TP: PyQt & Interaction gestuelle

Objectif

L'objectif de ce TP est triple:

- Manipuler les classes **QListWidget** et **QTimer**
- Se familiariser avec le système de reconnaissance $1 recognizer
- Développer une plateforme pour tester ce système de reconnaissance
- Développer une méthode de feedback pour guider l'exécution du geste

Travail à réaliser

Vous devez réaliser une interface permettant à l'utilisateur de 1) charger et afficher le vocabulaire de gestes ; 2) tester le 1$ recognizer (dessiner un geste et qu'il soit reconnu par le système). La figure 1 montre à quoi ressemblera cette interface. Vous serez ensuite libre de vous appuyez sur cette interface pour implémenter davantage de fonctionnalités.

Vous trouverez dans le code différents commentaires de type **#todo N** qui indique qu'il y a quelque chose à écrire ici pour l'étape **N**. **Attention**, pour une même étape il peut y avoir plusieurs todo (dans différentes méthodes/fonctions et parfois fichiers).

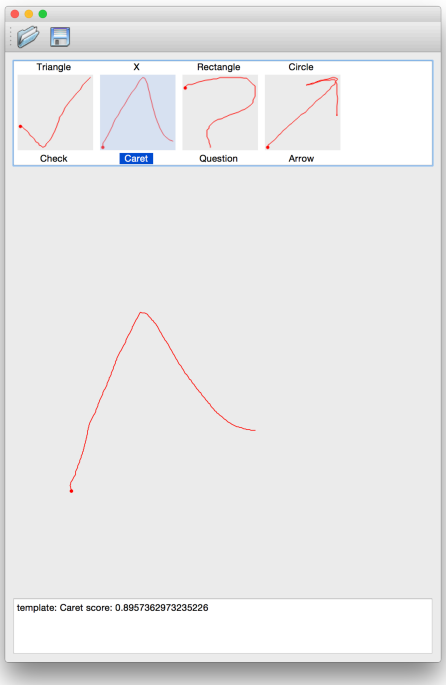

Figure 1. L'interface contenant une galerie de templates (haut), une zone pour tester le système de reconnaissance (centre), une zone de texte affichant l'historique des gestes reconnus.

Partie 1

Étape 1 : Squelette de l'application

Vous trouverez un squelette de l'application dans l'archive. La classe principale (avec la fonction main) est **MainWindow.py**.

Étape 2 : créer la galerie de templates

Pour créer une galerie d' « images », la classe **QListWidget** est particulièrement appropriée. Elle est facile d'utilisation et peut gérer une grande quantité d'information si nécessaire. Vous devez implémenter la méthode **create_template_gallery()** dans le fichier **MainWindow.py** et utiliserez les méthodes suivantes de **QListWidget**:

- **setViewMode()** pour indiquer que l'on souhaite afficher des icones dans cette liste
- **setUniformItemSizes()** pour que toutes icones aient la même taille
- **setIconSize()** pour définir la taille des icones. On pourra prendre par exemple : 100x100

Mettre à jour le code correspondant dans la méthode **__init__**

Étape 3 : Ajouter les templates à la galerie

Le fichier **onedol_ds.pkl** contient 16 classes de gestes différentes. Pour chaque classe, il y a plusieurs échantillons. Pour récupérez les coordonnées des templates (data) et les labels (labels), vous utiliserez les commandes suivantes dans **__init__** :

```
d = pickle.load( open( './onedol_ds.pkl', 'rb' ) )
data = d['dataset']
labels = d['labels']
```

On ne gardera qu'un échantillon par template pour avoir un système plus réactif ($1 Recognizer n'a besoin que d'un seul échantillon par classe contrairement à d'autres systèmes de reconnaissance, e.g. Rubine).

Ajouter les templates dans la galerie. Pour cela, vous devez implémenter la méthode **add_template_thumbnail()** en utilisant la classe **QlistWidgetItem** avec en paramètre un **QIcon** et un **QString** ainsi que la méthode **addItem()** de **QListWidget**. Appeler cette méthode dans **__init__**

A ce stade, vous pouvez visualiser les différents templates, faire un geste, mais celui-ci n'est pas reconnu. C'est l'objet de la partie 2.

