UNIVERSITÀ DEGLI STUDI
DI GENOVA

DIPARTIMENTO DI INGEGNERIA NAVALE, ELETTRICA, ELETTRONICA E DELLE TELECOMUNICAZIONI

CORSO DI STUDIO IN INGEGNERIA ELETTRONICA E TECNOLOGIE DELL'INFORMAZIONE

Tesi di Laurea Triennale

Ottobre 2019

Progetto e realizzazione di un sistema embedded per la valutazione di urti

(Design and development of an embedded device for impact assessment)

Candidati: Matteo Fresta e Matteo Pastorino Ghezzi

Relatore: Prof. Riccardo Berta

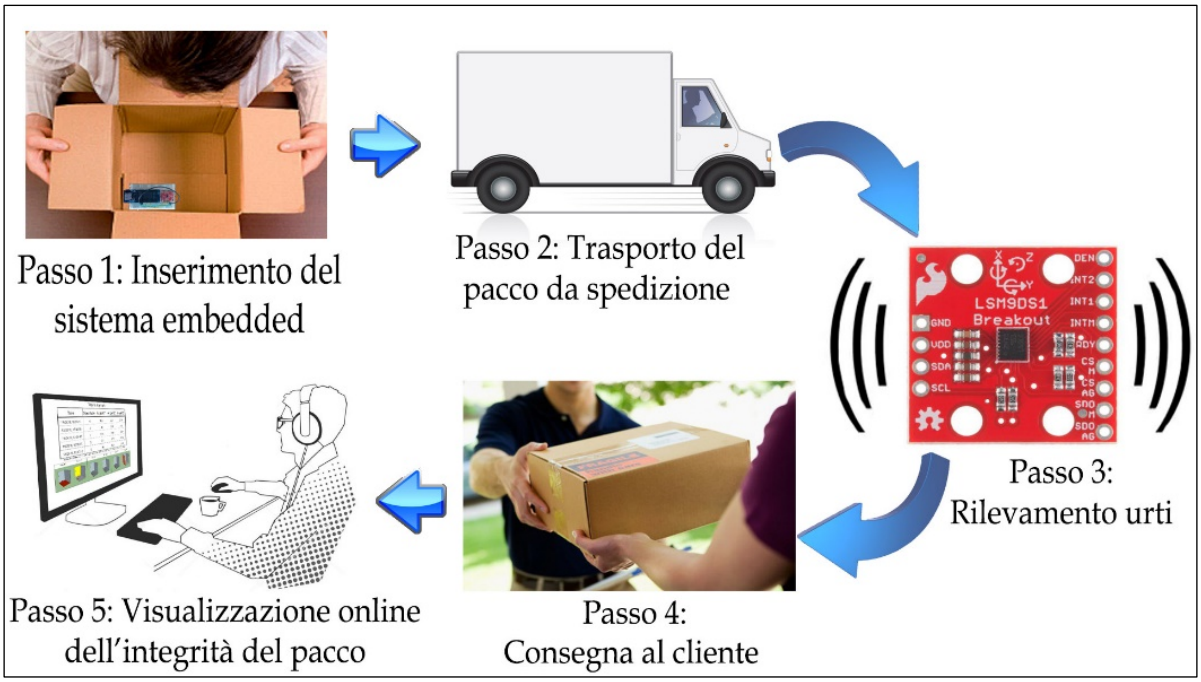
Sommario

Il presente elaborato è stato redatto con l'obiettivo di illustrare la realizzazione di un dispositivo per l'acquisizione remota di dati relativi alla sicurezza di un oggetto.
Il software acquisisce ed elabora i dati relativi agli urti che l'oggetto d'interesse subisce e successivamente le informazioni relative allo stato del prodotto vengono inviate ad una API (*Application Programming Interface*) chiamata *Atmophere* [1] nel caso il dispositivo sia connesso ad Internet, oppure salvate in locale con l'ausilio di una scheda microSD.
Infine, per consentire una rappresentazione dei dati *user-friendly*, è stata creata un'interfaccia web che consente di visualizzare i dati tramite una tabella e delle immagini, le quali indicano l'intensità degli urti sulle singole facce di un'ipotetica scatola.

Il progetto del seguente sistema embedded riunisce due realtà estremamente attuali: da una parte la sicurezza nonché verifica da parte dell'utente della corretta e adeguata spedizione di un articolo comprato su un portale di *e-commerce*, settore in continua espansione e sempre più rilevante a livello economico anche in Italia ([2],[3]), dall'altra la diffusione esponenziale di oggetti fisici della nostra vita quotidiana estesi alle potenzialità del web ed interconnessi tra loro (si stima che entro il 2020 i dispositivi connessi ad Internet saranno tra le 40-50 miliardi di unità, raddoppiando il numero del 2010).

1. Introduzione

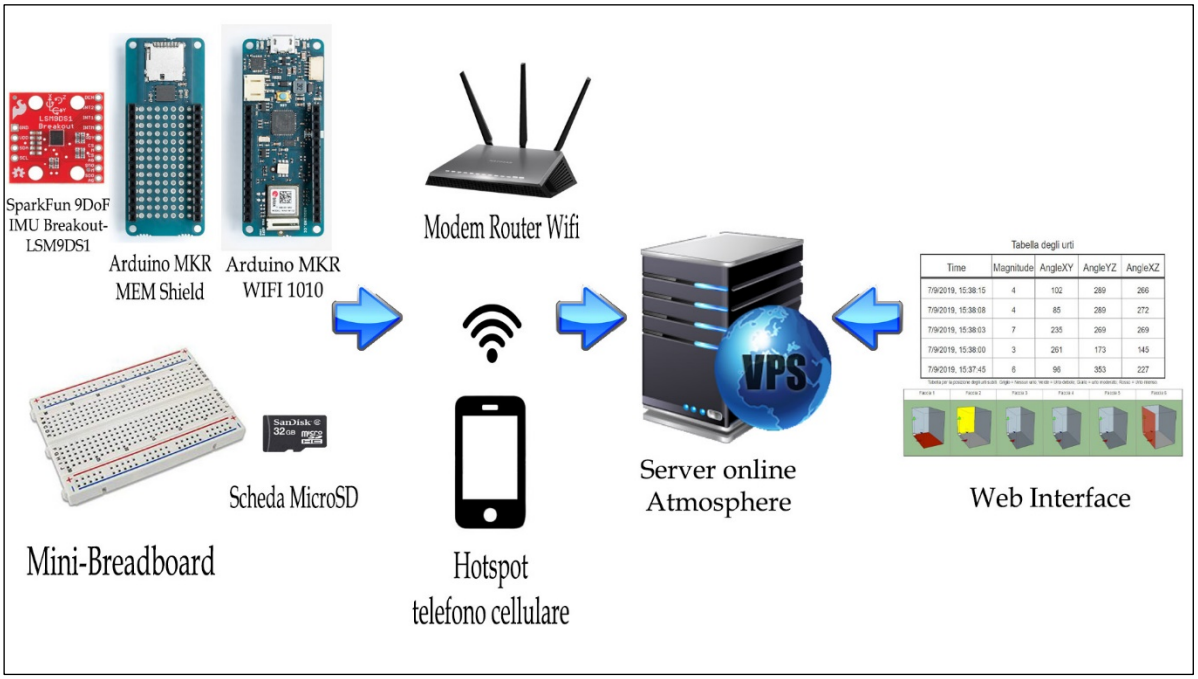
La presente tesi ha come oggetto il progetto e la realizzazione di un sistema embedded per la valutazione di urti, utilizzando strumenti di facile sviluppo ed economicamente accessibili quali Arduino come microcontrollore e storage di informazioni ed un sensore IMU; quest'ultimo permette di avere una puntuale conoscenza sul movimento di un oggetto.
Questo progetto può trovare applicazione nei settori delle spedizioni, della logistica e dei trasporti settori che, in un contesto di forte globalizzazione, costituiscono gli elementi chiave per la crescita economica di un paese.



