
ECOLE
POLYTECHNIQUE
DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES

INGÉNIEUR CIVIL BIOMÉDICAL

---

# GRAPHICAL USER INTERFACE DIDACTIC PALPATION DEVICE

---

**Active Medical Devices**
**ELEC-H424**

**Auteurs :**
Simon DEN HEANE
Thomas NINANE

**Date :**
16 Mai 2020

**Professeur :**
Antoine Nonclerq

**Superviseur :**
Adrien Debelle

# Table des matières

# 1 Introduction

La conception d'un appareil de palpation pour nodules pulmonaires est un projet entrepris conjointement par les services du BEAMS et du SAAS. L'objectif de ce dernier serait de remplacer les cours d'*Instrumentation* et d'*Automatique* actuellement donnés en BA3. Cela permettrait en effet aux futurs étudiants d'acquérir les mêmes compétences, mais de manière beaucoup plus didactique.

Dans le cadre du cours d'*Active Medical Devices*, il a donc été demandé de collaborer à ce projet en développant une interface graphique. Celle-ci permettrait de faire le lien entre les étudiants et l'appareil de palpation, en affichant notamment les données mesurées par les différents capteurs sous forme de graphiques, en offrant la possibilité d'enregistrer ces résultats dans un fichier, etc.

Au cours de l'année passée, une première ébauche d'interface graphique avait été mise en place par Andréa Thibaud durant son stage. Il s'agissait en réalité d'un fichier Excel permettant de faire l'affichage des données envoyées depuis la carte Arduino. Cette solution n'étant pas optimale, il a été décidé après discussion avec le superviseur du projet de réaliser l'interface graphique grâce à Python.

Il est important de noter que ce rapport ne suit pas exactement la structure d'un rapport scientifique classique. Celui-ci a en effet été écrit avec l'intention de servir de guide afin de permettre aux futurs étudiants de comprendre le code informatique et le fonctionnement de l'interface graphique avec plus de facilité.

Le code Python est disponible à cette adresse : `https://github.com/thomasninane/didactic_palpation_device`.

Le code LaTeX de ce document a été ajouté au projet GitHub afin que les explications concernant le code puissent être mises à jour en fonction de l'évolution du projet.

La section 3 explique le code Python qui permet de créer l'interface graphique. Afin de permettre une compréhension maximale, il faut noter que chaque méthode a aussi été commentée directement dans le code.

## 2 Cahier des charges

Afin de mieux visualiser les fonctionnalités devant être implémentées, un cahier des charges a été réalisé. Il est repris ci-dessous :

1. Présence de commentaires dans le code afin d'en faciliter la compréhension.
2. L'interface doit être intuitif et facile d'utilisation pour les étudiants qui s'en serviront durant les prochaines années.
3. Les courbes à afficher sont : commande, position, vitesse et force.
4. Les courbes doivent être affichées sur des graphiques différents.
5. Un même graphique doit pouvoir afficher 2 courbes : celle de la partie maître et celle de la partie esclave (exemple : commande maître et commande esclave sur le graphique "commande").
6. Explorer les limites de l'Arduino et de l'interface graphique concernant l'affichage et le rapatriement des données sur l'ordinateur. Idéalement, on voudrait acquérir les données toutes les millisecondes et actualiser l'interface toutes les 10 ou 100 millisecondes.
7. Pour l'affichage en temps **non-réel** :
   (a) Présence d'un bouton *SELECT FILE* permettant de choisir le fichier source (fichier *.txt*).
   (b) Implémentation d'une fonctionnalité permettant de générer les graphiques grâce à ce fichier source.
   (c) Présence de boutons permettant d'afficher ou de cacher les courbes souhaitées (exemple : on veut parfois uniquement se concentrer sur la partie maître et cacher les courbes de la partie esclave).
8. Pour l'affichage en temps **réel** :
   (a) Implémentation d'une fonctionnalité permettant d'acquérir de nouvelles données, de tracer les graphes en temps réel, de sauvegarder ces graphes et de sauvegarder les données acquises.
   (b) Ces nouvelles données sont envoyées par l'Arduino.
   (c) La communication entre l'Arduino et l'interface graphique doit se faire via un port série (port *COM*).
   (d) Implémentation de 3 cases d'input (valeur basse, valeur haute) afin de pouvoir appliquer un échelon sur la commande ainsi que la fréquence d'acquisition.
   (e) Implémentation d'un bouton *GO* qui enverrait les données des 3 cases d'input à l'Arduino, qui lancerait automatiquement l'acquisition par l'Arduino et l'affichage en temps réel des graphes.
   (f) Présence d'un bouton *STOP* qui envoie une instruction à l'Arduino lui demandant d'arrêter l'acquisition et l'envoi de nouvelles données.

(g) Présence d'un bouton *EXPORT* permettant d'exporter les données acquises sous forme de fichier *.txt*.

(h) Bouton *RECALL* permettant de relancer l'acquisition après que celle-ci ait été arrêtée sans perdre les données acquises préalablement.

9. Implémentation d'un message d'avertissement lorsqu'on clique sur la bouton "fermer le programme" indiquant à l'utilisateur que la fermeture de celui-ci entraîne la perte des acquisitions non sauvegardées.

10. Fonctionnalités supplémentaires :

    (a) Superposition de courbes sur le même graphe (force et position par exemple).

    (b) Convertir la commande en degrés

    (c) Possibilité de faire un zoom sur les graphiques.

    (d) Faire des vérifications avant d'envoyer les 3 valeurs des cases d'input (commande doit être entre 0 et 4095).

    (e) Il est possible que les données doivent être sauvegardées sur le microcontrôleur et que celles-ci soient envoyées lorsque l'acquisition est arrêtée. Il faut donc s'assurer que l'interface arrive à gérer cette opération.

# 3 Explication du code informatique

Le code informatique contient 7 fichiers Python. Ils sont cités ci-dessous et leur rôle est détaillé dans les sous-sections ci-après.

1. **main.py**
2. **MainPage.py**
3. **DrawPlotsParent.py**
4. **DrawPlotsFromFile.py**
5. **DrawPlotsRealTime.py**
6. **DataOutputWindow.py**
7. **GlobalConfig.py**

## 3.1 main.py

Ce fichier crée la fenêtre principale grâce au module *tkinter*. Elle contient donc la fonction *main* ainsi qu'une classe : *GUI*. Cette classe possède un attribut *frames* qui est un dictionnaire. Celui-ci va contenir les différentes pages qui sont créées dans la fenêtre principale.

Le constructeur crée 3 pages : *MainPage*, *DrawPlotsRealTime* et *DrawPlotsFromFile*. Cependant, lors du lancement de l'application, seule la page *MainPage* est affichée. Il faut donc bien comprendre la différence entre les fenêtres et les pages.

On y voit aussi la présence de la méthode *on_closing()* qui permet d'avoir une fenêtre d'avertissement lorsque l'utilisateur désire fermer le programme.

## 3.2 MainPage.py

Cette classe constitue une page. Elle hérite donc de *tkinter.Frame* et on y retrouve 3 éléments :

1. Le nom de la page : "MAIN PAGE".
2. Un bouton permettant d'accéder à la page "Draw Plots Real Time".
3. Un bouton permettant d'accéder à la page "Draw Plots From File".

**Note concernant le constructeur :** On y trouve un paramètre qui s'appelle *controller*. Il s'agit de la classe "GUI" car celle-ci contient la méthode permettant de changer la page qui est affichée dans la fenêtre principale (pour rappel, 3 pages sont créées lors du lancement du programme mais elles ne sont jamais affichées simultanément).

## 3.3 DrawPlotsParent.py

Ce fichier est utile afin d'éviter de réécrire du code qui est commun à *DrawPlotsRealTime* et *DrawPlotsFromFile*. Il va donc permettre d'implémenter de l'héritage car les 2 pages citées précédemment se ressemblent fortement. *DrawPlotsRealTime* et *DrawPlotsFromFile* héritent des

attributs et méthodes de *DrawPlotsParent*.

