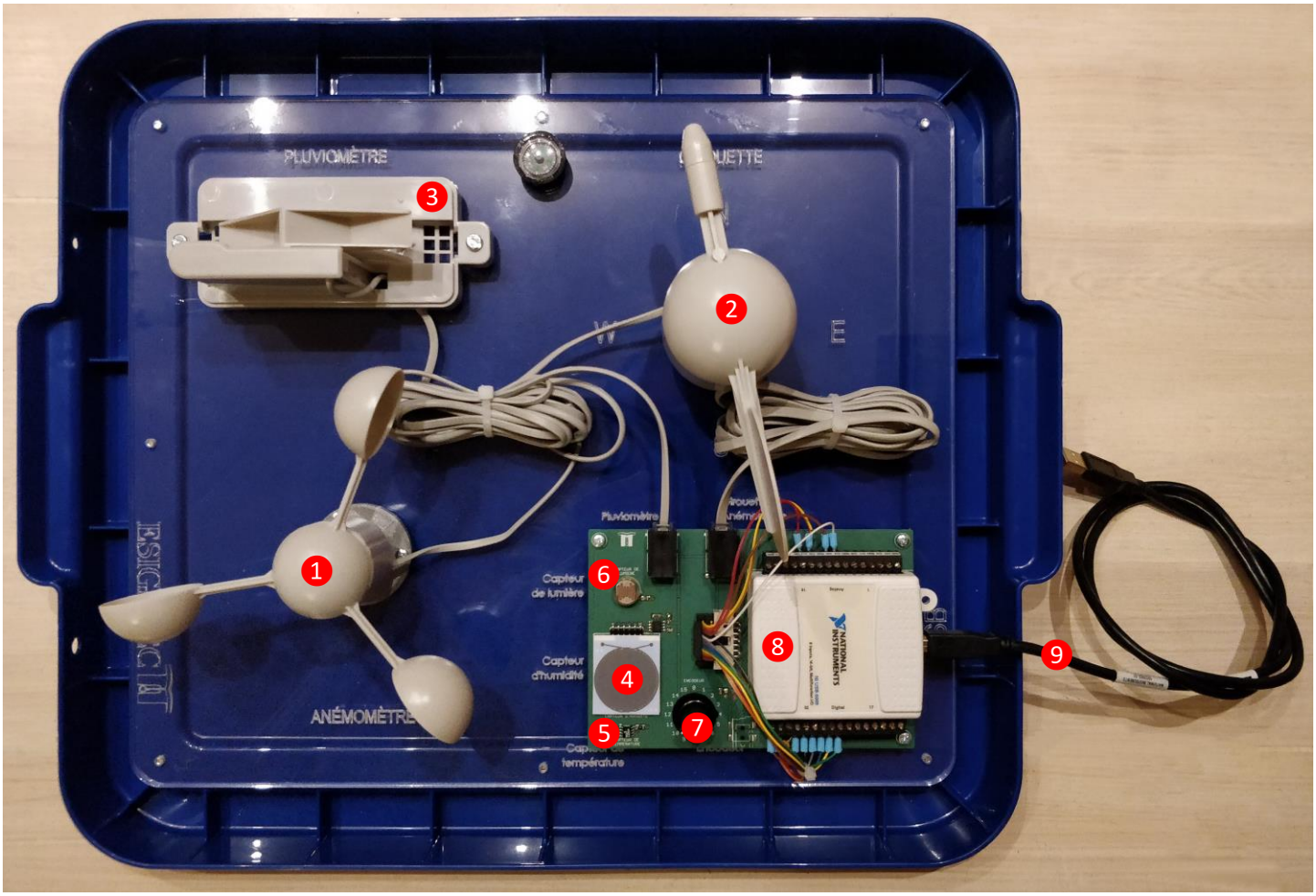
U.E. INSTRUMENTATION & SYSTÈMES

PARTIE 3 - Projet Python



Mis à jour le 8 décembre 2021 par J-Y CALAIS

Contexte & Objectifs

Ce projet de 32h, réparti en **16h de travail encadré** et **16h de travail personnel**, clôture l'UE Instrumentation & Systèmes. Vous travaillerez en petites **équipes** de 2 ou **3 étudiants maximum**.

L'objectif visé par cette 3ème et dernière partie est de faire la synthèse de vos connaissances dans l'utilisation de l'environnement de développement Python pour acquérir et traiter des mesures.

Vous devrez réaliser un programme Python d'acquisition, de traitement et d'analyse de données météorologiques.

