cajal

Release 0.30

Cámara Lab

Jul 21, 2023

OVERVIEW AND WALKTHROUGH

CAJAL is a Python package designed to explore and analyze the morphology of cells and its relationship with other single-cell data.

CAJAL leverages the Python Optimal Transport library to compute the Gromov-Wasserstein (GW) distance between every pair of cells in a given sample. This distance quantifies the degree to which the shape of one cell can be transformed into that of another with minimal stretching or bending. One of the key benefits of using the GW distance is that it does not require any prior knowledge or model for the morphology of the cells. This feature makes CAJAL suitable for studying arbitrarily heterogeneous mixtures of cells with highly complex and diverse morphologies that may defy straightforward classification.

The morphological distance produced by CAJAL is a bona-fide mathematical distance in a latent space of cell morphologies. In this latent space, each cell is represented by a point, and distances between cells indicate the amount of physical deformation needed to change the morphology of one cell into that of another. By formulating the problem in this way, CAJAL can make use of standard statistical and machine learning approaches to define cell populations based on their morphology; dimensionally reduce and visualize cell morphology spaces; and integrate cell morphology spaces across tissues, technologies, and with other single-cell data modalities, among other analyses.

CHAPTER

ONE

WHAT IS CAJAL?

CAJAL is a general computational framework for the multi-modal analysis and integration of single-cell morphological data. It builds upon recent advances in applied metric geometry and shape registration to enable the characterization of morphological cellular processes from a biophysical perspective and produce a mathematical distance function upon which algebraic and statistical analytic approaches can be built.

In its simplest form, the study of cell morphology involves comparing cell shapes irrespective of distance-preserving transformations such as rotations and translations. To facilitate this, CAJAL internally represents a cell as a list of points randomly sampled from the surface of the cell (usually between 50 and 200), together with a matrix of the (Euclidean or geodesic) pairwise distances between these points in the cell, known as the intracellular distance matrix.

In a hypothetical scenario where computational speed is infinite, comparing two intracellular distance matrices would be conducted as follows. Consider cells *A* and *B*, each containing 50 selected sample points. The distance between sample points *i* and *j* in cell *A* can be denoted as *A(i,j)*. If we have a pairing f between the sample points of *A* and *B*, we can consider f as an attempt to superimpose *A* on *B*. The distortion of *A* that arises from this pairing can be quantified by:

$$\Gamma_f = \max_{i,j \in A} |A(i,j) - B(f(i), f(j))|$$

This quantifies how much *A* has to be deformed or stretched in order to overlay it on *B* along the given pairing.

The Gromov-Hausdorff distance between *A* and *B* is then defined as the distortion arising from the best possible pairing, when all possible pairings are considered.

$$d_{GH}(A,B) = \min_{f: A \cong B} \max_{i,j \in A} |A(i,j) - B(f(i), f(j))|$$

Unfortunately, this quantity cannot be computed in practice, as there are 50! or about 3x10^64 ways to give a one-to-one pairing between the points of A and B, and we cannot search through all of these. Therefore, CAJAL relies on a more computationally efficient approximation, the Gromov-Wasserstein distance. Both the Gromov-Hausdorff distance and the Gromov-Wasserstein distance satisfy the axioms for a metric, giving a sensible and reasonably well-behaved notion of distance.

CAJAL provides tools to compute the pairwise Gromov-Wasserstein distance between all cells in a directory of cell image data and exploring, interpreting, and analyzing the resulting cell morphology latent space. For example, the user can use clustering approaches to identify groups of cells with similar morphology and predict features of new cells by comparing their shape with other cells. They can also investigate whether a cell feature is highly correlated with its morphology. CAJAL provides tools for exploring, interpreting, and analyzing the cell morphology latent space produced by CAJAL.

CAJAL is written and developed by the Cámara Lab at the University of Pennsylvania. More information about the theoretical foundations of CAJAL can be found at:

- Govek, K. W., et al. CAJAL enables analysis and integration of single-cell morphological data using metric geometry. Nature Communications 14, 3672 (2023).

- Memoli, F. On the use of Gromov-Hausdorff distances for shape comparison. Eurographics Symposium on Point-Based Graphics (2007).

- Memoli, F. Gromov–Wasserstein distances and the metric approach to object matching. Foundations of computational mathematics 11, 417-487 (2011).

- Memoli, F. & Sapiro, G. A theoretical and computational framework for isometry invariant recognition of point cloud data. Foundations of Computational Mathematics 5, 313-347 (2005).

CHAPTER

TWO

COMPUTING INTRACELLULAR DISTANCE MATRICES

CAJAL represents a cell as a finite set of uniformly sampled points from its outline together with a notion of distance between each pair of points. This cell data is internally represented as an intracellular distance matrix, where each rows and column corresponds to a point in the cell, and the entry at position (i, j) denotes the distance between points x_i and x_j. Typically, 50 to 200 sampled points per cell are sufficient for most applications.

To compute the Gromov-Wasserstein distance between two cells, users need to first convert their cell morphology data into intracellular distance matrices. In this regard, CAJAL offers functionality that supports three kinds of input data files neuronal tracing data (SWC files), 3D meshes (OBJ files), and 2D cell segmentation files (TIFF files). This section describes how to leverage this functionality to produce intracellular distance matrices that enable users to perform Gromov-Wasserstein distance computations.

2.1 Euclidean vs. geodesic distances

CAJAL supports two types of intracellular distances matrices: Euclidean distance, which is the ordinary straight-line distance through the ambient space, and geodesic distance, which is the length of the shortest path through the surface of the cell. The choice between using Euclidean or geodesic distance affects the types of deformations that CAJAL considers relevant when comparing the shape of two cells.

Using Euclidean distance to measure intracellular distances results in morphological distances that are insensitive to translations, rotations, or mirroring of a cell. However, bending or flexing a cell will change the morphological distance between that cell and other cells. On the other hand, using geodesic intracellular distances leads to morphological distances that are insensitive to translations, rotations, mirroring, bending, and flexing of the cells.

To illustrate the distinction, consider two pieces of string, A and B, both of which are twelve inches long. If A is laid out in a straight line and B is tightly coiled, then the Gromov-Wasserstein distance between them will be nontrivial in they are represented by Euclidean intracellular distance matrices. This is because one must bend B to straighten it out into a line segment. However, if they are represented by their geodesic distance matrices, then the Gromov-Wasserstein distance will be zero. This is because one can deform A into B without any stretching or elongating, as they are the same length.

2.2 Neuronal Tracing Data

CAJAL supports neuronal tracing data in the SWC specification defined here. You can find examples of *.swc files compatible with CAJAL can be found in the CAJAL Github repository under `CAJAL/data/swc_files`.

The package provides two functions that operate on directories of *.swc files. *cajal.sample_swc.compute_icdm_all_euclidean()* and *cajal.sample_swc.compute_icdm_all_geodesic()*. These functions generate intracellular distance matrices for each cell in the source directory and populate a *.csv file with the results.

For example, if you have a directory called */home/jovyan/CAJAL/CAJAL/data/swc_files* that contains *.swc files and you want to write the intracellular distance matrices to a *.csv file called */home/jovyan/CAJAL/CAJAL/data/swc_icdm.csv*, you can use the following code.

```
failed_cells = sample_swc.compute_icdm_all_euclidean(
    infolder = "/home/jovyan/CAJAL/CAJAL/data/swc_files",
    out_csv= "/home/jovyan/CAJAL/CAJAL/data/swc_icdm.csv",
    n_sample = 50,
    preprocess=swc.preprocessor_eu(
        structure_ids=[1,3,4],
        soma_component_only=True),
    num_processes = 8
    )
```

The *n_sample* argument specifies the number of points from each cell that will be sampled (we recommend between 50-100). The *num_processes* argument specifies the number of processes that will be launched in parallel, and we recommend setting it to the number of cores on your machine.

The *preprocess* argument is optional and can be used to filter out some neurons from being sampled, for reasons of data quality, and/or transform the remaining data before sampling from it. The argument is very flexible and can be used in many ways. For convenience, two specific use cases are built-in. The line *structure_ids = [1,3,4]* indicates that samples will only be drawn from the node types corresponding to 1, 3 and 4 in the SWC specification, i.e., the soma and basal and apical dendrites. This can be useful when the user has a mixture of full neuronal reconstructions and dendrite-only neuronal reconstructions and wants to discard the axons from the full neuronal reconstructions. To keep all node types, set *structure_ids = "keep_all_types"*. The argument *soma_component_only=True* indicates that the function will only sample from the unique component of the neuron containing the soma, and will write to an error log any neurons which do not contain a unique component containing nodes labeled as soma. This illustrates the basic function of the preprocessing function, in this example filtering out all neurons which don't have a unique soma node, and transforming the remaining neurons by discarding all components except the one containing the unique soma node. To keep all connected components, set *soma_component_only=False*.

The function returns a list called *failed_cells* that contains the names of the cells for which sampling was unsuccessful (i.e., the preprocessing function returned an error) together with the error itself. If the sampling is successful, the results are silently written to a file.

A similar functionality is implemented in *cajal.sample_swc.compute_icdm_all_geodesic()* with respect to the computation of intracellular geodesic distances.

2.3 3D meshes

CAJAL provides support for Wavefront *.obj 3D mesh files. The package expects each line of a mesh file to be one of the following.

- A comment, marked with a "#"
- A vertex, written as *v float1 float2 float3*
- A face, written as *f linenum1 linenum2 linenum3*

Examples of *.obj files compatible with CAJAL can be found in the CAJAL Github repository under the folder `CAJAL/data/obj_files`.

It is important to note that a *.obj file may contain several distinct connected components. By default, CAJAL separates these components into individual cells. However, in situations where a *.obj file is supposed to represent a single cell but has multiple disconnected components due to measurement errors, the package provides functionality to create a new mesh where all components are joined together by new faces. This allows for the computation of a geodesic

distance between points in the mesh. If the user wants to compute the Euclidean distance between points, such repairs are unnecessary, as the Euclidean distance is insensitive to connectivity.

CAJAL provides one batch-processing function that goes through all *.obj files in a given directory, separates them into connected components, computes intracellular distance matrices for each component, and writes all these square matrices to a *.csv file. For example,

```
failed_samples = sample_mesh.compute_icdm_all(
            infolder="/home/jovyan/CAJAL/data/obj_files",
            out_csv="/home/jovyan/CAJAL/data/sampled_pts/obj_geodesic_50.csv",
            metric = "segment",
            n_sample=50,
            num_cores=8,
            segment = True,
            method="heat"
            )
```

The arguments *infolder, out_csv, n_sample, metric* are as in *Neuronal Tracing Data*, except that *infolder* is a folder containing *.obj files rather than *.swc files.

If the Boolean flag *segment* is True, the function will break down each *.obj file into its connected components and treat them as individual, isolated cells. If *segment* is set to False, the function will treat each *.obj file as a single cell. If the user chooses the "geodesic" metric and the contents of a *.obj file are not connected, CAJAL will automatically attempt to "repair" the cell by modifying it to adjoin new paths between connected components, so that a geodesic distance between points can be defined.

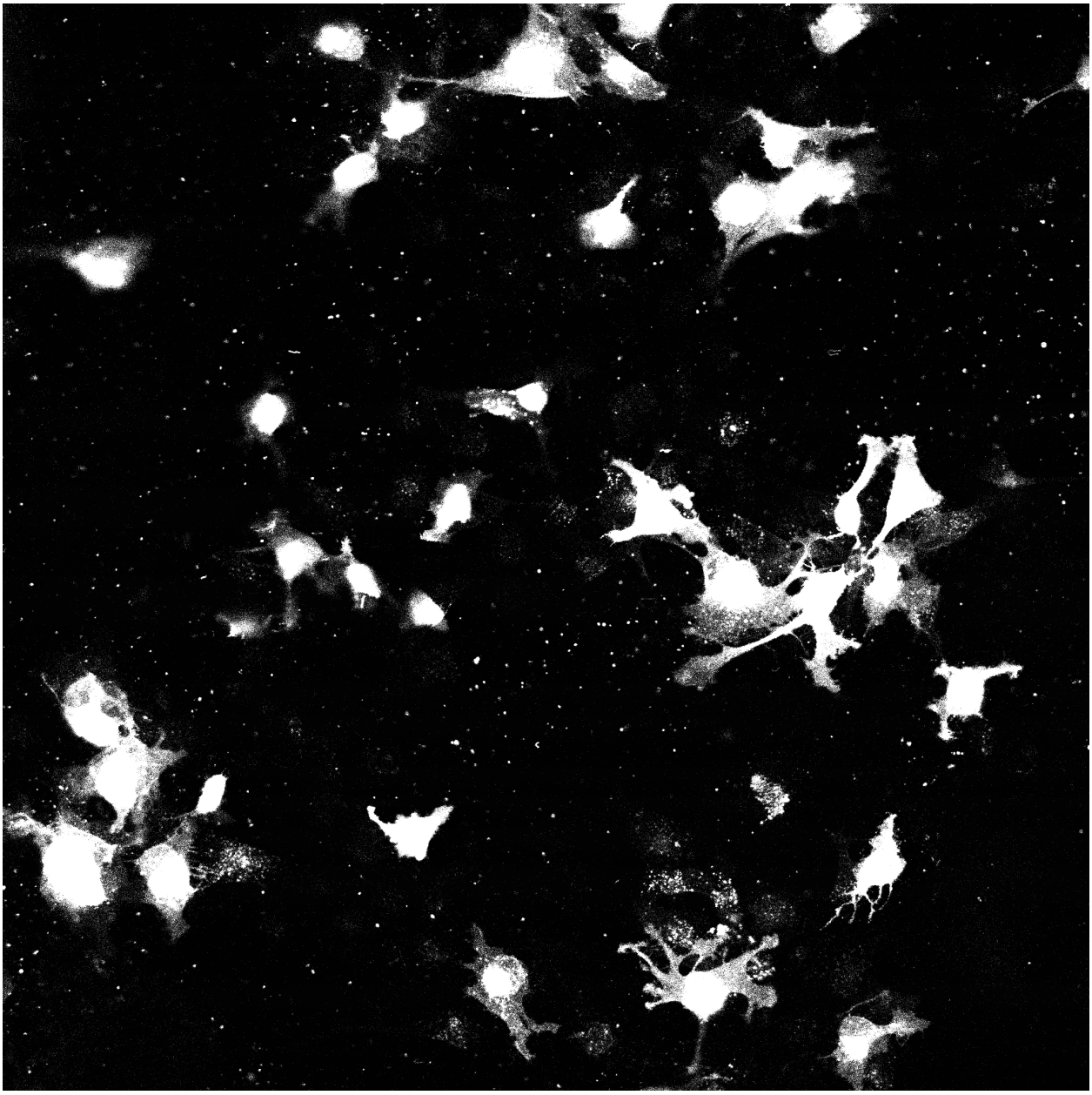
Warning: Modifying the data by adjoining new triangles to the mesh is an imputation of data which changes its topology. This presents the same thorny questions as in any other scenario when data is imputed, and the user should keep this in mind while interpreting the data. The functionality of "repairing" the cell is premised on the assumption that the *.obj file represents a single geometric object and that it fails to be connected for trivial reasons. If a *.obj file genuinely contains multiple distinct components, then the geodesic distances resulting from this process will not be meaningful.

