# Spécifications Techniques GIMOVapp

## I. Introduction

GIMOVapp est une interface graphique ergonomique et fluide pour la visualisation de données single cell RNA seq et multi-omiques. L'interface est sous la forme d'un site web accessible uniquement par le réseau du CRCL et sera hébergé sur un serveur interne.

Ce document a pour objectif de fournir toutes les spécifications techniques utilisées durant le projet. De l'organisations des fichiers, à l'architecture du code, le développement du logiciel est compartimenté en éléments dont le fonctionnement est expliqué segment par segment.

## Table des matières

# II. Le format Anndata et Mudata

Afin de permettre la visualisation d'un dataset, il est important d'avoir un bon format de stockage sur disque et en mémoire. Un format déjà existant est utilisé pour ne pas faire la conversion des datasets, évitant ainsi de nombreux problèmes possibles suite à des mises à jour. L'analyse de données est faite en R et en Python, les formats les plus récurrents dans ces langages sont `SingleCellExperiment`, `Seurat` et `Anndata`. Anndata a pour avantage d'être le format standard utilisé en analyse de données singlecell en Python. De cette façon, plusieurs datasets sont déjà fournis sous ce format, et les outils d'analyse Python sont déjà compatibles. Dans le but de combler le manque de flexibilité du format Anndata, le format Mudata est aussi pris en charge pour les datasets multi-omiques.

Anndata[1] est un package Python pour la manipulation de matrices de données annotées en mémoire et sur disque, positionné entre pandas et xarray. Anndata offre une large gamme de fonctionnalités efficaces en termes de calcul incluant, entre autres, le support des sparse data, les lazy operations et une interface PyTorch. Mudata est le format du package Muon[2] qui est un Framework Python de Anndata. Il se focalise sur l'analyse de données multimodales et multi-omiques, permettant en plus du singlecell RNA-seq de faire de l'ATAC-seq et du CITE-seq par exemple.

## A. Les données attendues par GIMOVapp

Le format Anndata (qui a pour extension de fichier .h5ad) est peu flexible contrairement à ce qu'on peut avoir l'habitude sur R. Cependant il reste de nombreuses variations possibles dans l'objet python que l'on doit spécifier pour éviter tout problème.

⚠ *Le format Anndata est en train de faire une grosse mise à jour. La matrice principale `X` va être fusionné avec le dictionnaire contenant les autres matrices optionnelles `layers`. La maj se fera sur plusieurs mois ou années, il faudra cependant faire des modifications pour le prendre en compte.*

⚠ Le fichier Anndata ou Mudata doit obligatoirement contenir la clef *`feature_types` dans `var` pour que le type donnée (rna, atac, prot) soit reconnue par GIMOVapp. Sinon le programme renverra un message d'erreur.*

- L'attribut `raw` n'est pas utilisé, les matrice de comptage brut, log normal ou scale supplémentaires à la matrice principale (stocké pour le moment dans `X`) devront être stocké dans `layers`.
- GIMOVapp fait la différence entre les différents types de données stockées dans les Observations `obs` et les Variables `var`. Il y en a 3, les numériques, les facteurs, et les strings. Il est important de vérifier que les données résultantes d'un clustering soit bien sous forme de facteurs.

[1] https://anndata.readthedocs.io/en/latest/
[2] https://muon.readthedocs.io/en/latest/

- GIMOVapp permet d'afficher les embeddings calculé en 3D. Ne pas hésiter à le faire, c'est toujours un plus.
- Les gènes marqueurs sont détectés quand l'objet dans `.uns` est un dictionnaire qui contient les clefs `names`, `pvals`, `scores`, `logfoldchanges` et `pvals_adj`. Par défaut le test de rang de Scanpy[3] utilise ce format, cependant si des gènes marqueurs sont importés il faut aussi qu'ils soient sous ce format. Pour afficher la heatmap correctement, il faut aussi garder en mémoire le groups utilisé par le test de rang, il est attendu dans `.uns[rank_test_name]['params']['groupby']`. Il faut aussi que ce groups soit présent sous le même nom dans ` .obs`.
- Un dataset spatial transcriptomique est lu comme du RNA seq classique à 2 différences prêt. Premièrement on doit retrouver l'embeddings `spatial` dans `.obsm`. Deuxièmement l'image de haute qualité doit être stocké sous forme de matrice numpy dans `.uns['spatial'][nomdudataset]['images']['hires']`. Il est aussi nécessaire d'avoir le facteur d'échelles dans `.uns['spatial'][nomdudataset]['scalefactors']['tissue_hires_scalef']` .
- Pour que la vélocité soit affiché avec un streamplot, `.obsm` doit contenir `velocity_umap` ainsi que `X_umap`.
- Dans l'onglet Pathways il est possible de charger une liste de gènes via un fichier csv ou xlsx. Seul la première colonne est utilisée pour charger les gènes, les autres sont ignorées.

