TD Modélisation - Informatique:
Interface Graphique (4h)

Objectifs principaux du TD

- Comprendre et mettre en œuvre les bases de la programmation d'interfaces graphiques avec PyQt5
- Développer progressivement une application fonctionnelle et interactive

Préambule sur l'interface graphique

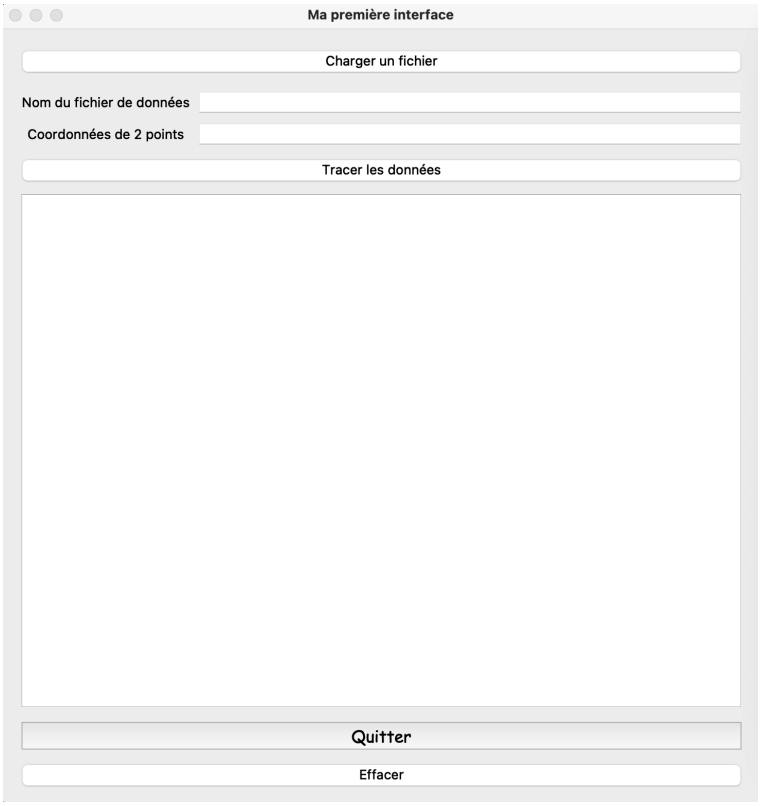
Ce sujet est largement inspiré du cours de Christine Fay-Varnier. D'autres tutoriels ou guides de référence peuvent être consultés à l'adresse suivante :

- `http://doc.qt.io/qt-5/widgets-tutorial.html`

Attention : les exemples sont décrits pour le langage C++. Il peut y avoir des différences entre Qt et PyQt. Certaines sont décrites ici : `https://doc.bccnsoft.com/docs/PyQt5/`.

Partie 1 - Prise en main d'une interface graphique

Les activités décrites ci-dessous vont permettre d'aboutir progressivement à la création de l'interface graphique suivante :



Activité 1 - Application et GUI

La création d'une application PyQt commence par l'instanciation d'un objet `QApplication`. Le programme minimal est le suivant :

```
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import Qt

def main():
    app = QApplication([])  # création d'une application
    r = app.exec_()

if __name__ == "__main__":
    main()
```

- La classe `QApplication` est la base de tout programme PyQt.
- La méthode `exec_()` démarre l'application.
- Rien ne s'affiche si aucune interface graphique (GUI) n'est créée.

En début du script, ajouter les lignes suivantes, permettant de créer la classe `GUI_Main` qui modélise l'interface de notre application :

```
class GUI_Main(QWidget):
    def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("Ma première interface")
```

Et modifier la fonction principale `main()` ainsi :

```
def main():
    # creation d'un application
    app = QApplication([])

    # creation d'une interface
    gui = GUI_Main()

    # affichage de l'interface
    gui.show()

    # lancement de l'application
    r = app.exec_()

if __name__ == "__main__":
    main()
```

Exécuter ce code pour vérifier que la fenêtre principale s'affiche correctement.

