Interaction gestuelle – $1 recognizer

Auteur : Gilles Bailly,
Contributeurs : Sylvain Malacria, Aurélien Tabard

Base de code pour démarrer :
https://drive.google.com/file/d/1MoPaJio6IXu2j-kjgrlNBzq5MhdKYaCI/

Objectifs

- 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) ;
3. Offrir du feedback sur la complétion et la reconnaissance du geste

La figure 1 montre à quoi ressemblera cette interface.

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

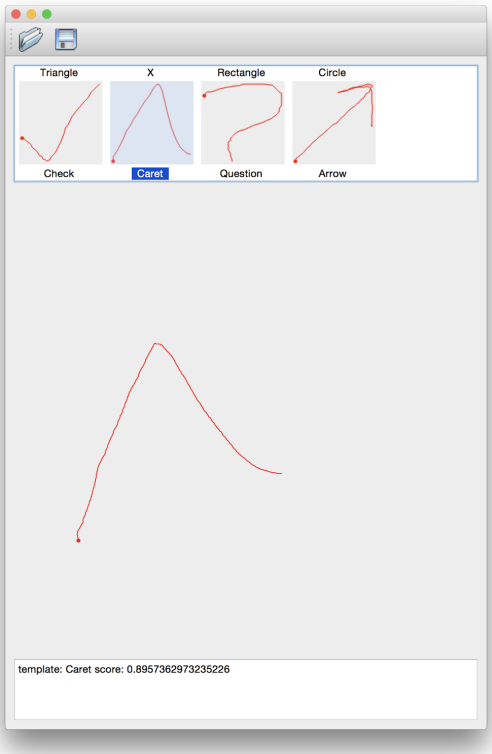


Figure 1. L'interface contenant une galerie de templates (haut), une zone pour tester le système de reconnaissance (centre), une zone de texte affichant l'historique des gestes reconnus.

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`.

Vous aurez besoin des packages suivants s'ils ne sont pas installés dans votre environnement python (ou si vous créez un nouvel environnement).

- `numpy`
- `PyQt5`

Étape 2 : créer la galerie de templates

Pour créer une galerie d' « images », la classe `QListWidget` (ref) 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 icônes dans cette liste. Utiliser par exemple `QListView.IconMode`
- `setUniformItemSizes()` pour que toutes icones aient la même taille
- `setIconSize()` pour définir la taille des icônes.
  On pourra prendre par exemple : `50x50 -> QSize(50,50)`

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 devriez avoir une interface qui ressemble à la figure 1. 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 créée dans la classe `Canvas` (voir 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` : créer un recognizer et ajouter les templates.

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 déjà codée dans la méthode `resample()`

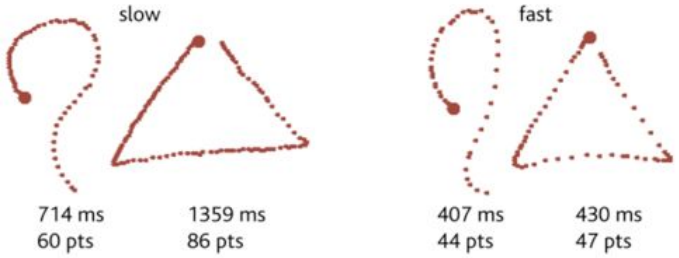
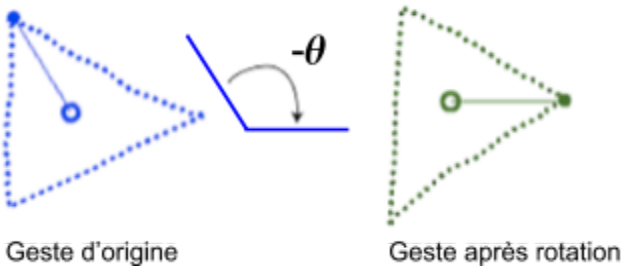


Figure 2. Illustration de l'influence de la vitesse d'exécution d'un geste sur le nombre de points

```
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
```

Figure 3. Algorithme de ré-échantillonage

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 réorientés 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 méthodes `rotateToZero()` et `RotateBy()` dans `onedollar.py`

```
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
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. l'algorithme de rotation : on calcule le centroide (déjà fait dans le code fourni), puis on calcule la rotation à effectuer. Pour cela on calcule l'atan2(y,x) disponible via numpy.arctan2(). **θ** appartient alors à]-**π**/2;**π**/2[

Voir numpy.concatenate

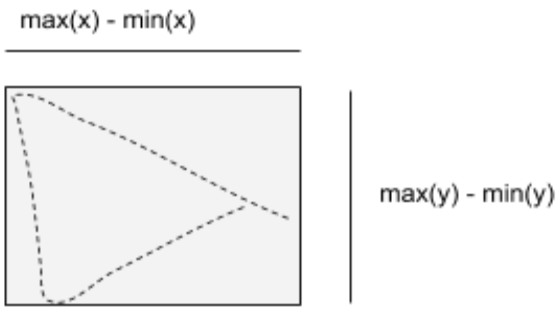
Étape 7 : Mise à l'échelle et translation

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 souci de simplicité, le centre du geste est translaté à l'origine.