Nella figura in alto, è presente una possibile situazione di utilizzo di questo sistema embedded.
Come mostrato al Passo 1, il nostro sistema verrà inserito in una posizione specifica della scatola da spedizione tenendo conto dell'orientazione del sensore IMU che servirà per l'analisi dei dati successiva: in figura è rappresentato il circuito su mini-breadboard, ma un utilizzo più pratico e robusto potrebbe essere una scheda dedicata con il sensore incorporato e di dimensioni molto più contenute, per non ridurre lo spazio all'interno della scatola a spese dell'oggetto da spedire.
Una volta inserito il sensore, il pacco verrà sigillato e spedito verso la destinazione (Passo 2); durante il tragitto il sensore IMU sarà completamente operativo e analizzerà gli eventuali urti durante lo spostamento (Passo 3). Durante questo percorso i dati da notificare verranno salvati su una scheda SD in attesa di essere inviati al server di riferimento.
Una volta giunto a destinazione il pacco viene consegnato al cliente (Passo 4) e il microcontrollore potrà collegarsi ad Internet e caricare i dati online.
L'utente finale (Passo 5) potrà infine accedere alla interfaccia web e visualizzare i dati notificati dal sistema embedded.
Tramite l'interfaccia web è possibile consultare la data esatta di ogni urto e l'intensità di ognuno di essi. Inoltre, si potrà anche avere una stima delle condizioni della scatola.

Questa innovazione potrebbe essere usata come garanzia a favore sia dei clienti che dei venditori: per esempio quando verranno segnalati urti eccessivi durante il trasporto e l'oggetto trasportato risulta danneggiato o non funzionante, l'utente potrà avvalersi di questi dati per la resa dell'oggetto o di un eventuale rimborso.

Le componenti principali del sistema sono brevemente presentate nella seguente figura e verranno analizzate più specificatamente nei paragrafi successivi.



La parte hardware del sistema embedded è formata da 3 moduli operanti più una scheda microSD, una mini-breadboard e dei fili di collegamento.
I moduli sono un sensore IMU (SparkFun 9DoF IMU Breakout - LSM9DS1) e due moduli della famiglia Arduino: un microcontrollore MKR WIFI 1010 e un supporto per schede microSD, l'MKR MEM Shield.
Per le fasi più importanti del funzionamento è necessaria la connessione Internet effettuabile tramite Modem Router Wifi o Hotspot da telefono cellulare.
Il server online utilizzato dal nostro progetto è Atmosphere [1] con il quale la parte hardware, dopo aver effettuato un login, condivide i dati.
Infine, è stata creata una Web Interface connessa ad Atmosphere da cui il cliente potrà in modo molto semplice, visualizzare i dati di interesse.

La tesi è articolata in tre paragrafi:

- nel primo vengono elencati e discussi i mezzi per la realizzazione del dispositivo, includendo, inoltre, considerazioni sugli approcci utilizzati per affrontare il problema;
- nel secondo si passa alla valutazione dei risultati ottenuti dopo aver testato il sistema in diverse condizioni di lavoro;
- nel terzo ed ultimo sono esposte alcune considerazioni finali e commenti.

2. Metodi e strumenti utilizzati

Le componenti principali del progetto sono:

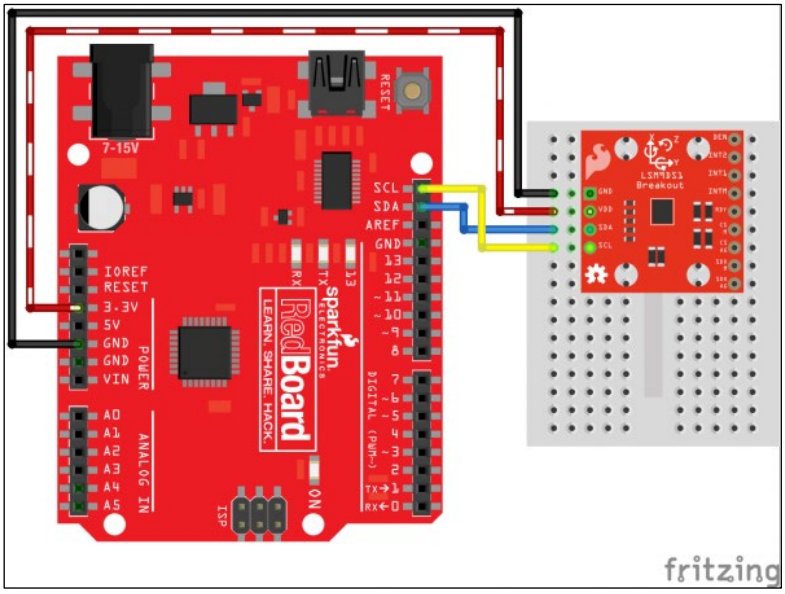
1. un microcontrollore (Arduino MKR WIFI 1010)
2. un sensore IMU (SparkFun 9DoF IMU Breakout - LSM9DS1)
3. un supporto per la scheda microSD (Arduino MKR MEM Shield). Queste tre componenti poi sono state montante su una *mini-breadboard* utilizzando dei semplici connettori.

Vediamo in dettaglio le singole parti partendo dal modulo Arduino, che ben si adatta alle nostre esigenze: la scheda infatti, oltre a presentare le caratteristiche tipiche di un Arduino, quali elevato numero di connettori I/O, un IDE semplice, un costo accessibile, ecc.., include un modulo WiFi, l'ESP32, che, grazie alla sua flessibilità e al suo consumo energetico ridotto, consente una più veloce programmazione di applicazioni basate sul IoT.
L'MKR WIFI 1010 presenta una porta USB che consente di alimentare il modulo a 5 V; il circuito però funziona a 3.3 V, differenziandosi per questo aspetto dagli altri microcontrollori della famiglia Arduino. Perciò la tensione massimo tollerato dai connettori I/O è 3.3 V (questo aspetto sarà importante durante lo studio del sensore).
Per programmare la scheda, Arduino fornisce un ambiente di sviluppo libero, comune a tutte le *board*, funzionante sia online che offline (in questo progetto si è scelto di utilizzare la versione desktop).
Essendo dunque il *software* comune a tutte le schede, è necessario installare il pacchetto specifico per il nostro Arduino.
Dopo l'installazione, il modulo è pronto per l'uso.

Il funzionamento dell'MKR MEM Shield è molto semplice: in generale, qualora non bastasse la memoria flash di 256 KB dell'MKR WIFI 1010, aggiunge un ulteriore spazio di 2 MB e, grazie allo slot per schede microSD, è possibile aggiungere diversi GB di memoria (nel caso in esame viene utilizzata una microSD da 1GB, spazio più che sufficiente per questo progetto).
Il sistema di comunicazione tra lo *shield* e la *board* è l'SPI. Grazie alla piena compatibilità con la famiglia MKR, il circuito dello *shield* opera a 3.3 V, ovvero la stessa tensione del 1010: questo dettaglio è importante poiché non si rende necessario l'utilizzo di un *level shifter*, la cui assenza non complica la realizzazione circuitale del sistema.

Di seguito viene descritto il chip responsabile dell'individuazione degli urti: l'IMU LSM9DS1 ospita un accelerometro a 3 assi, un giroscopio a 3 assi e un magnetometro a 3 assi per un totale di 9 gradi di libertà su un unico circuito integrato (IC, *integrated circuit*); le loro unità di misura sono espresse rispettivamente in m/s^2 o g (pari a circa 9.8 m/s^2; se l'oggetto non si muove, è misurato 1 g verso terra), DPS (*degrees per second*, °/s) e gauss (Gs).
All'interno di questo progetto, il sensore viene impiegato per percepire l'urto e la sua magnitudo, e per avere la conoscenza completa sulla posizione d'impatto dell'oggetto; inoltre per ogni misura possono essere regolate le scale, adattando le misurazioni rilevate a seconda dell'impiego dell'IC.
L'LSM9DS1 supporta sia SPI sia I^2C: nel progetto si è optato per l'utilizzo dell'I^2C, che richiede un numero minore di cavi per l'interconnessione rispetto a SPI (due cavi contro quattro).
Il datasheet inoltre specifica che la tensione d'esercizio è 3.3 V, quindi, come per il MEM Shield, non è previsto *level-shifting*.