2.4 Segmentation files

Image segmentation is the process of separating an image into distinct components to simplify the representations of objects. Morphological segmentation is one approach to image segmentation based on morphology. While CAJAL provides tools to sample from the cell boundaries of segmented image files, it is important to note that CAJAL is not a tool for image segmentation itself. Users are expected to segment and clean their own images.

To help users prepare their data for use with CAJAL, we provide a basic example using images from the CAJAL Github repository (`CAJAL/data/tiff_images`).

Let us consider the following image



The OpenCV package provides some basic functionality to clean image data and perform segmentation. Users can use the `cv.imread()` function to load *.tiff files into memory.

```
import tifffile

img=tifffile.imread(CAJAL/data/tiff_images/epd210cmd1l3_1.tif)
```

We then recommend collapsing the greyscale image to black and white and performing dilation followed by erosion and erosion followed by dilation to remove noise and small holes.

```
import cv2 as cv
import numpy as np

_, thresh = cv.threshold(img,100,255,cv.THRESH_BINARY)
```

(continues on next page)

(continued from previous page)

```
kernel = np.ones((5,5),np.uint8)
closing = cv.morphologyEx(thresh, cv.MORPH_CLOSE, kernel)
closethenopen = cv.morphologyEx(closing, cv.MORPH_OPEN,kernel)
```

Afterward, users can label each connected region of the image with a unique common color.

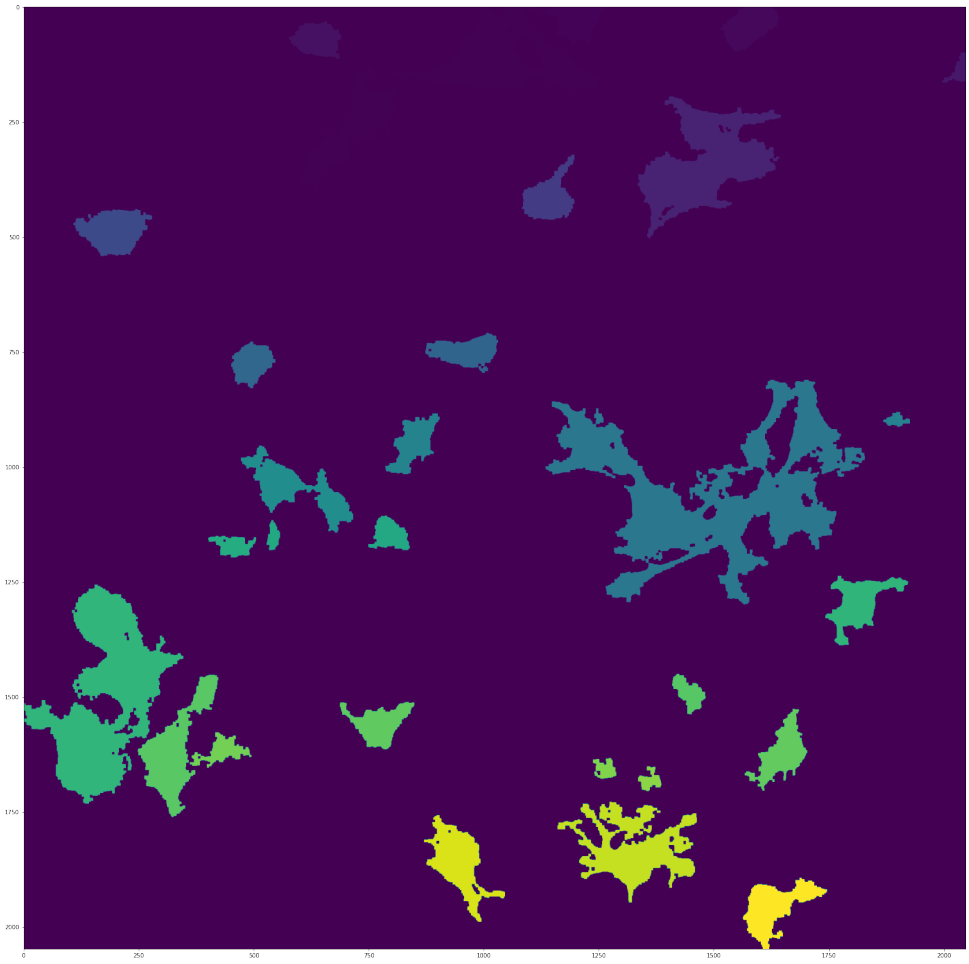
```
from skimage import measure

labeled_img = measure.label(closethenopen)
```

The image is still somewhat noisy, with a few specks in it. To despeckle it, we can remove all connected regions with fewer than 1000 pixels by grouping them into the background region, which is labelled with 0.

```
labels = np.unique(labeled_img, return_counts=True)
labels = (labels[0][1:],labels[1][1:])
remove = np.isin(labeled_img, labels[0][labels[1]<1000])
img_keep = labeled_img.astype(np.uint8)
img_keep[remove] = 0
```

We can use matplotlib to view the image from an interactive environment like Jupyter notebook.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.imshow(simplify_img_keep)
fig.set_size_inches(30, 30)
plt.show()
```



This image is representative of the type of images that CAJAL is meant to process: a 2D array of integers, where each cell is represented by a connected block of integers with the same value. Two distinct cells should have different values, and each cell should have a different value than the background.

We can write the cleaned image to a file using `tifffile.imwrite()`.

```
tifffile.imwrite('/home/jovyan/CAJAL/CAJAL/data/cleaned_file.tif',
img_keep, photometric='minisblack')
```

It is essential to note that this is only a toy example. For instance, in this image multiple overlapping cells have been grouped into a single mask. Users would normally discard such overlapping cells before analysis with CAJAL.

To sample points and compute intracellular distances from *.tiff / *.tif files like these, CAJAL provides the function *cajal.sample_seg.compute_icdm_all()*. This function takes an input directory full of cleaned *.tiff/*.tif files and an output directory as arguments. For each *.tiff file in the input directory, *cajal.sample_seg.*

compute_icdm_all() breaks the image down into its separate cells, samples a given number of points for each one, and writes the resulting resulting intracellular distance matrix for each cell to a single collective database for all files in the directory.

```
infolder ="/home/jovyan/CAJAL/CAJAL/data/tiff_images_cleaned/"
out_csv="/home/jovyan/CAJAL/CAJAL/data/tiff_sampled_50.csv"
sample_seg.compute_icdm_all(
               infolder,
               out_csv,
               n_sample = 50,
               num_cores = 8,
               background = 0,
               discard_cells_with_holes = False,
               only_longest = False
               )
```

infolder specifies the input directory of cleaned *.tiff/*.tif files, *db_name* indicates the name of the database file, and *n_sample* the number of points to sample from each cell. *background* is the index for the background color, which is 0 by default. If the flag *discard_cells_with_holes* is set to True, the function will ignore any cells that have multiple boundaries. The argument *only_longest* is only relevant if *discard_cells_with_holes* is False. In this case if *only_longest* is True, then the function only samples from the longest boundary of the cell instead of across all boundaries. Cells that meet the image boundary are discarded.

CHAPTER

THREE

COMPUTING GW DISTANCES

To compute the Gromov-Wasserstein (GW) distance between intracellular distance matrices, users can employ the function *cajal.run_gw.compute_gw_distance_matrix()*.

This section assumes that the user has already obtained the intracellular distance matrices for their cells. It is worth noting that the GW distance can be calculated using the same function regardless of how the intracellular distance matrices were computed and whether they represent the Euclidean or geodesic metric.

To use the function, the user should provide the path to an input *.csv database containing the intracellular distance matrices through the argument *intracell_csv_loc*. The output GW distance matrix will be saved in a new .csv file specified by the argument *gw_dist_csv_loc*.

```
run_gw.compute_gw_distance_matrix(
    intracell_csv_loc = "/home/jovyan/CAJAL/CAJAL/data/swc_icd.csv",
    gw_dist_csv_loc = "/home/jovyan/CAJAL/CAJAL/data/gw_dists.csv",
    num_processes = 8)
```

By default, the coupling matrices which represent the best possible pariting between two cells are not retained. However, *cajal.run_gw.compute_gw_distance_matrix()* also provides functionality to save the coupling matrices. This is for instance required for the computation of average cellular shapes.

Numpy should automatically parallelize the computation across multiple cores. Users on Windows can check the process manager, while those on Unix-based systems can use the "top" command to verify that all cores are being utilized.

Warning: Note that storing the coupling matrices will generate a large amount of data, which scales quadratically with the number of input cells. For example, if there are 150 cells with 50 sampled points each, the resulting database size may be approximately 180MB. File IO may also become a bottleneck in the computation. Therefore, users should exercise caution when saving the coupling matrices, especially when working with a large number of cells.

CHAPTER

FOUR

INFERRING ASSOCIATIONS WITH CELL MORPHOLOGY

The Laplacian Score is a statistical test implemented in CAJAL to determine whether differences in a numerical feature assigned to cells, $f : G \to \mathbb{R}$, such as the expression of a gene or the genotype of the cell in a given locus, are related to differences in cell morphology. Specifically, the Laplacian Score answers the question: if x and y are two cells with similar morphology, are $f(x)$ and $f(y)$ closer on average than if x and y were chosen randomly?

To perform this analysis, CAJAL uses the Gromov-Wasserstein distance between every pair of cells to construct an undirected graph G where nodes represent cells and edges connect cells with distances less than ε, a user-specified positive real parameter. The Laplacian score of f with respect to the graph G is positive number defined by

$$C_G(f) = \frac{\sum_{(i,j)\in E(G)}(f(i) - f(j))^2}{\mathrm{Var}_G(f)}$$

where $E(G)$ is the set of edges in the graph, i, j range over nodes of G, and $\mathrm{Var}_G(f)$ is the weighted variance of *f,* where the weight of node i is proportional to the number of neighbors of i in G. When the Laplacian Score is close to zero, this indicates that the values of f tend to be similar between connected cells.

To test the significance of the Laplacian Score, CAJAL provides a permutation test that shuffles the values of f across the nodes of G to generate a null distribution, from which a p-value can be computed. Additionally, CAJAL supports regression analysis to account for the influence of other covariates, $g_1, \ldots, g_n$, defined on G. Users can fit a multivariate linear regression model to remove the dependence of $C_G(f)$ on $C_G(g_1), \ldots, C_G(g_n)$, and evaluate whether the Laplacian Score of f is below what would be expected from the covariate features.

Overall, the Laplacian Score implemented in CAJAL provides a flexible approach for analyzing the relationship between cell morphology and numerical features, with the ability to account for other covariates and assess statistical significance.

More information about the theoretical foundations of the Laplacian score can be found at:

- Govek, K. W., et al. CAJAL enables analysis and integration of single-cell morphological data using metric geometry. Nature Communications 14 (2023) 3672.

- Govek, K. W., Yamajala, V. S., and Camara, P. G. Clustering-Independent Analysis of Genomic Data using Spectral Simplicial Theory. PLOS Computational Biology 15 (2019) 11.

- He, X., Cai, D., and Niyogi, P. Laplacian Score for Feature Selection In Advances in neural information processing systems (2005) 507-514.

CHAPTER

FIVE

COMPUTING AVERAGE CELL SHAPES

When computing the Gromov-Wasserstein distance between two cells X and Y, the optimal transport algorithm returns two pieces of information:

1. A *coupling matrix*, which represents the optimal probabilistic mapping of X onto Y that minimizes the distortion.
2. The distortion induced by this optimal coupling matrix, which is the Gromov-Wasserstein distance.

We can utilize the coupling matrix to construct a morphological average of a group or cluster of cells. CAJAL implements an algorithm called `avg_shape_spt()` to construct this morphological average. In brief, the algorithm proceeds as follows:

- Identify the *medoid cell* of the cluster, which is the cell that has the lowest average distance to the other cells.
- Use the optimal coupling matrices to reorient every other cell with respect to the medoid, so that they can be directly compared.
- Rescale all intracellular distance matrices to unit step size to ensure that differences in overall size do not distort the comparison.
- Cap the distance between points within each cell at 2. This destroys information about the global structure of the geodesic distances, preventing very distant points from having an outsize effect.
- Compute the arithmetic mean of all distance matrices, where the distance between any two points in the averaged matrix is the average distance between the corresponding pairs of points in each cell in the cluster.
- For neurons, construct a shortest-path tree through the weighted graph encoded by the average distance matrix. This tree represents the average neuronal morphology of the cluster.

CHAPTER

SIX

TUTORIAL 1: PREDICTING THE MOLECULAR TYPE OF NEURONS

To demonstrate some of the main functionalities of CAJAL, here we perform some basic analysis on a set of neuron morphological reconstructions obtained from the Allen Brain Atlas. To facilitate the analysis, we provide a compressed *.tar.gz file containing the *.SWC files of 509 neurons used in this example, which can be downloaded directly from this link. In this tutorial we assume that the SWC files are located in the folder `/home/jovyan/swc`. More information about this dataset can be found at:

- Gouwens, N. W. et al. Classification of electrophysiological and morphological neuron types in the mouse visual cortex. Nat Neurosci 22, 1182-1195 (2019).

For this analysis, we focus on the morphology of the dendrites and exclude the axons of the neurons. To achieve this, we set `structure_ids = [1,3,4]`, which tells CAJAL to only sample points from the soma and the basal and apical dendrites. We sample 100 points from each neuron and compute the Euclidean distance between each pair of points in that neuron using the following code:

[2]:
```
import cajal.sample_swc
import cajal.swc

cajal.sample_swc.compute_icdm_all_euclidean(
    infolder="/home/jovyan/swc",
    out_csv="/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv",
    preprocess=cajal.swc.preprocessor_eu(
        structure_ids=[1,3,4],
        soma_component_only=False),
    n_sample=100,
    num_processes=8)  # num_processes can be set to the number of cores on your machine
```