Partie 2

L'algorithme de reconnaissance prend en entrée une liste de points qui correspond à la position du pointeur de la souris à l'écran au cours du temps. Cette liste de points suit les traitements successifs suivants :

1. Ré-échantillonnage pour avoir un nombre de points constant indépendamment de la fréquence d'échantillonnage du périphérique (nombre d'acquisition du signal par seconde) et de la vitesse d'exécution du geste

2. Rotation du geste pour le rendre indépendant de l'angle suivant lequel il est exécuté par rapport à l'écran

3. Mise à l'échelle du geste pour le rendre indépendant de la taille suivant laquelle il est réalisé

4. Translation du geste pour le rendre indépendant de la position à laquelle il est réalisé

La liste de points obtenue est alors comparée à chacun des templates existants en appliquant une série d'ajustements angulaires pour trouver l'alignement optimal. Le template correspond à un geste de référence auquel on associe une liste de points et un nom. Chaque comparaison permet de calculer une note comprise entre 0 et 1 qui reflète le degré de similarité entre le geste réalisé et le template, calculé comme une distance euclidienne entre les deux gestes. Le template qui obtient le score le plus élevé est considéré comme le geste exécuté.

Étape 4 : Ajouter les templates au $1 recognizer

Le système de reconnaissance est implémenté dans le fichier **onedollar.py**. Une instance a été créé dans la class **Canvas** (**Canvas.py**). Vous utiliserez la méthode **addTemplate()** de la classe **oneDollar** pour ajouter chaque template. Mettre à jour la méthode **__init__** de la classe **MainWindow**.

Étape 5 : Ré-échantillonage

Le nombre de points d'un geste dépend de la fréquence d'échantillonnage du périphérique d'entrée et de la vitesse à laquelle le geste est exécuté. Pour rendre les gestes directement comparables, les M points d'un chemin sont ré-échantillonnés en N points espacés à intervalles réguliers. L'article propose d'utiliser la valeur 64 pour N. Cette étape est deja codée dans la méthode **resample()**

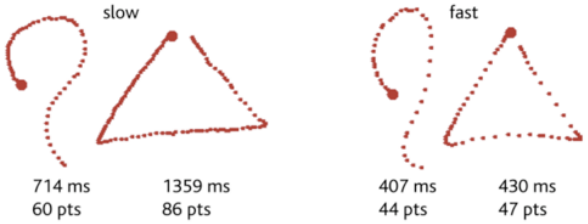

Figure 2. Illustration de l'influence de la vitesse d'exécution d'un geste sur le nombre de points

Figure 3. Algorithme de rééchantillonage

```
RESAMPLE(points, n)
1   I ← PATH-LENGTH(points) / (n − 1)
2   D ← 0
3   newPoints ← points_0
4   foreach point p_i for i ≥ 1 in points do
5     d ← DISTANCE(p_{i-1}, p_i)
6     if (D + d) ≥ I then
7       q_x ← p_{i-1}_x + ((I − D) / d) × (p_i_x − p_{i-1}_x)
8       q_y ← p_{i-1}_y + ((I − D) / d) × (p_i_y − p_{i-1}_y)
9       APPEND(newPoints, q)
10      INSERT(points, i, q)   // q will be the next p_i
11      D ← 0
12    else D ← D + d
13  return newPoints
PATH-LENGTH(A)
1   d ← 0
2   for i from 1 to |A| step 1 do
3     d ← d + DISTANCE(A_{i-1}, A_i)
4   return d
```

Étape 6 : Rotation basée sur l'angle « indicatif »

Afin de rendre les gestes indépendants de l'angle suivant lequel ils sont exécutés par rapport à l'écran, ils doivent être ré-orientés avant de pouvoir être comparés. L'algorithme consiste d'abord à trouver le centre du geste (valeur moyenne des abscisses et des ordonnées des points) puis à utiliser le premier point pour effectuer une rotation de l'ensemble des points afin que le premier point du geste et le centre soient alignés suivant l'horizontal. Cette rotation sera affinée lors de l'étape de reconnaissance.

