Projet GISTRE

```
|
|--- data
|      |--- human
|      |      |--- data{x}.json
|      |--- normal
|      |      |--- data{x}.json
|      |--- obstacle
|             |--- data{x}.json
|--- src
|      |--- config.json
|      |--- generate_data.py
|      |--- main.py
|      |--- train_data.py
|      |--- utils.py
|--- stm32
|      |--- *
|--- install_dependencies.sh
|--- README.md
```

Vous pouvez récupérer ce projet en utilisant la commande suivante :
`git clone git@github.com:memory-Ieak/ROBOT.git`

Des détails plus approfondis sur le projet seront fournis ultérieurement.

I - Objectif du projet

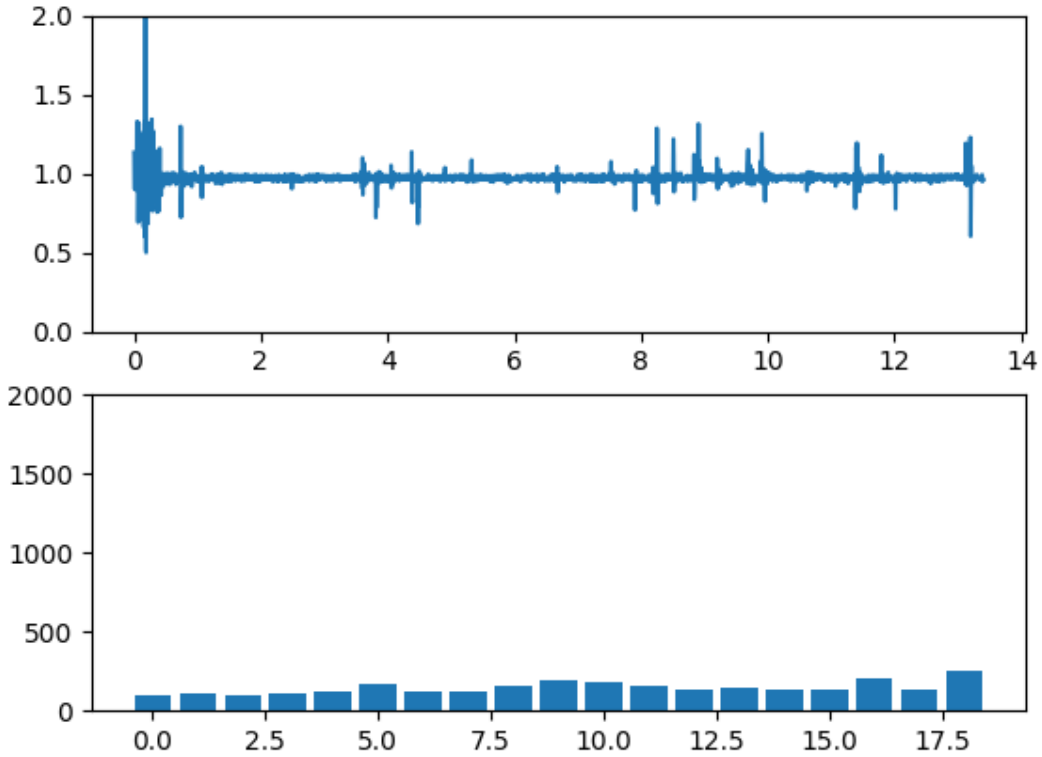
L'objectif de ce projet est d'exploiter un accéléromètre afin de détecter des vibrations sur une voiture miniature et de les caractériser. Nous chercherons à différencier trois scénarios distincts :

- La voiture a subi une variété de choc
- La voiture est manipulé par un humain
- La voiture ne se trouve dans aucune de ces deux situations.