Une station météo est à votre disposition, elle intègre les capteurs suivants *(voir photo page précédente)* :

- ❶ Anémomètre
- ❷ Girouette
- ❸ Pluviomètre
- ❹ Capteur d'humidité
- ❺ Capteur de température
- ❻ Capteur de luminosité
- ❼ Encodeur rotatif
- ❽ Boitier d'acquisition de données NI DAQ USB 6009
- ❾ Câble USB

ATTENTION **: Les stations météo sont fragiles. Utilisez-les avec précaution.**

Le «kit» des fichiers relatifs au Projet Station Météo Python est disponible sur :

Page ENT -> Mes cours -> UE Instrumentation & Systèmes -> Partie 3 -> Kit Projet Station Météo Python

Vous y trouverez :

- Ce document.
- Des sources à compléter pour démarrer votre projet.
- Les datasheets des capteurs.

À l'issue de cette 3ème partie, vous remettrez à votre enseignant :
l'ensemble du projet Python réalisé (fichiers sources, fichier de mesures généré par le programme)

Ce projet est noté sur **40** points répartis de la façon suivante :

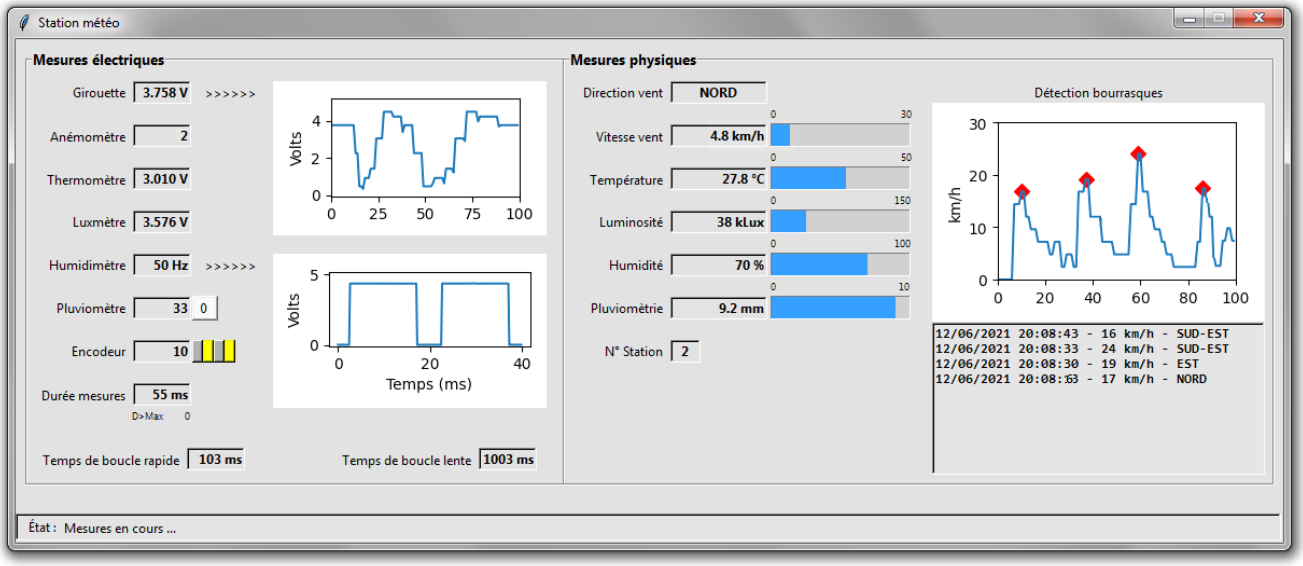
- Fonctionnalités : **15** points
- Qualité des sources : **10** points
- Qualité de l'interface utilisateur : **10** points
- Documentation du code : **5** points

Cahier des charges

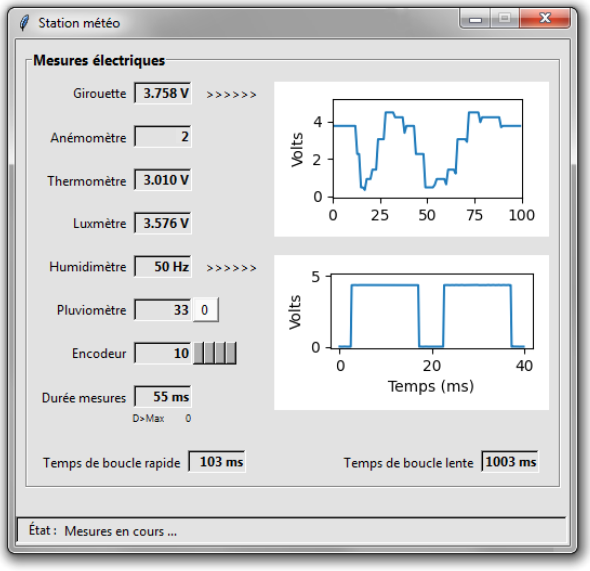
L'application que vous devez concevoir réalisera les traitements suivants :

- lecture de l'anémomètre toutes les secondes, lecture des autres capteurs toutes les 100ms;
- affichage des **mesures électriques**;
- tracé de la **courbe de variation** de la tension de la **girouette**;
- graphe du **signal périodique** du capteur d'**humidité**;
- affichage des **valeurs booléennes** de l'**encodeur**;
- **remise à zéro** de la mesure du **pluviomètre**;
- affichage de la **durée des mesures** des capteurs, et des temps réalisés par les boucles à 100ms et 1sec;
- affichage de l'**état du logiciel** (mesures en cours, erreur d'acquisition ...);
- affichage des mesures électriques converties en **mesures physiques**;
- détection des **pics de vitesse du vent** (bourrasques);
- affichage de l'**historique** des pics détectés;
- enregistrement des pics détectés dans un **fichier texte**.