```
100%|| 508/509 [06:02<00:00,  1.40it/s]
```

[2]:
```
[]
```

Once the sampling is completed, we compute the Gromov-Wasserstein distance between each pair of neurons. To compute the Gromov-Wasserstein distance matrix we use the code:

[3]:
```
import cajal.run_gw

cajal.run_gw.compute_gw_distance_matrix(
    "/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv",
    "/home/jovyan/swc_bdad_100pts_euclidean_GW_dmat.csv",
    num_processes=8)
```

```
100%|| 129286/129286 [03:52<00:00, 556.95it/s]
```

[3]:
```
(array([[  0.        ,  76.53525355,  48.81215985, ...,  36.25765651,
          39.63267218, 107.27192268],
```

(continues on next page)

(continued from previous page)

```
        [ 76.53525355,   0.        ,  90.55259238, ...,  69.27173625,
          82.74822498,  50.54451328],
        [ 48.81215985,  90.55259238,   0.        , ...,  26.48503494,
          16.99102489, 129.81156708],
        ...,
        [ 36.25765651,  69.27173625,  26.48503494, ...,   0.        ,
          21.15960915, 107.41792624],
        [ 39.63267218,  82.74822498,  16.99102489, ...,  21.15960915,
           0.        , 121.93211717],
        [107.27192268,  50.54451328, 129.81156708, ..., 107.41792624,
         121.93211717,   0.        ]]),
 None)
```

We can visualize the resulting space of cell morphologies using UMAP:

[1]:
```
import plotly.io as pio
pio.renderers.default = 'iframe'

import cajal.utilities
import umap
import plotly.express

# Read GW distance matrix
cells, gw_dist_dict = cajal.utilities.read_gw_dists("/home/jovyan/swc_bdad_100pts_
↪euclidean_GW_dmat.csv", header=True)
gw_dist = cajal.utilities.dist_mat_of_dict(gw_dist_dict)

# Compute UMAP representation
reducer = umap.UMAP(metric="precomputed", random_state=1)
embedding = reducer.fit_transform(gw_dist)

# Visualize UMAP
plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in cells])
```

```
2023-07-20 21:31:24.609344: I tensorflow/core/platform/cpu_feature_guard.cc:193] This␣
↪TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use␣
↪the following CPU instructions in performance-critical operations:  SSE4.1 SSE4.2 AVX␣
↪AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler␣
↪flags.
/opt/conda/lib/python3.10/site-packages/umap/umap_.py:1780: UserWarning: using␣
↪precomputed metric; inverse_transform will be unavailable
  warn("using precomputed metric; inverse_transform will be unavailable")
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels␣
↪instead.
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Once the user has computed the Gromov-Wasserstein distances between cells it is possible to cluster the cells using standard clustering techniques. The Louvain and Leiden clustering algorithms are two commonly used algorithms to identify communities within large networks; they can be adapted to finite metric spaces by constructing a k-nearest-neighbors graph on top of the metric space. CAJAL provides access to both of these clustering algorithms. When combined with a low-dimensional embedding tool such as the UMAP algorithm, the user can plot the clusters in a 2-dimensional embedding and visualize them. Here, we use the Leiden algorithm to cluster the neurons based on their morphology:

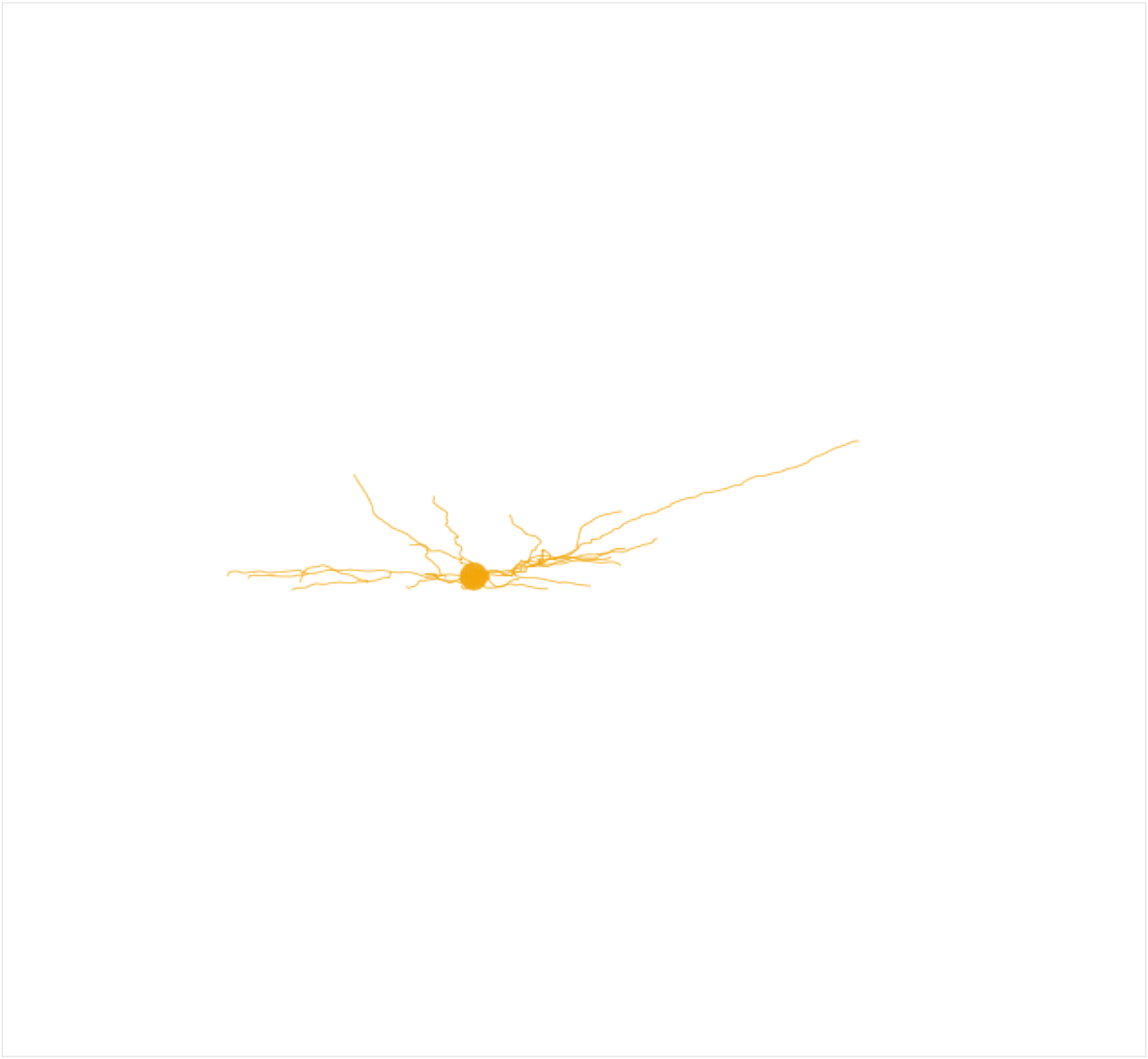
[2]:
```
clusters = cajal.utilities.leiden_clustering(gw_dist, seed=1)
plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in cells],
                       color = [str(m) for m in clusters])
```

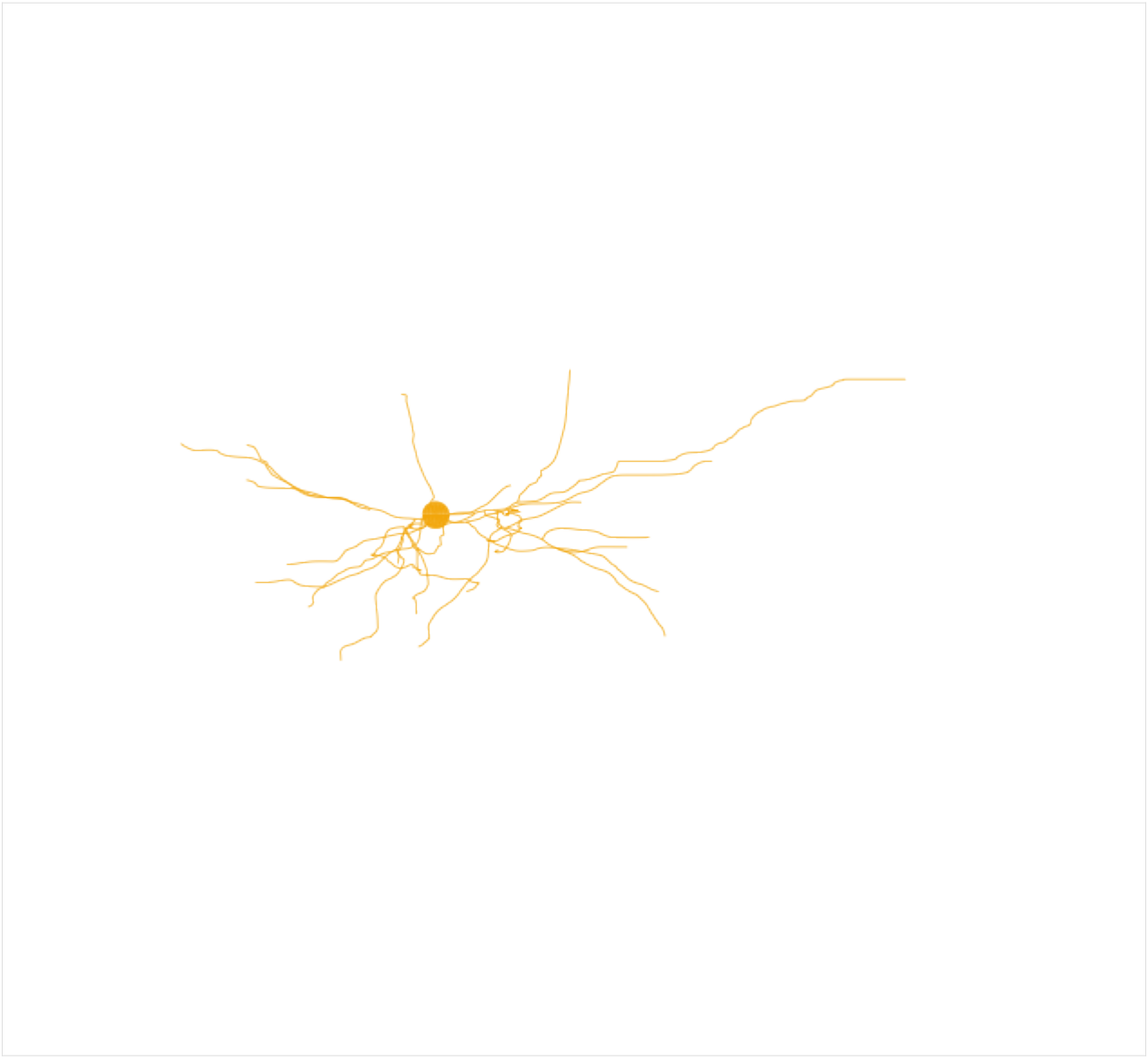
Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

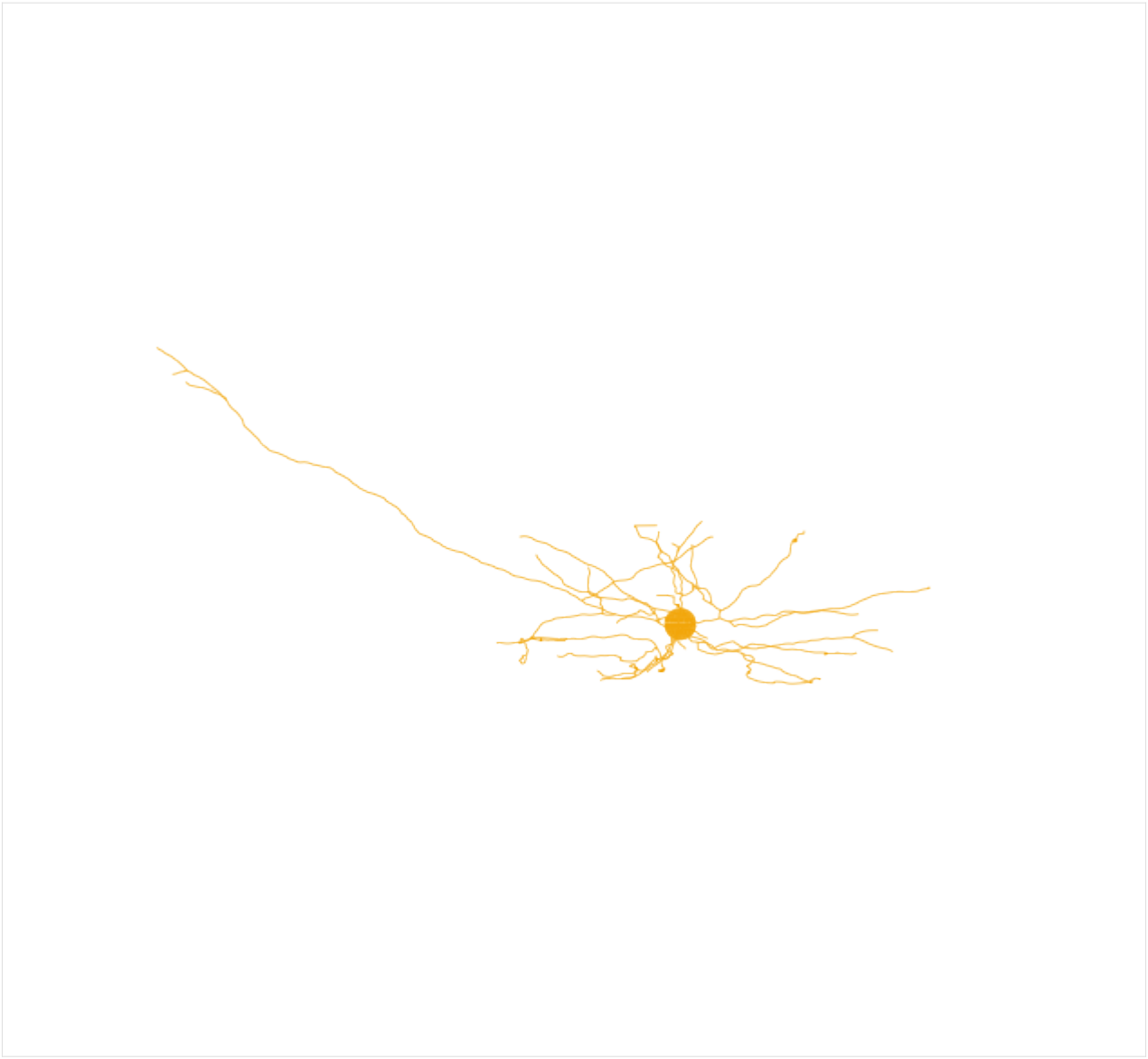
As expected, cells belonging to the same cluster have similar morphology. For example, let us visualize some of the cells in the brown cluster (cluster 8) using the Python package NAVis (which can be installed using `pip install navis`):