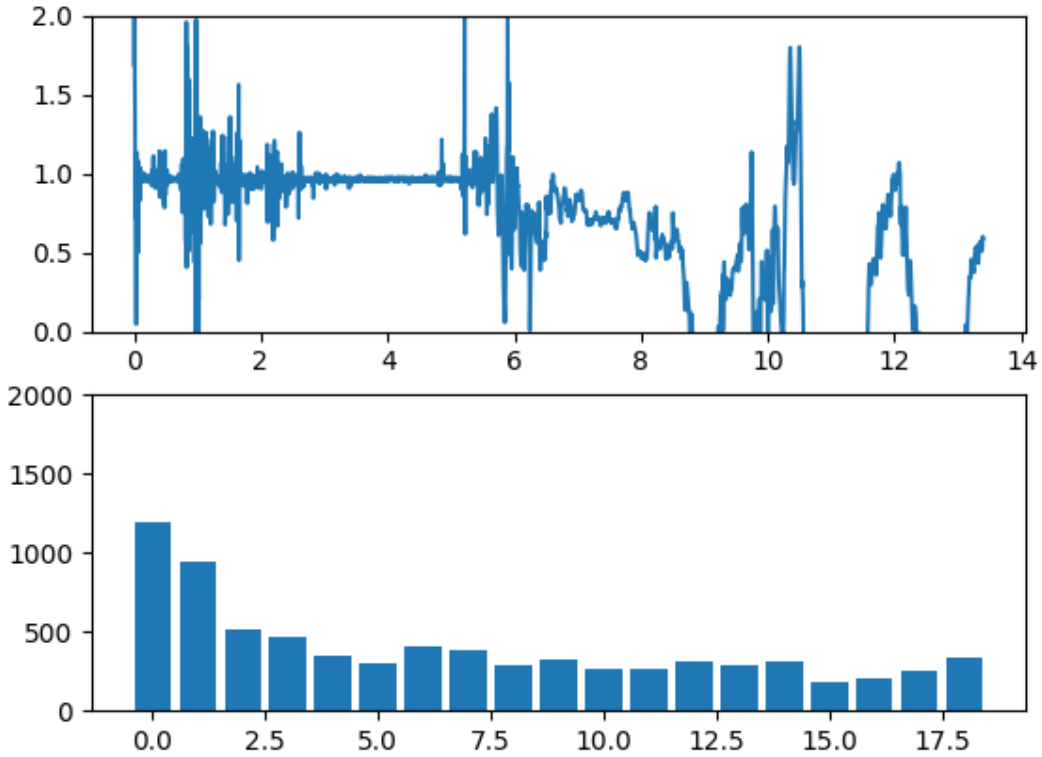
II - Problématiques du projets

La première problématique consiste à identifier correctement les fréquences significatives pour chaque cause de vibration sur le capteur.