PARTIES À RÉALISER PENDANT LE PROJET



Un projet réalisant les fonctionnalités de base est fourni :



Les 7 voies de mesure sont lues à travers 4 ports :

Port analogique

AI.2, AI.3, AI.6, AI.7 : humidité, girouette, température, luminosité

Port numérique n°0

P0.7 : pluviomètre, 1 bit

Port numérique n°1

P1.0 à P1.3 : encodeur rotatif du numéro de station, 4 bits

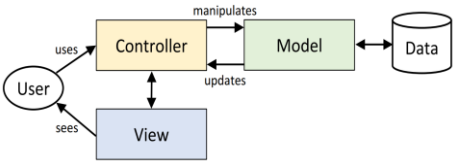
Port compteur

PFIO : anémomètre

Analyse de l'application Station Météo

Le projet de station météo de base fourni utilise l'architecture logicielle «MVC».

Il est toujours préférable de séparer le code de gestion de l'Interface Utilisateur du reste du programme, comme cela est préconisé dans le modèle de conception (Design Pattern) **MVC : Model-View-Controller**.
Cela facilite les évolutions de l'interface utilisateur, et permet une meilleure réutilisabilité des sources.



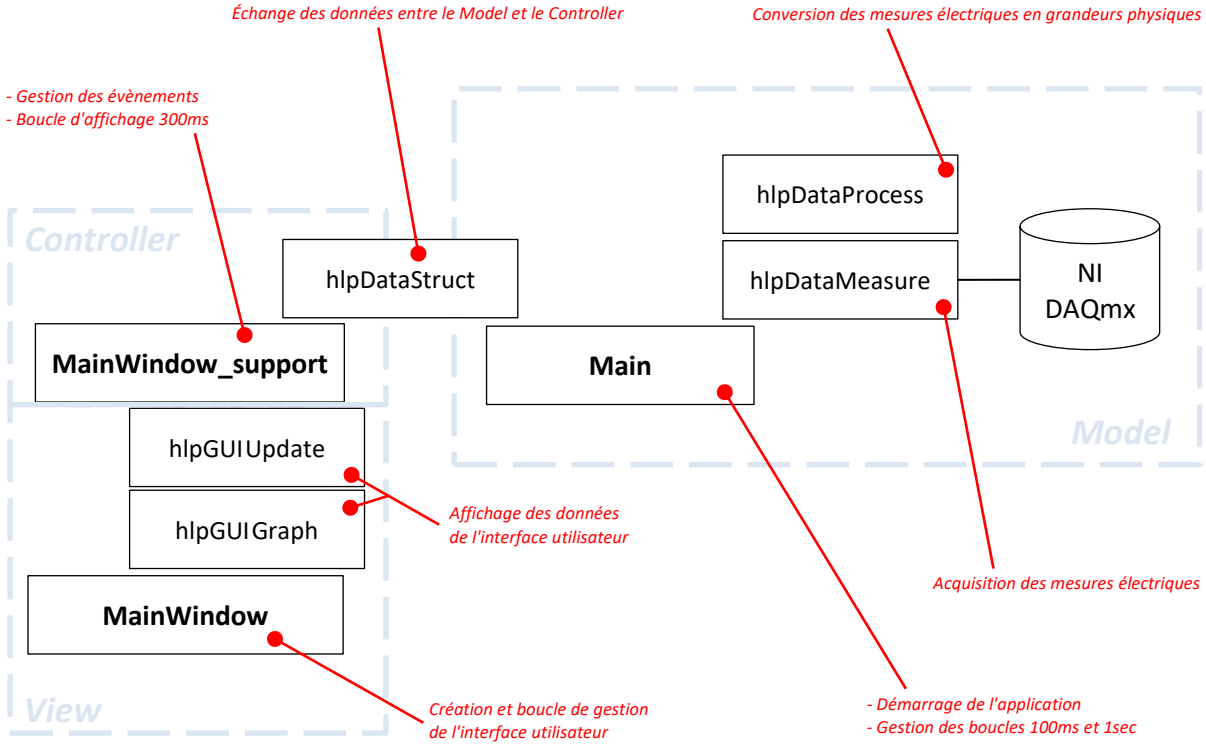
C'est particulièrement pertinent dans le cas de l'utilisation de PAGE qui génère les deux fichiers «View & Controller».