```
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
```

Écrivez la méthode `ScaleToSquare()`. Vous pouvez utiliser les méthodes **amax** et **amin** de **numpy** pour calculer la largeur et la hauteur de la bounding box. Utilisez la variable `self.square_size` pour le paramètre size du pseudo code.
`TranslateToOrigin()` est déjà implémenté.



Calcul de la bounding-box

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 `score` correspondant.

`recognize()` **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 avec `distanceAtBestAngle()`.

Écrivez la méthode `recognize()`. Pour le calcul, vous utiliserez la variable `self.square_size`. Le pseudocode considère que les points passés en paramètre ont déjà été corrigés *(= vérifiez que les étapes précédentes sont bien exécutées sur les points du geste avant de l'envoyer à la reconnaissance).*

```
RECOGNIZE(points, templates)
1   b ← +∞
2   foreach template T in templates do
3     d ← DISTANCE-AT-BEST-ANGLE(points, T, -θ, θ, θΔ)
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, θa, θb, θΔ)
1   x1 ← φθa + (1 – φ)θb
2   f1 ← DISTANCE-AT-ANGLE(points, T, x1)
3   x2 ← (1 – φ)θa + φθb
4   f2 ← DISTANCE-AT-ANGLE(points, T, x2)
5   while |θb – θa| > θΔ do
6     if f1 < f2 then
7       θb ← x2
8       x2 ← x1
9       f2 ← f1
10      x1 ← φθa + (1 – φ)θb
11      f1 ← DISTANCE-AT-ANGLE(points, T, x1)
12    else
13      θa ← x1
14      x1 ← x2
15      f1 ← f2
16      x2 ← (1 – φ)θa + φθb
17      f2 ← DISTANCE-AT-ANGLE(points, T, x2)
18  return MIN(f1, f2)
DISTANCE-AT-ANGLE(points, T, θ)
1   newPoints ← ROTATE-BY(points, θ)
2   d ← PATH-DISTANCE(newPoints, Tpoints)
3   return d
PATH-DISTANCE(A, B)
1   d ← 0
2   for i from 0 to |A| step 1 do
3     d ← d + DISTANCE(Ai, Bi)
4   return d / |A|
```

Match points against a set of templates. The size variable on line 7 of RECOGNIZE refers to the size passed to SCALE-TO in Step 3. The symbol φ equals ½(-1 + √5). We use θ=±45° and θΔ=2° on line 3 of RECOGNIZE. Due to using RESAMPLE, we can assume that A and B in PATH-DISTANCE contain the same number of points, i.e., |A|=|B|.

Même si une 1ère rotation de correction a déjà eu lieu, cette dernière était grossière. Pour la phase de reconnaissance il faut aligner le geste à chaque template pour pouvoir effectuer un calcul de distance correct. C'est ce que fait la recherche de l'angle optimal via `distanceAtBestAngle()` (code fourni). Cette recherche s'appuie sur la méthode du nombre d'or pour minimiser le nombre d'itérations vers la solution optimale (10 itérations au maximum avec cette technique). Voir la section "An Analysis of Rotation Invariance" de l'article pour comprendre le calcul et sa comparaison à du hill climbing ou du brute force.

A ce stade, le $1 recognizer est implémenté. Votre outil devrait ressembler à celui de la figure 1. Et vous devriez être en mesure d'avoir un résultat du recognizer dans votre console.

Nous allons maintenant travailler le feedback donné à l'utilisateur dans la partie 3.

Partie 3

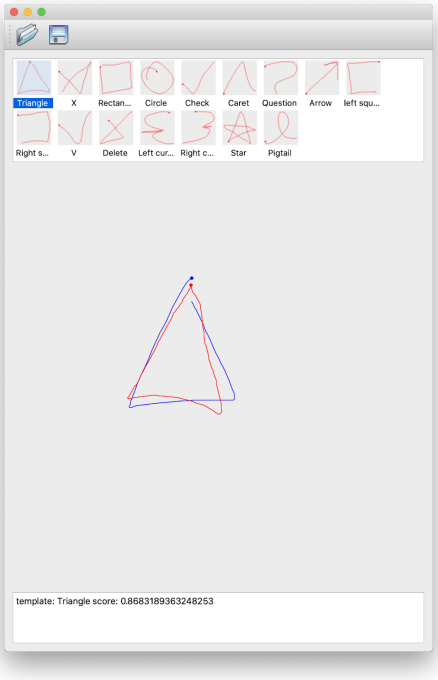
La classe `Canvas` permet d'utiliser le $1 recognizer. Elle hérite de `QWidget` et permet de récupérer le tracé de l'utilisateur et de l'afficher. La méthode `recognize_gesture()` est appelée quand l'utilisateur relâche le bouton de la souris.

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 « highlighter » le template correspondant dans la galerie et afficher le score dans la zone de texte du bas de la fenêtre. Pour cela nous allons utiliser des signaux de PyQt :

a. Créer un signal (e.g. `selected_template`) dans la classe `Canvas` 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.
b. Connecter ce signal (dans la `class MainWindow`), au slot `set_action_on_gesture` de la classe `MainWindow`.
c. Modifier la méthode `set_action_on_gesture` en utilisant la méthode `setCurrentRow()` de `QListWidget`, pour « highlighter » le template reconnu.
d. Émettre le signal depuis `recognize_gesture` (décommenter le code).

Étape 10 : Afficher un feedback statique



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-échantillonné (`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 affichée dans le canvas soit celle du geste ré-échantillonné.
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 même nombre de points car ils ont été ré-échantillonnés), il s'agit juste d'une moyenne pondérée, la fonction `interpolate()` est fournie.