## B. L'utilisation des jeux de données

Une fois le dataset correctement formaté, il peut être sauvegardé sous le bon format avec la fonction `write`. Il devra ensuite être placé dans le dossier `datasets`. Il faudra ensuite l'ajouter dans le fichier contenant tous les datasets à importer ainsi que donner le code à entrer pour l'utiliser.

Lors du lancement de GIMOVapp ce fichier sera lu, et permettra de charger le dataset quand le premier utilisateur le demandera. Chaque dataset ne peux être charger qu'une seul fois pour économiser de la RAM, il est donc commun à tous les utilisateurs qui ont demandé à l'utiliser. Le partage des données entre les utilisateurs ainsi que le chargement et le déchargement des datasets est expliqué en détail dans la partie `La gestion des utilisateurs et des datasets`

[3] https://scanpy.readthedocs.io/en/stable/

# III. Dash et Plotly

La conception de GIMOVapp demande énormément de graphiques et de retour interactifs entre plusieurs composants. L'utilisation d'un Framework est alors nécessaire, pour ne pas repartir de zéro, avec l'html, le css, les scripts js et les graphiques à faire manuellement. Dash a été choisis pour sa compatibilité avec python, ce qui facilite l'utilisation du format Anndata, ainsi que pour ses graphiques et composants simple à mettre en place. Au-delà de sa simplicité apparente Dash[4] permet de créer une interface complète avec la gestion de nombreuses fonctions qui font le lien entre le Python et je JavaScript.

## A. Fonctionnement général

Écrit avec Plotly.js et React.js[5], Dash est idéal pour créer et déployer des applications avec des interfaces utilisateur personnalisées. Il est particulièrement adapté à tous ceux qui travaillent avec des données. Grâce à quelques modèles simples, Dash fait abstraction de toutes les technologies et de tous les protocoles nécessaires à la création d'une application Web complète avec une visualisation interactive des données.

Dash fonctionne avec 2 éléments principaux, les composants et les callbacks. Les callbacks permettent de faire le lien entre plusieurs composant qui vont pouvoir interagir entre eux. Par exemple un menu déroulant va permettre de sélectionner une variable à observer. Le callback va alors appeler une fonction qui va recréer la figure avec la bonne variable.

## B. Les composants html

La majorité des composants utilisé dans GIMOVapp sont des composants html ou des dérivés. Il s'agit de classes JavaScripts, qui ont été traduit en python. On peut alors simplement ajouter un bouton dans notre interface en ajoutant la classe correspondante dans le layout principale du programme. Dash propose un large choix de composants qui vont permettre d'accélérer la création de l'interface.

```python
from dash import html

html.Div([
    html.Div('Example Div', style={'color': 'blue', 'fontSize': 14}),
    html.P('Example P', className='my-class', id='my-p-element')
], style={'marginBottom': 50, 'marginTop': 25})
```

That Dash code will render this HTML markup:

```html
<div style="margin-bottom: 50px; margin-top: 25px;">

    <div style="color: blue; font-size: 14px">
        Example Div
    </div>

    <p class="my-class", id="my-p-element">
        Example P
    </p>

</div>
```

[4] https://plotly.com/dash/ & https://dash.plotly.com/
[5] https://fr.reactjs.org/

Les composants html contienne plusieurs attributs importants. Les plus importants sont `Children`, `Id` et `ClassName`, les 2 derniers sont d'ailleurs directement liés aux balises html comportant aussi les propriétés id et class. En html on peut encapsuler plusieurs balises entre elles, de la même façon, Children est une liste comportant ses composants fils. `Id` est un identifiant unique nécessaire pour utiliser les callbacks, et `ClassName` va permettre de lier notre composant à des classes CSS.

⚠ *Contrairement à l'image ci-dessus il est important de ne pas utiliser l'attribut `style`, pour une meilleure organisation de GIMOVapp tout le css doit être gérer dans le stylesheet. Pour modifier un composant on peut lui donner un `className` que l'on modifiera dans le stylesheet.*

## C. Les composants non html

Dash propose aussi des composants non html, qui sont utilisé comme des outils. Le premier est `dcc.Interval`. Il permet d'appeler en boucles un callback après tous les certains temps donnés en millisecondes. Il est utilisé pour mettre à jour des informations automatiquement et dans la gestion des utilisateurs, afin de suivre leurs connexions et leurs déconnexions.