Avant de détailler le contenu de ce fichier, il est important d'acquérir certains concepts de base du module *tkinter*. Ce module permet de générer des interfaces graphiques et est doté des éléments suivants :

1. **LabelFrame :** permet de créer un cadre vide. Ce cadre peut contenir un nom ou ne pas être nommé et ses bords peuvent être rendus invisibles. L'utilisation de cadres permet de faciliter le placement des différents éléments au sein d'une même page.
2. **Button :** permet de créer un bouton, de lui donner un nom et de lier ce bouton à une commande/action.
3. **Entry :** permet de créer un champ de texte afin de récupérer des informations entrées par l'utilisateur.
4. **CheckButton :** permet de créer une boite à cocher. Ce *CheckButton* doit être lié à une variable de type *IntVar()* qui vaut 0 ou 1 selon que la case soit cochée ou non. Lorsqu'il est coché ou décoché, il est possible d'exécuter une action.
5. **Label :** permet de créer une ligne de texte qui peut être affichée sur l'interface.

Une fois l'élément créé, il faut ensuite l'afficher. Ceci peut se faire grâce à la méthode *pack()* ou *grid(row, column)*. Il est important de noter que ces deux méthodes ne peuvent pas être utilisées simultanément au sein d'un même *LabelFrame*. La première méthode va simplement placer les éléments l'un en-dessous de l'autre, alors que la seconde permet de déterminer plus précisément l'endroit où l'on désire rajouter l'élément.

### 3.3.1 Méthodes statiques

Les méthodes statiques sont celles qui ne nécessitent pas l'existence d'un objet pour être exécutées. En python, cela revient à ne jamais avoir besoin d'un élément (attribut ou méthode) commençant par "self." car ce mot clé est équivalent à "this." en Java (voir cours INFO-H2001). Ce fichier contient 2 méthodes statiques :

1. **convert_position_to_degrees :** la position envoyée par l'Arduino peut être convertie en degrés grâce à cette fonction afin de mieux visualiser à quoi cette mesure correspond.
2. **convert_command_to_amps :** la commande envoyée par l'Arduino est une valeur comprise entre 0 (correspondant à -2A) et 4095 (correspondant à 2A). Si l'étudiant le désire, il peut afficher la commande en ampères grâce à cette fonction.

### 3.3.2 Constructeur

Le constructeur contient tous les éléments communs aux deux pages qui sont utilisés afin d'afficher les graphes en temps réel ou non. Ces éléments sont détaillés ci-dessous :

1. **Upper Frame :** ce cadre est utile afin d'afficher le titre de la page et d'avoir un bouton "retour" qui permet de retourner à la page principale.

2. **Main Frame :** ce cadre est l'élément principal de la page. Il contient 2 éléments : un grand cadre (*Figure Frame*) qui lui-même contient une figure à 4 graphes (4 "subplots") et le cadre de droite (*Right Side Frame*).
3. **Right Side Frame :** ce cadre se situe à droite des graphiques. Il contient toutes les options qui permettent d'interagir avec les graphes et de manipuler les données. On y retrouve plusieurs sous-cadres :
   (a) **Window Specific Frame :** c'est ce cadre qui va contenir les options qui ne sont pas communes entre *DrawPlotsRealTime* et *DrawPlotsFromFile*.
   (b) **Output Window Button :** permet de créer une nouvelle fenêtre afin d'afficher les données contenues dans le fichier texte ou celles qui ont été acquises grâce à l'Arduino.
   (c) **Plots Options Frame :** ce cadre va contenir 4 sous-cadres. Chaque sous-cadre contient les options spécifiques de la courbe à laquelle il se rapporte (vitesse, position, commande et force). Par exemple, pour la courbe de la commande, on a 3 options : afficher la commande en ampères, afficher les données de la partie maître et afficher les données de la partie esclave.

Voici, ci-dessous, un schéma pouvant être utile à la visualisation des différents éléments de la page et à leur positionnement sur l'interface graphique :

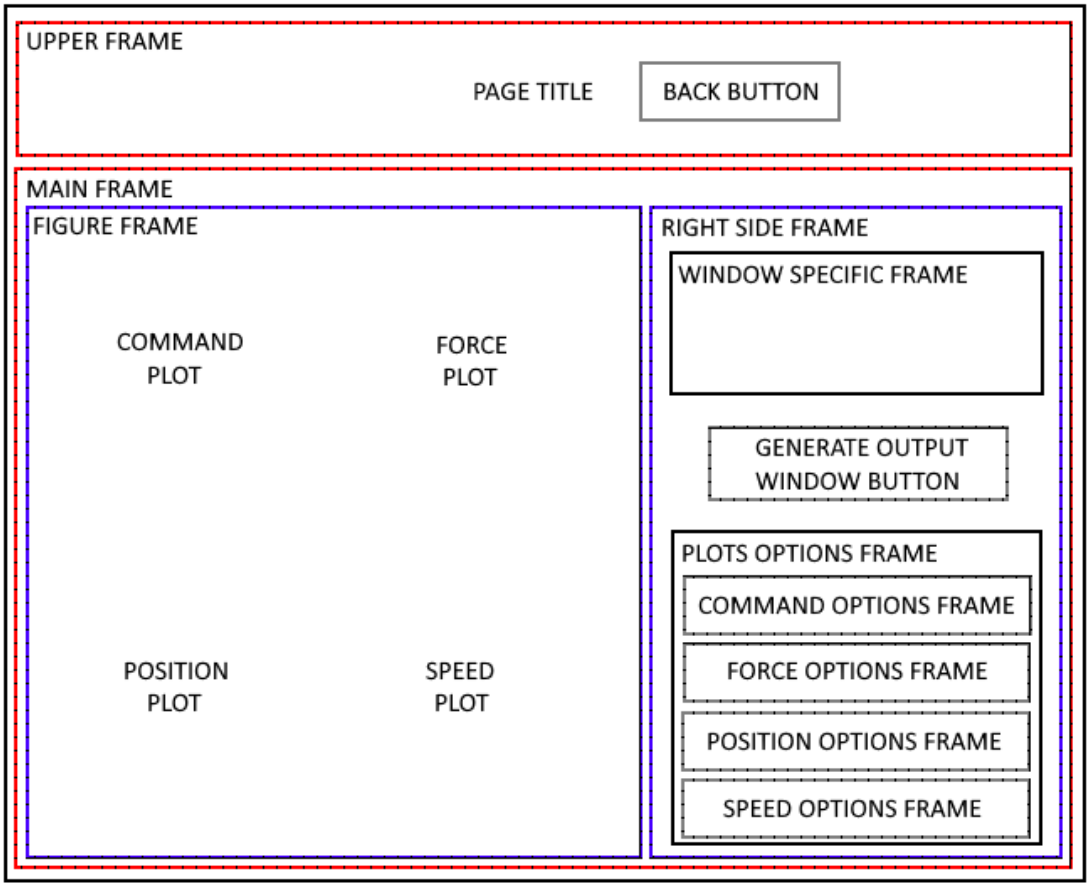


FIGURE 1 – Schéma de la page *DrawPlotsParent*

Il faut aussi noter qu'il est de bonne pratique d'initialiser tous les attributs dans le constructeur. Ceux-ci doivent donc être initialisés à *None*. Les 2 figures ci-dessous aident à comprendre cette procédure :

```
self.saveRecordingLabelFrame = tk.LabelFrame(self.rightSideLabelFrame, text="SAVE RECORDING", pady=10)
self.saveRecordingLabelFrame.grid(row=1, column=0)

self.filenameEntry = None
self.saveFileButton = None
self.fill_save_recording_label_frame()
```

FIGURE 2 – Code dans le constructeur

```
def fill_save_recording_label_frame(self):
    tk.Label(self.saveRecordingLabelFrame, text="FILENAME:").grid(row=0, column=0, padx=10)

    self.filenameEntry = tk.Entry(self.saveRecordingLabelFrame, borderwidth=3, width=40)
    self.filenameEntry.grid(row=0, column=1)
    self.filenameEntry.insert(0, "Data")

    self.saveFileButton = tk.Button(self.saveRecordingLabelFrame, text='SAVE FILE',
                                    width=10, height=1,
                                    state=tk.DISABLED,
                                    command=lambda: self.save_data_as_txt())
    self.saveFileButton.grid(row=0, column=2, padx=10)
```

FIGURE 3 – Méthode attribuant des objets aux attributs initialisés à "None" dans le constructeur. Le contenu de cette méthode n'est pas écrit dans le constructeur afin d'en alléger l'écriture.