DocumentationQListView Class

The QListView class provides a list or icon view onto a model.

Inherited by : QListWidget and QUndoView

Public Types

enum	Flow { LeftToRight, TopToBottom }
enum	LayoutMode { SinglePass, Batched }
enum	Movement { Static, Free, Snap }
enum	ResizeMode { Fixed, Adjust }
enum	ViewMode { ListMode, IconMode }

viewMode : ViewMode

This property holds the view mode of the QListView.

This property will change the other unset properties to conform with the set view mode. QListView-specific properties that have already been set will not be changed, unless clearPropertyFlags() has been called.

Setting the view mode will enable or disable drag and drop based on the selected movement. For ListMode, the default movement is Static (drag and drop disabled); for IconMode, the default movement is Free (drag and drop enabled).

Access functions:

QListView::ViewMode	viewMode() const
void	setViewMode(QListView::ViewMode *mode*)

uniformItemSizes : bool

This property holds whether all items in the listview have the same size

This property should only be set to true if it is guaranteed that all items in the view have the same size. This enables the view to do some optimizations for performance purposes.

By default, this property is false.

This property was introduced in Qt 4.1.

Access functions:

bool	uniformItemSizes() const
void	setUniformItemSizes(bool *enable*)

iconSize : QSize

This property holds the size of items' icons

Setting this property when the view is visible will cause the items to be laid out again.

Access functions:

QSize	iconSize() const

void	setIconSize(const QSize &*size*)

QListWidgetItem::QListWidgetItem(const QIcon &*icon*, const QString &*text*, QListWidget **parent* = nullptr, int *type* = Type)

Constructs an empty list widget item of the specified *type* with the given *icon*, *text* and *parent*. If the parent is not specified, the item will need to be inserted into a list widget with QListWidget::insertItem().

This constructor inserts the item into the model of the parent that is passed to the constructor. If the model is sorted then the behavior of the insert is undetermined since the model will call the '<' operator method on the item which, at this point, is not yet constructed. To avoid the undetermined behavior, we recommend not to specify the parent and use QListWidget::insertItem() instead.

currentRow : int

This property holds the row of the current item.

Depending on the current selection mode, the row may also be selected.

Access functions:

int	currentRow() const
void	setCurrentRow(int *row*)
void	setCurrentRow(int *row*, QItemSelectionModel::SelectionFlags *command*)

QTimer Class

The QTimer class provides repetitive and single-shot timers.

Public Functions

	QTimer(QObject **parent* = nullptr)
virtual	~QTimer()
QMetaObject::Connection	callOnTimeout(Functor *slot*, Qt::ConnectionType *connectionType* = Qt::AutoConnection)
QMetaObject::Connection	callOnTimeout(const QObject **context*, Functor *slot*,

QMetaObject::Connection	callOnTimeout(const QObject *_receiver_, MemberFunction *_slot_, Qt::ConnectionType _connectionType_ = Qt::AutoConnection)
int	interval() const
std::chrono::milliseconds	intervalAsDuration() const
bool	isActive() const
bool	isSingleShot() const
int	remainingTime() const
std::chrono::milliseconds	remainingTimeAsDuration() const
void	setInterval(int _msec_)
void	setInterval(std::chrono::milliseconds _value_)
void	setSingleShot(bool _singleShot_)
void	setTimerType(Qt::TimerType _atype_)
void	start(std::chrono::milliseconds _msec_)
int	timerId() const
Qt::TimerType	timerType() const

interval : int

This property holds the timeout interval in milliseconds

The default value for this property is 0. A QTimer with a timeout interval of 0 will time out as soon as all the events in the window system's event queue have been processed.

Setting the interval of an active timer changes its timerId().

Access functions:

int	interval() const
void	setInterval(int *msec*)
void	setInterval(std::chrono::milliseconds *value*)

Distance

$$d_i = \frac{\sum_{k=1}^{N} \sqrt{\left(C[k]_x - T_i[k]_x\right)^2 + \left(C[k]_y - T_i[k]_y\right)^2}}{N}$$

Le geste (transformé) C est comparé au template T_i pour trouver la distance d_i entre chaque point des deux gestes.