La guida fornita dal produttore del chip *SparkFun*, [4] illustra un esempio di collegamento usando l'interfaccia I^2C: come si può notare nella figura di seguito, oltre ai due cavi per l'alimentazione, sono necessari solo altri due per la comunicazione tra microcontrollore e sensore.



Connessione tra LSM9DS1 e microcontrollore RedBoard; l'immagine presentata differisce di poco dal sistema effettivamente usato all'interno di questa tesi

Dopo aver approfondito la configurazione hardware, passiamo a quella software: per interfacciarsi con l'IMU, è necessario scaricare la libreria Arduino fornita dal produttore del chip. I file presenti all'interno della libreria sono quattro: tre file .h e un file .cpp.
Gli *header file* definiscono: i registri interni per il giroscopio/accelerometro e magnetometro (*LSM9DS1_Registers.h*), i tipi e gli *enum* usati dalla classe LSM9DS1 (*LSM9DS1_Types.h*) ed infine i prototipi delle classi (*SparkFunLSM9DS1.h*).
All'interno di *SparkFunLSMDS1.cpp,* sono implementate tutte le funzioni della classe LSMDS1, da quelle ad alto livello, come la lettura/scrittura dei registri dell'IMU, a quelle a basso livello, come le operazioni di lettura e scrittura hardware.

Oltre alla libreria sopra illustrata, si devono impiegare altre tre librerie: *ArduinoJSON* [5], *ArduinoHttpClient* [6] e *WiFiUdp* [7].
La prima serve a convertire in formato JSON i dati ricevuti dal sensore: ciò è necessario poiché le POST alla rotta */measurements* di Atmosphere accettano solamente il formato JSON mentre la seconda consente di gestire agevolmente l'iterazione con i *web server* direttamente da Arduino (l'uso di questa libreria è strettamente legato ad un'altra libreria denominata *WiFiNINA*, necessaria ad abilitare l'antenna WiFi integrata nel MKR).
Infine, la terza libreria WiFiUdp.h è utilizzata per connettersi a un server NTP (Network Time Protocol) e ottenere l'ora globale esatta da cui far partire il nostro sistema.

All'accensione del sistema, la prima *routine* contenuta nel *setup* è *firstConnection*: la funzione, dopo un iniziale controllo della versione del firmware utilizzato dal modulo WiFi, provvede alla connessione della rete scelta, i cui parametri, ssid e password, sono presenti all'interno di un file.h dedicato (*arduino_secrets.h*); se il login alla rete va a buon fine, la variabile *status* assume valore pari a 3, numero che indica la presenza di connessione ad una rete WiFi (*WiFi.status() = WL_CONNECTED*).

```
Attempting to connect to SSID: Telefono Mi
Connected to wifi
SSID: Telefono Mi
IP Address: 192.168.43.147
signal strength (RSSI):-47 dBm
```

Su monitor seriale, queste sono le prime righe in caso di connessione riuscita

Essendo la verifica della connessione all'interno di un ciclo *while*, l'ESP32 tenta di connettersi fino ad ottenere esito positivo; in questo progetto non viene valutato il caso in cui la prima connessione non si verifichi, questo perché supponiamo che, come già esposto all'interno del primo paragrafo, il seguente dispositivo sia contenuto all'interno di un imballaggio per spedizioni e che quindi, al momento della spedizione, il sistema possa connettersi alla rete WiFi presente all'interno del magazzino.

Successivamente, la funzione *imuSettings* permette all'utente di variare il *range* dell'accelerometro (variabile in un intervallo compreso tra -16 e +16g) e, successivamente, controlla se il sensore e la scheda microSD sono collegati; se non vengono rilevati problemi, il metodo *begin* permette di aggiornare il sensore con le impostazioni scelte.
Le operazioni di *setup* continuano con l'acquisizione, previa autorizzazione, di un *token* (dall'API Atmophere): se la comunicazione è andata a buon fine (verificabile stampando a video lo *status code* della richiesta), si ottiene una stringa al cui interno è presente il *token*. Quest'ultimo dovrà essere sempre inserito all'interno di un *header* per ogni chiamata alle API.
In generale la sicurezza è un aspetto critico all'interno dell'IoT [8]: l'autenticazione basata su *token* consente di identificare, ed eventualmente accettare, ciascuna richiesta mandata dal *client*.
Ogni API dovrebbe generare questo "gettone" in maniera pseudo-casuale e forte a livello criptografico.
L'impiego di questo tipo di autorizzazione all'interno del seguente elaborato soddisfa i requisiti minimi di sicurezza ma è bene ricordare che, in generale, a seconda dell'applicazione e dei dispositivi usati, è necessario prendere ulteriore precauzioni (per quanto tempo è valido lo stesso *token*, più utenti possono avere lo stesso, è sufficiente usarne solo uno, ecc…).

L'ultima operazione del setup è la funzione *getSecondsServer* che serve per ottenere l'ora globale esatta per permettere al nostro sistema di avere un record temporale degli eventi.
Per ottenere l'ora ci colleghiamo a un NTP (Network Time Protocol), che è un protocollo Internet standard (IP) per sincronizzare gli orologi dei computer connessi alla rete e può essere utilizzato per sincronizzare tutti i dispositivi di rete con Tempo Coordinato Universale (UTC) in pochi millisecondi [9].
Il *Coordinated Universal Time* (UTC) è uno standard temporale mondiale, strettamente correlato al GMT (Greenwich Mean Time); il vantaggio di utilizzare UTC è che questo non varia, ovvero è lo stesso in tutto il mondo.
NTP imposta dunque gli orologi dei computer su UTC, il client applica qualsiasi offset di fuso orario locale o offset di ora legale. In questo modo i client possono sincronizzarsi con i server indipendentemente dalla posizione e dalle differenze di fuso orario.

Vista l'importanza di questo dato, nel progetto è presente una ricorsività dentro codice: il sistema richiede, tramite una connessione Udp, il pacchetto contenente l'informazione (è per non ripetere dato tante volte) utile al nostro sistema.
Nei sistemi, il tempo viene rappresentato in secondi rispetto alla mezzanotte (UTC) del 1° gennaio 1970 e in questo modo viene salvata ora e data sul nostro Sistema.
Questo dato viene utilizzato quando è necessario notificare la data e l'ora di un urto utilizzando la variabile *secondsStart* di ritorno dalla funzione *getSecondsServer* e a questa viene sommata la funzione *seconds()* definita come *millis()*/1000, dove *millis()* è la funzione di Arduino che calcola i millisecondi trascorsi dall'accensione del Sistema.

Come è noto per qualsiasi Arduino, dopo il *setup* segue la funzione *loop*, facilmente intuibile dal nome della funzione stessa e permette di reiterare il codice scritto al suo interno, consentendo al programma di cambiare e rispondere a seconda dell'evento preso in esame; nel caso di un rilevatore di urti, il *loop* aspetta fino all'istante di un colpo.
L'analisi dell'impatto (nel codice sviluppata sotto *impactAnalysis*) inizia con la lettura, tramite funzioni di libreria, dell'accelerometro.
Per scelta progettuale, la routine dell'impatto viene chiamata ogni 2 ms, ovvero qualsiasi evento si verifichi all'interno di 2 millisecondi in seguito all'urto, viene ignorato dal microcontrollore; inoltre, onde evitare "falsi positivi" (per esempio, rimbalzi successivi all'urto), è stata aggiunta una variabile *vibration* che funziona come *loop counter*: se il contatore, impostabile a piacere, è maggiore di 0, l'urto rilevato è in realtà un rimbalzo ed è dunque ignorato ai fini dell'analisi.