Deuxièmement on retrouve un composant très important `dcc.Store` il permet de stocker des informations propre à un utilisateur, sans variable global. Dans GIMOVapp il va garder en mémoire le code du dataset afin de pouvoir y accéder librement. Il contient aussi diverses informations sur le dataset qui pourra être utilisé par plusieurs callback pour la création des graphiques.

Il existe plusieurs autre composant comme `dcc.download` qui combiné à `dcc.send` permet d'importer des images et fichiers en tout genre du serveur jusqu'à l'utilisateur.

⚠ *Il existe plusieurs types de storage, qui font varier la portée des informations retenue. GIMOVapp utilise `session`, qui ne sauvegarde pas les données après la fermeture de la fenêtre.*

## D. Les callbacks

Les callbacks sont des décorateurs Python qui permettent de lier des Inputs et des Output de composants (grâce à leur id) avec une fonction fait soi-même. Chaque composant possède des attributs caractéristiques de leur fonction (un slider a par exemple un attribut `value` qui est correspond au nombre sur lequel le curseur est) qui peut être utiliser comme Input, appelant le callback à chacune de ses modifications, comme Output, modifiant cette valeur, ou comme State, permettant de récupérer cette valeur du slider sans appeler de callback par sa modification (utile dans le cas où l'on ne veut pas régénérer une figure à chaque fois par exemple)

⚠ *Les composants (html et non html) ne peuvent avoir leur id lié à un Output que par un seul callback. Il est donc important de faire attention à l'organisation du programme pour ne pas avoir à modifier un composant par plusieurs fonctions. C'est pour cela que dans Overview/callbacks.py on retrouve une fonction gérant toutes les modifications liées au Store.*

Il existe 2 classes très pratiques fournie par Dash pour les callbacks :

- PreventUpdate : permet de quitter les callbacks sans update les Outputs, en effet `return None` ne fonctionne pas car il est attendu autant de d'objet dans le return que d'output. On l'utilise comme lorsqu'on déclare une erreur `raise PreventUpdate`.
- Dash.no_update : dans le cas de plusieurs Outputs permet de ne pas modifier l'un d'eux. Exemple `return « Loading Datasets », Dash.no_update, fig1`. Le deuxième Output du callback ne sera pas mis à jour.

⚠ *Lors du lancement de Dash tous les callbacks sont appelés une fois avec toutes les valeurs d'Input à None. Il est alors très pratique (voir important) de vérifier le type des entrés avant d'utiliser les Inputs et ensuite utiliser PreventUpdate si nécessaire.*

### E. Les figures Plotly

Plotly en plus de fournir Dash fournit aussi de nombreuses figures compatibles avec numpy et pandas, grâces au package `Plotly.express` et `Plotly.graph_objects`. Les fonctions liées aux callbacks vont pouvoir récupérer les Inputs des composants et les utiliser comme paramètre localement, ainsi que le jeu de données charger en mémoire globalement. On va ensuite donnée les infos du fichier Anndata à une fonction express ou graph_objects qui va nous donner une figure. On va ensuite retourner cette figure, qui sera récupérer comme Output par le callback.

Certaine fonction demande de parser plus ou moins le fichier Anndata, par exemple utiliser un histogramme est simple, cependant un sankey plot ou alluvial plot demande une fonction à part (et plus longue) pour récupérer les données sous le bon format.