### 3.3.3 Méthodes

On distingue 2 catégories de méthodes : celles qui servent à ajouter des éléments graphiques dans les cadres (boutons, cases à cocher, etc.) et celles qui permettent d'effectuer des tâches ou des actions lorsque l'utilisateur clique sur un élément de l'interface.

Les méthodes appartenant à la première catégorie sont utiles afin d'alléger le constructeur et de compartimenter les différentes étapes réalisées lors de la création des pages *DrawPlotsRealTime* et *DrawPlotsFromFile*. Toutes les méthodes présentes dans *DrawPlotsParent* sont reprises ci-dessous :

1. **fill_upper_frame :** Remplit *Upper Frame* avec le titre de la page et le bouton "retour" permettant de retourner au menu principal.
2. **fill_figure_frame :** Remplit *Figure Frame* avec une figure contenant 4 graphiques. Ce cadre est positionné à gauche et au centre de *Main Frame*.
   On y trouve les méthodes *FigureCanvasTkAgg()*, *get_tk_widget()* et *draw()* qui sont des méthodes spécifiques à l'affichage de figures dans une interface graphique avec le module *tkinter*.
3. **fill_plots_options_frame :** Remplit *Plots Options Frame*. On y retrouve une boucle qui va générer un cadre pour chaque type de graphique (commande, force, position et vitesse). Cette méthode réalise les opérations suivantes :
   (a) Création d'une zone de texte afin de donner un nom aux graphiques lors de leur sauvegarde sur le disque dur de l'ordinateur.
   (b) Création d'un bouton engendrant la sauvegarde du graphique. Ce bouton est donc lié à la commande *self.save_plot()*. Lors de l'assignation de cette commande, on observe la méthode *partial(function, function_parameters)*. **Elle permet d'éviter un bug de l'interface lié à la boucle "for". Il faut donc éviter de modifier cette ligne**.

(c) Création des cases à cocher en appelant la méthode *create_check_button()*. Celles-ci dépendent du type de graphe ce qui explique les nombreuses conditions "if". Par exemple, il n'y a pas de force pour la partie maître et il ne faut donc pas créer de case, il faut créer une case supplémentaire pour la commande afin de la convertir en ampères, etc.

4. **create_check_button :** Cette méthode crée une case à cocher. La case qui est créée est liée à la commande *refresh_all_plots()* si l'affichage n'est pas en temps réel. Cette commande va actualiser les graphes afin d'afficher ce que l'utilisateur désire (par exemple, si la case convertissant la commande en ampères est cochée, il faut que le graphique de la commande soit actualisé pour prendre en compte ce changement). A l'inverse, si l'affichage est en temps réel, il ne faut pas lier la case à une commande car les graphiques sont rafraîchis automatiquement.
5. **clear_all_plots :** permet d'effacer toutes les courbes affichées sur les graphiques en conservant les noms des axes et les titres.
6. **refresh_all_plots :** méthode qui est redéfinie dans les classes filles mais qui doit être définie dans la classe mère afin d'éviter une erreur de type "method out of scope".
7. **activate_or_deactivate_save_plot_buttons :** permet d'activer ou de désactiver tous les boutons liés à la sauvegarde des graphiques.
8. **generate_data_output_window :** crée une nouvelle fenêtre qui affiche les données acquises via l'Arduino ou celles du fichier texte.
9. **destroy_data_output_window :** ferme la fenêtre qui affiche les données. Cette fonction est utile lorsqu'un nouveau fichier texte est sélectionné ou qu'une nouvelle acquisition est démarrée.
10. **add_date_to_save_name_entries :** lorsque l'utilisateur désire sauvegarder des données sur le disque dur de l'ordinateur, il est important de rajouter la date et l'heure afin d'éviter que ces nouvelles données écrasent des fichiers déjà enregistrés préalablement. Cette opération est réalisée grâce à cette méthode.
11. **save_plot :** cette fonction permet de démarrer la sauvegarde d'un graphique. Elle appelle une des deux méthodes ci-dessous.
12. **save_plot_normal_axis :** permet la sauvegarde d'un graphe tout en respectant le choix de l'utilisateur d'afficher ou de cacher une des courbes (maître ou esclave).
13. **save_plot_special_axis :** permet la sauvegarde de la commande en ampères ou de la position en degrés tout en respectant le choix de l'utilisateur d'afficher ou de cacher une des courbes (maître ou esclave).

### 3.4 DrawPlotsFromFile.py

Ce fichier permet de tracer les graphes à partir d'un fichier texte. Il hérite des méthodes et attributs de la classe *DrawPlotsParent*. Ce choix permet de fortement simplifier le constructeur de cette classe ainsi que les méthodes qui doivent y être écrites.

### 3.4.1 Méthodes statiques

Ce fichier contient une seule méthode statique : *create_data_frame()*. Lorsqu'un fichier texte est sélectionné, il faut rajouter les données qu'il contient à un *data frame* (df) car c'est celui-ci qui est utilisé pour tracer et sauvegarder les courbes.

### 3.4.2 Constructeur

Le constructeur appelle la méthode *super().___init___()* qui permet l'utilisation du constructeur de la classe mère. Il faut ensuite remplir *Window Specific Frame* des éléments graphiques qui sont spécifiques à la classe fille.

Le seul bouton qui doit être rajouté est un bouton permettant de sélectionner un fichier texte sur le disque dur de l'ordinateur. Afin d'éviter toute ambiguïté, lorsqu'un fichier est sélectionné, il est important d'afficher son nom. Ce bouton et le nom du fichier sont rajoutés à *File Selection Frame* qui lui-même est ajouté à *Window Specific Frame*.

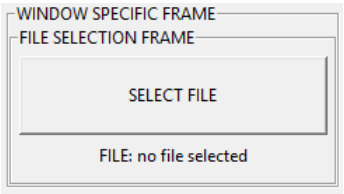


FIGURE 4 – DrawPlotsFromFile - WindowSpecificFrame

### 3.4.3 Méthodes

Cette classe utilise 3 méthodes qui lui sont spécifiques :

1. **fill_file_selection_frame :** remplit *File Selection Frame* afin d'alléger le constructeur.
2. **import_recording :** permet d'importer un fichier texte. Lorsque le fichier est sélectionné, le *data frame* va être créé grâce à la méthode statique *create_data_frame()* et utilisé afin d'afficher les courbes sur les 4 graphiques.
3. **refresh_all_plots :** permet de rafraîchir les graphiques en prenant en compte les options cochées par l'utilisateur. Lorsque les graphiques ont été actualisés, la méthode va activer les boutons permettant leur sauvegarde. Cette méthode est appelée dès qu'un fichier texte a été choisi ou dès que l'état d'une case à cocher est modifié.

### 3.4.4 Capture d'écran

La figure ci-dessous montre à quoi ressemble la page *DrawPlotsFromFile* (note : afin de désengorger l'interface graphique, les bords et les noms de certains cadres ont été rendus invisibles).