L'exercice **HelloWorld**, réalisé pendant les séances de TD, utilise deux fichiers Python générés par PAGE :
HelloWorld.py (View) : effectue l'affichage graphique de l'application (fenêtre Windows)
HelloWorld_support.py (Controller) : répond aux interactions de l'utilisateur
Dans cet exercice il n'y avait pas d'accès à des données externes nécessitant un troisième fichier «Model».

L'exercice **TestDAQmx** utilise, lui aussi, les deux fichiers générés par PAGE.
Cependant, pour ne pas compliquer inutilement les sources, l'acquisition a été réalisée dans le «Controller» plutôt que d'être placée un troisième fichier «Model».

L'application Station Météo fournie développe une architecture beaucoup plus rigoureuse et utilise trois fichiers principaux, selon le principe MVC, épaulés par des utilitaires (helpers).

Structure logicielle de l'application Station Météo



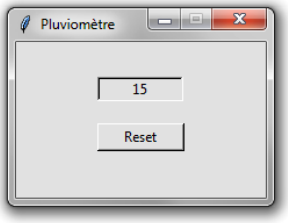
Avant de travailler sur l'application Station Météo, vous pourrez vous familiariser avec l'architecture MVC en étudiant l'exemple décrit en page suivante.

Exemple modèle applicatif «MVC»

Cet exemple utilise l'architecture MVC à 3 fichiers suivante :

Model	: Main.py	démarre l'application, réalise les mesures et la mise en forme des données
View	: MainWindow.py	construit l'Interface Utilisateur
Controller	: MainWindow_support.py	traite les actions de l'utilisateur

Dans cet exemple, on réalise l'acquisition de l'entrée **Pluviomètre - P0.7** de la Station Météo, produisant une **impulsion de 1 à 0** à chaque **basculement du godet**. Un **compteur** est **incrémenté** et affiché sur l'Interface Utilisateur à chaque front descendant. Un **bouton** de l'Interface Utilisateur permet une **remise à zéro du compteur**.



L'**interface utilisateur** générée par PAGE s'exécute dans un **thread «bloquant»**, on n'en sort que lorsque la fenêtre est fermée.

Pour pouvoir réaliser des mesures en boucle en même temps que l'Interface Utilisateur est utilisée, il faut mettre en place un **thread de mesure qui s'exécute en «parrallèle»**.
La librairie **continuous_threading** facilite la mise en place d'un thread répétitif dont la **cadence** peut être définie.

Le dialogue entre «Controller» et «Model» est réalisé par deux variables :
GUINum contient la valeur à afficher;
GUIBtnReset indique un appui sur le bouton Reset.

MainWindow_support.py *(extrait)*

```
#-- Custom part ---------------------------------
GUIBtnReset = False
GUINum      = 0

def GUIDisplay():          Mise à jour visuelle de la valeur
    global w               numérique toutes les 300 ms

    w.TextNum["text"] = "{0:d}".format(GUINum)
    root.after(300, GUIDisplay)

def GUIStart():
    import MainWindow
    MainWindow.vp_start_gui()
#---------------------------------------------------

def init(top, gui, *args, **kwargs):
    global w, top_level, root
    w = gui
    top_level = top
    root = top

    #-- Custom part ----------------------------
    GUIDisplay()
    #-------------------------------------------

def BtnReset(p1):
    #-- Custom part ----------------------------
    global GUIBtnReset
    GUIBtnReset = True
    #-------------------------------------------
```

Main.py *(extrait)*

```
#-- DAQmx functions -----------------------------------------------------
def DAQmxRead():
    global bPrevRead

    if MainWindow_support.GUIBtnReset:          RaZ compteur si bouton Reset appuyé
        MainWindow_support.GUIBtnReset = False
        MainWindow_support.GUINum      = 0
    else:
        b = bTask.read()                        Incrément compteur si détection front descendant
        if (bPrevRead and (not b)) : MainWindow_support.GUINum += 1
        bPrevRead = b

#-- Timed Loop functions ------------------------------------------------
import continuous_threading

def TimedLoopStart():                           Création d'un thread cadencé toutes les 100ms
    global th                                   + ouverture voie de mesure

    DAQmxOpen()
    th = continuous_threading.PeriodicThread(0.1, TimedLoopJob)
    th.start()

def TimedLoopJob():  # 100ms loop
    DAQmxRead()                                 Lecture voie de mesure

def TimedLoopStop():
    global th

    th.join()                                   Attente fin exécution complète du thread avant son arrêt
    DAQmxClose()                                + fermeture voie de mesure

#-- Application start/stop ----------------------------------------------
if __name__ == "__main__":
    TimedLoopStart()                            Démarrage boucle cadencée de mesure
    MainWindow_support.GUIStart()               Démarrage Interface Utilisateur (jusqu'à sa fermeture)
    TimedLoopStop()                             Arrêt boucle cadencée de mesure
```