Activité 2 - Widgets

Toujours dans la méthode `__init__`, ajouter la ligne suivante :

```
# Création d'un bouton
self.b_quitter_ = QPushButton("Quitter", self)
```

Un widget peut en contenir un autre. Ici, on a inséré un widget `QPushButton` dans le widget principal.

On trouve une liste synthétique des widgets principaux à l'adresse suivante : `https://www.tutorialspoint.com/pyqt/pyqt_basic_widgets.htm`

Ajouter les lignes suivantes :

- On a modifié le libellé affiché sur le bouton et la police de caractère.
- On aurait pu mettre directement le bon libellé lors de la création du bouton.

```
self.b_quitter_.setFont(QFont("Comic Sans MS", 16))
```

Créer ensuite deux nouveaux boutons avec les intitulés suivants, en gardant la police de caractères par défaut :

- "Charger un fichier"
- "Tracer les donnees"

Tester le résultat de votre programme. Il est évident qu'il faudra organiser l'affichage des différents boutons.

Activité 3 - Le layout

À terme, notre interface va être organisée sous forme d'une grille de **7 lignes et 2 colonnes**, numérotées à partir de 0. On placera les widgets dans cette grille grâce au `layout`.

Ajouter les lignes d'instructions suivantes :

```
monLayout = QGridLayout()

#placement des boutons : numéro de ligne, numéro de colonne, nombre de lignes
#et de colonnes occupées, p.ex. b_charger_fichier_ sur la 1e ligne, 1e colonne,
#disposé sur 1 ligne et 2 colonnes
monLayout.addWidget(self.b_charger_fichier_, 0, 0, 1, 2)
monLayout.addWidget(self.b_tracer_donnees_, 3, 0, 1, 2)
monLayout.addWidget(self.b_quitter_, 5, 0, 1, 2)

self.setLayout(monLayout)
```

- Le `layout` permet de gérer l'agencement des widgets dans la fenêtre.
- Ici, on a utilisé un layout sous forme de grille (`QGridLayout`).
- Les lignes vides sont automatiquement ignorées.

Activité 4 - Gestion des événements

Il existe une multitude d'événements susceptibles de changer l'état d'une application ou des données qu'elle traite. Ces événements peuvent résulter d'une action de l'utilisateur (comme un clic), ou bien de l'activation d'une connexion Internet, du signal envoyé par une horloge, etc. Dans tous les cas, trois éléments sont concernés :

- la source de l'événement, qui est l'objet changeant d'état (comme un bouton qui passe à l'état cliqué),
- l'objet événement lui-même, qui porte des informations sur ce changement d'état,
- la cible de l'événement, c'est-à-dire l'objet qui va devoir réagir à la survenue de l'événement.

Ajouter les lignes suivantes après la création du bouton, puis tester l'effet sur le comportement du bouton :

```
# Connexion du clic du bouton à la fermeture de l'application
self.b_quitter_.clicked.connect(qApp.quit)
```

On vient de "lier" (bind) le signal de "clic" sur le bouton à l'exécution d'une fonction système qui permet de fermer l'application. Ce qu'on appelle un SLOT.

À la suite de la méthode __init__, définir les méthodes suivantes :

```
def charger_fichier(self):
    print("Début du chargement")

def tracer_donnees(self):
    print("Début du traçage")
```

Après la création des boutons respectifs, ajouter les lignes suivantes dans la méthode __init__, puis tester l'effet sur le comportement des boutons : `http://www.courspython.com/interfaces.html`

```
self.b_charger_fichier_.clicked.connect(self.charger_fichier)
self.b_tracer_donnees_.clicked.connect(self.tracer_donnees)
```

Attention : ne pas confondre les boutons (exemple : `b_charger_fichier_`) et les méthodes qu'on associe à ces boutons (exemple : `charger_fichier`)

On peut remarquer que pour chaque bouton créé, on fait à chaque fois la même chose, on crée un bouton et on lui associe une méthode. On va regrouper ces actions dans une méthode "générique". Toujours dans la classe `Gui_Main`, définir la méthode `create_button` suivante :

```
def create_button(self, libelle, method):
    """ creation d'un bouton et association avec l'action à lancer au clic """
    button = QPushButton(libelle, self)
    button.clicked.connect(method)
    return button
```

Modifier la création des boutons pour utiliser cette méthode générique. L'appel à la fonction `create_button` remplace les instructions de création du bouton et de la connexion avec la méthode associée.