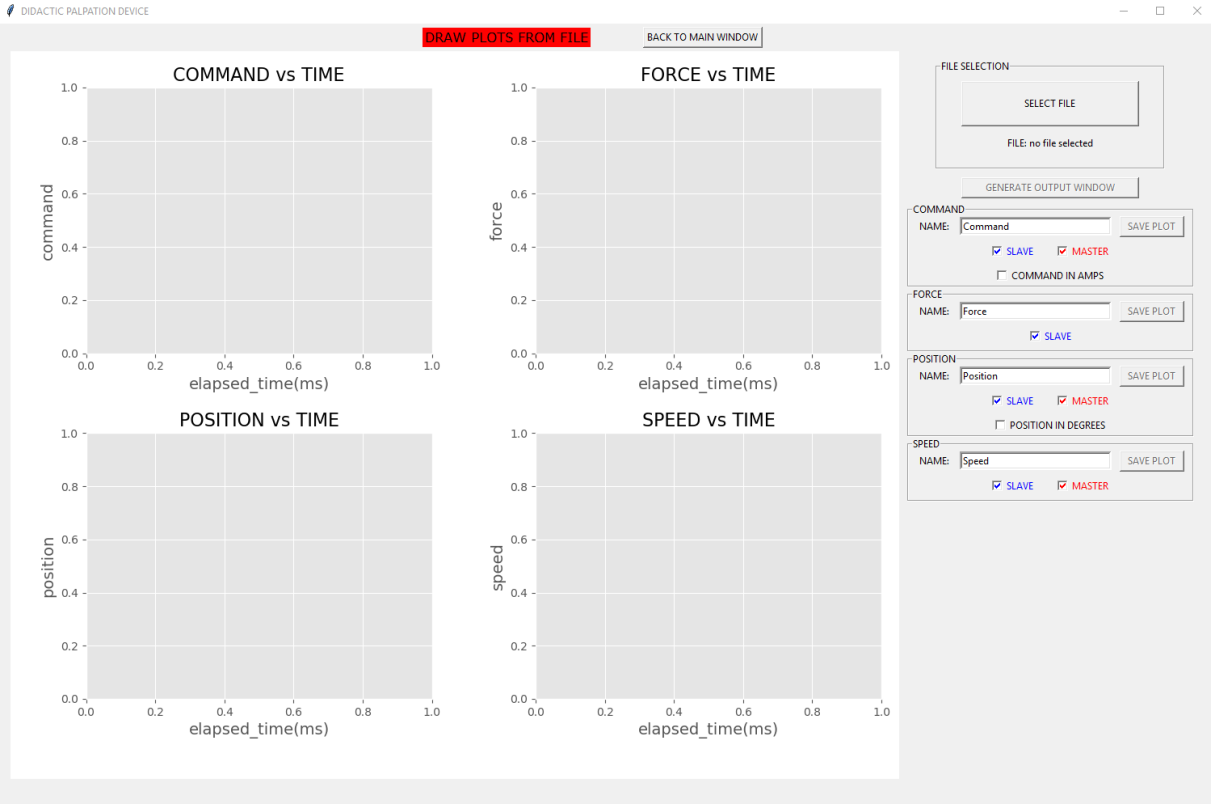


FIGURE 5 – Capture d'écran : DrawPlotsFromFile

### 3.5 DrawPlotsRealTime.py

Ce fichier permet de tracer les graphes à partir des données qui sont envoyées par l'Arduino. Il permet aussi de sauvegarder toutes les données reçues sous forme de fichier texte. Tout comme *DrawPlotsFromFile*, cette classe hérite des attributs et des méthodes de *DrawPlotsParent*.

Cette fenêtre peut fonctionner de 2 manières :

1. Si un Arduino est connecté à l'ordinateur, celui-ci envoie les données au programme et l'interface les affiche sur les graphes en temps réel.
2. Dans le cas où un Arduino n'est pas connecté, il est quand même possible de simuler une acquisition de données en temps réel. Ceci peut-être utile si l'utilisateur n'a pas accès au microcontrôleur mais qu'il désire quand même expérimenter avec l'interface graphique.

**Note concernant le microcontrôleur :** Il ne faut pas oublier de modifier le paramètre *COMMUNICATION_PORT* qui se situe dans le fichier *GlobalConfig*. De plus, l'Arduino doit être connecté à l'ordinateur avant le lancement du programme.

### 3.5.1 Méthodes statiques

Ce fichier contient une seule méthode statique : *load_simulation_file()* qui est utile afin de simuler une acquisition de données. Elle charge un fichier texte qui à la même structure que lorsque c'est le microcontrôleur qui envoie les données. Ces données vont ensuite être utilisées par la méthode *simulate_real_time_data_acquisition()*. Le fichier qui est utilisé pour cette simulation est défini par *SIMULATE_DATA_ACQUISITION_FILE* qui se situe dans *GlobalConfig*.

### 3.5.2 Constructeur

Tout comme pour *DrawPlotsFromFile*, le constructeur appelle la méthode *super().___init___()* et il faut ensuite remplir *Window Specific Frame* avec les éléments qui sont spécifiques à *DrawPlotsRealTime*. Ceux-ci sont décrits ci-dessous :

1. Un menu déroulant permettant de choisir si l'acquisition se fait grâce à un Arduino ou si elle est simulée à partir d'un fichier texte pré-existant (note : si le microcontrôleur n'est pas connecté ou que le port COM est mal configuré, seule l'option "simulation" est visible).
2. **Acquisition Parameters Frame :** ce cadre contient tous les éléments qui sont nécessaires au lancement d'une nouvelle acquisition par le microcontrôleur.
3. **Save Recording Frame :** ce cadre contient tous les éléments qui sont nécessaires à la sauvegarde des données reçues sous forme de fichier texte.
4. **Refresh Plots Button :** lorsque l'acquisition est finie, le rafraîchissement des graphes ne se fait plus automatiquement. Par conséquent, si l'utilisateur désire modifier l'affichage (cacher une courbe ou alors modifier les unités pour la commande ou la position), il faut rafraîchir manuellement les graphes.

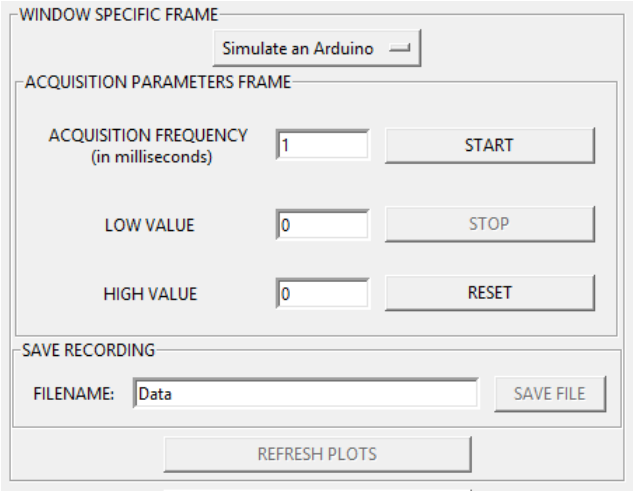


FIGURE 6 – DrawPlotsRealTime - WindowSpecificFrame

### 3.5.3 Méthodes

Les méthodes utilisées par cette classe sont reprises ci-dessous :

