

    self.ring_buffer_x[sensorID]["values"].append(xDecAccl3)
    self.ring_buffer_x[sensorID]["times"].append(time.time())
    self.ring_buffer_y[sensorID]["values"].append(yDecAccl3)
    self.ring_buffer_y[sensorID]["times"].append(time.time())
    self.ring_buffer_z[sensorID]["values"].append(zDecAccl3)
    self.ring_buffer_z[sensorID]["times"].append(time.time())

    return xDecAccl1, yDecAccl1, zDecAccl1, \
        xDecAccl2, yDecAccl2, zDecAccl2, \
        xDecAccl3, yDecAccl3, zDecAccl3
```

Imagen 5.8 – *Handler.py, función obtainAccelerometerValues*

El último paso para obtener los valores del acelerómetro lo realiza la función "Hex2Dec" detallada en la ***Imagen 5.9***. Esta función recibe como parámetro los datos obtenidos por los acelerómetros para cada uno de los ejes x, y, z y para cada una de las 3 muestras que se reciben en cada trama y realiza la conversión de los mismos de hexadecimal a decimal.

```
# Function to convert hexadecimal values to decimal values
def Hex2Dec(self, hex):
    try:
        i = eval("0x" + hex)
        if i >= 2**15:
            result = i - 2**16
        else:
            result = i

        return result
    except Exception:
        logging.error("[BLE_HANDLER_HEX2DEC] HEX_VALUE: %s", hex)
        logging.error("An error ocurred during the hexadecimal to decimal conversion")
```

Imagen 5.9 – *Handler.py, función Hex2Dec*

Bibliografía

- **SeeedStudio Beaglebone Green Wireless**

https://beagleboard.org/getting-started
https://beagleboard.org/latest-images
https://elinux.org/Beagleboard:BeagleBoneBlack_Debian#Stretch_Snapshot_iot
http://wiki.seeedstudio.com/BeagleBone_Green_Wireless/
https://etcher.io
https://www.digikey.com/en/maker/blogs/2017/how-to-setup-wifi-on-the-beaglebone-black-wireless

- **H3LIS33DL**

https://github.com/ControlEverythingCommunity/H3LIS331DL
https://www.st.com/resource/en/datasheet/h3lis331dl.pdf

- **Python**

https://pypi.org/project/smbus-cffi/
https://docs.python.org/2/library/collections.html
https://docs.python.org/2/library/configparser.html
https://docs.python.org/2/library/logging.html
https://docs.python.org/2/howto/logging.html
https://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/installation-on-ubuntu
https://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/using-the-bbio-library
https://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/gpio

- **BLE-WIFI Gateway iGS01**

https://www.ingics.com/doc/iGS01/BLE_WiFi_Gateway_iGS01_Specification.pdf
https://www.ingics.com/doc/iGS01/AP001_iGS01_quick_start_guide.pdf
https://www.ingics.com/doc/iGS01/BLE_WiFi_Gateway_iGS01_User_Guide.pdf
https://www.ingics.com/doc/iGS01/AP005_iGS01_verify_http.pdf
https://www.ingics.com/doc/iGS01/AP009_iGS01_connect_to_AWS_IoT.pdf
https://www.ingics.com/doc/iGS01/AP004_iGS01_telnet_command.pdf
https://www.ingics.com/doc/iGS01/AP007_iGS01_payload_filter.pdf

- **Sensor Beacon iBS01G**

https://www.ingics.com/doc/iBS01/iBS01_briefing.pdf
https://www.ingics.com/doc/iBS01/Sensor_Beacon_iBS01_Specification.pdf
https://www.ingics.com/doc/iBS01/Sensor_Beacon_iBS01_User_Guide.pdf
https://www.ingics.com/doc/iBS01/iBS01_payload_format.pdf
https://play.google.com/store/apps/details?id=com.ingics.tag.igstagconfig&utm_source=global_co&utm_medium=prtnr&utm_content=Mar2515&utm_campaign=PartBadge&pcampaignid=MKT-Other-global-all-co-prtnr-py-PartBadge-Mar2515-1

- **Estado del arte**

[1]. Autor: Pinar Arpiñar-Avsar y Abdullah Ruhi Soylu.
Título: "Consistencia en los patrones de aceleración de jugadores de fútbol con diferentes niveles de habilidad".
https://g-se.com/consistencia-en-los-patrones-de-aceleracion-de-jugadores-de-futbol-con-diferentes-niveles-de-habilidad-1287-sa-F57cfb271e3989

[2]. Autor: José Andrés Sánchez y Ramón Marañón López.
Título: “Sistema de registro de parámetros cinemáticos intracíclicos en el nado”.
https://www.researchgate.net/profile/Jose_Sanchez24/publication/292983622_Sistema_de_registro_de_parametros_cinematicos_intraciclicos_en_el_nado_el_acelerometro/links/56b3b17108ae1f8aa45352b5/Sistema-de-registro-de-parametros-cinematicos-intraciclicos-en-el-nado-el-acelerometro.pdf

[3]. Autor: Vicente Romo Pérez, Juan Carlos Burguillo Rial, Eduardo Rodríguez Fernández y Javier García Nuñez.
Título: “Gestión de la información a través de un sistema de razonamiento basado en casos con acelerómetro como instrumento de medida”.
http://reined.webs.uvigo.es/index.php/reined/article/view/51/41