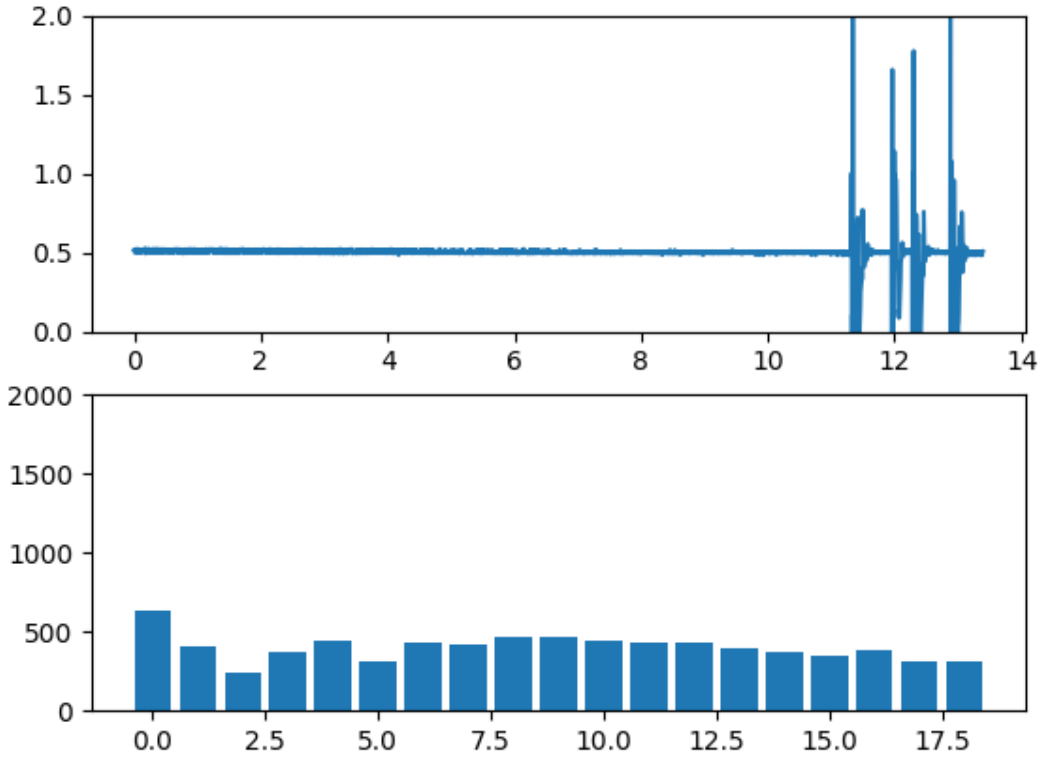
Pour analyser ces variations, nous avons examiné trois scénarios distincts :



Dans ce cas, les fréquences sont uniformes, illustrant la condition normale du capteur sans variations significatives.

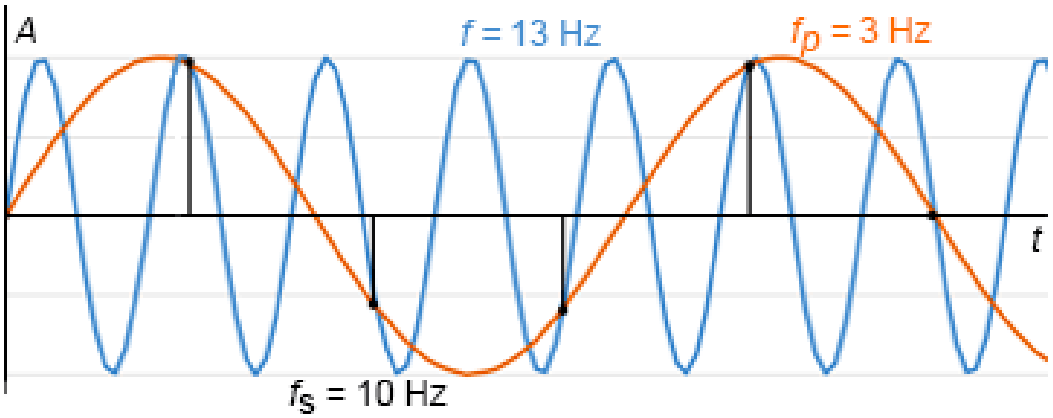


Les basses fréquences sont davantage affectées lors de légères variations du capteur, démontrant des variations distinctes dans la plage de fréquences.