Activité 5 - Les LineEdit

- Entre la création des deux premiers boutons, ajouter les lignes d'instructions suivantes :

```
self.label_filename_ = QLabel("Nom du fichier de données", self)
self.label_filename_.setAlignment(Qt.AlignCenter)
self.edit_filename_ = QLineEdit(self)
```

- Ajouter les lignes d'instructions permettant, grâce au layout, de placer ces deux widgets sur la 3ème ligne.
- Modifier la méthode `charger_fichier` ainsi :

```
def charger_fichier(self):
    print("Nom du fichier de données : ", self.edit_filename_.text())
```

- Tester l'effet de ce que vous venez d'écrire en saisissant un nom de fichier dans la zone d'édition puis en cliquant sur le bouton de chargement du fichier.
- Ajouter de façon similaire un label et un edit permettant de saisir les coordonnées des points. On donnera une largeur maximale de 50 au widget `edit` grâce à la méthode `setMaximumWidth`.
- Compléter en créant et en plaçant le bouton `Effacer`.

Activité 6 - Boîtes de dialogue

Écrire la méthode `charger_fichier` qui récupère le nom d'un fichier que nous souhaitons ouvrir.

Ceux qui le souhaitent peuvent chercher à utiliser `QFileDialog.getOpenFileName` et `setText`. `https://doc.qt.io/qt-6/qfiledialog.html`

Attention : en PyQt5, `getOpenFileName` retourne un tuple (nom du fichier et type).

Activité 7 - Dessiner

On va créer un widget pour pouvoir dessiner et tracer les segments définis par des points. On utilise les objets scènes qui permettent de créer une scène (`QGraphicsScene`) et de l'afficher dans un second temps (`QGraphicsView`).

Le premier objet, de type `QGraphicsScene`, contient les objets graphiques, le second, de type `QGraphicsView`, est une vue (éventuellement transformée) de la scène.

```
wid, hgt = 600, 400
self.scene_ = QGraphicsScene(0, 0, wid, hgt, self)
self.view_ = QGraphicsView(self.scene_, self)
self.view_.setMinimumSize(wid + 100, hgt + 100)  # pour avoir une petite marge
```

`http://www.bogotobogo.com/Qt/Qt5_QGraphicsView_QGraphicsScene.php`

Placer l'objet `self.view_` dans le layout au-dessus du bouton `Quitter`.

Écrire la méthode `tracer_donnees` **:**

- La méthode qui permet d'ajouter un segment à la scène est la méthode `addLine`.

- Dans cet exercice simplifié, l'utilisateur saisira directement dans l'interface graphique les quatre coordonnées `x1 y1 x2 y2` correspondant aux deux points du segment. Ces valeurs seront utilisées telles quelles pour tracer le segment sur la `QGraphicsScene`. Exemple de saisie : `100 200 300 400`.
- **Remarque :** si les valeurs saisies sont trop petites ou trop grandes, elles risquent de dépasser les limites de la scène et ne pas être visibles. Pour gérer ce cas, il serait nécessaire de créer une fonction qui adapte automatiquement les coordonnées du "monde réel" à l'échelle de la zone de dessin. Cela pourra être un exercice complémentaire pour ceux qui souhaitent aller plus loin.

```
self.scene_.addLine(x1, y1, x2, y2)
```

Ajouter la méthode suivante, liée au bouton `Effacer`, qui permet d'effacer le dessin :

```
def reset(self):
    """
    Efface la scène de dessin.
    """
    self.scene_.clear()
```

Partie 2 - Développement d'une interface graphique pour l'analyse d'images Sentinel-2 (TD2)

Objectifs

L'objectif de cette partie est de développer progressivement une interface graphique **qui permet** :

- De charger un fichier image au format TIFF (par exemple issu de Sentinel-2).
- D'afficher une bande spécifique de l'image choisie par l'utilisateur.
- De visualiser l'histogramme correspondant à la bande affichée.
- D'afficher les informations essentielles sur l'image (dimensions, nombre de bandes, projection, résolution).