1. **fill_acquisition_parametersl_frame :** remplit *Acquisition Parameters Frame*. On y retrouve les éléments suivants :
   (a) L'intervalle (en millisecondes) entre 2 mesures acquises via l'Arduino.
   (b) La valeur basse de la commande.
   (c) La valeur haute de la commande.
   (d) Un bouton "start" qui appelle la méthode *start_recording()*.
   (e) Un bouton "stop" qui appelle la méthode *stop_recording()*.
   (f) Un bouton "reset" qui appelle la méthode *reset_recording()*.
2. **fill_save_recording_frame :** remplit *Save Recording Frame*. Permet d'enregistrer les données acquises sous forme de fichier texte tout en nommant ce fichier selon les désirs de l'utilisateur.
3. **start_recording :** cette méthode peut soit transmettre les valeurs utiles à l'Arduino (intervalle d'acquisition, valeur basse et valeur haute de la commande) et lui ordonner de commencer l'acquisition, soit charger un fichier qui va être utilisé afin de simuler une acquisition de données sans microcontrôleur via l'initialisation d'un Thread.
4. **stop_recording :** si l'acquisition se fait grâce au microcontrôleur, celui-ci reçoit une commande d'arrêt. Sinon, le Thread qui simule l'arrivée périodique de données est arrêté. De plus, le rafraîchissement automatique des graphes est interrompu.
5. **reset_recording :** cette méthode est utile afin de permettre une nouvelle acquisition en supprimant les données reçues, en effaçant les courbes sur les graphes et en activant ou désactivant certains boutons.
6. **save_data_as_txt :** permet de sauvegarder les données reçues sur le disque dur de l'ordinateur.
7. **get_acquisition_parameters :** permet de récupérer les paramètres d'acquisition qui ont été entrés par l'utilisateur. Cette méthode vérifie que ces valeurs sont correctes et, le cas échéant, donne un message d'erreur.
8. **send_acquisition_parameters_to_arduino :** envoie les paramètres d'acquisition vers le microcontrôleur. Voici un exemple d'envoi : "c10-1000-3000-1".
   (a) La lettre "c" est utile afin de signifier qu'un ordre est envoyé (commande). Toute information qui est envoyée vers l'Arduino et qui ne commence pas par la lettre "c" est ignorée.
   (b) 10 représente la fréquence d'acquisition.
   (c) 1000 représente la valeur basse de la commande.
   (d) 3000 représente la valeur haute de la commande.
   (e) 1 ordonne au microcontrôleur de démarrer l'acquisition.
9. **simulate_real_time_data_acquisition :** les données chargées par *load_simulation_file()* sont stockées dans *self.simulation_data*. Cette méthode tourne dans un Thread et va copier

une ligne de *self.simulation_data* (la ligne dont l'index correspond à *self.simulation_step*) pour ajouter les valeurs aux listes qui contiennent les données acquises.

10. **real_time_data_acquisition :** cette méthode permet l'acquisition des données provenant de l'Arduino. Elle contient une boucle qui est exécutée tant que l'acquisition est en cours. On commence par lire le port série et séparer les différentes données envoyées (les valeurs envoyées vers l'ordinateur sont séparées par le caractère ";"). On rajoute ensuite les données à la liste adéquate (afin de mieux comprendre les opérations, il est utile de lire la section 3.5.5).
11. **create_data_frame :** permet de créer un *data frame* à partir des données acquises via l'Arduino. Ce *data frame* est ensuite utilisé pour sauvegarder les graphiques et pour sauvegarder ces données sous forme de fichier texte.
12. **refresh_all_plots :** permet de rafraîchir les graphiques en prenant en compte les options cochées par l'utilisateur. Lorsque l'acquisition est terminée, la méthode va activer les boutons permettant leur sauvegarde. Cette méthode est appelée dès que l'utilisateur appuie sur le bouton *Start* et est ensuite exécutée automatiquement toutes les 100 millisecondes (voir *PLOTTING_FREQUENCY* dans *GlobalConfig*).

### 3.5.4 Capture d'écran

La figure ci-dessous montre à quoi ressemble la page *DrawPlotsRealTime* (note : afin de désengorger l'interface graphique, les bords et les noms de certains cadres ont été rendus invisibles).

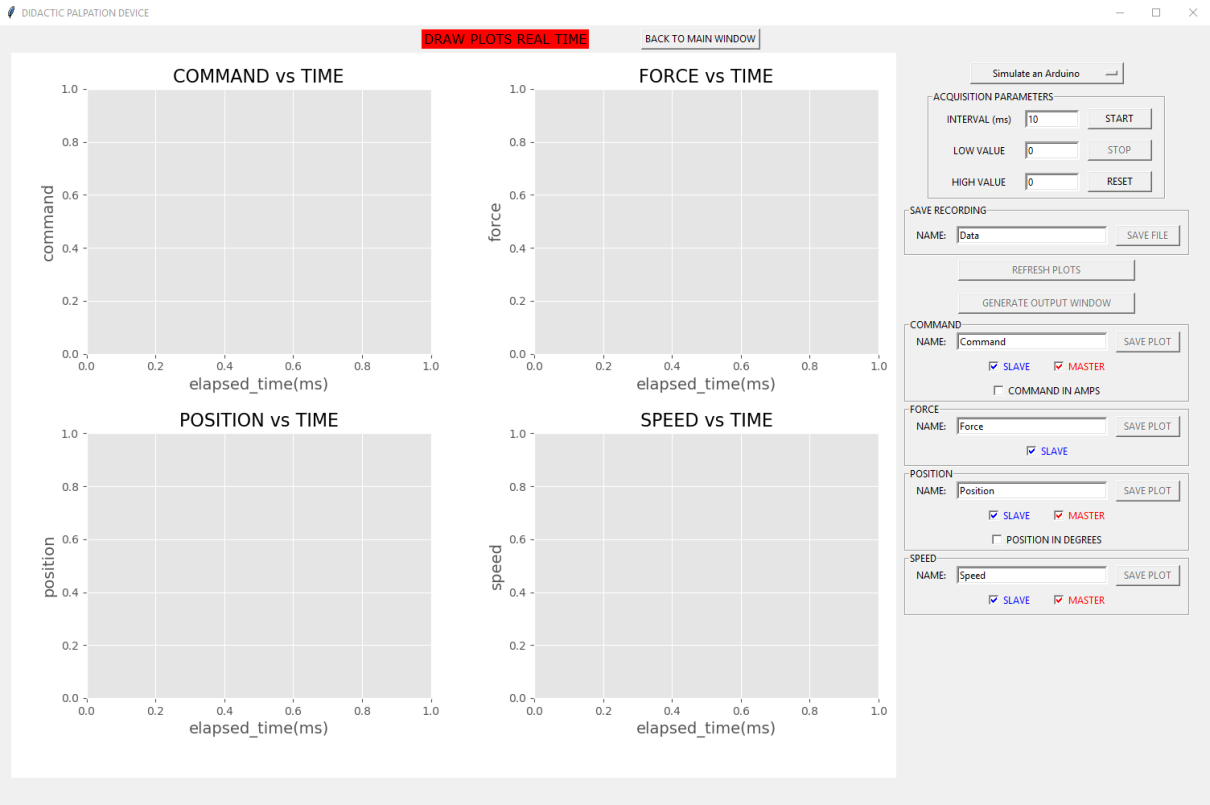

FIGURE 7 – Capture d'écran : DrawPlotsRealTime

### 3.5.5 Remarque concernant l'acquisition en temps réel

Lors de l'affichage des graphiques via le fichier texte, les données sont directement ajoutées à un *data frame* car l'utilisation de celui-ci facilite la manipulation des colonnes (et donc l'affichage des graphiques ainsi que la sauvegarde de ceux-ci).

Cependant, lors de l'acquisition en temps réel, les données arrivent extrêmement rapidement et l'ajout d'une ligne à un *data frame* existant n'est pas efficace du tout car cette méthode nécessite de copier l'entièreté du *data frame* (l'opération prend parfois plusieurs millisecondes). Ceci provoque l'accumulation des données dans le *buffer* du microcontrôleur, ce qui à 2 conséquences critiques. La première est que les données affichées par le programme ne sont plus des données en temps réel et la seconde est que lorsque ce *buffer* est plein, il est automatiquement vidé ce qui provoque une perte énorme de données (à peu près 2 secondes de données sont perdues toutes les 6 secondes pour une fréquence d'acquisition de 10 millisecondes).

Afin de palier à ce problème, la procédure suivante est utilisée : un dictionnaire est créé et celui-ci contient 15 listes qui sont initialement vides. Les clés de ce dictionnaire sont les noms des colonnes (voir *DATA_FRAME_COLUMNS* dans *GlobalConfig*). Lorsqu'une nouvelle ligne de données est envoyée vers le programme, chaque élément est ajouté à la liste à laquelle il correspond car l'ajout d'un élément à une liste est extrêmement rapide.