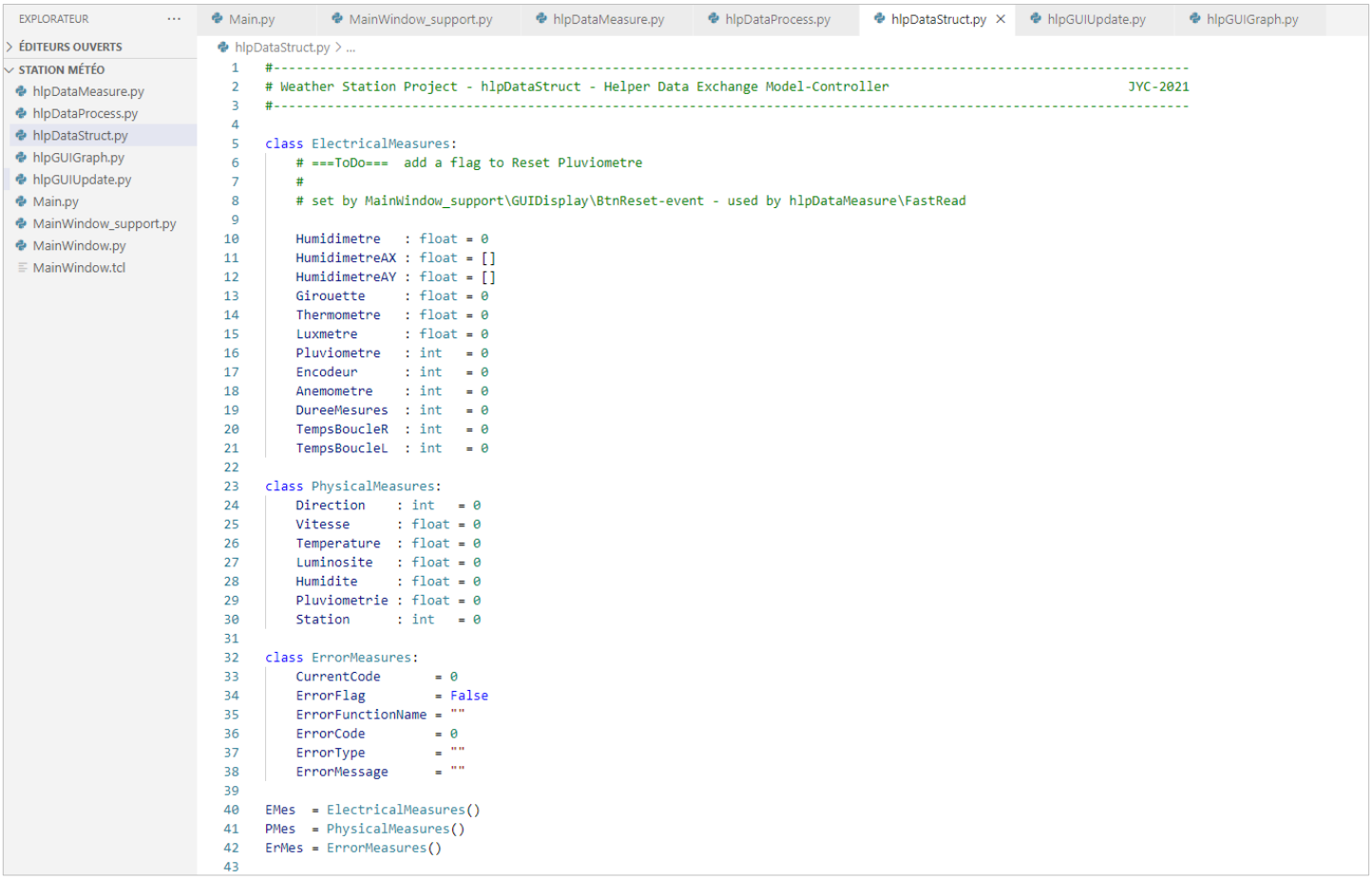
Les sources présentés ci-dessus sont des extraits exposant les parties essentielles de l'application.
Il est important d'étudier les sources complets de l'exemple pour bien comprendre le concept MVC.

Réalisation du projet Station Météo

Ouverture du projet avec Visual Studio Code

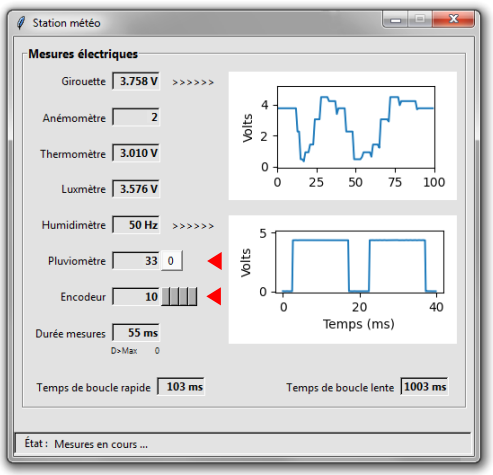
Après avoir extrait les fichiers sources dans votre répertoire de travail, ouvrez le dossier Station Météo avec Visual Studio Code (**menu Fichier\Ouvrir le dossier ...**).