La funzione *Impact* è stata adattata da un codice per il rilevamento di collisioni tra veicoli [10] al cui interno troviamo il calcolo dell'intensità dell'urto, ricorrendo al teorema di Pitagora: il codice originale presentava una rilevazione di urti bidimensionale per calcolare la parete laterale della macchina che ha

subito l'urto, invece per il nostro sistema abbiamo adattato il codice tridimensionalmente seguendo le specifiche del progetto.
Le variabili utilizzate sono le accelerazioni sui tre assi e il modulo della forza dell'urto è calcolato utilizzando questi valori per ricavare il modulo del vettore risultante. Il risultato è successivamente confrontato con un valore denominato *sensitivity*, anch'esso impostabile a piacere dall'utente: se il valore del modulo è maggiore della sensibilità impostata, considero valido l'urto (e pongo uguale ad 1 un flag per evidenziare l'impatto avvenuto e visualizzare su seriale i dettagli).
Inoltre, questi valori vengono conservati in memoria per confrontarli con i dati ottenuti all'analisi successiva.
Per evitare di contare più volte lo stesso urto, si confrontano i dati acquisiti con quelli precedenti salvati in memoria e calcolo la differenza per ricavare il vettore finale.
Quindi, per riassumere, il rilevamento di un urto o meno è vincolato da tre fattori:

- il tempo intercorso tra un urto e il successivo
- la verifica di eventuali rimbalzi successivi all'impatto
- la sensibilità scelta per il sensore.

Oltre all'intensità dell'urto, è rilevante conoscere anche il punto in cui si è verificato.
Per fare ciò, sono state introdotte tre variabili, che indicano gli angoli d'impatto sui piani XY, YZ e XZ: il calcolo dei valori è stato reso possibile grazie alla funzione *atan2(y,x)*, che compare per la prima volta nel linguaggio di programmazione FORTRAN, ed indica l'angolo in radianti tra il semiasse positivo delle x e un punto di coordinate (x,y) che giace su di esso.
L'intervallo di valori restituibili dalla funzione va da -π e +π ma, per una migliore lettura dei risultati, è stato scelto di convertire questi ultimi da radianti a gradi e modificare l'intervallo da 0° a 360°: se il valore originario è maggiore di 360°, viene sottratto l'angolo giro, se è minore di 360°, viene sommato.
Per verificare la correttezza dei dati acquisiti, all'interno della funzione sono stati aggiunti dei *print* per visualizzare sul monitor seriale di Arduino IDE: questo strumento ha permesso in fase di programmazione di testare il programma e, una volta concluso, di avere immediatamente disponibili i valori che il sensore riceveva.

Dopo l'analisi dell'impatto tramite le sue quantità principali sopraelencate, i dati possono essere inviati all'API o salvati su SD: se è presente una connessione Internet, verificabile tramite la funzione *connection*, viene chiamata la funzione *postOnlineData*.
All'interno di *connection*, a differenza di *firstConnection,* il tentativo di collegamento è eseguito una sola volta e, in caso di fallimento, un messaggio annuncia l'utilizzo della scheda SD da questo momento in avanti.
In *postOnlineData*, dopo l'acquisizione del solito *token*, i dati acquisiti sono convertiti in oggetti JSON e inseriti a loro volta in documenti di tipo JSON (in [5] sono chiamati *JsonDocument*, necessari anche a creare le *reference* ad un array, detti *JsonArray*): questa operazione in realtà è eseguita sia che ci sia connessione sia che manchi, predisponendo già in questa fase, i valori che verranno inviati all'API dalla scheda microSD al momento della riconnessione.
In seguito, la funzione *postMeasureFunction*, prendendo in ingresso il *JsonObject* appena creato ed il *token*, esegue la POST alla rotta indicata; come è possibile notare nell'immagine sottostante, la variabile *httpClient* annuncia l'inizio della richiesta ed in seguito specifica, tramite il metodo *post()*, la rotta.
In aggiunta alla rotta di Atmophere, vengono specificati alcuni *header*, in cui spicca l'autorizzazione tramite *token*, ed infine il corpo della richiesta è inviato.
Per verificare che la POST sia andata a buon fine, è stampato a schermo lo *status code* della richiesta e il *body* della risposta.

```
void postOnlineData(int value) {
  String token = getToken();
  //ora parte seconda post
  measureObject["thing"] = "product";
  measureObject["feature"] = "impact";
  measureObject["device"] = "impact-sensor";
  //if (!arrayComplete)
  //{
  if (value == 0) {
    firstCompleteNested();//nel caso in cui all'impatto ho connessione
  }
  if (value == 1)
  {
    SDCompleteNested();
  }

  //arrayComplete=true;
  //}
  //else
  //{
  //  setCompleteNested();
  //}

  postMeasureFunction(measureObject, token);

  int statusCode2 = httpClient.responseStatusCode();
  String response2 = httpClient.responseBody();// è il body di ritorno

  Serial.print("Status code2: ");
  Serial.println(statusCode2);
  Serial.print("Response: ");
  Serial.println(response2);
}
```

La funzione postOnlineData

```
void postMeasureFunction(JsonObject& measureData, String token) {
  String dataStr2 = "";
  serializeJson(measureData, dataStr2);   // https://arduinojson.org/v6/api/json/serializejson/
  httpClient.beginRequest();
  httpClient.post("/v1/measurements");
  httpClient.sendHeader("Content-Type", "application/json");
  httpClient.sendHeader("Content-Length", dataStr2.length());
  httpClient.sendHeader("Authorization", token);  //da fare quando ricevi il token per inviare i successivi dati
  httpClient.beginBody();
  httpClient.print(dataStr2);
  httpClient.endRequest();

  wifi.flush();
}
```

La funzione postMeasureFunction

Tornando al *loop*, nel caso in cui la connessione Internet sia assente al momento dell'impatto, onde evitare di perdere informazione, i dati vengono temporaneamente salvati su SD all'interno di un file denominato *datalog.txt*; prima di scrivere all'interno di esso, si verifica che il file esista, altrimenti ne viene creato uno.
Ciascuna stringa all'interno di *datalog.txt* è composta dalle stesse quantità principali precedentemente elencate e al termine della scrittura viene visualizzato un messaggio di conferma.
Quando il dispositivo è *offline*, ad ogni impatto rilevato, la scheda tenta comunque di connettersi: questo controllo è stato aggiunto ipotizzando che ci sia stata solo una momentanea perdita di connessione e che quindi la rete Internet sia nuovamente disponibile nel breve periodo.

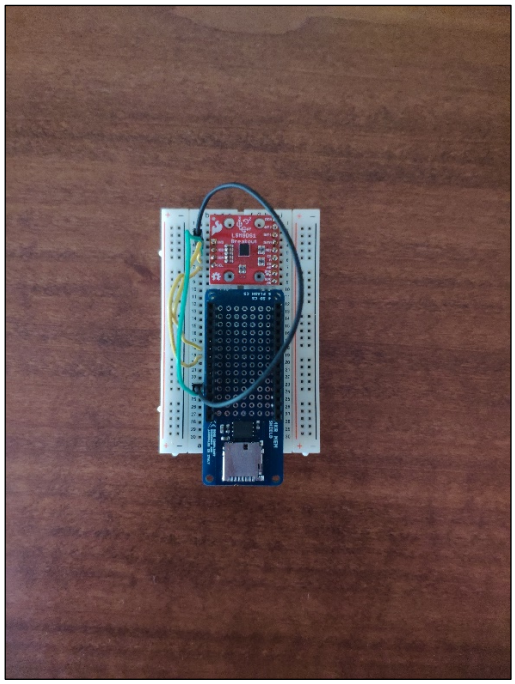
```
Impact detected!!       Magnitude:3.22   AngleXY:136     AngleYZ:328     AngleXZ:211


Attempting to connect to SSID: Telefono Mi
No wifi, using SD memory
ho salvato l'urto su scheda SD
Righe salvate su SD = 5
```

Esempio di urto salvato su SD

Al termine di queste operazioni, sempre all'interno del *loop*, ogni 5 secondi è verificata la connessione e si possono verificare queste due situazioni:

- se il controllo avviene quando il dispositivo è connesso ad Internet, l'esecuzione procede come di consueto
- se la verifica giunge a dispositivo *offline*, la funzione *readSDandPost* è chiamata.