Activité 1 - Création de l'application et de la fenêtre principale

Comme en Partie 1, il faut créer une instance de `QApplication` et définir une classe principale `GuiTiffViewer`. Exemple de début de script à compléter :

```
class GuiTiffViewer(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Tiff Viewer")
        # à compléter : création des widgets
```

Activité 2 - Création des widgets

Nous allons créer les widgets suivants :

- Bouton "Charger un fichier TIFF".
- Champ de texte pour afficher le chemin du fichier.
- Champ de texte pour saisir le numéro de bande.
- Bouton "Afficher Bande".
- Zone texte pour afficher les informations du fichier.
- Bouton "Quitter".

Remarque : nous pouvons utiliser la fonction générique `create_button(libelle, method)` pour simplifier la création des boutons et l'association des méthodes.

Activité 3 - Organisation du layout

Utilisez un `QGridLayout` pour organiser les widgets dans l'interface. Pensez à :

- Placer les boutons sur les bonnes lignes.
- Mettre les champs de texte côte à côte avec leurs labels.
- Prévoir une grande zone pour l'affichage des graphiques.

Activité 4 - Gestion des événements

- Associer le clic sur le bouton "Charger un fichier TIFF" à une méthode `charger_fichier`.
- Associer le clic sur le bouton "Afficher Bande" à une méthode `afficher_bande`.
- Gérer le bouton "Quitter" pour fermer l'application.

Activité 5 - Méthode `charger_fichier`

- Ouvrir une boîte de dialogue pour sélectionner un fichier TIFF.
- Afficher le chemin du fichier dans le champ texte.
- Charger le fichier avec `gdal.Open` et afficher les informations principales du fichier dans la zone texte. On peut formater ces informations dans une chaîne de caractères, p.ex.:

```
    info = f"Taille: {ds.RasterXSize} x {ds.RasterYSize}\n"
    info += f"Bandes: {ds.RasterCount}\n"
    info += f"Projection: {ds.GetProjection()}\n"
    info += f"Résolution pixel: {ds.GetGeoTransform()[1]}"
    self.info_label.setText(info)
```

Activité 6 - Affichage d'une bande TIFF et intégration de Matplotlib

Dans cette activité, nous allons lire une bande d'un fichier TIFF et afficher simultanément l'image et son histogramme en utilisant Matplotlib intégré dans PyQt5 via un `FigureCanvas`.

Intégration de Matplotlib dans PyQt5 : créer la figure et le canvas dans le constructeur `__init__` :

```
self.figure = plt.figure(figsize=(8,4))
self.canvas = FigureCanvas(self.figure)
```

N'oubliez pas d'ajouter le canvas dans le layout.
Ensuite, **créer une méthode** `afficher_bande`, dans laquelle on doit :

- **Lire la bande choisie par l'utilisateur et récupérer le numéro de bande depuis le champ texte** `edit_band` **:**

```
    band_num = int(self.edit_band.text())
```

- Accéder à la bande correspondante et lire les valeurs sous forme de tableau NumPy.
- **Afficher l'image en gris et l'histogramme (sous forme de sous-figures, i.e. subplot).**
- **Mettre à jour l'affichage :**

```
    self.canvas.draw()
```

Résultat attendu

L'interface doit permettre de charger un fichier TIFF, choisir une bande, et afficher :

- L'image de la bande choisie.
- L'histogramme correspondant.
- Les informations principales sur le fichier.