En double-cliquant sur les sources (panneau de gauche), ceux-ci s'affichent dans des onglets (panneau de droite).



7 fichiers doivent être **modifiés** pour réaliser le cahier des charges (voir ci-dessus).
Les modifications sont repérées dans les sources par le marqueur **«===ToDo===»**.

Le **projet** fourni est **exécutable**, avec quelques fonctionnalités incomplètes (RaZ pluviomètre, voyants encodeur).



Gestion bouton Remise à Zéro Pluviomètre

Ajouter l'évènement associé au bouton avec le logiciel PAGE, et mettre à jour les fichiers **MainWindow.py** et **MainWindow_support.py**. (voir exemple HelloWorld réalisé en TD)

ATTENTION : il ne faut pas écraser le contenu du fichier support, mais y ajouter uniquement la nouvelle fonctionnalité. (bouton [Update] dans la boite de dialogue de génération du fichier).

A l'identique de ce qui est fait dans l'exemple MVC, ajouter un flag dans **hlpDataStruct.py** pour créer un canal de communication entre «Controller» et «Model». Effectuer ce transfert dans **Main.py**.
Créer la fonctionnalité de remise à zéro du compteur pluviomètre dans **hlpDataMeasure.py**.

Gestion des indicateurs bolléens de l'encodeur

Dans le fichier **hlpGUIUpdate.py**, mettre à jour les indicateurs **EP1_0 à EP1_3** (correspondant aux ports P1.0 à P1.3 du boitier USB) en testant l'état des bits correspondants de EMes.Encodeur.

Vérification des mesures électriques

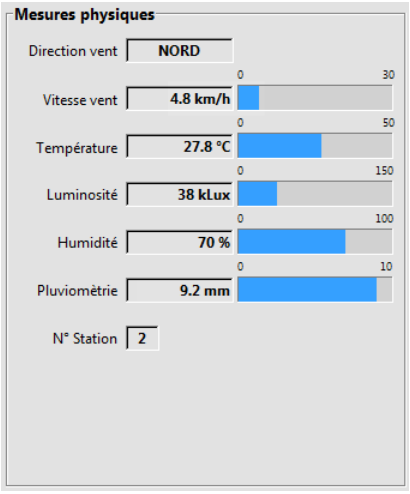
Il est important de **vérifier** soigneusement le **fonctionnement des 7 voies de mesure** :

- humidité signal rectangulaire, fréquence approximative de 25Hz (humide) à 100Hz (sec);
- girouette variation non linéaire de la tension de 0,5V à 4,5V, en 8 paliers (8 points cardinaux);
- température tension d'environ 3V pour 25°, varie précisément de 10mV par degré;
- luminosité tension approximative de 0V pour un éclairement très fort à 4,8V dans le noir;
- pluviomètre 1 impulsion pour 0,2794mm (1mm = 1ℓ/m²);
- anémomètre 1 impulsion par seconde pour un vent de 2,4km/h (force 1 sur l'échelle de Beaufort);
- encodeur code Gray sur 4 bits inversés.

Relever les valeurs de la girouette et de l'encodeur, elles seront utilisées pour la conversion des mesures.

Modification de l'Interface Utilisateur Graphique

Avec le logiciel PAGE, ajouter un **panneau Mesures Physiques** (LabelFrame) et y inclure les indicateurs numériques (Label), les indicateurs graphiques (Progressbar), les annotations (Message).
Les alias des indicateurs de mesures électriques sont préfixés par un «E» (EGirouette, EAnemometre ...).
Les alias des indicateurs de mesures physiques seront préfixés par un «P» (PDirection, PVitesse ...).



L'effet "boîte enfoncée" des indicateurs numériques s'obtient par l'attribut «relief».

Modifier l'attribut «maximum» des ProgressBar selon la grandeur affichée.

Conversion des Mesures Électriques en Grandeurs Physiques

Les conversions sont réalisées par le fichier **hlpDataProcess.py**.

Déduire les **équations de conversion** des mesures **à partir des plages de valeurs** décrites dans le paragraphe «vérification des mesures électriques» de la page précédente.

La grandeur «Humidité» sera bornée entre 0 et 100%.
La grandeur «Luminosité» sera bornée entre 0 et 150klux.

La **conversion de l'encodeur rotatif** transformera le **code Gray** en une suite de valeurs entières variant de 0 à 15.
Cette conversion s'obtient en utilisant le tableau des 16 valeurs relevées lors de la vérification des mesures électriques.
À chaque valeur mesurée correspond la valeur à afficher dans un tableau de conversion.