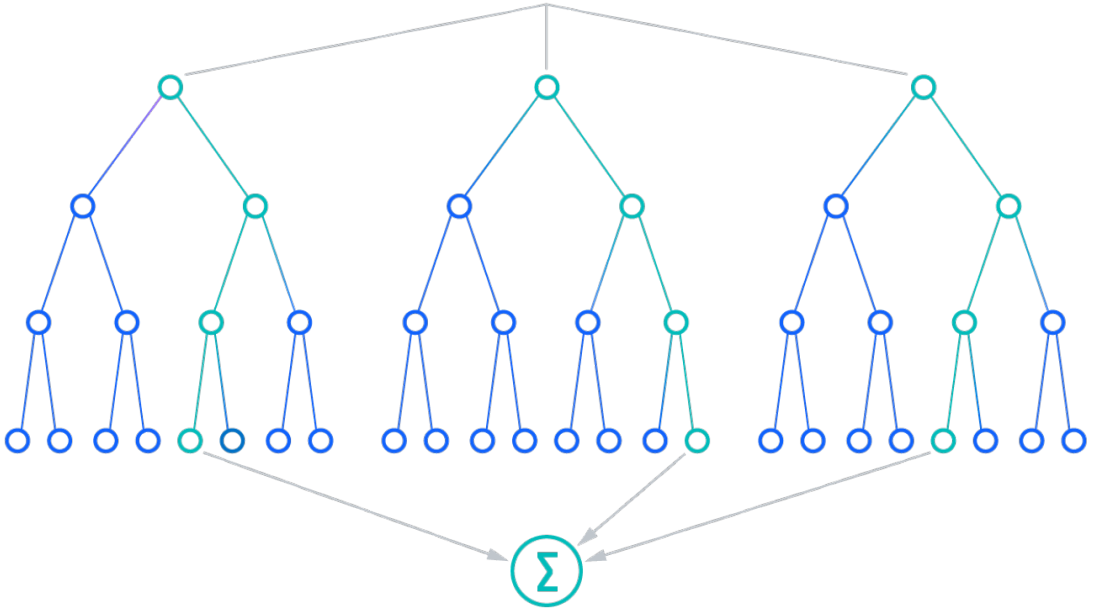


Des pics de fréquence apparaissent clairement lorsque le capteur subit des variations importantes, mettant en évidence des changements significatifs dans les fréquences détectées.

La deuxième problématique concerne la réactivité de la détection de la cause des vibrations. En effet, pour prendre correctement les mesures des vibrations afin de réaliser une transformée de Fourier significative, il est nécessaire d'avoir une plage de temps importante et un taux d'échantillonnage adapté. Nous avons donc choisi de réaliser un échantillonnage toutes les 5 millisecondes afin de prévenir tout risque d'aliasing.



La troisième problématique concerne l'utilisation d'un modèle d'intelligence artificielle pour interpréter les données. À cet égard, notre choix s'est porté sur l'utilisation d'une forêt aléatoire.



Une forêt aléatoire est un algorithme d'apprentissage automatique qui rassemble les résultats de plusieurs arbres de décision pour produire une prédiction unique. Ces arbres, non corrélés entre eux, sont générés par le biais d'une méthode appelée bagging, qui repose sur la sélection aléatoire d'échantillons d'entraînement avec remplacement. L'algorithme de forêt aléatoire introduit également le concept de feature randomness, où un sous-ensemble aléatoire de fonctions est utilisé pour créer des arbres non corrélés. Cette approche réduit le risque de surajustement, améliore la flexibilité du modèle et permet une meilleure interprétation des données.

III - Technologie / matériel utilisée

Matériel:

- MPU6050
- STM32 NUCLEO-F401RE
- Ubuntu 20.04

Technologie:

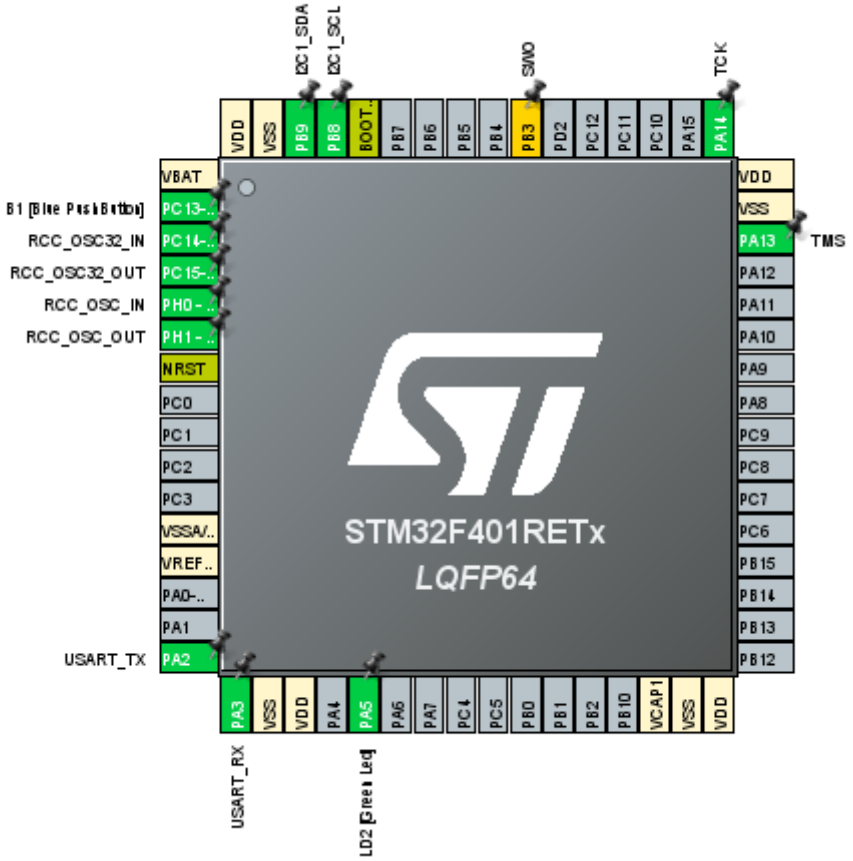
- Tensor Flow
- numpy (fft, etc…)
- C (STM32)
- Python 3 (Ubuntu 20.04)

IV - Utilisation du projet

Accédons d'abord au répertoire stm32.

```
cd stm32/
```

Ce dossier représente un projet STM32CubeMX et peut être ouvert avec:



Dans notre configuration, nous avons configuré une liaison I2C pour communiquer avec le capteur et une liaison UART pour échanger des données avec un ordinateur.

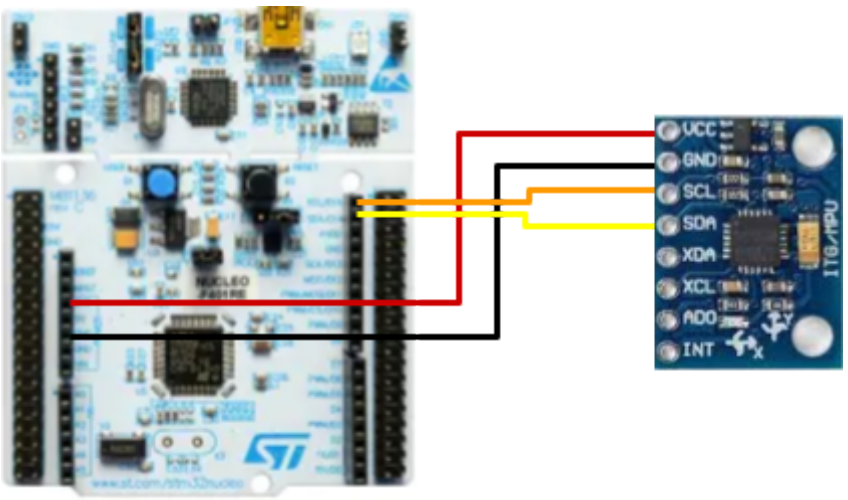
Comme mentionné précédemment, notre intérêt se concentre sur l'accéléromètre, échantillonné toutes les 5 millisecondes.
La structure des données est la suivante:

```
struct __attribute__ ((packed)) MPU6050_accel_data
{
    uint8_t SOH;
    float accX, accY, accZ;
    uint8_t EOT;
};
```

La trame de données est représentée comme:

SOH	accX	accY	accZ	EOT
0x01	0x????	0x????	0x????	0x04

Le câblage du capteur avec le stm32:



Pour flasher le microcontrôleur, utilisez la commande :

```
make flash
```

À présent, le microcontrôleur est prêt à fonctionner.