Écrivez les methods **rotateToZero()** et **RotateBy().** ATAN corresponds à la methode atan2 de numpy

```
ROTATE-TO-ZERO(points)
1   c ← CENTROID(points)   // computes (x̄, ȳ)
2   θ ← ATAN (c_y − points_0_y, c_x − points_0_x)   // for -π ≤ θ ≤ π
3   newPoints ← ROTATE-BY(points, -θ)
4   return newPoints
ROTATE-BY(points, θ)
1   c ← CENTROID(points)
2   foreach point p in points do
3     q_x ← (p_x − c_x) COS θ − (p_y − c_y) SIN θ + c_x
4     q_y ← (p_x − c_x) SIN θ + (p_y − c_y) COS θ + c_y
5     APPEND(newPoints, q)
6   return newPoints
```

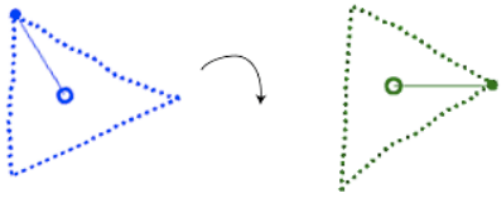

Figure 4. Algorithme de rotation

Étape 7 : Mise à l'échelle et translation

Après rotation, le geste est mis à l'échelle pour tenir dans un carré de référence afin de le rendre invariant à la taille suivant laquelle il est exécuté. Cette mise à l'échelle est non uniforme si bien qu'un rectangle et un carré seront considérés comme identiques. Après mise à l'échelle, le geste est translaté à un point de référence pour être indépendant de la position à laquelle il est exécuté. Par soucis de simplicité, le centre du geste est translaté à l'origine. Écrivez la méthode **ScaleToSquare()**. Vous pouvez utiliser les méthodes **max** et **min** du numpy pour calculer la largeur et la hauteur de la bouding box, ainsi que la variable **self.square_size** (au lieu du paramètre size)**. TranslateToOrigin()** est déjà implémenté.

```
SCALE-TO-SQUARE(points, size)
1   B ← BOUNDING-BOX(points)
2   foreach point p in points do
3     q_x ← p_x × (size / B_width)
4     q_y ← p_y × (size / B_height)
5     APPEND(newPoints, q)
6   return newPoints
TRANSLATE-TO-ORIGIN(points)
1   c ← CENTROID(points)
2   foreach point p in points do
3     q_x ← p_x − c_x
4     q_y ← p_y − c_y
5     APPEND(newPoints, q)
6   return newPoints
```

Étape 8 : Reconnaissance

Nous avons maintenant toutes les méthodes pour implémenter la méthode **recognize()** qui prend en paramètre le geste dessiné par l'utilisateur et retourne le label du template le plus proche, son id et le socre corrspondant. Cette méthode calcule la distance entre le geste de l'utilisateur et chaque template. Cette comparaison est faite en déterminant l'angle de rotation entre les deux figures qui minimise cette distance. Cette recherche de l'angle optimal est obtenue par l'utilisation de la technique Golden Section Search pour minimiser le nombre d'itérations vers la solution optimale (10 itérations au maximum avec cette technique).