Questa funzione inizialmente verifica la presenza di *datalog.txt* all'interno della scheda SD e solo successivamente, tramite un ciclo *while*, "legge" parola per parola fino alla fine della riga.
Per inserire ciascuna quantità, è stato utilizzato uno *switch-case* poiché sappiamo che l'ordine di lettura delle stringhe è uguale a quello del *JsonObject* precedentemente creato e già predisposto in *postOnlineData*, funzione che viene chiamata al termine dello *switch-case*.
Dopo ogni stringa inviata ad Atmophere, questa viene rimossa dal file; al termine della lettura di tutte le righe, *datalog.txt* è definitivamente cancellato poiché ha termine la sua funzione di conservazione temporanea dei dati, essendo questi ultimi reperibili direttamente dall'API.



Il dispositivo effettivamente utilizzato nel corso della sperimentazione

Riassumendo

SETUP:

- Prima connessione alla rete internet tramite modem router o hotspot da cellulare.
- Preparazione e *settings* dei parametri del sensore IMU
- Richiesta e acquisizione del token dal server API Atmosphere

LOOP:

- Analisi impatto (Loop ogni 2ms fino ad evento)
- Verifica connessione alla rete internet:

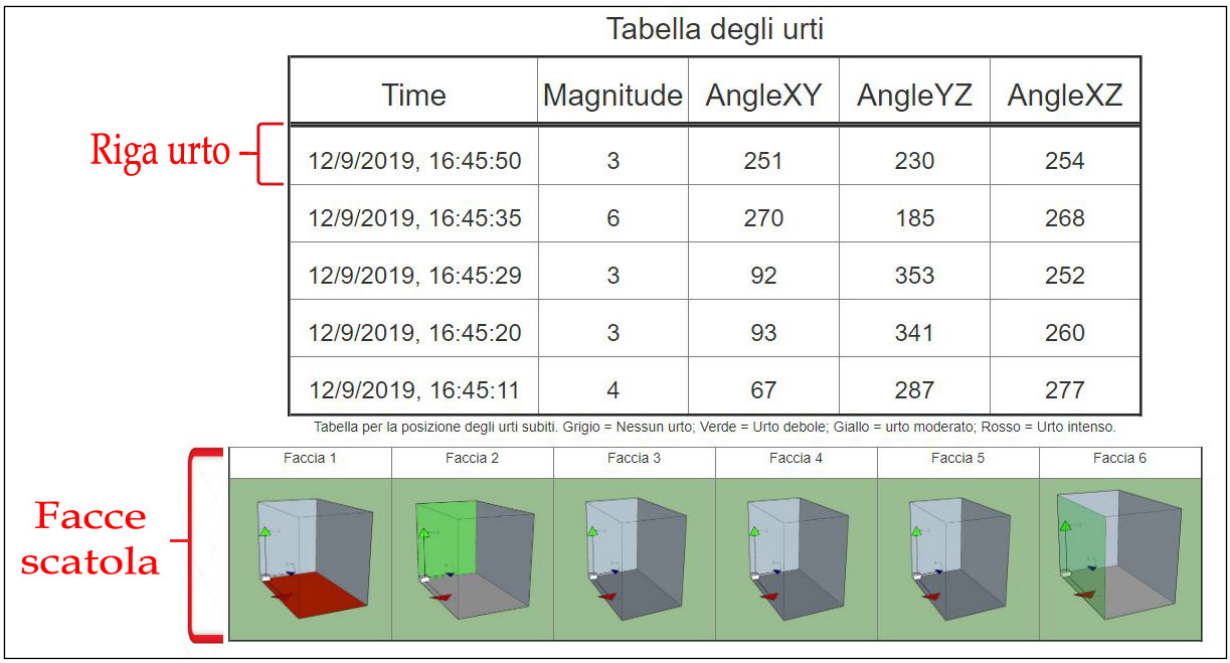
1. **Connesso alla rete:**
    - Funzione *PostOnlineData* di tipo "0" ovvero creo il file Json con i dati già disponibili nelle variabili e effettuo post ad Atmosphere
    - Verifico risposta positiva dal server
    - Ritorno al loop iniziale

2. **Non connesso alla rete:**
    - Salvo i dati in ordine sulla scheda SD e incremento di 1 la variabile *dataPrintedSD* per notificare il salvataggio di una riga sul file *datalog.txt*
    - Torno al loop iniziale e ogni 5 secondi controllo se la connessione viene ristabilita
    - Se il sistema è connesso e ho righe salvate sulla scheda SD, chiamo la funzione *postOnlineData* di tipo "1": Prendo da memoria i dati di interesse e li organizzo in formato JSON per poi inviarli al server. Dopo questa operazione decremento di 1 *dataPrintedSD* e ripeto la post fino all'annullamento della variabile.
    - Infine, cancello il file *datalog.txt* e torno al loop iniziale

Prendiamo ora in considerazione l'interfaccia web scritta in linguaggio html e con utilizzo dello script.
Per prima cosa è stata costruita una tabella per visualizzare a pagina gli ultimi 5 record salvati sul server Atmosphere (questa impostazione potrà in futuro essere modificata a seconda delle esigenze su quali record visualizzare e in che ordine).
Le intestazioni della tabella sono rispettivamente: *Time* per la data, *Magnitude* per l'intensità dell'urto, *AngleXY*, *AngleYZ*, *AngleXZ*.
I dati per riempire questi campi vengono ottenuti tramite una *get* al server Atmosphere alla seguente rotta:

"http://test.atmosphere.tools/v1/measurements?filter={"thing":"product"}&limit=1&page=1"
(questo per la prima riga della tabella).

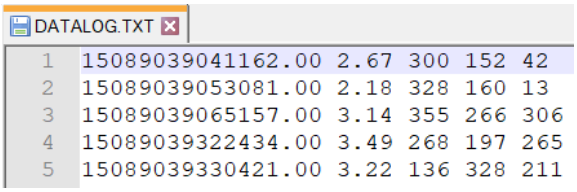
Il file in risposta dal server viene trasformato in formato JSON e vengono estrapolati tutti i dati di interesse e inseriti nella tabella e così viene ripetuto per tutte le righe.
Ogni volta che viene ricevuta una risposta, è eseguito un algoritmo che utilizza i dati contenuti nei tre campi degli angoli ed è calcolato, su un ideale parallelepipedo rappresentante il pacco da spedizione, la faccia su cui avviene l'impatto.
Per la creazione del parallelepipedo sono state prese le misure di un pacco di spedizione standard: Larghezza 40.16 cm, Altezza 32.86 cm, Profondità 25.8 cm.
Mediante l'uso di una variabile per ogni faccia e l'utilizzo di soglie decise arbitrariamente, viene calcolato un punteggio di danno subito per ogni lato: questo valore viene utilizzato per scegliere quale foto dell'archivio mostrare nell'interfaccia.
Come descritto in figura i colori corrispondono a: Grigio = Nessun urto; Verde = Urto debole; Giallo = urto moderato; Rosso = Urto intenso.



Interfaccia web visibile agli utenti.