Cette procédure implique 2 changements :

1. La méthode *refresh_all_plots()* doit utiliser les listes du dictionnaire plutôt que les colonnes du *data frame* afin d'afficher les graphiques en temps réel (voir les différences entre la méthode de *DrawPlotsFromFile* et celle de *DrawPlotsRealTime*).
2. Une fois l'acquisition de données terminée, il est nécessaire de rajouter toutes les colonnes du dictionnaire à un *data frame* afin de permettre la sauvegarde des graphiques et des données sur l'ordinateur.

## 3.6 DataOutputWindow.py

Cette classe constitue une nouvelle fenêtre qui est appelée lorsque l'utilisateur clique sur le bouton *Generate Output Window*. Elle permet d'afficher les données acquises via l'Arduino ou alors celles qui se situent dans un fichier texte.

Afin d'accéder aux attributs nécessaires à l'affichage des données, il faut avoir accès à *DrawPlotsParent*. Ceci se fait grâce au paramètre *parent* qui se situe dans le constructeur.

Comme l'entièreté des données ne peut être affichée sur l'écran (dans la direction horizontale), une barre de défilement horizontale a été rajoutée. De plus, si les données proviennent d'un

fichier texte, le nom de ce fichier est affiché en haut de la fenêtre.

### 3.7 GlobalConfig.py

Ce fichier est utile afin de définir certains paramètres du programme sans devoir explorer l'ensemble des différents fichiers du projet. Voici quelques paramètres qui y sont définit :

1. **APP_GEOMETRY :** définit la dimension de la fenêtre principale.
2. **OUPUT_WINDOW_GEOMETRY :** définit la dimension de la fenêtre qui affiche les données acquises via l'Arduino ou celles contenues dans un fichier texte.
3. **FIGURE_X et FIGURE_Y :** les dimensions de la figure qui contient les 4 graphes.
4. **COMMUNICATION_PORT :** le port *COM* qui est en contact avec l'Arduino. Cette information peut être trouvée en allant dans le gestionnaire de périphériques (voir figure 8 ci-dessous).
5. **BAUDRATE :** La vitesse de transmission des données entre l'Arduino et l'ordinateur. Cette valeur doit être identique dans le code Python et le code Arduino.
6. **SIMULATE_DATA_ACQUISITION_FILE :** il est possible de simuler une acquisition en temps réel sans Arduino. Celle-ci se fait à partir d'un fichier texte pré-existant et son chemin est défini ici.
7. **SIMULATE_DATA_ACQUISITION_INTERVAL :** l'intervalle entre l'ajout de 2 lignes de données lorsque l'acquisition en temps réel est simulée via un fichier texte.
8. **PLOTTING_FREQUENCY :** l'intervalle entre 2 actualisations des graphiques lors de l'affichage en temps réel (en millisecondes).
9. **DATA_FRAME_COLUMNS :** les noms des grandeurs qui sont envoyées par l'Arduino. Il faut faire attention car une modification de cet élément a des conséquences sur l'ensemble du code Python et Arduino.
10. **PLOT_TYPES :** les noms des 4 types de graphes : commande, force, position et vitesse.

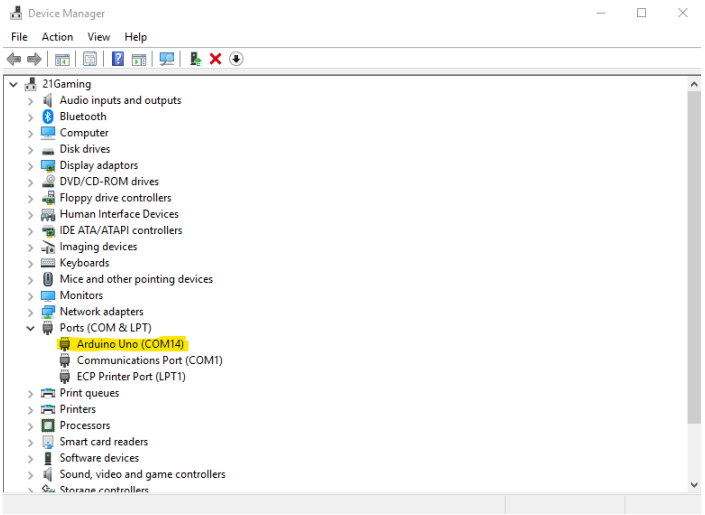


FIGURE 8 – Port COM

# 4 Modifications du code de l'Arduino

Comme nous n'avons pas accès au code Arduino qui a été utilisé afin de générer les premiers fichiers *.txt* (voir dossier "src/old"), nous recommandons les modifications suivantes :

1. Envoyer toutes les grandeurs de temps en millisecondes et non en microsecondes.
2. Rajouter les lignes de codes nécessaires à la réception des paramètres d'acquisitions provenant de Python.
3. Rajouter le temps écoulé depuis la première mesure.
4. S'assurer que les données sont envoyées dans l'ordre correct (voir *DATA_FRAME_COLUMNS* dans *GlobalConfig*). C'est à dire :
   (a) index
   (b) interval(ms)
   (c) time(ms)
   (d) command_slave
   (e) position_slave
   (f) speed_slave
   (g) command_master
   (h) position_master
   (i) speed_master
   (j) force_slave
   (k) elapsed_time(ms)

Le code Arduino qui a été créé afin de recevoir les paramètres d'acquisition provenant de Python et d'envoyer des données aléatoires vers l'interface se situe dans le dossier "arduino/acquiring_and_sending_data".

# 5 Limitations lors de l'envoi de données du microcontrôleur vers Python

Comme dit précédemment, lorsque l'acquisition se fait en temps réel, les graphiques sont actualisés toutes les 100 millisecondes. Cependant, l'acquisition de données par le microcontrôleur est nettement plus rapide. Il faut donc s'assurer que le microcontrôleur est assez puissant que pour envoyer les données toutes les millisecondes.

## 5.1 Arduino Uno

Bien que le dispositif utilise un Arduino Due, comme les deux types d'Arduino sont à notre disposition, nous avons pensé que quantifier les différences entre les vitesses d'exécution du code serait intéressant. Pour ce faire, le délai entre l'acquisition de 2 mesures par l'Arduino a été mis à une valeur nulle. L'acquisition a ensuite été lancée et les données rapatriées par le programme Python.

On observe ci-dessous que l'exécution de la boucle principale prend 4 à 5 millisecondes ce qui ne permet pas d'acquérir de nouvelles données toutes les millisecondes.

| index | interval(ms) | time(ms) |
|---|---|---|
| 0 | 0 | 3787 |
| 1 | 4 | 3791 |
| 2 | 4 | 3795 |
| 3 | 5 | 3800 |
| 4 | 4 | 3804 |
| 5 | 4 | 3808 |
| 6 | 4 | 3812 |
| 7 | 4 | 3816 |
| 8 | 4 | 3820 |
| 9 | 4 | 3824 |
| 10 | 4 | 3828 |
| 11 | 4 | 3832 |
| 12 | 4 | 3836 |
| 13 | 5 | 3841 |
| 14 | 5 | 3846 |
| 15 | 4 | 3850 |
| 16 | 4 | 3854 |
| 17 | 4 | 3858 |
| 18 | 4 | 3862 |
| 19 | 4 | 3866 |
| 20 | 4 | 3870 |
| 21 | 4 | 3874 |
| 22 | 4 | 3878 |
| 23 | 6 | 3884 |
| 24 | 4 | 3888 |
| 25 | 4 | 3892 |

FIGURE 9 – Capture d'écran des acquisitions réalisées via l'Arduino Uno avec un délai nul.

Nous voulons cependant attirer l'attention du lecteur sur le fait que le code Arduino utilisé génère des données aléatoirement avant de les envoyer vers l'interface graphique. Le temps d'exécution peut donc différer des résultats obtenus ci-dessus lorsque les données envoyées sont celles qui correspondent aux différents capteurs du dispositif.