Écrivez les methodes **distanceAtBestAngle()** et **recognize()**. Pour le calcul, vous utiliserez les variables **self.square_size** and Phi initialisées en haut du fichier.

```
RECOGNIZE(points, templates)
1   b ← +∞
2   foreach template T in templates do
3     d ← DISTANCE-AT-BEST-ANGLE(points, T, -θ, θ, θ_Δ)
4     if d < b then
5       b ← d
6       T' ← T
7   score ← 1 − b / 0.5√(size² + size²)
8   return ⟨T', score⟩
DISTANCE-AT-BEST-ANGLE(points, T, θ_a, θ_b, θ_Δ)
1   x_1 ← φθ_a + (1 − φ)θ_b
2   f_1 ← DISTANCE-AT-ANGLE(points, T, x_1)
3   x_2 ← (1 − φ)θ_a + φθ_b
4   f_2 ← DISTANCE-AT-ANGLE(points, T, x_2)
5   while |θ_b − θ_a| > θ_Δ do
6     if f_1 < f_2 then
7       θ_b ← x_2
8       x_2 ← x_1
9       f_2 ← f_1
10      x_1 ← φθ_a + (1 − φ)θ_b
11      f_1 ← DISTANCE-AT-ANGLE(points, T, x_1)
12    else
13      θ_a ← x_1
14      x_1 ← x_2
15      f_1 ← f_2
16      x_2 ← (1 − φ)θ_a + φθ_b
17      f_2 ← DISTANCE-AT-ANGLE(points, T, x_2)
18  return MIN(f_1, f_2)
DISTANCE-AT-ANGLE(points, T, θ)
1   newPoints ← ROTATE-BY(points, θ)
2   d ← PATH-DISTANCE(newPoints, T_points)
3   return d
PATH-DISTANCE(A, B)
1   d ← 0
2   for i from 0 to |A| step 1 do
3     d ← d + DISTANCE(A_i, B_i)
4   return d / |A|
```

A ce stade, le $1 recognizer est implémenté, mais vous ne pouvez pas encore l'utiliser. C'est l'objet de la partie 3.

Partie 3

La class Canvas permet d'utiliser le $1 recognizer. Elle hérite de **QWidget** et permet de récupérer le tracé de l'utilisateur ainsi que de l'afficher. La méthode **recognize_gesture()** est appelée quand l'utilisateur relâche le bouton de la souris.

Étape 9 : Tester le $1 recognizer

Une trace s'affiche dans la console pour tester votre système de reconnaissance. Pour aller plus loin, on souhaite « highligher » le template correspondant dans la galerie et afficher le score dans la zone de texte. Pour cela :
Une erreur s'affiche lorsque vous effectuez un geste dans la fenêtre. Pour y remédier, créer un signal (e.g. selected_template) dans la classe **Canva**s à émettre quand un geste est reconnu. Le signal aura trois paramètres **'QString', int, float** (exactement comme ca) correspondant au label, l'identifiant du template et le score. Connecter ce signal (dans la class **MainWindow**, au slot **set_action_on_gesture()** de la classe **MainWindow.** Mettre à jour cette méthode en utilisant la méthode **setCurrentRow() de QListWidget.**

Étape 10 : Afficher un feedback local (static)

On souhaite maintenant que le template reconnu s'affiche dans le voisinage du geste réalisé. Mettre à jour la méthode **display_feedback()** de la classe **Canvas**. Vous utiliserez la méthode **get_feedback()**. Cette méthode applique plusieurs transformations géométriques (rotation, translation, mise à l'échelle) pour que le template s'affiche « correctement ». Cette méthode a besoin d'une instance du geste de l'utilisateur ré-échantilloné (**resampled_gesture dans OneDollar**). Modifier la méthode **recognize** de **OneDollar** pour conserver cette instance.

Étape 11 : Afficher un feedback dynamique

On souhaite maintenant avoir un feedback dynamique, une animation qui transforme le chemin du geste réalisé en chemin « parfait » (correspondant au Template). Pour faire cette animation, il est nécessaire que les deux gestes aient le même nombre de points.

a- Mettre à jour la trace de **self.path** dans **display_feedback()** de **Canvas** pour que la trace affiché dans le canvas soit celle du geste ré-échantillioné.
b- Créer un timer à l'aide de la classe **QTimer** dans la méthode **__init__** de Canvas, le connecter au slot **timeout()**.
c- Dans la methode **display_feedback()** de **Canvas,** lancer le timer avec la méthode **start().** Utiliser la méthode **setInterval()** pour définir le temps entre deux appels à la méthode.
d- Dans la méthode **timeout()** : faire une interpolation linéaire entre **self.path** et **self.termination**. L'interpolation devra se faire sur chaque point (path et termination ont le meme nombre de points car ils ont été ré-échantillonés)