[6]:
```
import navis

cells_cluster_8 = [n for m, n in zip(clusters, cells) if m==8]
cluster_8 = [navis.read_swc("/home/jovyan/swc/" + n + ".swc") for n in cells_cluster_8]

cluster_8[1].plot2d()
cluster_8[2].plot2d()
cluster_8[3].plot2d()
```

[6]:
```
(<Figure size 720x720 with 1 Axes>, <Axes3DSubplot: >)
```







NAVis offers many other great functionalities, including interactive 3D visualizations of the neurons. More information about those functionalities can be found at the NAVis documentation.

We can also compute the medoid of the cluster, i. e. the most central neuron of the cluster (and therefore a good representative of the morphologies present in the cluster), and visualize it:

[7]:
```
medoid = navis.read_swc("/home/jovyan/swc/" +
                        cajal.utilities.identify_medoid(cells_cluster_8, gw_dist_dict) +
                        ".swc")
medoid.plot2d()
```

[7]:
```
(<Figure size 720x720 with 1 Axes>, <Axes3DSubplot: >)
```

The file `CAJAL/data/cell_types_specimen_details.csv` in the GitHub repository of CAJAL contains metadata for each of the neurons in this example, including the layer, Cre line, etc. Here we color the above UMAP representation by the cortical layer of each neuron:

[3]:
```
import pandas

metadata = pandas.read_csv("CAJAL/data/cell_types_specimen_details.csv")
metadata.index = [str(m) for m in metadata["specimen__id"]]
metadata = metadata.loc[cells]

plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in cells],
                       color = metadata["structure__layer"])
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

As shown in the visualization, different cortical layers seem to be associated with specific regions of the cell morphology space. We can quantify statistically the association using the Laplacian score:

[9]:
```
import cajal.laplacian_score
import numpy
import pandas
from scipy.spatial.distance import squareform

# Build indicator matrix
layers = numpy.unique(metadata["structure__layer"])
indicator = (numpy.array(metadata["structure__layer"])[:,None] == layers)*1

# Compute the Laplacian score
laplacian = pandas.DataFrame(cajal.laplacian_score.laplacian_scores(indicator,
                                    gw_dist,
                                    numpy.median(squareform(gw_dist)),
                                    permutations = 5000,
                                    covariates = None,
                                    return_random_laplacians = False)[0])
laplacian.index = layers

print(laplacian)
```

```
     feature_laplacians  laplacian_p_values  laplacian_q_values
1              0.976439              0.0002             0.00048
2/3            0.968968              0.0002             0.00048
4              0.970067              0.0002             0.00048
5              0.987542              0.0002             0.00048
6a             0.992997              0.0020             0.00240
6b             0.992043              0.0022             0.00220
```

We observe that indeed all cortical layers are significantly associated with distinct regions of the cell morphology space with false discovery rates (FDRs) < 0.05.

We could perform a similar analysis with other features. Alternatively, we could build a classifier to predict the value of each feature based on the position of the cells in the cell morphology space. For example, each neuron in the dataset is derived from a specific Cre driver line, which preferentially labels distinct neuronal types. Neurons from the same Cre driver line therefore tend to have similar morphologies, and a Laplacian score analysis would show that many Cre driver lines are significantly associated with distinct regions of the cell morphology space. As a result, it is possible to predict the Cre driver line of a neuron based on its morphological features.

To accomplish this, we train a nearest-neighbors classifier on the GW distance matrix and evaluate its accuracy using 7-fold cross-validation:

[10]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_score

cre_lines = numpy.array(metadata["line_name"])

clf = KNeighborsClassifier(metric="precomputed", n_neighbors=10, weights="distance")
cv = StratifiedKFold(n_splits=7, shuffle=True)
```

(continues on next page)

(continued from previous page)

```
accuracy = cross_val_score(clf, X=gw_dist, y=cre_lines,cv=cv)

numpy.mean(accuracy)
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/model_selection/_split.py:684:␣
↪UserWarning:

The least populated class in y has only 1 members, which is less than n_splits=7.
```

[10]:
```
0.2829963035442487
```

Similarly, we can compute the Matthews correlation coefficient of the classification, which appropriately weights the error arising from misclassifying elements of smaller classes:

[11]:
```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import matthews_corrcoef

cvp = cross_val_predict(clf, X=gw_dist, y=cre_lines, cv=cv)

print(matthews_corrcoef(cvp, cre_lines))
```

```
0.23180785506113458
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/model_selection/_split.py:684:␣
↪UserWarning:

The least populated class in y has only 1 members, which is less than n_splits=7.
```

CHAPTER

SEVEN

TUTORIAL 2: GENETIC DETERMINANTS OF NEURONAL MORPHOLOGY

We will illustrate the utility of the Laplacian score in identifying genes that contribute to the neuronal plasticity in the C. elegans. This example utilizes a dataset consisting of 799 3D neuronal reconstructions of the C.elegans DVB neuron across various mutant and control strains during days 1 to 5 of adulthood. The dataset can be downloaded from the following folder. In this tutorial we assume that the SWC files are located in the folder `CAJAL/data_worm/swc`. The DVB neuron is an excitatory GABAergic motor interneuron located in the dorso-rectal ganglion of the worm, and is known to undergo post-developmental neurite outgrowth in males. This outgrowth alters the neuron's morphology and synaptic connectivity, contributing to changes in the spicule protraction step of male mating behavior. More information about this dataset can be found at:

- Hart, M. P. & Hobert, O. Neurexin controls plasticity of a mature, sexually dimorphic neuron. Nature 553, 165-170, (2018).
- Govek, K. W. et al. CAJAL enables analysis and integration of single-cell morphological data using metric geometry. Nature Communications 14, 3672, (2023).

To begin our analysis, we calculate the Gromov-Wasserstein distance between each pair of cells. For the sake of time, here we just sample 50 points per cell. This computation typically requires 20-30 minutes to complete on a standard desktop computer. A larger number of sampled points would offer better results, but would also increase the computing time.

[2]:
```
import cajal.sample_swc
import cajal.swc
import cajal.run_gw

cajal.sample_swc.compute_icdm_all_geodesic(
    infolder="CAJAL/data_worm/swc/",
    out_csv="CAJAL/data_worm/c_elegans_icdm.csv",
    preprocess=cajal.swc.preprocessor_geo(
        structure_ids="keep_all_types"),
    n_sample=50,
    num_processes=8)  # num_processes can be set to the number of cores on your machine

cajal.run_gw.compute_gw_distance_matrix(
    "CAJAL/data_worm/c_elegans_icdm.csv",
    "CAJAL/data_worm/c_elegans_gw_dist.csv",
    num_processes=8)
```

```
100%|| 798/799 [00:31<00:00, 25.03it/s]
100%|| 318801/318801 [02:09<00:00, 2460.13it/s]
```

[2]:
```
(array([[0.        , 5.11936112, 3.68814622, ..., 4.50253393, 3.74849009,
         2.41605872],
        [5.11936112, 0.        , 4.16953034, ..., 5.1718854 , 3.86677755,
         3.06617002],
        [3.68814622, 4.16953034, 0.        , ..., 5.37682889, 3.85930797,
         2.66918667],
        ...,
        [4.50253393, 5.1718854 , 5.37682889, ..., 0.        , 3.52210097,
         4.01724968],
        [3.74849009, 3.86677755, 3.85930797, ..., 3.52210097, 0.        ,
         3.07758098],
        [2.41605872, 3.06617002, 2.66918667, ..., 4.01724968, 3.07758098,
         0.        ]]),
 None)
```

We can generate a UMAP plot that visualizes the cell morphology space, with each point colored according to the age of each worm in days. The metadata for each neuron in this example is provided in the file `CAJAL/data/c_elegans_features.csv`, which can be found in the GitHub repository of CAJAL. This metadata includes information such as the age of the worm in days and the genotype of each gene (0: wild-type; 1: mutant).

[1]:
```
import plotly.io as pio
pio.renderers.default = 'iframe'

import cajal.utilities
import umap
import pandas
import plotly.express

# Read GW distance matrix
cells, gw_dist_dict = cajal.utilities.read_gw_dists("CAJAL/data_worm/c_elegans_gw_dist.
↪csv", header=True)
gw_dist = cajal.utilities.dist_mat_of_dict(gw_dist_dict)

# Compute UMAP representation
reducer = umap.UMAP(metric="precomputed", random_state=1)
embedding = reducer.fit_transform(gw_dist)

# Download metadata
metadata = pandas.read_csv("CAJAL/data_worm/c_elegans_features.csv", index_col = "cell_
↪name")

# Visualize UMAP
plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in cells],
                       color = [str(m) for m in metadata["day"]])
```

```
2023-07-20 21:18:46.439529: I tensorflow/core/platform/cpu_feature_guard.cc:193] This␣
↪TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use␣
↪the following CPU instructions in performance-critical operations:  SSE4.1 SSE4.2 AVX␣
↪AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler␣
```

(continues on next page)

(continued from previous page)

```
↪flags.
/opt/conda/lib/python3.10/site-packages/umap/umap_.py:1780: UserWarning: using␣
↪precomputed metric; inverse_transform will be unavailable
  warn("using precomputed metric; inverse_transform will be unavailable")
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels␣
↪instead.
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Unsurprisingly, the age of the worm plays a significant role in shaping the morphology of its neurons. This is evident in the UMAP representation above, which reveals that neurons of different ages cluster in distinct regions of the UMAP. To quantify this association, we can use the Laplacian score:

[4]:
```
import cajal.laplacian_score
import numpy
from scipy.spatial.distance import squareform

laplacian = pandas.DataFrame(cajal.laplacian_score.laplacian_scores(numpy.array(metadata[
↪"day"]).reshape(799,1),
                                        gw_dist,
                                        numpy.median(squareform(gw_dist)),
                                        permutations = 5000,
                                        covariates = None,
                                        return_random_laplacians = False)[0])

print(laplacian)
```

```
   feature_laplacians  laplacian_p_values  laplacian_q_values
0            0.951357              0.0002              0.0002
```

A very small p value suggests a strong association between the age of the worm and the morphology of the DVB neuron.

Moving forward, our goal is to identify mutations that impact the morphology of the DVB neuron. To achieve this, we will rely on the Laplacian score once again. However, it is essential to consider the unequal representation of worms with a given genotype across different ages in the dataset. To address this issue, we will account for the uneven distribution of ages for each genotype. As an example, we will investigate the impact of deleterious mutations in the unc-25 gene. Let us first look at their distribution in the cell morphology space:

[2]:
```
plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in cells],
                       color = [str(m) for m in metadata["unc-25"]])
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

The UMAP representation reveals that cells with a deleterious mutation in unc-25 exhibit similar morphology, a finding supported by the small p-value of the Laplacian score of unc-25 in the cell morphology space:

[6]:
```
laplacian = pandas.DataFrame(cajal.laplacian_score.laplacian_scores(numpy.array(metadata[
↪"unc-25"]).reshape(799,1),
                                        gw_dist,
                                        numpy.median(squareform(gw_dist)),
                                        permutations = 5000,
                                        covariates = None,
                                        return_random_laplacians = False)[0])

print(laplacian)
```

```
   feature_laplacians  laplacian_p_values  laplacian_q_values
0            0.995027              0.0018              0.0018
```

However, most of the samples with a mutation in unc-25 were obrained from worms with ages 1 or 3 days:

[7]:
```
metadata.loc[metadata["unc-25"]==1,"day"].value_counts()
```

[7]:
```
1    18
3     6
Name: day, dtype: int64
```

This leads to the question: is the comparable morphology of neurons with a deleterious mutation in unc-25 attributed to the mutation itself or the similar age of the worms? To address this issue, we can employ the Laplacian score but treating the age of the worm as a covariate:

[8]:
```
laplacian = pandas.DataFrame(cajal.laplacian_score.laplacian_scores(numpy.array(metadata.
↪iloc[:,0:11]),
                                        gw_dist,
                                        numpy.median(squareform(gw_dist)),
                                        permutations = 5000,
                                        covariates = numpy.array(metadata["day"]),
                                        return_random_laplacians = False)[0])
laplacian.index = metadata.columns.values.tolist()[0:11]

print(laplacian)
```

```
        feature_laplacians  laplacian_p_values  laplacian_q_values    beta_0  \
nrx-1             0.996896              0.005799            0.012757  1.000429
mir-1             1.000011              0.124175            0.151770  1.009661
unc-49            0.996668              0.007399            0.013564  1.004363
nlg-1             0.994696              0.001000            0.003666  0.960108
unc-25            0.995027              0.002799            0.007698  0.898466
unc-97            0.961754              0.000200            0.002200  0.993265
lim-6             1.000981              0.304539            0.334993  1.036849
lat-2             0.994027              0.000800            0.004399  1.008793
ptp-3             0.999590              0.080184            0.110253  0.985277
sup-17            0.997795              0.014597            0.022938  0.981026
pkd-2             1.001690              0.547690            0.547690  1.000995

          beta_1  beta_1_p_value  regression_coefficients_fstat_p_values  \
nrx-1   0.001071    4.695718e-01                             9.391437e-01
mir-1  -0.008178    7.174294e-01                             5.651412e-01
unc-49 -0.002882    5.790428e-01                             8.419143e-01
nlg-1   0.041295    2.050789e-03                             4.101577e-03
unc-25  0.102830    1.535813e-12                             3.071627e-12
```

(continues on next page)

(continued from previous page)

```
unc-97  0.008190    2.834938e-01                              5.669877e-01
lim-6  -0.035374    9.925361e-01                              1.492786e-02
lat-2  -0.007340    6.945934e-01                              6.108133e-01
ptp-3   0.016133    1.312833e-01                              2.625666e-01
sup-17  0.020424    7.884423e-02                              1.576885e-01
pkd-2   0.000425    4.880074e-01                              9.760148e-01

        laplacian_p_values_post_regression  laplacian_q_values_post_regression
nrx-1                             0.005999                            0.021996
mir-1                             0.084383                            0.116027
unc-49                            0.007199                            0.015837
nlg-1                             0.006999                            0.019246
unc-25                            0.146771                            0.179386
unc-97                            0.000200                            0.002200
lim-6                             0.056789                            0.089239
lat-2                             0.000600                            0.003299
ptp-3                             0.181964                            0.200160
sup-17                            0.041392                            0.075885
pkd-2                             0.554289                            0.554289
```

Upon examining the table, we note that the q-value of unc-25 shifts from 0.008 to 0.18 after adjusting for the covariate effect. Consistent with this, the F-statistic suggests a considerable impact of the covariate on the Laplacian score of unc-25, as evidenced by the low p-value of the F-statistic.

CHAPTER

EIGHT

TUTORIAL 3: COMPUTING MORPHOLOGICAL DISTANCES IN LARGE DATASETS

The Gromov-Wasserstein distance between two cells with 100 points takes about 9ms to compute on a standard desktop computer. The number of pairs grows quadratically with the number of cells, and so the total runtime can become large in datasets with several thousands of cells.

For large datasets we provide two tools to reduce the necessary computation, as well as a hybrid of these.

In *[1]*, the author establishes several lower bounds for the Gromov-Wasserstein (GW) distance. CAJAL implements one of the fastest bounds, the second lower bound (SLB) *[2]*. For many downstream analyses, such as clustering and dimensional reduction, it is not crucial to know the exact values between disparate cells, and it is enough to know the precise Gromov-Wasserstein distance only for cells that are close to each other in the morphology space. Since the SLB is a fast lower bound to the GW distance, it can be used to quickly identify pairs of cells that are located far apart in the morphology space so that their precise GW distance does not need to be precisely computed.

Let us illustrate how the computation of the SLB using CAJAL works on the same neuronal dataset as in Tutorial 1. We start with the file of intracellular distances computed in Tutorial 1:

[1]:
```
from cajal.qgw import slb_parallel

slb_parallel(
    "/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv",
    num_processes =8,                                  # num_processes can be set to the␣
↪number of cores on your machine
    out_csv = "/home/jovyan/slb_dists.csv")
```

```
100%|| 129286/129286 [00:00<00:00, 325877.15it/s]
```

The SLB is somewhat crude as an approximation of Gromov-Wasserstein, as it is only a lower bound, but it only takes a ~6 seconds to compute for this dataset.

To get a better sense of the SLB accuracy, let us compare the SLB with the GW distance computed for each pair of cells in Tutorial 1:

[1]:
```
import plotly.io as pio
pio.renderers.default = 'iframe'

from cajal.utilities import read_gw_dists, dist_mat_of_dict
from cajal.run_gw import cell_iterator_csv
import plotly.express

names, _ = zip(*cell_iterator_csv("/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv"))
names=list(names)
```

(continues on next page)

(continued from previous page)

```
_, gw_dist_dict = read_gw_dists("/home/jovyan/swc_bdad_100pts_euclidean_GW_dmat.csv",␣
↪True)
gw100_dist_table = dist_mat_of_dict(gw_dist_dict, names, as_squareform=False)

_, slb_dist_dict =  read_gw_dists("/home/jovyan/slb_dists.csv", True)
slb_dist_table = dist_mat_of_dict(slb_dist_dict, names, as_squareform=False)

fig = plotly.express.scatter(x=slb_dist_table,
                        y=gw100_dist_table,
                        template="simple_white",
                        labels={
                          "x" : "SLB",
                          "y" : "GW distance"})
fig.update_traces(marker={'size': 1})
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

The mean and stadard deviation of the relative difference between the GW distance and the SLB in this example are:

[3]:
```
import numpy

print(numpy.mean((gw100_dist_table-slb_dist_table)/gw100_dist_table))
print(numpy.std((gw100_dist_table-slb_dist_table)/gw100_dist_table))
```

```
0.22281620822119072
0.19661884027625978
```

Although the SLB is only a lower bound for the Gromov-Wasserstein distance, using it alone is already fairly accurate as a classifier. Let us repeat the same analysis presented in Tutorial 1 for predicting the molecular type of neurons but using the SLB instead of the GW distance:

[4]:
```
import pandas
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_predict
from sklearn.metrics import matthews_corrcoef
from scipy.spatial.distance import squareform

metadata = pandas.read_csv("CAJAL/data/cell_types_specimen_details.csv")
metadata.index = [str(m) for m in metadata["specimen__id"]]
metadata = metadata.loc[names]

cre_lines = numpy.array(metadata["line_name"])

clf = KNeighborsClassifier(metric="precomputed", n_neighbors=10, weights="distance")
cv = StratifiedKFold(n_splits=7, shuffle=True)

cvp = cross_val_predict(clf, X=squareform(slb_dist_table), y=cre_lines, cv=cv)

print(matthews_corrcoef(cvp, cre_lines))
```

```
0.21540117286365484
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/model_selection/_split.py:684:␣
↪UserWarning:

The least populated class in y has only 1 members, which is less than n_splits=7.
```

The Matthews correlation coefficient is only slightly lower than the one we derived in the Tutorial 1 using the GW distance.

If the SLB distances between all cells are known, one may envision a simple algorithm*[3]* to compute the k nearest neighbors of any given cell under the Gromov-Wasserstein metric. If k is chosen sufficiently large, this is enough to understand the local structure of the morphology space and cluster it.

The second tool we provide is an implementation of the quantized Gromov-Wasserstein distance proposed by Chowdhury, Miller and Needham*[4]*. Given cells X and Y, the quantized Gromov-Wasserstein distance is given as follows: 1. Partition the points of X and Y into n clusters. Let X^n, Y^n be the set of medoids of these clusters; X^n can be thought of as the best possible approximation to X in the GW morphology space by a set with at most n points. 2. Computing the optimal Gromov-Wasserstein transport plan between the subspaces X^n and Y^n formed by the medoids of each cluster. 3. Extend this to a global transport plan between X and Y by pairing points within paired clusters by their distance from the medoid, and compute the distortion associated to this transport plan. This approximation gives an acceptable tradeoff between precision and computation time.

Below, for each cell we cluster the 100 sampled points into 25 clusters.

[5]:
```
from cajal.qgw import quantized_gw_parallel

quantized_gw_parallel(
    intracell_csv_loc="/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv",
    num_processes=8,
    num_clusters=25,
    out_csv="/home/jovyan/quantized_gw.csv")
```

```
100%|| 129286/129286 [00:32<00:00, 4025.92it/s]
```

This is about ten times faster to compute than the GW distance and provides a better approximation to the GW distance than the SLB.

[2]:
```
_, qgw_dist_dict = read_gw_dists("/home/jovyan/quantized_gw.csv", header=True)
qgw_dmat = dist_mat_of_dict(qgw_dist_dict, cell_names=names, as_squareform=False)

fig = plotly.express.scatter(x=qgw_dmat,
                         y=gw100_dist_table,
                         template="simple_white",
                         labels={
                           "x" : "Quantized GW distance",
                           "y" : "GW distance"})
fig.update_traces(marker={'size': 1})
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

We can see that the mean and stadard deviation of the relative difference between the GW distance and the quantized GW distance are smaller than those for the SLB:

[7]:
```
print(numpy.mean((gw100_dist_table-qgw_dmat)/gw100_dist_table))
print(numpy.std((gw100_dist_table-qgw_dmat)/gw100_dist_table))
```

```
-0.10198643981692142
0.13250463499313633
```

Finally, CAJAL implements an approach that combine these two tools (quantized Gromov-Wasserstein and SLB) in an integrated analysis method which allows the user to reduce computation time in three simultaneous ways:

1. by computing only the k nearest neighbors of each cell and estimating the rest roughly using SLB
2. by accepting a small fraction of errors in the reported nearest neighbors list of each cell (i.e., 98% of the nearest neighbors are correct)
3. by using the quantized GW distance as a proxy for the true GW distance

[8]:
```
from cajal.qgw import combined_slb_quantized_gw
combined_slb_quantized_gw(
    "/home/jovyan/swc_bdad_100pts_euclidean_icdm.csv",
    "/home/jovyan/swc_bdad_100pts_euclidean_gw_estimator.csv",
    num_processes=8,
    num_clusters=25,
    accuracy = 0.95,
    nearest_neighbors = 30)
```

The above command does the following:

1. Compute the pairwise SLB between any two cells in the given list of intracell distance matrices.
2. For each cell X, identify the 30 estimated nearest neighbors $X_1, \ldots, X_{30}$ based on the SLB distance matrix, and compute the QGW distance $QGW_{25}(X, X_k)$ for all 30 pairs.
3. For each remaining cell pair X, Y, we estimate the probability that $QGW_{25}(X, Y)$ is small enough to "injure" the existing purported list of nearest neighbors. We sort the cell pairs in descending order by this probability and compute the QGW_{25} distance between pairs in this list until the expected number of "injuries" remaining is less than 5% of the nearest neighbor table, so that of the reported 30 nearest nearest neighbors to each point, 28.5 are expected to be correct.
4. For all remaining cells we estimate the correct distance based on the SLB.

As expected, a UMAP representation based on the fast QGW/SLB combined estimator is very close to the UMAP that we computed in Tutorial 1 for the ame data using the full GW distance:

[3]:
```
import umap
import pandas

_, slbqgw_dist_dict = read_gw_dists("/home/jovyan/swc_bdad_100pts_euclidean_gw_estimator.
↪csv", header=False)
slbqgw_sq_dist = dist_mat_of_dict(slbqgw_dist_dict, cell_names=names)

# Load metadata
metadata = pandas.read_csv("CAJAL/data/cell_types_specimen_details.csv")
metadata.index = [str(m) for m in metadata["specimen__id"]]
metadata = metadata.loc[names]

# Compute UMAP representation
reducer = umap.UMAP(metric="precomputed", random_state=1)
```

(continues on next page)