## 5.2 Arduino Due

A nouveau, on peut observer que l'exécution de la boucle principale prend 4 à 5 millisecondes. Il n'est donc pas possible, même avec une Arduino Due, d'acquérir de nouvelles données toutes les millisecondes.

| index | interval(ms) | time(ms) |
|---|---|---|
| 0 | 0 | 6627 |
| 1 | 4 | 6631 |
| 2 | 4 | 6635 |
| 3 | 4 | 6639 |
| 4 | 4 | 6643 |
| 5 | 4 | 6647 |
| 6 | 4 | 6651 |
| 7 | 4 | 6655 |
| 8 | 4 | 6659 |
| 9 | 4 | 6663 |
| 10 | 4 | 6667 |
| 11 | 4 | 6671 |
| 12 | 4 | 6675 |
| 13 | 4 | 6679 |
| 14 | 5 | 6684 |
| 15 | 4 | 6688 |
| 16 | 4 | 6692 |
| 17 | 4 | 6696 |
| 18 | 4 | 6700 |
| 19 | 4 | 6704 |
| 20 | 5 | 6709 |
| 21 | 4 | 6713 |
| 22 | 4 | 6717 |
| 23 | 4 | 6721 |
| 24 | 4 | 6725 |
| 25 | 4 | 6729 |

FIGURE 10 – Capture d'écran des acquisitions réalisées via l'Arduino Due avec un délai nul.

Il semblerait donc que l'Arduino Due n'est pas plus puissante que l'Arduino Uno, et ne permet donc pas d'acquérir des données plus rapidement. Cependant, comme mentionné plus haut, les données utilisées pour réaliser ce test ont été générées aléatoirement par le code Arduino. Nous pensons qu'en opérant la même expérience mais avec les données mesurées par les différents capteurs, une différence de vitesse d'exécution plus marquée pourrait être observable. On peut en effet envisager l'hypothèse que l'Arduino Due soit plus puissante que l'Arduino Uno pour effectuer la conversion de la valeur expérimentale analogique en une valeur digitale.

# 6 Remarques supplémentaires

## 6.1 Remarques concernant le cahier des charges

Au fur et à mesure de l'avancement du projet, l'utilité du bouton *Recall* est devenue floue. Par conséquent, il a été décidé de ne pas l'implémenter.

La superposition de la force et de la position sur le même graphique ne constituant pas une fonctionnalité indispensable, il a été décidé de ne pas l'implémenter. Toutefois, le code actuel permet de rajouter cette fonction assez facilement : il suffirait de rajouter une boite à cocher dans le cadre dédié aux options de la force ("Force Options Frame" sur la figure 1). Lorsque cette boite est cochée, il faudrait créer un deuxième axe Y sur le graphique de la force (afin d'avoir une échelle différente) et y afficher les données de la position. L'implémentation de cette fonctionnalité nécessite, en plus de rajouter la boite à cocher, la modification de la méthode *refresh_all_plots()* de *DrawPlotsRealTime* et de *DrawPlotsFromFile*.

Le but principal de l'acquisition en temps réel est d'afficher les données reçues sur les graphes. Cependant, il est apparu vers la fin du projet qu'il pourrait être nécessaire de stocker les données sur le microcontrôleur et de les envoyer vers l'interface lorsque l'acquisition est terminée. Il faut donc s'assurer que le code Python est adapté à cette contrainte ce qui semble être le cas lorsque l'on regarde la méthode *real_time_data_acquisition* de *DrawPlotsRealTime*.

## 6.2 Remarque concernant de possibles problèmes d'affichage de l'interface

L'interface graphique peut ne pas s'afficher correctement sur tous les écrans (Comme présenté sur la figure 14, certains éléments peuvent être hors des cadres et il est donc impossible d'interagir avec eux). Pour régler ce problème, deux méthodes existent :

1. Modifier *FIGURE_X* et *FIGURE_Y* dans *GlobalConfig*. Ceci va modifier la taille de la figure comprenant les 4 graphes ce qui peut permettre d'afficher tous les éléments si la figure est réduite.
2. Changer la mise à l'échelle dans Windows ("Windows scaling"). La procédure est expliquée ci-dessous.

Il faut commencer par faire un clique droit sur le bureau et choisir l'option "paramètres d'affichage" (display settings en anglais).

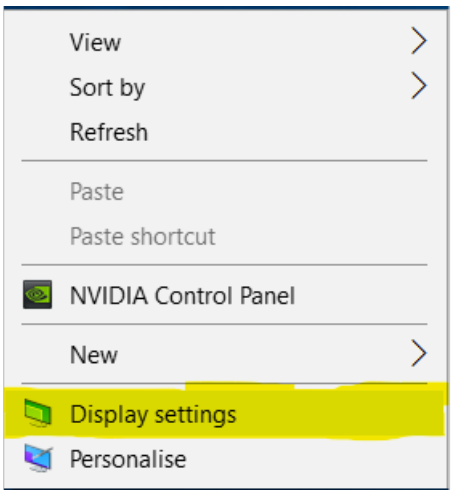


FIGURE 11 – Menu lorsqu'un clique droit est effectué sur le bureau

On voit ci-dessous à gauche que la mise à l'échelle est de 125%. La mise à l'échelle minimale est de 100% et elle permet de maximiser le nombre d'éléments qui peuvent être affichés à l'écran.

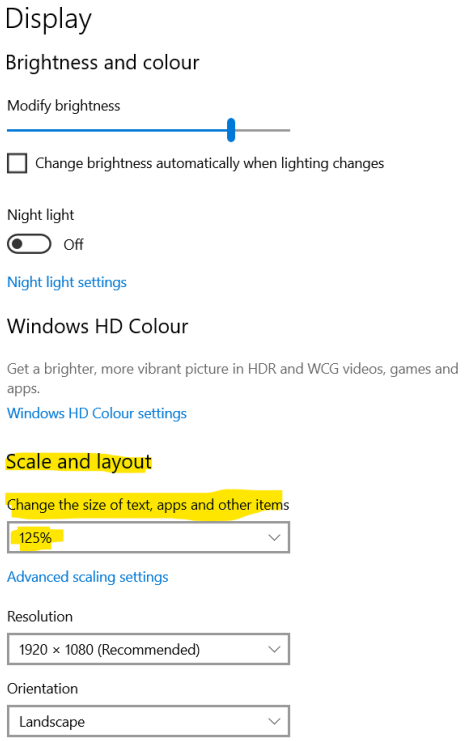


FIGURE 12 – Mise à l'échelle : 125%

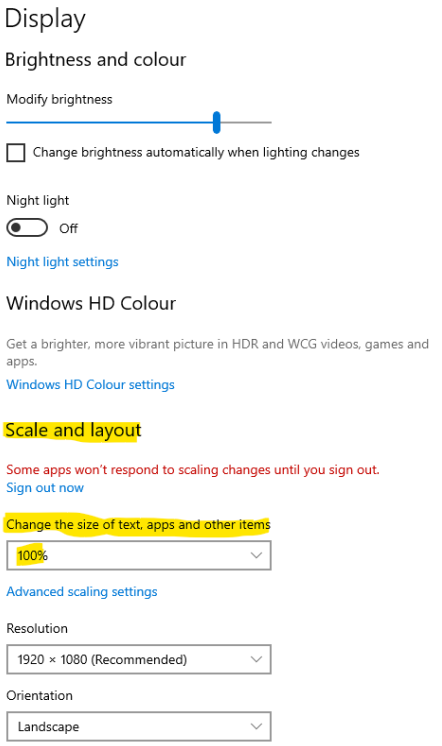


FIGURE 13 – Mise à l'échelle : 100%

Les deux images ci-dessous permettent de voir l'impact du changement de ce paramètre.