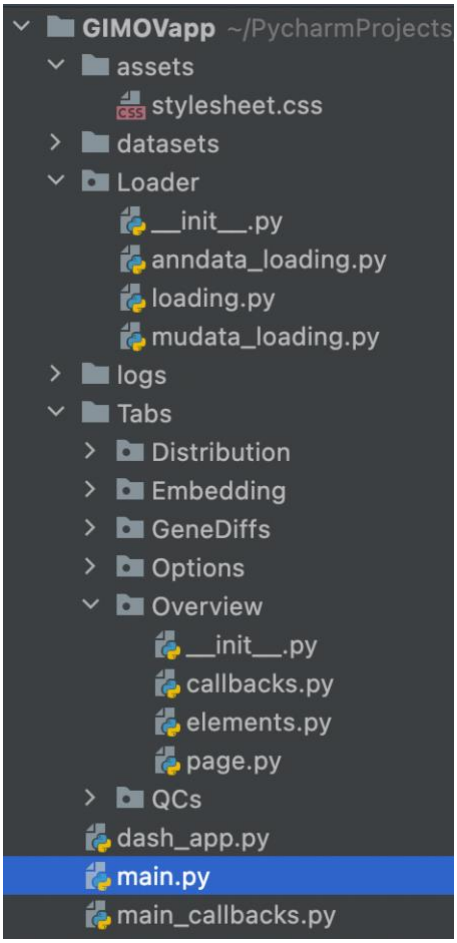
⚠ *Chaque plot à une limite de taille des données en entrée à partir du quelle la fonction va être très lente. Cette limite dépend aussi de l'ordinateur de l'utilisateur (la fonction tourne en JavaScript en local). Il est nécessaire de vérifier cette limite pour chaque plot et de soit fournir un paramètre pour ajuster le nombre de cellules affichées (comme pour la umap) soit de n'afficher qu'un certain pourcentage des cellules (10% pour la heatmap) après vérification que cela ne change pas la figure.*

# IV. Organisation du programme

*⚠ Dash utilise une fonction de debug qui lance le programme 2 fois. Il faut faire attention avec les fonctions asynchrone qui peuvent aussi être lancé 2 fois.*

## A. L'organisation des fichiers

Les fichiers sont organisés comme l'image ci-contre. On retrouve de bas en haut :




- `main_callbacks.py` qui contient les fonctions concernant l'application en général, que ce soit pour la gestion des utilisateurs, l'écritures des logs ou autre.
- `main.py` qui contient le layout principal de GIMOVapp ainsi que la fonction pour lancer l'application.
- `dash_app.py` contient tous les objets qui vont être utiliser avec des variables globales à travers le programme. On les met dans un fichier à part pour éviter les importation circulaire (arrêtant le programme)
- `Tabs` qui contient un dossier par onglet dans GIMOVapp. Chaque dossier à les 4 même fichiers, `__init__.py`, `callbacks.py` qui contient tous les callbacks liés à l'onglets, `elements.py` qui contient tous les composants de l'onglet et `page.py` qui arrange les composants comme montré sur l'image du `B. les composants HTML`.
- `logs` qui contient tous les fichiers logs sorties par le programme
- `Loader` contient plusieurs scripts permettant de charger en mémoire et parser les fichier Anndata et Mudata.
- `datasets` contient les datasets, ainsi que le fichier contenant le code d'accès de chaque datasets et d'un fichier buffer permettant de ne pas lancer les fonction asynchrones 2 fois (comme expliqué dans le ⚠ au-dessus)
- `assets` contient le `stylesheet.css` qui est automatiquement pris en compte par Dash

*⚠ `assets` et `stylesheet.css` ne doivent ni être renommés ni être déplacés pour être pris en compte par Dash*

## B. La structure du programme

Les différents types d'éléments du programmes, composant html, css, callbacks, fonctions python, ont été séparé en plusieurs scripts au maximum afin de clarifier le code. C'est pour cela que dans chaque dossier de `Tabs` on retrouve la même structure, avec les composant séparés du rendu html, séparé du callback. Faire ça a pour avantage de minimiser les changements à faire lors d'une modification. Le changement de css n'impact que le fichier stylesheet.css et la modification d'un callback n'entrainera pas de problème sur la disposition des composants.

Cette méthode permet aussi de faciliter l'ajout d'onglet. Il suffit pour cela de rajouter un onglet dans le `main`, puis de créer le dossier correspondant avec ses 3 fichiers principaux. On retrouve alors un peu de redondance, mais c'est pour garder une indépendance totale entre les différents onglets. On peut donc travailler dessus sans modifier le reste.

Le script `main.py` est le cœur du programme, il va appeler tous les autres scripts nécessaires au fonctionnement de GIMOVapp. Les callbacks présents dans `main_callbacks.py` et dans les différents dossiers correspondant aux onglets ont juste besoin d'être importés dans le `main` afin d'être pris en compte par Dash. Ensuite on doit importer tous les layouts de chaque onglet dans le `main` pour les attribués aux onglets dans l'interface.