(continued from previous page)

```
embedding = reducer.fit_transform(slbqgw_sq_dist)

# Visualize UMAP colored by cortical layer
plotly.express.scatter(x=embedding[:,0],
                       y=embedding[:,1],
                       template="simple_white",
                       hover_name=[m + ".swc" for m in names],
                       color = metadata["structure__layer"])
```

```
2023-07-20 21:05:35.256736: I tensorflow/core/platform/cpu_feature_guard.cc:193] This␣
↪TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use␣
↪the following CPU instructions in performance-critical operations:  SSE4.1 SSE4.2 AVX␣
↪AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler␣
↪flags.
/opt/conda/lib/python3.10/site-packages/umap/umap_.py:1780: UserWarning:

using precomputed metric; inverse_transform will be unavailable

OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels␣
↪instead.
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

8.1 Notes

1.[^](#cite_ref-1) Mémoli, F. P. Gromov–Wasserstein Distances and the Metric Approach to Object Matching. Found Comput Math (2011) 11, 417–487.

2.[^](#cite_ref-2) Specifically we have implemented the expression which appears on the right hand side of the first inequality of Corollary 6.2 on page 462, for p = 2. This expression is not directly named in the paper. We do not compute the quantity Mémoli calls SLB, as it is too computationally expensive for our purposes.

3.[^](#cite_ref-3) A simple algorithm for computing the nearest neighbors of a cell in the Gromov-Wasserstein morphology space if the SLB distance is known is as follows: 1. First, sort all other cells by their SLB2 distance from c_0, as $c_1, c_2, c_3, \ldots$. 2. Next, compute the Gromov-Wasserstein distance $GW(c_0, c_j)$, as $j = 1, 2, 3, \ldots$. Write e_j^k for the k-th element of the set $GW(c_0, c_1), GW(c_0, c_2), \ldots, GW(c_0, c_j)$ when these are ordered from least to greatest. (e_j^k is only defined when $k \leq j$). Continue computing $GW(c_0, c_j)$ until j reaches a value ℓ such that for all $i > \ell$, $SLB(c_0, c_i) > e_\ell^k$. Because SLB is a lower bound, at this point, the k nearest neighbors of c_0 are contained in the set $\{c_1, \ldots, c_\ell\}$.

4.[^](#cite_ref-4) Chowdhury, S., Miller, D., Needham, T. (2021). Quantized Gromov-Wasserstein. In: Oliver, N., Pérez-Cruz, F., Kramer, S., Read, J., Lozano, J.A. (eds) Machine Learning and Knowledge Discovery in Databases. Research Track. ECML PKDD 2021. Lecture Notes in Computer Science(), vol 12977. Springer, Cham. https://doi.org/10.1007/978-3-030-86523-8_49

CHAPTER

NINE

PROCESSING SWC FILES

CAJAL supports neuronal tracing data in the SWC spec as specified here: http://www.neuronland.org/NLMorphologyConverter/MorphologyFormats/SWC/Spec.html

The sample_swc.py file contains functions to help the user sample points from an *.swc file.

class NeuronNode(*sample_number: int*, *structure_id: int*, *coord_triple: tuple[float, float, float]*, *radius: float*, *parent_sample_number: int*)

A NeuronNode represents the contents of a single line in an *.swc file.

Parameters

- **sample_number** (*int*) –
- **structure_id** (*int*) –
- **coord_triple** (*tuple[float, float, float]*) –
- **radius** (*float*) –
- **parent_sample_number** (*int*) –

class NeuronTree(*root:* NeuronNode, *child_subgraphs: list[*cajal.swc.NeuronTree*]*)

A NeuronTree represents one connected component of the graph coded in an *.swc file.

Parameters

- **root** (NeuronNode) –
- **child_subgraphs** (*list[*cajal.swc.NeuronTree*]*) –

class cajal.swc.**SWCForest**

A swc.SWCForest is a list of swc.NeuronTree's. It is intended to be used to represent a list of all connected components from an SWC file. An SWCForest represents all contents of one SWC file.

read_swc(*file_path: str*) → tuple[*swc.SWCForest*, dict[int, *cajal.swc.NeuronTree*]]

Construct the graph (forest) associated to an SWC file. The forest is sorted by the number of nodes of the components

An exception is raised if any line has fewer than seven whitespace separated strings.

Parameters
file_path (*str*) – A path to an *.swc file.

Returns
(forest, lookup_table), where lookup_table maps sample numbers for nodes to their positions in the forest.

Return type
tuple[*swc.SWCForest*, dict[int, *cajal.swc.NeuronTree*]]

One can alternately represent an SWC file in a simple list format, rather than using a nested class structure. The class structure may be more elegant, but we have encountered a number of SWCs so far where the depth of the graph associated to an SWC exceeds the default stack limit of Python, and so recursive algorithms on graphs are prone to stack overflow errors. A list is more amenable to iterative algorithms. In particular, if the user wants to serialize the data of an SWC graph (for example, to pass it between two processes or threads) they should recast it as a list so that the Python serialization function does not cause a stack overflow.

linearize(*forest:* swc.SWCForest) → list[*cajal.swc.NeuronNode*]

Linearize the SWCForest into a list of NeuronNodes where the sample number of each node is just its index in the list plus 1.

Parameters
forest (swc.SWCForest) – An SWCForest to be linearized.

Returns
A list *linear* of NeuronNodes. The list *linear* represents a directed graph which is isomorphic to *forest*; under this graph isomorphism, the xyz coordinates, radius, and structure identifier will be preserved, but the fields *parent_sample_number* and *sample_number* will not be. Instead, we will have *linear[k].sample_number*==*k+1* for each index *k*. (This index shift is clearly error-prone with Python's zero-indexing of lists, but it seems common in SWC files.)

Return type
list[*cajal.swc.NeuronNode*]

In addition to having "standardized" indices, this is a breadth-first linearization algorithm. It is guaranteed that:

1. The graph is topologically sorted in that parent nodes come before child nodes.
2. Each component is contained in a contiguous region of the list, whose first element is of course the root by (1.)
3. Within each component, the nodes are organized by level, so that the first element is the root, indices 2..n are the nodes at depth 1, indices n+1 .. m are the nodes at depth 2, and so on.

forest_from_linear(*ell: list[*cajal.swc.NeuronNode*]*) → *swc.SWCForest*

Convert a list of `swc.NeuronNode`'s to a graph.

Parameters
ell (*list[*`cajal.swc.NeuronNode`*]*) – A list of `swc.NeuronNode`'s where ell[i].sample_number == i+1 for all i. It is assumed that *ell* is topologically sorted, i.e., that parents are listed before their children, and that roots are marked by -1.

Returns
An `swc.SWCForest` containing the contents of the graph.

Return type
swc.SWCForest

write_swc(*outfile: str*, *forest:* swc.SWCForest) → None

Write *forest* to *outfile*. Overwrite whatever is in *outfile*.

This function does not respect the sample numbers and parent sample numbers in *forest*. They will be renumbered so that the indices are contiguous and start at 1.

Parameters

- **outfile** (*str*) – An absolute path to the output file.
- **forest** (swc.SWCForest) –

Return type
None

If the user is batch-processing all *.swc files in a given directory, it is appropriate to include a filtering function so that the user does not accidentally crash the program by trying to read a non-SWC file into memory. Such extraneous files could include backup text files automatically generated by a text editor or by the operating system, hidden files, log files, or lists of cell indices. Therefore the user has the option to supply a "name validation" function which returns either True or False for each file name in the directory, only the filenames which pass the name validation test will be sampled from. The default name validation function is this one:

default_name_validate(*filename: str*) → bool

If the file name starts with a period '.', the standard hidden-file marker on Linux, return False. Otherwise, return True if and only if the file ends in ".swc" (case-insensitive).

Parameters
filename (*str*) –

Return type
bool

The user should be warned that passing information between distinct Python processes is costly, and the following function is not recommended if the user wants to employ multiprocessing, as any child process which takes cells from this iterator as input will incur high overhead by serializing and copying the data between processes. For multiprocessing/parallelization it is better to give each process its own list of file names to operate on, and let them read the files independently.

cell_iterator(*infolder: str, name_validate: ~typing.Callable[[str], bool] = <function default_name_validate>*) → Iterator[tuple[str, *swc.SWCForest*]]

Construct an iterator over all SWCs in a directory (all files ending in *.swc or *.SWC).

Parameters

- **infolder** (*str*) – A path to a folder containing SWC files.
- **name_validate** (*Callable[[str], bool]*) –

Returns
An iterator over pairs (name, forest), where "name" is the file root (everything before the period in the file name) and "forest" is the forest contained in the SWC file.

Return type
Iterator[tuple[str, *swc.SWCForest*]]

The following function is very useful for sampling from fragments of a neuron. .. autofunction:: cajal.swc.filter_forest

keep_only_eu(*structure_ids: Container[int]*) → Callable[[*swc.SWCForest*], *swc.SWCForest*]

Given *structure_ids*, a (list, set, tuple, etc.) of integers, return a filtering function which accepts an `swc.SWCForest` and returns the subforest containing only the node types in *structure_ids*.

Example: *keep_only([1,3,4])(forest)* is the subforest of *forest* containing only the soma, the basal dendrites and the apical dendrites, but not the axon.

The intended use is to generate a preprocessing function for *swc.read_preprocess_save*, *swc.batch_filter_and_preprocess*, or *sample_swc.compute_and_save_intracell_all_euclidean*, see the documentation for those functions for more information.

Parameters
structure_ids (*Container[int]*) – A container of integers representing types of neuron nodes.

Returns
A filtering function taking as an argument an SWCForest *forest* and returning the subforest of *forest* containing only the node types in *structure_ids*.

Return type
Callable[[*swc.SWCForest*], *swc.SWCForest*]

preprocessor_geo(*structure_ids: Union[Container[int], Literal['keep_all_types']]*) → Callable[[*swc.SWCForest*], *NeuronTree*]

This preprocessor strips the tree down to only the components listed in *structure_ids* and also trims the tree down to a single connected component. This is similar to `swc.keep_only_eu()` and the user should consult the documentation for that function. Observe that the type signature is also different. The callable returned by this function is suitable as a preprocessing function for `sample_swc.read_preprocess_compute_geodesic()` or `sample_swc.compute_and_save_intracell_all_geodesic()`.

Parameters
structure_ids (*Union[Container[int], Literal['keep_all_types']]*) –

Return type
Callable[[*swc.SWCForest*], NeuronTree]

preprocessor_eu(*structure_ids: Union[Container[int], Literal['keep_all_types']]*, *soma_component_only: bool*) → Callable[[*swc.SWCForest*], Union[Err[str], *swc.SWCForest*]]

Parameters

- **structure_ids** (*Union[Container[int], Literal['keep_all_types']]*) – Either a collection of integers corresponding to structure ids in the SWC spec, or the literal string 'keep_all_types'.
- **soma_component_only** (*bool*) – Indicate whether to sample from the whole SWC file, or only from the connected component containing the soma. Whether this flag is appropriate depends on the technology used to construct the SWC files. Some technologies generate SWC files in which there are many unrelated connected components which are "noise" contributed by other overlapping neurons. In other technologies, all components are significant and the authors of the SWC file were simply unable to determine exactly where the branch should be connected to the main tree. In order to get sensible results from the data, the user should visually inspect neurons with multiple connected components using a tool such as Vaa3D https://github.com/Vaa3D/release/releases/tag/v1.1.2 to determine whether the extra components should be regarded as signal or noise.

Returns
A preprocessing function which accepts as argument an SWCForest *forest* and returns a filtered forest containing only the nodes listed in *structure_ids*. If *soma_component_only* is True, only nodes from the component containing the soma will be returned; otherwise nodes will be drawn from across the whole forest. If *soma_component_only* is True and there is not a unique connected component whose root is a soma node, the function will return an error.

Return type
Callable[[*swc.SWCForest*], *Union*[*Err*[str], *swc.SWCForest*]]

total_length(*tree:* NeuronTree) → float

Return the sum of lengths of all edges in the graph.

Parameters
tree (NeuronTree) –

Return type
float

weighted_depth(*tree:* NeuronTree) → float

Return the weighted depth/ weighted height of the tree, i.e., the maximal geodesic distance from the root to any other point.

Parameters
tree (NeuronTree) –

Return type
float

discrete_depth(*tree:* NeuronTree) → int

Returns
The height of the tree in the unweighted or discrete sense, i.e. the longest path from the root to any leaf measured in the number of edges.

Parameters
tree (NeuronTree) –

Return type
int

node_type_counts_tree(*tree:* NeuronTree) → dict[int, int]

Returns
A dictionary whose keys are all structure_id's in *tree* and whose values are the multiplicities with which that node type occurs.

Parameters
tree (NeuronTree) –

Return type
dict[int, int]

node_type_counts_forest(*forest:* swc.SWCForest) → dict[int, int]

Returns
a dictionary whose keys are all structure_id's in *forest* and whose values are the multiplicities with which that node type occurs.

Parameters
forest (swc.SWCForest) –

Return type
dict[int, int]

num_nodes(*tree:* NeuronTree) → int

Returns
The number of nodes in *tree*.

Parameters
tree (NeuronTree) –

Return type
int

read_preprocess_save(*infile_name: str*, *outfile_name: str*, *preprocess: Callable[[*swc.SWCForest*], Union[Err[T],* swc.SWCForest*,* NeuronTree*]]*) → Union[Err[T], Literal['success']]

Read the *.swc file *file_name* from disk as an *SWCForest*. Apply the function *preprocess* to the forest. If preprocessing returns an error,return that error. Otherwise, write the preprocessed swc to outfile and return the string "success".

This function exists mostly for convenience, as it can be called in parallel on several files at once without requiring a large amount of data to be communicated between processes.

Parameters

- **infile_name** (*str*) –
- **outfile_name** (*str*) –
- **preprocess** (*Callable[[swc.SWCForest], Union[Err[T], swc.SWCForest, NeuronTree]]*) –

Return type
Union[*Err*[*T*], *Literal*['success']]

get_filenames(*infolder: str, name_validate: ~typing.Callable[[str], bool] = <function default_name_validate>*) → tuple[list[str], list[str]]

Get a list of all files in infolder. Filter the list by name_validate. :return: a pair of lists (cell_names, file_paths), where *file_paths* are the paths to cells we want to sample from, and *cell_names[i]* is the substring of *file_paths[i]* containing only the file name, minus the extension; i.e., if file_paths[i] is "/home/jovyan/files/abc.swc" then cell_names[i] is "abc".

See `swc.default_name_validate()` for an example of a name validation function.

Parameters

- **infolder** (*str*) –