Entrez maintenant dans le dossier src en utilisant la commande suivante :

```
cd src/
```

À l'intérieur, le dossier est subdivisé en plusieurs fichiers, concentrons-nous d'abord sur config.json.

Ce fichier joue un rôle crucial dans la configuration du projet. Il offre une flexibilité significative en permettant la modification aisée du comportement du projet.

```
{
    "folders": ["human", "normal", "obstacle"],
    "serial": {
        "port": "/dev/ttyACM0",
        "baudrate": 115200
    },
    "data": {
        "format": "=BfffB",
        "begin": 1,
        "end": 4,
        "sampling": 5,
        "time": 13.4
    },
    "labels": {
        "human": 0,
        "normal": 1,
        "obstacle": 2
    },
    "prompts": {
```

```
        "0" : "manipulé par un humain",
        "1" : "normal",
        "2" : "a subit un choc"
    }
}
```

Les paramètres dans ce fichier sont ajustables, offrant une façon simple de personnaliser le comportement du système en fonction des besoins spécifiques. Cela simplifie également les ajustements pour s'adapter à des scénarios particuliers sans nécessiter de modifications directes du code source.

Cette fois, notre attention se porte sur les scripts Python du dossier src.

utils.py représente une collection de fonctions utilisées par d'autres scripts, réduisant ainsi la redondance dans le code. Voici la documentation des différentes fonctions :

utils.load_json(nom_fichier)

Paramètre: nom_fichier : *string*
Nom du json à charger

Returns: donnees : *dict*
Les données chargées à partir du fichier JSON.

utils.save_json(nom_fichier, liste)

Paramètre: nom_fichier : *string*
Nom du json à sauvegarder
liste : *list*
Liste de données à sauvegarder.

utils.Rx_accel_data()

Returns: (acc_x, acc_y, acc_z) : *(float, float, float)*
Données d'accélération.

utils.histogramme(data_x, label=None)

Paramètre: data_x: *list*
Données en entrée.
label: *string*
Étiquette associée aux données.

Returns: hist : *np.array*
Histogramme des données.

generate_data.py a pour objectif de récupérer des données et de les sauvegarder dans le dossier data/, en les associant au label d'intérêt. Ceci facilite l'entraînement du modèle. Voici son utilisation et les arguments disponibles:

```
usage: generate_data.py [-h] [--filename FILENAME] [--label LABEL] [--index
INDEX]

Script de collecte de données et de prédiction avec modèle.

optional arguments:
    -h, --help            montre le message d'aide puis exit.
    --filename FILENAME   nom du fichier pour sauvegarder les données.
    --label LABEL         dossier pour sauvegarder les données.
    --index INDEX         indice de départ pour les fichiers.
```

train_data.py joue un rôle essentiel dans l'entraînement du modèle en utilisant l'ensemble des données du dossier data/. Voici la documentation de la fonction associée :

train_data.train_model()

Returns: modele : *RandomForestClassifier*
Modèle entraîné.

main.py a pour objectif de déterminer en temps réel l'état du capteur. Voici son utilisation et les arguments disponibles:

```
usage: main.py [-h] [--plot] [--val]

Script de collecte de données et de prédiction avec modèle.

optional arguments:
  -h, --help        montre le message d'aide puis exit.
  --plot            activer le tracé des données.
  --val             activer le print des données.
```

V - Amélioration

- Communication bluetooth
- Intégrer le modèle dans le micro contrôleur
- Ajout de donnée/plus de label
- Intégrer dans une voiture miniature avec une led RGB

VI - References

- https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
- https://www.tensorflow.org/?hl=fr
- https://www.st.com/resource/en/datasheet/stm
- https://www.ibm.com/fr-fr/topics/random-forest