3. Sperimentazione e risultati

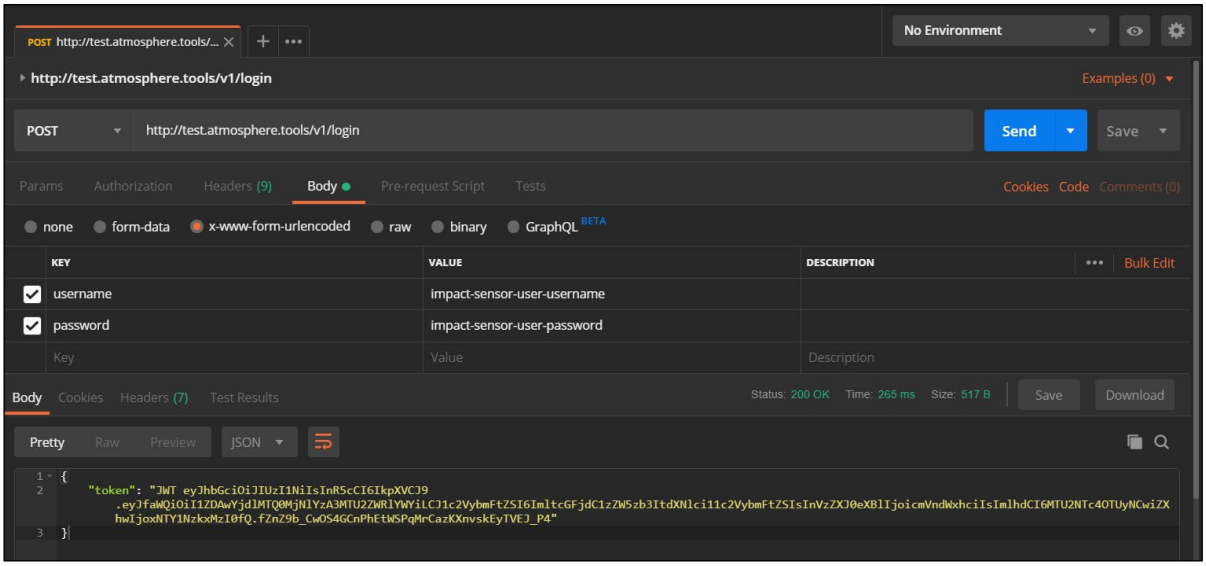
Al termine dell'analisi hardware e software del progetto, il sistema è stato testato, caricando il codice nel microcontrollore.
Durante le prime verifiche di funzionamento, è stato messo da parte la sezione relativa all'invio dei dati all'API e dunque le uniche funzioni esaminate in questa fase sono state quelle relative all'impatto e del salvataggio su scheda SD (quindi in assenza di connessione).
Per visualizzare i dati *real-time*, come scritto nel secondo paragrafo, l'ambiente di sviluppo di Arduino mette a disposizione uno strumento che consente alla scheda di comunicare con il computer tramite porta USB, il monitor seriale; inoltre, collegando la scheda al computer, permette di alimentarla.
Il primo *test* eseguito sul sistema è il salvataggio dei dati su SD, ma prima di partire con gli urti, verifichiamo che Arduino ci avverta nel caso non venga rilevata la scheda SD al momento dell'accensione: se la scheda non è presente o è presente un guasto nello *shield*, è restituito il messaggio *"Card failed, or not present"*.
All'interno della funzione deputata alla scrittura, è nuovamente effettuato un controllo sulla scheda: infatti, come già descritto in precedenza, *SD.open* apre/crea il file *datalog.txt*, ma se l'operazione non va a buon fine, viene visualizzato un errore, dovuto ad una possibile assenza della SD (rimozione forzata volontaria) o difetto sul MKR (un urto troppo inteso potrebbe aver danneggiato il sistema).
Se non vi sono errori, l'urto è regolarmente salvato e un messaggio di conferma sottolinea il totale delle righe scritte all'interno del file.
Date le ipotesi assunte, ricolleghiamo il sistema al PC ed eseguiamo il secondo *test*: simulare cinque urti, scollegare il dispositivo ed aprire il file tramite un editor di testo sul computer.
Il risultato di tale esperimento è visibile in figura: dati cinque urti, all'interno del file sono presenti cinque righe che mostrano in ordine cronologico, intensità ed angoli dei singoli urti.



Il file datalog.txt

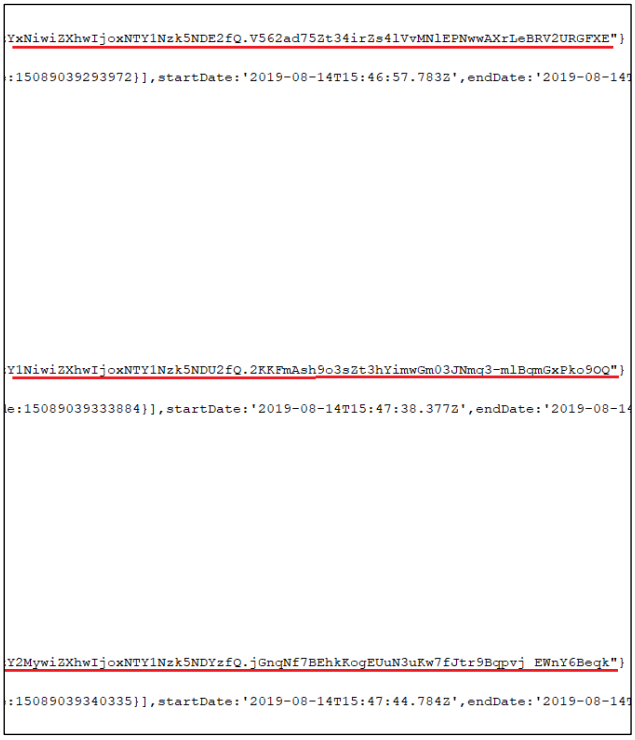
Confermate le ipotesi di funzionamento a dispositivo *offline*, la sperimentazione è continuata includendo le funzioni strettamente legate all'invio dei dati; prima però è stato ritenuto opportuno testare le rotte per invocare le API Atmosphere.
Infatti, un aspetto cruciale per poter fare le chiamate è il possesso del *token*, ottenibile eseguendo una POST alla rotta *http://test.atmosphere.tools/v1/login,* il *body* cui al suo interno contiene un *username* e una *password.*
Senza utilizzare direttamente il sistema costruito, esiste un programma che consente di avviare richieste ad un API in modo veloce ed efficiente: Postman.
Questa applicazione supporta diverse richieste http ed è possibile aggiungere degli *header* a quest'ultime.

Nell'immagine che segue, è visibile l'interfaccia del programma al momento della POST: la risposta alla richiesta è molto più dettagliata rispetto a quella di un semplice browser poiché è ben evidente lo *status code*, il tempo impiegato dal momento della richiesta e la dimensione della risposta.
Inoltre, Postman dà la possibilità di scegliere il formato con cui visualizzare il *body* (XLM, HTML, JSON, ecc…).



L'interfaccia (con risposta) di Postman al momento del login

Il *token* ricevuto in risposta è coerente con quanto dichiarato in precedenza, ovvero è un codice generato in maniera pseudo-casuale.
Verificato il corretto funzionamento tramite applicazione, ora siamo sicuri che *getToken* fornirà in uscita un codice simile a quello presente in figura, sempre diverso ad ogni impatto (segue immagine).

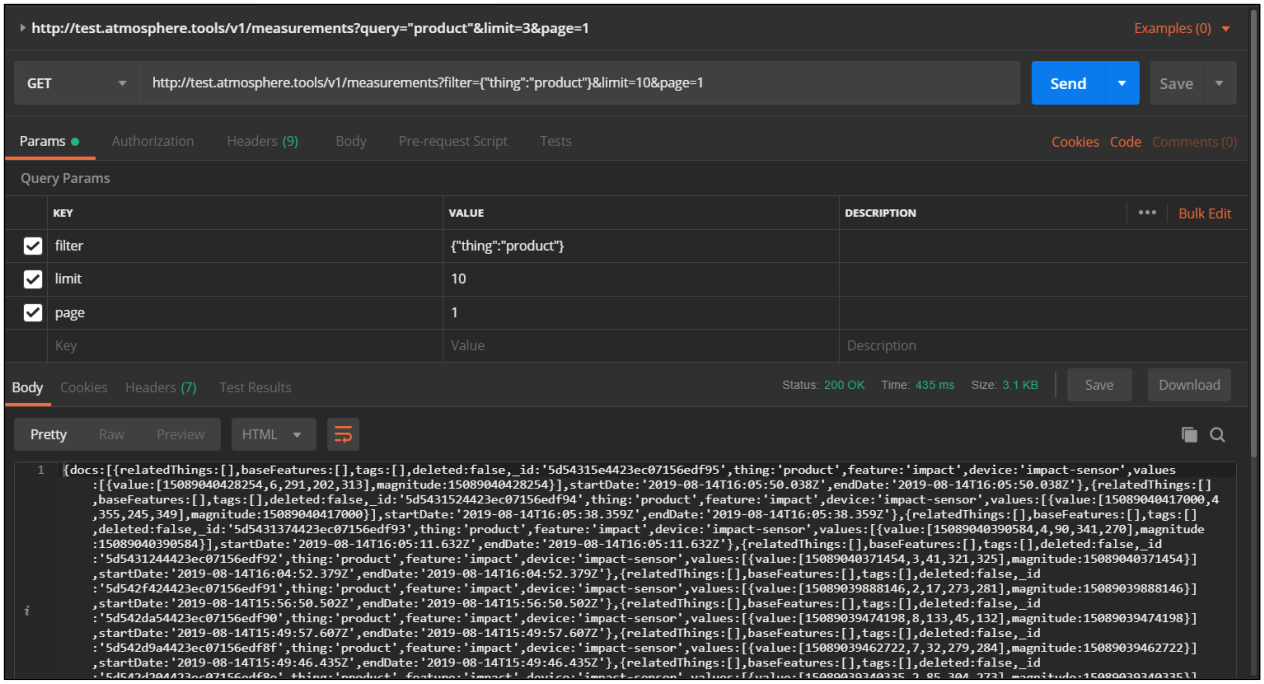


Dettaglio di tre token al momento della chiamata: sebbene la prima parte rimanga uguale, le lettere sono diverse verso la fine