- **name_validate** (*Callable[[str], bool]*) –

Return type
tuple[list[str], list[str]]

batch_filter_and_preprocess(*infolder: str, outfolder: str, preprocess: ~typing.Callable[[swc.SWCForest], ~typing.Union[~cajal.utilities.Err[~cajal.utilities.T], swc.SWCForest, ~cajal.swc.NeuronTree]], parallel_processes: int, err_log: ~typing.Optional[str], suffix: ~typing.Optional[str] = None, name_validate: ~typing.Callable[[str], bool] = <function default_name_validate>*) → None

Get the set of files in infolder. Filter down to the filenames which pass the test *name_validate*, which is responsible for filtering out any non-swc files.For the files in this filtered list, read them into memory as `swc.SWCForest`'s. Apply the function *preprocess* to each forest. *preprocess* may return an error (essentially just a message contained in an error wrapper) or a modified/transformed SWCForest, i.e., certain nodes have been filtered out, or certain components of the graph deleted. If *preprocess* returns an error, write the error to the given log file *err_log* together with the name of the cell that caused the error. Otherwise, if *preprocess* returns an SWCForest, write this SWCForest into the folder *outfolder* with filename == cellname + suffix + '.swc'.

Parameters

- **infolder** (*str*) – Folder containing SWC files to process.
- **outfolder** (*str*) – Folder where the results of the filtering will be written.
- **err_log** (*Optional[str]*) – A file name for a (currently nonexistent) *.csv file. This file will be written to with a list of all the cells which were rejected by *preprocess* together with an explanation of why these cells could not be processed.
- **preprocess** (*Callable[[swc.SWCForest], Union[Err[T], swc.SWCForest, NeuronTree]]*) – A function to filter out bad SWC forests or transform them into a more manageable form.
- **parallel_processes** (*int*) – Run this many Python processes in parallel.
- **suffix** (*Optional[str]*) – If a file in infolder has the name "abc.swc" then the corresponding file written to outfolder will have the name "abc" + suffix + ".swc".
- **name_validate** (*Callable[[str], bool]*) – A function which identifies the files in *infolder* which are *.swc files. The default argument, `swc.default_name_validate()`,

checks to see whether the filename has file extension ".swc", case insensitive, and discards files starting with '.', the marker for hidden files on Linux. The user may need to write their own function to ensure that various kinds of backup /autosave files and metadata files are not read into memory.

Return type
None

For computing geodesic distances, it is more convenient to have a data structure with the weights precomputed and attached to the edges, so we introduce an alternate representation for a neuron where coordinates are forgotten and only the weighted tree structure remains. These objects can be smaller than the original NeuronTrees.

class WeightedTreeRoot(*subtrees: 'list[WeightedTreeChild]'*)

Parameters
subtrees (*list[cajal.weighted_tree.WeightedTreeChild]*) –

class WeightedTreeChild(*subtrees: 'list[WeightedTreeChild]'*, *depth: 'int'*, *unique_id: 'int'*, *parent: 'WeightedTree'*, *dist: 'float'*)

Parameters

- **subtrees** (*list[cajal.weighted_tree.WeightedTreeChild]*) –
- **depth** (*int*) –
- **unique_id** (*int*) –
- **parent** (*sample_swc.WeightedTree*) –
- **dist** (*float*) –

A `cajal.weighted_tree.WeightedTree` is either a *cajal.weighted_tree.WeightedTreeRoot* or a *cajal.weighted_tree.WeightedTreeChild*.

WeightedTree_of(*tree:* NeuronTree) → *WeightedTreeRoot*

Convert a NeuronTree to a WeightedTree. A node in a WeightedTree does not contain a coordinate triple, a radius, a structure_id, or a parent sample number.

Instead, it contains a direct pointer to its parent, a list of its children, and (if it is a child node) the weight of the edge between the child and its parent.

In forming the WeightedTree, any node with both a parent and exactly one child is eliminated, and the parent and the child are joined directly by a single edge whose weight is the sum of the two original edge weights. This reduces the number of nodes without affecting the geodesic distances between points in the graph.

Parameters
tree (NeuronTree) – A NeuronTree to be converted into a WeightedTree.

Returns
The WeightedTree corresponding to the original NeuronTree.

Return type
WeightedTreeRoot

CHAPTER

TEN

SAMPLING FROM SWC FILES

get_sample_pts_euclidean(*forest: list[*cajal.swc.NeuronTree*]*, *step_size: float*) → list[numpy.ndarray[Any, numpy.dtype[numpy.float64]]]

Sample points uniformly throughout the forest, starting at the roots, at the given step size.

Returns
a list of (x,y,z) coordinate triples, represented as numpy floating point arrays of shape (3,). The list length depends (inversely) on the value of *step_size*.

Parameters

- **forest** (*list[cajal.swc.NeuronTree]*) –
- **step_size** (*float*) –

Return type
list[numpy.ndarray[*Any*, numpy.dtype[numpy.float64]]]

icdm_euclidean(*forest: list[*cajal.swc.NeuronTree*]*, *num_samples: int*) → ndarray[Any, dtype[float64]]

Compute the (Euclidean) intracell distance matrix for the forest, with n sample points. :param forest: The cell to be sampled. :param num_samples: How many points to be sampled. :return: A condensed (vectorform) matrix of length n* (n-1)/2.

Parameters

- **forest** (*list[cajal.swc.NeuronTree]*) –
- **num_samples** (*int*) –

Return type
ndarray[*Any*, *dtype*[*float64*]]

geodesic_distance(*wt1: Union[*WeightedTreeRoot, WeightedTreeChild*]*, *h1: float*, *wt2: Union[*WeightedTreeRoot, WeightedTreeChild*]*, *h2: float*) → float

Let p1 be a point in a weighted tree which lies at height h1 above wt1. Let p2 be a point in a weighted tree which lies at height h2 above wt2. Return the geodesic distance between p1 and p2.

Parameters

- **wt1** (*Union[WeightedTreeRoot, WeightedTreeChild]*) – A node in a weighted tree.
- **h1** (*float*) – Represents a point *p1* which lies *h1* above *wt1* in the tree, along the line segment connecting *wt1* to its parent. *h1* is assumed to be less than the distance between *wt1* and *wt1.parent*; or if *wt1* is a root node, *h1* is assumed to be zero.
- **wt2** (*Union[WeightedTreeRoot, WeightedTreeChild]*) – A node in a weighted tree.
- **h2** (*float*) – Represents a point *p2* which lies *h2* above *wt2* in the tree, along the line segment connecting *wt2* to its parent. Similar assumptions as for *h1*.

Return type
float

get_sample_pts_geodesic(*tree:* NeuronTree, *num_sample_pts: int*) → list[tuple[Union[*cajal.weighted_tree.WeightedTreeRoot*, *cajal.weighted_tree.WeightedTreeChild*], float]]

Sample points uniformly throughout the body of *tree*, starting at the root, returning a list of length *num_sample_pts*.

"Sample points uniformly" means that there is some scalar *step_size* such that a point *p* on a line segment of *tree* will be in the return list iff its geodesic distance from the origin is an integer multiple of *step_size*.

Returns
a list of pairs (wt, h), where *wt* is a node of *tree*, and *h* is a floating point real number representing a point *p* which lies a distance of *h* above *wt* on the line segment between *wt* and its parent. If *wt* is a child node, *h* is guaranteed to be less than the distance between *wt* and its parent. If *wt* is a root, *h* is guaranteed to be zero.

Parameters
- **tree** (NeuronTree) –
- **num_sample_pts** (*int*) –

Return type
list[tuple[*Union*[*cajal.weighted_tree.WeightedTreeRoot*, *cajal.weighted_tree.WeightedTreeChild*], float]]

icdm_geodesic(*tree:* NeuronTree, *num_samples: int*) → ndarray[Any, dtype[float64]]

Compute the intracell distance matrix for *tree* using the geodesic metric. Sample *num_samples* many points uniformly throughout the body of *tree*, compute the pairwise geodesic distance between all sampled points, and return the matrix of distances.

Returns
A numpy array, a "condensed distance matrix" in the sense of `scipy.spatial.distance.squareform()`, i.e., an array of shape (*num_samples* * *num_samples* - 1/2,). Contains the entries in the intracell geodesic distance matrix for *tree* lying strictly above the diagonal.

Parameters
- **tree** (NeuronTree) –
- **num_samples** (*int*) –

Return type
ndarray[*Any*, *dtype*[*float64*]]

compute_icdm_all_euclidean(*infolder: str, out_csv: str, n_sample: int, preprocess: ~typing.Callable[[list[cajal.swc.NeuronTree]], ~typing.Union[~cajal.utilities.Err[~cajal.utilities.T], list[cajal.swc.NeuronTree]]] = <function <lambda>>, num_processes: int = 8*) → list[tuple[str, cajal.utilities.Err[T]]]

For each *.swc file in infolder, read the *.swc file into memory as an SWCForest, *forest*. Apply a preprocessing function *preprocess* to *forest*, which can return either an error message (because the file is for whatever reason unsuitable for processing or sampling) or a potentially modified SWCForest *processed_forest*. Sample n_sample many points from the neuron, evenly spaced, and compute the Euclidean intracell matrix. Write the resulting intracell distance matrices for all cells passing the preprocessing test to a csv file with path *out_csv*.

Parameters
- **infolder** (*str*) – Directory of input *.swc files.

- **out_csv** (*str*) – Output file to write to.
- **n_sample** (*int*) – How many points to sample from each cell.
- **preprocess** (*Callable[[list[cajal.swc.NeuronTree]], Union[Err[T], list[cajal.swc.NeuronTree]]]*) – *preprocess* is expected to be roughly of the following form:

  1. Apply such-and-such tests of data quality and integrity to the SWCForest. (For example, check that the forest has only a single connected component, that it has only a single soma node, that it has at least one soma node, that it contains nodes from the axon, that it does not have any elements whose structure_id is 0 (for 'undefined'), etc.)
  2. If any of the tests are failed, return an instance of `utilities.Err` with a message explaining why the *.swc file was ineligible for sampling.
  3. If all tests are passed, apply a transformation to *forest* and return the modified *new_forest*. (For example, filter out all axon nodes to focus on the dendrites, or filter out all undefined nodes, or filter out all components which have fewer than 10% of the nodes in the largest component.)

  If *preprocess(forest)* returns an instance of the `utilities.Err` class, this file is not sampled from, and its name is added to a list together with the error returned by *preprocess*. If *preprocess(forest)* returns a SWCForest, this is what will be sampled. By default, no preprocessing is performed, and the neuron is processed as-is.
- **num_processes** (*int*) – the intracell distance matrices will be computed in parallel processes, num_processes is the number of processes to run simultaneously. Recommended to set equal to the number of cores on your machine.

Returns
List of pairs (cell_name, error), where cell_name is the cell for which sampling failed, and *error* is a wrapper around a message indicating why the neuron was not sampled from.

Return type
list[tuple[str, cajal.utilities.Err[~T]]]

compute_icdm_all_geodesic(*infolder: str, out_csv: str, n_sample: int, num_processes: int = 8, preprocess: ~typing.Callable[[list[cajal.swc.NeuronTree]], ~typing.Union[~cajal.utilities.Err[~cajal.utilities.T], ~cajal.swc.NeuronTree]] = <function <lambda>>*) → list[tuple[str, cajal.utilities.Err[T]]]

This function is substantially the same as *cajal.sample_swc.compute_icdm_all_euclidean()* and the user should consult the documentation for that function. However, note that *preprocess* has a different type signature, it is expected to return a *NeuronTree* rather than an *SWCForest*. There is not a meaningful notion of geodesic distance between points in two different components of a graph.

The default preprocessing is to take the largest component.

Parameters

- **infolder** (*str*) –
- **out_csv** (*str*) –
- **n_sample** (*int*) –
- **num_processes** (*int*) –
- **preprocess** (*Callable[[list[cajal.swc.NeuronTree]], Union[Err[T], NeuronTree]]*) –

Return type
list[tuple[str, cajal.utilities.Err[~T]]]

CHAPTER

ELEVEN

PROCESSING OBJ MESHES

CAJAL supports cell morphology data in the form of Wavefront *.obj files.

A *.obj file should consist of a series of lines, either - comments starting with "#" (discarded) - a vertex line, starting with "v" and followed by three floating point xyz coordinates - a face line, starting with f and followed by three integers which are indices for the vertices

All other lines will be ignored or discarded.

For examples of compatible mesh files see the folder /CAJAL/data/obj_files in the CAJAL Git repository.

The sample_mesh.py file contains functions to help the user sample points from an *.obj file and compute the geodesic distances between points.

class cajal.sample_mesh.**VertexArray**

A `sample_mesh.VertexArray` is a numpy array of shape (n, 3), where n is the number of vertices in the mesh.

Each row of a `sample_mesh.VertexArray` is an XYZ coordinate triple for a point in the mesh.

Value
numpy.typing.NDArray[numpy.float_]

class cajal.sample_mesh.**FaceArray**

A FaceArray is a numpy array of shape (m, 3) where m is the number of faces in the mesh. Each row of a FaceArray is a list of three natural numbers, corresponding to indices in the corresponding VertexArray, representing triangular faces joining those three points.

Value
numpy.typing.NDArray[numpy.int_]

read_obj(*file_path: str*) → Tuple[*sample_mesh.VertexArray*, *sample_mesh.FaceArray*]

Reads in the vertices and triangular faces of a .obj file.

Parameters
file_path (*str*) – Path to .obj file

Returns

Ordered pair *(vertices, faces)*, where:

- *vertices* is an array of 3D floating-point coordinates of shape *(n,3)*, where *n* is the number of vertices in the mesh
- *faces* is an array of shape *(m,3)*, where *m* is the number of faces; the *k*-th row gives the indices for the vertices in the *k*-th face.

Return type
Tuple[*sample_mesh.VertexArray*, *sample_mesh.FaceArray*]

compute_icdm_all(*infolder: str*, *out_csv: str*, *metric: Union[Literal['euclidean'], Literal['geodesic']]*, *n_sample: int = 50*, *num_processes: int = 8*, *segment: bool = True*, *method: Union[Literal['networkx'], Literal['heat']] = 'heat'*) → List[str]

Go through every Wavefront *.obj file in the given input directory *infolder* and compute intracell distances according to the given metric. Write the results to output *.csv file named *out_csv*.

Parameters

- **infolder** (*str*) – Folder full of *.obj files.
- **out_csv** (*str*) – Output will be written to a *.csv file titled *out_csv*.
- **metric** (*Union[Literal['euclidean'], ~typing.Literal['geodesic']]*) – How to compute the distance between points.
- **n_sample** (*int*) – How many points to sample from each cell.
- **num_processes** (*int*) – Number of independent processes which will be created. Recommended to set this equal to the number of cores on your machine.
- **method** (*Union[Literal['networkx'], ~typing.Literal['heat']]*) – How to compute geodesic distance. The "networkx" method is more precise, and takes between 5 - 15 seconds for a cell with 50 sample points. The "heat" method is a faster but rougher approximation, and takes between 0.05 - 0.15 seconds for a cell with 50 sample points. This flag is not relevant if the user is sampling Euclidean distances.