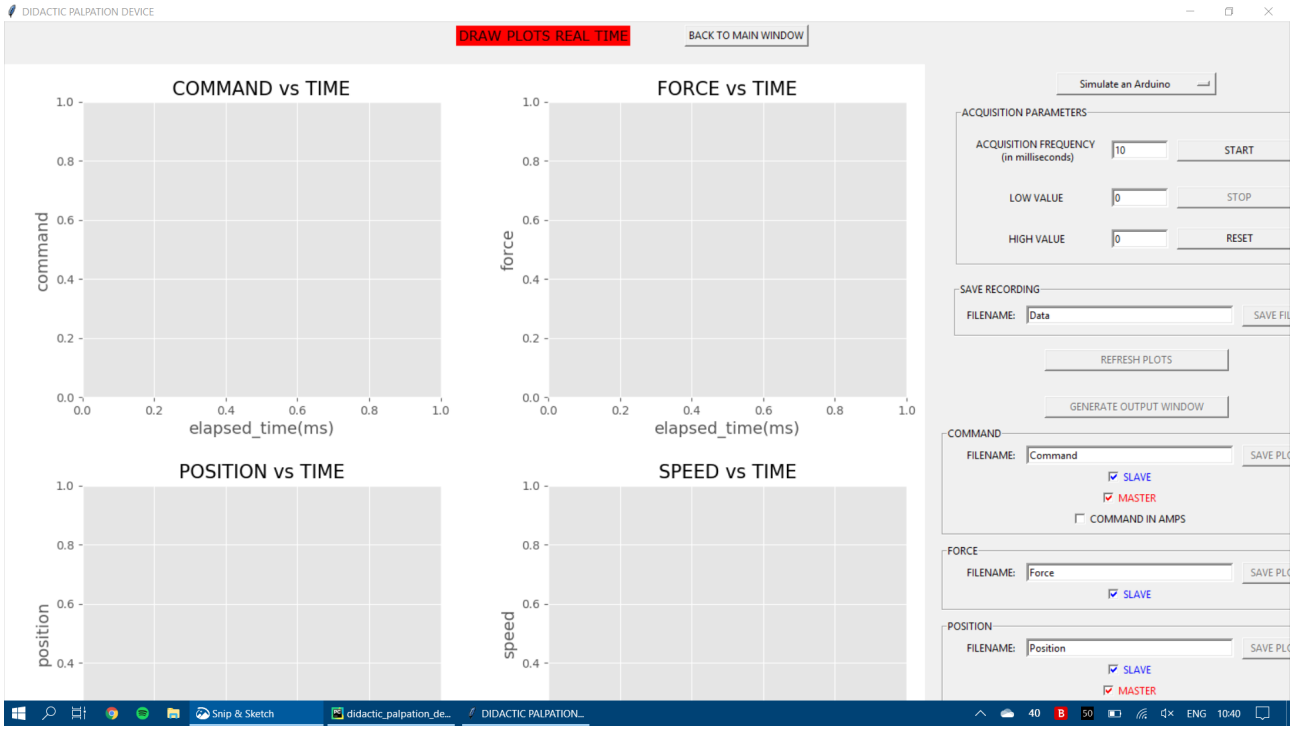


FIGURE 14 – Mise à l'échelle : 125%

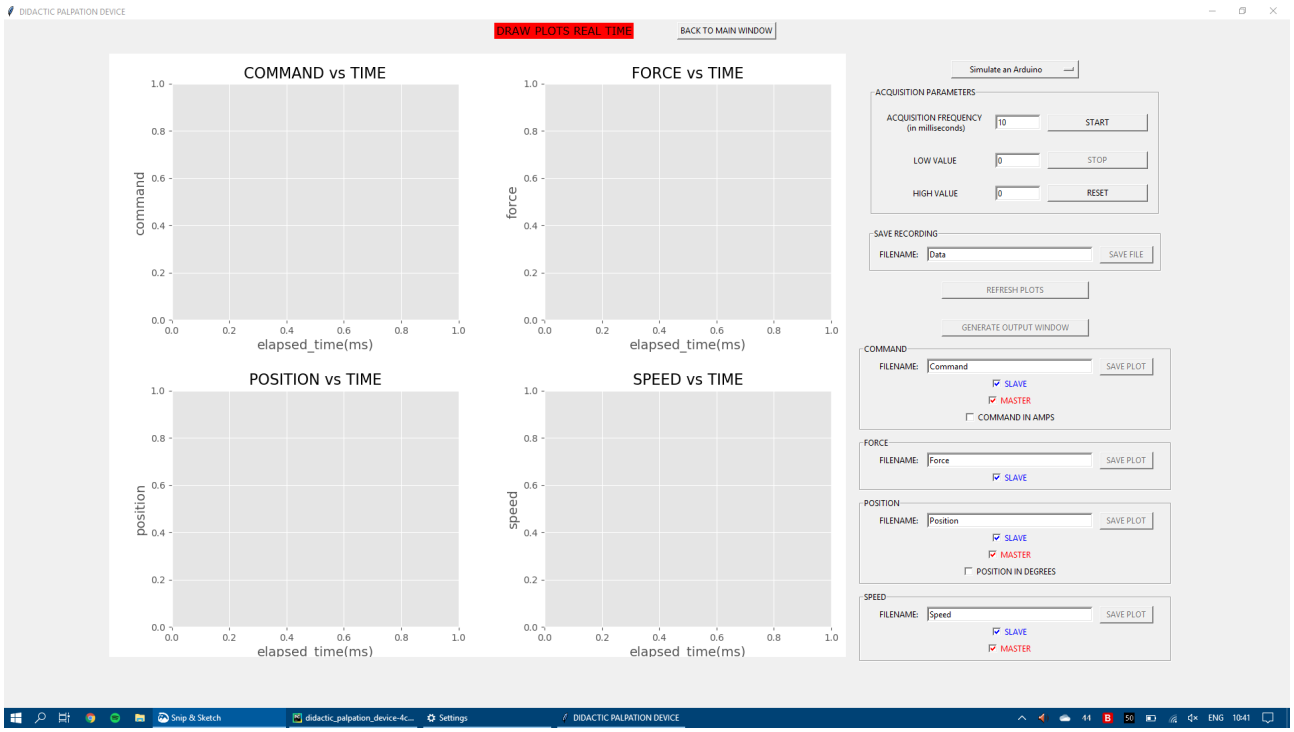


FIGURE 15 – Mise à l'échelle : 100%

# 7 Développements futurs

Pour conclure, nous proposons ci-dessous des recherches et vérifications complémentaires afin de s'assurer le bon fonctionnement de l'interface.

1. Nous recommandons de s'assurer que le programme est utilisable sur toute taille/résolution d'écran et, le cas échéant, implémenter les changements nécéssaires.
2. Lorsque l'acquisition des données dure longtemps, les graphes deviennent illisibles. Il pourrait être intéressant de n'afficher que les 10 ou 20 dernières secondes de l'acquisition afin d'assurer une meilleure lisibilité.
3. Investiguer s'il est en effet possible pour l'Arduino d'envoyer les données toutes les millisecondes grâce au code du microcontrôleur qui est spécifique au *didactic palpation device*.
4. Nous avons parfois observé des légères pertes de données lorsque l'acquisition dure longtemps. Ceci est typique d'un *buffer* plein. Nous pensons que ce problème peut être lié au temps requis pour ajouter les données aux différentes listes. En effet, lorsqu'une liste est initialisée, elle est représentée sous forme de tableau de taille fixe (par exemple, de taille $n$). Lorsque cette liste est pleine après le rajout de n éléments, l'ajout du $n+1$ élément nécessite de créer un nouveau tableau de taille $2n$, d'y copier les $n$ éléments et d'y ajouter l'élément $n+1$. Lorsque les tableaux deviennent grands, cette opération peut prendre plusieurs millisecondes [1] ce qui peut provoquer un embouteillage dans le *buffer*. Il serait donc intéressant de faire des investigations complémentaires afin de s'assurer que le problème provient bien de cette opération et d'y trouver des solutions.

1. D'après nos tests. Le code pour afficher le temps requis pour l'opération est encore présent (voir *real_time_data_acquisition* dans *DrawPlotsRealTime*