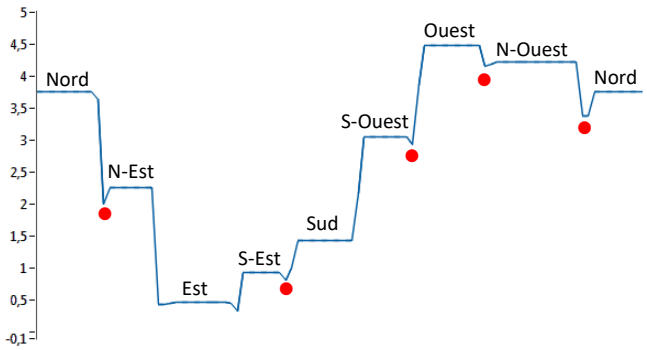
La **conversion de la girouette** nécessite un algorithme spécifique.

La position de la girouette s'obtient en comparant la valeur mesurée à chacune des 8 valeurs du tableau obtenu lors de la vérification des mesures électriques. La comparaison doit être réalisée avec une marge de tolérance, les mesures sont légèrement différentes entre les stations.
Lorsque la mesure correspond à une valeur du tableau, l'indice de cette valeur dans le tableau correspond à la position de la girouette.

La conversion de la girouette produira donc une **suite de valeurs** entières, variant **de 0 à 7**, et correspondant aux **8 points cardinaux** (N - NE - E - SE - S - SO - O - NO).
Les valeurs transitoires (●) ne seront pas décodées.

D'une manière générale, pour toute **tension non décodée**, le calcul produira en sortie la **dernière valeur valide**.

La faible différence de tension entre les positions Ouest et Nord-Ouest déterminera la **tolérance sur la tension** qui peut être employée pour détecter la position de la girouette.



Affichage des Grandeurs Physiques

L'affichage sur l'Interface Utilisateur est réalisé par le fichier **hlpGUIUpdate.py**.
Utiliser les modèles de code fournis dans ce fichier pour accéder aux composants de l'interface.

S'inspirer de l'affichage des mesures électriques pour le **format d'affichage des grandeurs physiques** (précision, unité).
Limiter aux bornes les valeurs affichées dans les **ProgressBar**.

Exercice de détection des pics dans une série de valeurs

Avant d'ajouter la fonctionnalité de détection des bourrasques de vent, vous concevrez une fonction permettant de détecter la position d'un «pic» dans un tableau de valeurs.

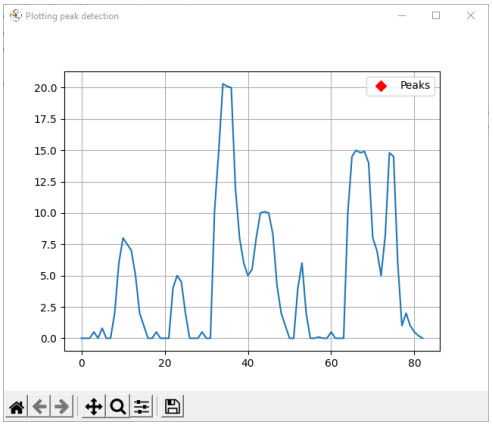
L'exercice Peaks contient le code nécessaire pour générer une courbe contenant plusieurs pics.
En vous inspirant des sources disponibles par le lien ci-dessous (et en simplifiant), vous complèterez la fonction PeakDetect pour obtenir la courbe du bas.

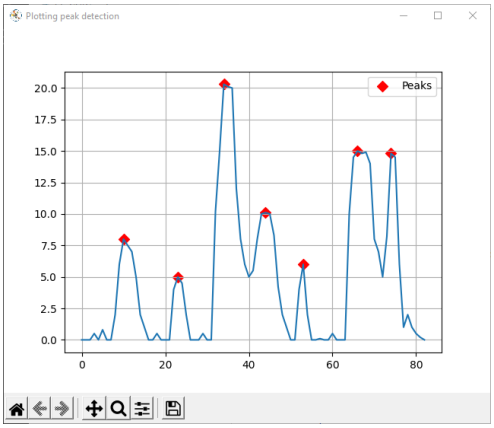
https://gist.github.com/sixtenbe/1178136

La fonction PeakDetect renvoie un **tableau** contenant les **positions des pics** détectés dans le tableau de valeurs en entrée.
Le paramètre **«Delta»** permet d'éviter plusieurs détections de pics lorsque la bourrasque forme un plateau.