Cosa succede dunque quando, al momento di un urto, la scheda tenta di “postare” i valori in Atmophere? Dal monitor seriale di Arduino, grazie alle *print* inserite nel codice, le richieste sembrano essere andate a buon fine poiché lo *status code* della POST è pari a 200 e nel protocollo http le risposte nella categoria “2xx” sono *Successful* (nello specifico, 200 è “OK”).
La prova conclusiva per la buona riuscita delle chiamate all’API è utilizzare nuovamente Postman: eseguiamo per tanto una GET alla rotta */measurements*, così facendo però in risposta troveremo misure di altri dispositivi che non sono correlati al nostro progetto; per ovviare a questo problema, */measurements* accetta alcuni parametri, simili ad un DBMS (*Database management system*), per ottimizzare la visualizzazione dei dati. [11]
Consultando la documentazione di Atmosphere, sono fornite solo due risposte tramite codici http: 200 in cui segue un facsimile della lista delle misurazioni, e 400, che indica una richiesta errata da parte dell’utente (la famiglia di risposte “4xx” segnalano un *Client error*)
Quindi la nuova GET presenta sotto la scheda *Params* di Postman, le parole chiavi:

- *filter*, in cui la nostra *thing* è chiamata *product*,
- *limit*, che indica quante misurazioni si intende visualizzare per pagina,
- *page*, che indica il numero di pagina richiesto.

L’insieme dei parametri e i suoi risultati sono visibili nell’immagine che segue:



Nell'esempio sopraesposto, tramite alcuni parametri, visibili anche nella barra degli indirizzi, Postman restituisce la prima pagina degli ultimi 10 urti di product

Dopo una panoramica sul funzionamento di Postman e dopo aver approfondito Atmosphere, si passa al terzo *test* del progetto: eseguire degli urti (nuovamente cinque per esempio), inviare i dati all'API e verificare tramite Postman se i dati stampati sul monitor seriale di Arduino coincidono con il *body*, opportunamente filtrato, dell'applicazione.
Nuovamente le nostre ipotesi sono confermate: la prima figura rappresenta un dettaglio degli urti dall'IDE di Arduino, la seconda invece è il *body* della risposta, filtrando gli ultimi cinque risultati.

```
Impact detected!!       Magnitude:3.22    AngleXY:85      AngleYZ:39      AngleXZ:84
Impact detected!!       Magnitude:2.54    AngleXY:230     AngleYZ:142     AngleXZ:137
Impact detected!!       Magnitude:3.79    AngleXY:4       AngleYZ:273     AngleXZ:301
Impact detected!!       Magnitude:3.10    AngleXY:264     AngleYZ:228     AngleXZ:265
Impact detected!!       Magnitude:4.46    AngleXY:113     AngleYZ:6       AngleXZ:166
```


```
{docs:[{relatedThings:[],baseFeatures:[],tags:[],deleted:false,_id:'5d56ccb94423ec07156edf9e',thing:'product',feature:'impact',device:'impact-sensor',values:[
{value:[15089039049797,4,113,6,166],magnitude:15089039049797}],startDate:'2019-08-16T15:33:13.778Z',endDate:'2019-08-16T15:33:13.778Z'},{relatedThings:[],
baseFeatures:[],tags:[],deleted:false,_id:'5d56ccaf4423ec07156edf9d',thing:'product',feature:'impact',device:'impact-sensor',values:[{value:[15089039039718,
3,264,228,265],magnitude:15089039039718}],startDate:'2019-08-16T15:33:03.690Z',endDate:'2019-08-16T15:33:03.690Z'},{relatedThings:[],baseFeatures:[],tags:[]
,deleted:false,_id:'5d56cca84423ec07156edf9c',thing:'product',feature:'impact',device:'impact-sensor',values:[{value:[15089039032867,4,4,273,301],
magnitude:15089039032867}],startDate:'2019-08-16T15:32:56.688Z',endDate:'2019-08-16T15:32:56.688Z'},{relatedThings:[],baseFeatures:[],tags:[],deleted:false,
_id:'5d56cca04423ec07156edf9b',thing:'product',feature:'impact',device:'impact-sensor',values:[{value:[15089039024854,3,230,142,137],
magnitude:15089039024854}],startDate:'2019-08-16T15:32:48.910Z',endDate:'2019-08-16T15:32:48.910Z'},{relatedThings:[],baseFeatures:[],tags:[],deleted:false,
_id:'5d56cc9a4423ec07156edf9a',thing:'product',feature:'impact',device:'impact-sensor',values:[{value:[15089039018490,3,85,39,84],magnitude:15089039018490}]
,startDate:'2019-08-16T15:32:42.328Z',endDate:'2019-08-16T15:32:42.329Z'}],total:97,limit:5,page:1,pages:20}
```


Da notare come su Arduino, i dati sono in ordine cronologico (dal più vecchio al più recente), mentre su Postman in ordine inverso (dal più recente al più vecchio)

L'ultima funzione da testare è la POST dei dati precedentemente salvati su SD: questa è la parte più delicata della tesi poiché è necessario convertire dei caratteri salvati su file .txt in oggetti JSON, che è l'unico formato che Atmosphere accetta (vedi [11]).
I dettagli inerenti alla conversione sopracitata sono già stati illustrati nel paragrafo precedente.
Il *modus operandi* per questo *test* è il seguente: connettere ad Internet Arduino (ricorda che la prima connessione è mandatoria per il funzionamento del progetto), scollegare il WiFi, simulare tre urti (che

verranno salvati su microSD) e ripristinare la connessione, verificando che i tre urti salvati vengano "postati".

Postman conferma ciò che il monitor seriale annunciava: le righe sono state lette, convertite per ogni loro campo e inviate all'API

Nella figura in alto, abbiamo conferma di quanto preannunciato. Durante la scrittura del codice, si è scelto di sottolineare come ciascuna proprietà dell'urto fosse stata salvata nella variabile corretta ed è per questo che prima della POST, compaiono delle righe che confermano ciò.

Sebbene Postman sia uno strumento potente e molto utile in fase di sviluppo, non è sicuramente di facile interpretazione al momento della lettura dei dati.

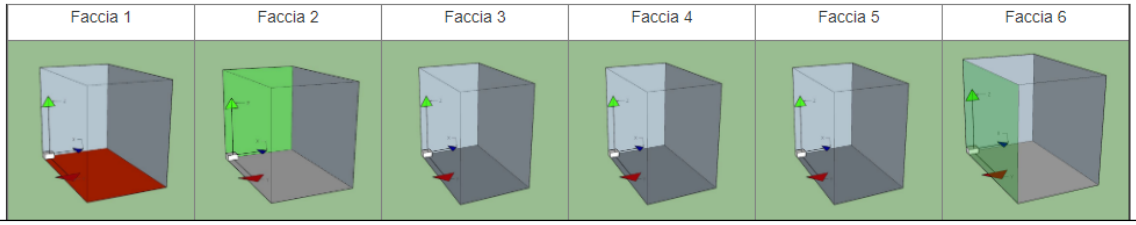
Per questo motivo è stata creata una pagina web che risulta agli occhi dell'utente finale di facile comprensione: la pagina dunque presenta una tabella che raccoglie gli ultimi cinque urti, elencando i tipici attributi già discussi in precedenza.