L'ajout des onglets d'analyse (Pathways et Clustering) force une légère modification dans la structure du programme. En effet toutes les données pouvant être affiché par GIMOVapp (observations, embeddings, etc.) sont stockées dans un `dcc.store`. Or l'architecture de Dash fait qu'il n'est possible de modifier cette valeur que dans un unique callback. Il est donc nécessaire de renvoyer le résultat des différents onglets d'analyse dans un même callback dans `Overview`. Pour cela on ajoute au callback l'input d'une div html invisible qui sera modifié pour contenir les informations à rajouter dans `dcc.store`. Le callback va à l'aide du contexte modifié `dcc.store` en fonction d'où provient l'input ( chargement de données, Clustering, Pathways, déconnexion, etc.)

# V. La gestion des utilisateurs et des datasets

## A. Importation des graphiques

Dash propose dans ses graphiques interactifs une fonction pour sauvegarder au format png le graphique actuel. C'est une fonction bien intégrée et rapide, cependant elle ne convient pas à l'utilisation de GIMOVapp. En effet la taille et la qualité de l'image peut être insuffisant pour l'utilisateur, ou même sous le mauvais format.

Pour régler ce problème la seule solution actuellement trouvé et d'ajouter un bouton extérieur au graphique interactif. Le bouton va appeler un callback qui va récupérer le graphique voulu, puis l'écrire sur le serveur et enfin l'envoyer à l'utilisateur. De cette façon on peut télécharger l'image sous le format et la taille voulu. Actuellement le format svg est privilégier car les dimensions ne sont pas à indiquées (format vectorielle)

⚠ *En contre partie de sa flexibilité la méthode pour télécharger les graphiques sous format svg est très lente. Premièrement parce que la figure récupérer par le callback est un dictionnaire qu'il faut retransformer en objet figure de Plotly. Deuxièmement car l'écriture sous format svg de la figure est lente.*

## B. Connexions et déconnexions

Les utilisateurs de GIMOVapp utilisent des datasets de plusieurs gigas chargés en mémoire sur le serveur. Il est donc important de suivre la connexion et déconnexion des utilisateurs afin de n'utiliser que le minimum de mémoire possible.

Un dictionnaire garde en mémoire global, les utilisateurs par dataset chargé en mémoire. À la déconnexion d'un utilisateur le dictionnaire est vérifié, s'il n'est plus utilisé alors il est déchargé. De la même façon un dataset n'est chargé qu'une seule fois, si le dataset est déjà chargé alors il sera partagé par les utilisateurs (ayant le code d'accès).

Un timer intégré avec dcc.Interval (et donc indépendant entre chaque utilisateur) permet de savoir combien de temps passe entre chaque update de graphique. Cela permet de détecter les utilisateurs inactifs, puis de réinitialiser les datasets utilisées ainsi que les données dans sa session.

Dash ne permet pas de détecter la déconnexion d'un utilisateur, il est pour primordial de le faire car si un utilisateur quitte l'interface sans être inactif avant, le dataset ne sera jamais déchargé. Afin de palier à ce problème un dictionnaire global enregistre pour chaque utilisateur, son ID, IP, ainsi que la dernière fois que le timer à appeler la fonction qui vérifie l'état de l'utilisateur. Une fois l'utilisateur déconnecter le timer ne fonctionne plus et le dictionnaire n'est plus mis à jour. Une fonction asynchrone se lance toutes les minutes pour vérifier les utilisateurs qui n'ont pas reçu leur mise à jour par le timer. Au bout de 2min (soit environ 2 intervalles du timer) La fonction asynchrone considère l'utilisateur comme déconnecté et le supprime des différents objets globaux, tout en déchargeant un dataset si nécessaire.

## C. Les logs

Les logs permettent en cas de problème quelconque avec GIMOVapp d'avoir une trace des interactions entre les utilisateurs et l'interface. Les logs sont actuellement affichés sur le terminal et enregistrer dans un fichier a part. Ils enregistrent pour le moment la connexion d'un utilisateur, sa déconnexion, le chargement et déchargement d'un dataset.

Les logs de GIMOVapp ont un format spécifique qui permet de parser les fichiers résultants plus facilement. On retrouve les champs suivant séparer par une tabulation :

- La date sous le format `aaaa-mm-jj`
- L'heure sous le format `hh:mm:ss`
- L'adresse ip de l'utilisateur ou `GIMOV-system` si l'action vient du programme lui même
- Le numéro de session ou -1 si l'action vient du programme lui même
- Le message indiquant la nature de l'action qui vient d'être réalisé.

Afin de garder ce format plus facilement, l'écriture des logs passe par une fonction dans main_callbacks.py qu'il faut importer dans le fichier voulu pour être utiliser.