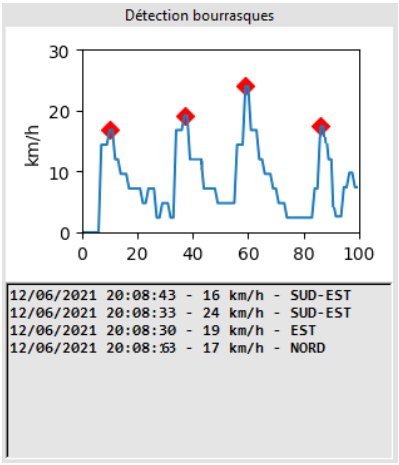
Exercice Peaks MatPlot.py > ...

```
"""-------------------------------------------------------------------------------
Completer la fonction PeakDetect declaree ci-dessous,
s'inspirer du source disponible au lien suivant (fonction "peakdetect") :
   https://gist.github.com/sixtenbe/1178136
-------------------------------------------------------------------------------"""

#-------------------------------------------------------------------------------
def PeakDetect(Values, Delta):
   return []
#-------------------------------------------------------------------------------

import os; os.system('cls')
import matplotlib.pyplot as plt

#Defining the x and y arrays
y = [0, 0, 0, 0.5, 0, 0.8, 0, 0, 2, 6, 8, 7.5, 7, 5, 2, 1, 0, 0, 0.5, 0, 0, 0,
4, 5, 4.5, 2, 0, 0, 0, 0.5, 0, 0, 10.2, 15, 20.3, 20.1, 20, 12, 8, 6, 5, 5.5,
8, 10, 10.1, 10, 8.3, 4.2, 2, 1, 0, 0, 4, 6, 2, 0, 0, 0.1, 0, 0, 0.5, 0, 0, 0,
10, 14.5, 15, 14.8, 14.9, 14, 8, 7,5, 8.2, 14.8, 14.5, 6, 1, 2, 1, 0.5, 0.2, 0]

x = range(len(y))

#Find peaks
peak_pos    : int   = []
peak_height : float = []

peak_pos = PeakDetect(Values = y, Delta = 2)
for item in peak_pos:
   peak_height.append(y[item])

#Plotting
fig = plt.figure("Plotting peak detection")
ax  = fig.subplots()

ax.plot(x,y)
ax.scatter(peak_pos, peak_height, color = 'r', s = 50, marker = 'D', label = 'Peaks')
ax.legend()
ax.grid()

plt.show()
```





Détection des bourrasques de vent



Avec le logiciel PAGE, vous ajouterez les éléments suivants (voir ci-contre) :

- une **zone de dessin** (Canvas)
- une **zone de texte** (Label)

Enregistrer les modifications (Ctrl+S) et générer le fichier Python GUI.

Le graphe dessiné est de type «déroulant», et représente les variations de la vitesse du vent dans le temps.
La fonction **PeakDetect**, précédemment développée, permet de «marquer» les maximums de vitesse détectés.

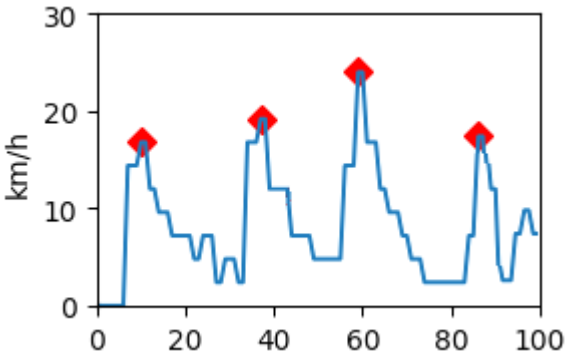
Une liste affiche la date, l'heure, la vitesse et la direction du vent pour chaque pic de vitesse présent sur le graphe.

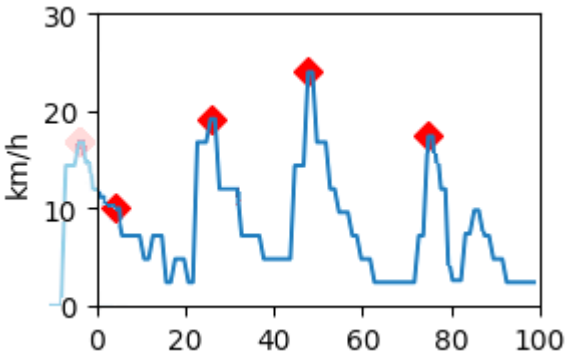
L'essentiel du code est implémenté dans le fichier **hlpGUIGraph.py**.
Vous complèterez la **classe PeakPlot** en vous inspirant de la **classe ChartPlot**, très similaire.

En plus du tableau d'historique de la vitesse du vent, il faudra aussi mémoriser un historique de la date+heure et un historique de la direction du vent.

La fonction **Plot**, en plus de tracer la courbe de vitesse et les pics, retournera la **liste des pics de vitesse détectés** sous la forme d'un tableau de chaines de caractères contenant les informations décrites ci-avant.

Lors du défilement du graphe, lorsqu'un pic de vitesse «sortira» de la fenêtre de visualisation, la fonction PeakDetect détectera un nouveau pic de vitesse, qui n'existe pas réellement. Ce défaut devra être éliminé.





Dans le fichier **hlpGUIUpdate.py** , la liste des pics de vitesse produite par la fonction **Plot** sera concaténée en une seule chaine de caractères pour être transmise à l'afficheur de texte sous le graphe.

Les **pics de vitesse** détectés seront **enregistrés en continu dans un fichier texte**, en évitant les doublons.