Insieme alla tabella, a fondo pagina sono state aggiunte sei immagini, ciascuna delle quali rappresentano una faccia di un ipotetico imballaggio.
Ogni lato della scatola sintetizza le conseguenze degli ultimi cinque impatti: ad ogni urto la pagina si aggiorna e quindi la rappresentazione grafica visualizza, con l'utilizzo di quattro diversi colori (grigio nessun urto, verde urto lieve, giallo moderato e rosso intenso) l'intensità degli impatti stessi.
La scala di assegnazione dei colori è stata arbitraria: a seconda dell'intensità dell'urto, espresso sempre in g, viene aumentato un contatore, il cui valore finale determina il colore della faccia.

Tabella degli urti

Time	Magnitude	AngleXY	AngleYZ	AngleXZ
12/9/2019, 16:45:50	3	251	230	254
12/9/2019, 16:45:35	6	270	185	268
12/9/2019, 16:45:29	3	92	353	252
12/9/2019, 16:45:20	3	93	341	260
12/9/2019, 16:45:11	4	67	287	277

Tabella per la posizione degli urti subiti. Grigio = Nessun urto; Verde = Urto debole; Giallo = urto moderato; Rosso = Urto intenso.



L'interfaccia grafica sopra descritta

La seguente applicazione è espandibile a piacere: se si volesse realizzare, per esempio, una *app* per *smartphone*, l'utente potrebbe decidere di visualizzare più urti e filtrare i risultati a seconda delle sue necessità, potendo quindi tracciare e approfondire lo stato della spedizione in base ai criteri selezionati.

4. Contributo personale e considerazioni conclusive

Dopo aver esposto dettagliatamente quali componenti sono state utilizzate, come sono state impiegate e che risultati abbiamo ottenuto, è importante descrivere come è stato affrontato il problema di rilevare gli impatti ricevuti da un oggetto e come presentarli all'utente.

Il progetto è ispirato, come accennato nel secondo paragrafo, da un sistema di rilevamento di urti installato su un'automobile: il contributo approntato al codice (reperibile in nota [10]) è stato quello di definire personalmente i tre angoli denominati XY, YZ e XZ. Sebbene questa possa sembrare una complicazione al fine della realizzazione del progetto, senza di essi, non sarebbe stato possibile progettare l'interfaccia web: nell'approcciare la seguente tesi infatti uno degli obiettivi cardine è stato quello di renderla fruibile anche a persone non abituate a confrontarsi con il mondo dell'elettronica e della programmazione.
L'unico modo per visualizzare i dati era leggere la risposta JSON ricevuta dal framework Atmophere tramite il programma Postman ma, grazie ad una *web application* creata da zero, la lettura risulta più agevole.
In aggiunta alla tabella, le immagini forniscono in maniera intuitiva, tramite una scala cromatica, una sintesi delle condizioni dell'imballaggio o dell'oggetto su cui è stato montato il sistema.
Per arrivare ad ottenere le immagini con le facce della scatola colorate, si è reso dunque necessario creare delle variabili *ad hoc*, non impiegando quelle già fornite dal sensore tramite gli angoli di Eulero (le variabili predefiniti all'interno del sensore IMU sono *pitch* e *roll*).

Poiché il codice prende appunto ispirazione dalla collisione di una macchina, durante la sperimentazione del sistema sono stati eseguiti numerosi test per impostare al meglio la soglia di decisione per il rilevamento o meno di un urto: settando infatti una soglia troppo bassa, il rischio è di rilevare qualcosa anche quando il sensore è fermo (un semplice tocco su un tavolo veniva considerato come un urto dal sensore!).

Da questa problematica, risolta alla fine sperimentalmente, è nato uno spunto di riflessione sull'impiego nel mondo reale del seguente progetto: ipotizzando infatti il suo utilizzo in ambito logistico, a seconda dell'oggetto che il corriere intende spedire, è sufficiente modificare una sola variabile per aumentate/diminuire la sensibilità del sensore. Per fare un esempio, la spedizione di un tavolo in vetro necessita di una soglia di sensibilità maggiore rispetto alla spedizione di un tavolo in legno o di materiale meno fragile.

Un altro problema sorto durante la realizzazione del sistema è stata la comunicazione tra i dati salvati su SD e il loro invio al server, in particolare la conversione da caratteri salvati su file .txt a variabili in JSON: alla fine la soluzione è stata la lettura carattere per carattere della singola linea salvata in memoria, escludendo alcuni caratteri speciali (per esempio lo spazio).
Poiché ci siamo serviti delle API di Atmophere, lo studio di come quest'ultimo elabora e salva i dati è stato utile alla soluzione del problema sopracitato.

È chiaro che questo progetto può rappresentare il primo passo verso qualcosa di più specifico ed elaborato, aggiungendo funzionalità: per esempio è possibile sfruttare il giroscopio a tre assi presente nel sensore per avvertire l'utente qualora il pacco sia sottosopra o appoggiato su uno dei due lati, oppure utilizzando sia l'accelerometro che il giroscopio, rilevare una possibile caduta del pacco.
Questa funzionalità è già presente negli *smartwatch* più recenti di Apple: quando viene rilevata una caduta, l'orologio ti dà la possibilità di chiamare soccorsi.

La seguente caratteristica può essere impiegata sempre durante una spedizione: se per esempio, il corriere effettua una frenata brusca, può essere avvertito della caduta di un imballaggio.

Oltre ad un arricchimento delle funzionalità del sistema, lo stesso *hardware* utilizzato qui è perfettibile: per ottimizzare costi e spazio, è possibile montare tutte le componenti su una *board* dedicata, rendendo quindi disponibile il sistema ad una eventuale commercializzazione.

Per concludere, questo elaborato ci ha permesso, oltre ad approfondire aspetti già affrontati in classe ed applicarli "sul campo", di confrontarci per la prima volta con la realizzazione di un progetto, partendo solamente dalle tre componenti *hardware* presentate: definito l'argomento, per prima cosa, si è studiato il microcontrollore e il suo ambiente di sviluppo. Ottenute i fondamentali per lavorare con Arduino, è cominciato il lavoro di ricerca su Internet di possibili progetti che si avvicinassero a quello presentato in queste pagine, e dopo aver individuato una base da cui partire, sono state aggiunte le funzionalità per adattare il progetto a quanto desiderato.

Durante la realizzazione di questa tesi, ci siamo accorti di quanta attenzione, cura e lavoro sono necessari per portare a termine un progetto relativamente semplice come questo, aspetto spesso sottovalutato o ignorato nel corso degli studi teorici.

5. Riferimenti bibliografici

https://store.arduino.cc/mkr-wifi-1010
https://store.arduino.cc/mkr-mem-shield

[1]https://drive.google.com/file/d/10dYphKwHbEJ-b69iEGDVpFh5caEc4vpM/view?usp=sharing

[2]
https://www.repubblica.it/economia/2019/04/09/news/commercio_vendite_stabili_a_febbraio_boom_dell_online-223606416/

[3]https://www.ilsole24ore.com/art/il-commercio-elettronico-fa-crescere-stipendi-digitale-ACZN3AY

[4]https://learn.sparkfun.com/tutorials/lsm9ds1-breakout-hookup-guide?_ga=2.217884755.452313816.1563890620-213251003.1554896041

[5]https://arduinojson.org/

[6]https://github.com/arduino-libraries/ArduinoHttpClient

[7]https://www.arduino.cc/en/Reference/WiFiUDPConstructor

[8]
https://docs.google.com/presentation/d/1kG5wEGNEoX74x_A3O_gOF-BERaPSLVQK3iFFOlzSMWE/edit?usp=sharing

[9]https://lastminuteengineers.com/esp8266-ntp-server-date-time-tutorial/

[10]
https://electronics.stackexchange.com/questions/156352/understanding-how-to-use-an-accelerometer-to-detect-vehicle-collisions

[11]http://test.atmosphere.tools/api-doc.html#/Measurement/get_measurements