- **segment** (*bool*) – If *segment* is True, each *.obj file will be segmented into its set of connected components before being returned, so an *.obj file with multiple connected components will be understood to contain multiple distinct cells. If *segment* is False, each *.obj file will be understood to contain a single cell, and points will be sampled accordingly. If *segment* is False and the user chooses "geodesic", in the event that an *.obj file contains multiple connected components, the function will attempt to "repair" the *.obj file by adjoining new faces to the complex so that a sensible notion of geodesic distance can be computed between two points. The user is warned that this imputing of data carries the same consequences with regard to scientific interpretation of the results as any other kind of data imputation for incomplete data sets.

Returns

Names of cells for which sampling failed because the cells have fewer than *n_sample* points.

Return type

List[str]

CHAPTER

TWELVE

SAMPLING FROM SEGMENTED IMAGES

cell_boundaries(*imarray: ndarray[Any, dtype[int64]]*, *n_sample: int*, *background: int = 0*, *discard_cells_with_holes: bool = False*, *only_longest: bool = False*) → List[Tuple[int, ndarray[Any, dtype[float64]]]]

Sample n coordinates from the boundary of each cell in a segmented image, skipping cells that touch the border of the image

Parameters

- **imarray** (*ndarray[Any, dtype[int64]]*) – 2D segmented image where the pixels belonging to different cells have different values
- **n_sample** (*int*) – number of pixel coordinates to sample from boundary of each cell
- **background** (*int*) – value of background pixels, this will not be saved as a boundary
- **discard_cells_with_holes** (*bool*) – if discard_cells_with_holes is true, we discard any cells with more than one boundary (e.g., an annulus) with a warning. Else, the behavior is determined by only_longest.
- **only_longest** (*bool*) – if discard_cells_with_holes is true, only_longest is irrelevant. Otherwise, this determines whether we sample points from only the longest boundary (presumably the exterior) or from all boundaries, exterior and interior.

Returns

list of float numpy arrays of shape (n_sample, 2) containing points sampled from the contours.

Return type

List[*Tuple*[int, *ndarray*[*Any*, *dtype*[*float64*]]]]

compute_icdm_all(*infolder: str*, *out_csv: str*, *n_sample: int*, *num_processes: int = 8*, *background: int = 0*, *discard_cells_with_holes: bool = False*, *only_longest: bool = False*) → None

Read in each segmented image in a folder (assumed to be .tif), save n pixel coordinates sampled from the boundary of each cell in the segmented image, skipping cells that touch the border of the image.

Parameters

- **infolder** (*str*) – path to folder containing .tif files.
- **out_csv** (*str*) – path to csv file to save cell boundaries.
- **n_sample** (*int*) – number of pixel coordinates to sample from boundary of each cell
- **discard_cells_with_holes** (*bool*) – if discard_cells_with_holes is true, we discard any cells with more than one boundary (e.g., an annulus) with a warning. Else, the behavior is determined by only_longest.
- **background** (*int*) – value which characterizes the color of the background pixels, this will not be saved as a boundary

- **only_longest** (*bool*) – if discard_cells_with_holes is true, only_longest is irrelevant. Otherwise, this determines whether we sample points from only the longest boundary (presumably the exterior) or from all boundaries, exterior and interior.
- **num_processes** (*int*) – How many threads to run while sampling.

Returns
None (writes to file)

Return type
None

CHAPTER

THIRTEEN

RUNNING GROMOV-WASSERSTEIN

class cajal.run_gw.**Distribution**

A run_gw.Distribution is a numpy array of shape (n,), with values nonnegative and summing to 1, where n is the number of points in the set.

Value
numpy.typing.NDArray[numpy.float_]

class cajal.run_gw.**SquareMatrix**

A SquareMatrix is a numpy array of shape (n, n) where n is the number of points in the space; in our applications a SquareMatrix is usually a distance matrix, a symmetric matrix with zeros along the diagonal.

Value
numpy.typing.NDArray[numpy.float_]

icdm_csv_validate(*intracell_csv_loc: str*) → None

Raise an exception if the file in intracell_csv_loc fails to pass formatting tests; else return None.

Parameters
intracell_csv_loc (*str*) – The (full) file path for the CSV file containing the intracell distance matrix.

Return type
None

The file format for an intracell distance matrix is as follows:

- A line whose first character is '#' is discarded as a comment.
- The first line which is not a comment is discarded as a "header" - this line may contain the column titles for each of the columns.
- Values separated by commas. Whitespace is not a separator.
- The first value in the first non-comment line should be the string 'cell_id', and all values in the first column after that should be a unique identifier for that cell.
- All values after the first column should be floats.
- Not including the cell id in the first column, each row except the header should contain the entries of an intracell distance matrix lying strictly above the diagonal, as in the footnotes of https://docs.scipy.org/doc/scipy/reference/ generated/scipy.spatial.distance.squareform.html

cell_iterator_csv(*intracell_csv_loc: str*) → Iterator[tuple[str, *run_gw.SquareMatrix*]]

Parameters
intracell_csv_loc (*str*) – A full file path to a csv file.

Returns
an iterator over cells in the csv file, given as tuples of the form (name, dmat). Intracell distance matrices are in squareform.

Return type
Iterator[tuple[str, *run_gw.SquareMatrix*]]

cell_pair_iterator_csv(*intracell_csv_loc: str*, *chunk_size: int*) → Iterator[tuple[tuple[int, str, *run_gw.SquareMatrix*], tuple[int, str, *run_gw.SquareMatrix*]]]

Parameters

- **intracell_csv_loc** (*str*) – A full file path to a csv file.
- **chunk_size** (*int*) – How many lines to read from the file at a time. Does not affect output.

Returns
an iterator over pairs of cells, each entry is of the form ((indexA, nameA, distance_matrixA),(indexB, nameB, distance_matrixB)), where *indexA* is the line number in the file, and *indexA* < *indexB*.

Return type
Iterator[tuple[tuple[int, str, *run_gw.SquareMatrix*], tuple[int, str, *run_gw.SquareMatrix*]]]

This is almost equivalent to itertools.combinations(cell_iterator_csv(intracell_csv_loc),2) but with more efficient file IO.

gw_pairwise_parallel(*cells: list[tuple[*run_gw.SquareMatrix, run_gw.Distribution*]]*, *num_processes: int*, *names: Optional[list[str]] = None*, *gw_dist_csv: Optional[str] = None*, *gw_coupling_mat_csv: Optional[str] = None*, *return_coupling_mats: bool = False*) → tuple[*run_gw.SquareMatrix*, Optional[list[tuple[int, int, run_gw.RectangularMatrix]]]]

Compute the pairwise Gromov-Wasserstein distances between cells, possibly along with their coupling matrices.

If appropriate file names are supplied, the output is also written to file. If computing a large number of coupling matrices, for reduced memory consumption it is suggested not to return the coupling matrices, and instead write them to file.

Parameters

- **cells** (*list[tuple[run_gw.SquareMatrix, run_gw.Distribution]]*) – A list of pairs (A,a) where *A* is a squareform intracell distance matrix and *a* is a probability distribution on the points of *A*.
- **num_processes** (*int*) – How many Python processes to run in parallel for the computation.
- **names** (*Optional[list[str]]*) – A list of unique cell identifiers, where names[i] is the identifier for cell i. This argument is required if gw_dist_csv is not None, or if gw_coupling_mat_csv is not None, and is ignored otherwise.
- **gw_dist_csv** (*Optional[str]*) – If this field is a string giving a file path, the GW distances will be written to this file. A list of cell names must be supplied.
- **gw_coupling_mat_csv** (*Optional[str]*) – If this field is a string giving a file path, the GW coupling matrices will be written to this file. A list of cell names must be supplied.
- **return_coupling_mats** (*bool*) – Whether the function should return the coupling matrices. Please be warned that for a large number of cells, *couplings* will be large, and memory consumption will be high. If *return_coupling_mats* is False, returns *(gw_dmat, None)*. This

argument is independent of whether the coupling matrices are written to a file; one may return the coupling matrices, write them to file, both, or neither.

Returns

If *return_coupling_mats* is True, returns *(gw_dmat, couplings)*, where gw_dmat is a square matrix whose (i,j) entry is the GW distance between two cells, and *couplings* is a list of tuples (i,j, coupling_mat) where *i,j* are indices corresponding to positions in the list *cells* and *coupling_mat* is a coupling matrix between the two cells. If *return_coupling_mats* is False, returns *(gw_dmat, None)*.

Return type

tuple[*run_gw.SquareMatrix*, *Optional*[list[tuple[int, int, run_gw.RectangularMatrix]]]]

compute_gw_distance_matrix(*intracell_csv_loc: str*, *gw_dist_csv_loc: str*, *num_processes: int*, *gw_coupling_mat_csv_loc: Optional[str] = None*, *return_coupling_mats: bool = False*, *verbose: Optional[bool] = False*) → tuple[*run_gw.SquareMatrix*, Optional[list[tuple[int, int, run_gw.RectangularMatrix]]]]

Compute the matrix of pairwise Gromov-Wasserstein distances between cells. This function is a wrapper for *cajal.run_gw.gw_pairwise_parallel()* except that it reads icdm's from a file rather than from a list. For the file format of icdm's see *cajal.run_gw.icdm_csv_validate()*.

Parameters

- **intracell_csv_loc** (*str*) – A file containing the intracell distance matrices for all cells.
- **gw_dist_csv_loc** (*str*) –
- **num_processes** (*int*) –
- **gw_coupling_mat_csv_loc** (*Optional[str]*) –
- **return_coupling_mats** (*bool*) –
- **verbose** (*Optional[bool]*) –

Return type

tuple[*run_gw.SquareMatrix*, *Optional*[list[tuple[int, int, run_gw.RectangularMatrix]]]]

For other parameters see *cajal.run_gw.gw_pairwise_parallel()*.

CHAPTER

FOURTEEN

SECOND LOWER BOUND AND QUANTIZED GROMOV-WASSERSTEIN

slb_parallel_memory(*cell_dms: Collection[ndarray[Any, dtype[float64]]]*, *num_processes: int*, *chunksize: int = 20*) → ndarray[Any, dtype[float64]]

Compute the SLB distance in parallel between all cells in *cell_dms*. :param cell_dms: A collection of distance matrices. Probability distributions other than uniform are currently unsupported. :param num_processes: How many Python processes to run in parallel :param chunksize: How many SLB distances each Python process computes at a time

Returns
a square matrix giving pairwise SLB distances between points.

Parameters

- **cell_dms** (*Collection[ndarray[Any, dtype[float64]]]*) –
- **num_processes** (*int*) –
- **chunksize** (*int*) –

Return type
ndarray[*Any*, *dtype*[*float64*]]

slb_parallel(*intracell_csv_loc: str*, *num_processes: int*, *out_csv: str*, *chunksize: int = 20*) → None

Compute the SLB distance in parallel between all cells in the csv file *intracell_csv_loc*. The files are expected to be formatted according to the format in *cajal.run_gw.icdm_csv_validate()*.

Parameters

- **cell_dms** – A collection of distance matrices
- **num_processes** (*int*) – How many Python processes to run in parallel
- **chunksize** (*int*) – How many SLB distances each Python process computes at a time
- **intracell_csv_loc** (*str*) –
- **out_csv** (*str*) –

Return type
None

class quantized_icdm(*cell_dm: ndarray[Any, dtype[float64]]*, *p: ndarray[Any, dtype[float64]]*, *num_clusters: int*)

This class represents a "quantized" intracell distance matrix, i.e., a metric measure space which has been equipped with a given clustering; it contains additional data which allows for the rapid computation of pairwise GW distances across many cells. Users should only need to understand how to use the constructor.

Parameters

- **cell_dm** (*ndarray[Any, dtype[float64]]*) – An intracell distance matrix in square-form.
- **p** (*ndarray[Any, dtype[float64]]*) – A probability distribution on the points of the metric space
- **num_clusters** (*int*) – How many clusters to subdivide the cell into; the more clusters, the more accuracy, but the longer the computation.

quantized_gw_parallel(*intracell_csv_loc: str*, *num_processes: int*, *num_clusters: int*, *out_csv: str*, *chunksize: int = 20*, *verbose: bool = False*) → None

Compute the quantized Gromov-Wasserstein distance in parallel between all cells in a family of cells.

Parameters

- **intracell_csv_loc** (*str*) – path to a CSV file containing the cells to process
- **num_processes** (*int*) – number of Python processes to run in parallel
- **num_clusters** (*int*) – Each cell will be partitioned into *num_clusters* many clusters.
- **out_csv** (*str*) – file path where a CSV file containing the quantized GW distances will be written
- **chunksize** (*int*) – How many q-GW distances should be computed at a time by each parallel process.
- **verbose** (*bool*) –

Return type
None

combined_slb_quantized_gw_memory(*cell_dms: Collection[ndarray[Any, dtype[float64]]]*, *num_processes: int*, *num_clusters: int*, *accuracy: float*, *nearest_neighbors: int*, *verbose: bool*, *chunksize: int = 20*)

Compute the pairwise SLB distances between each pair of cells in *cell_dms*. Based on this initial estimate of the distances, compute the quantized GW distance between the nearest with *num_clusters* many clusters until the correct nearest-neighbors list is obtained for each cell with a high degree of confidence.

The idea is that for the sake of clustering we can avoid computing the precise pairwise distances between cells which are far apart, because the clustering will not be sensitive to changes in large distances. Thus, we want to compute as precisely as possible the pairwise GW distances for (say) the 30 nearest neighbors of each point, and use a rough estimation beyond that.

Parameters

- **cell_dms** (*Collection[ndarray[Any, dtype[float64]]]*) – a list or tuple of square distance matrices
- **num_processes** (*int*) – How many Python processes to run in parallel
- **num_clusters** (*int*) – Each cell will be partitioned into *num_clusters* many clusters for the quantized Gromov-Wasserstein distance computation.
- **chunksize** (*int*) – Number of pairwise cell distance computations done by each Python process at one time.
- **out_csv** – path to a CSV file where the results of the computation will be written
- **accuracy** (*float*) – This is a real number between 0 and 1, inclusive.
- **nearest_neighbors** (*int*) – The algorithm tries to compute only the quantized GW distances between pairs of cells if one is within the first *nearest_neighbors* neighbors of the other; for all other values, the SLB distance is used to give a rough estimate.

- **verbose** (*bool*) –

combined_slb_quantized_gw(*input_icdm_csv_location: str*, *gw_out_csv_location: str*, *num_processes: int*, *num_clusters: int*, *accuracy: float*, *nearest_neighbors: int*, *verbose: bool = False*, *chunksize: int = 20*) → None

This is a wrapper around *cajal.qgw.combined_slb_quantized_gw_memory()* with some associated file/IO. For all parameters not listed here see the docstring for *cajal.qgw.combined_slb_quantized_gw_memory()*.

Parameters

- **input_icdm_csv_location** (*str*) – file path to a csv file. For format for the icdm see *cajal.run_gw.icdm_csv_validate()*.
- **gw_out_csv_location** (*str*) – Where to write the output GW distances.
- **num_processes** (*int*) –
- **num_clusters** (*int*) –
- **accuracy** (*float*) –
- **nearest_neighbors** (*int*) –
- **verbose** (*bool*) –
- **chunksize** (*int*) –

Returns
None.

Return type
None

CHAPTER

FIFTEEN

LAPLACIAN SCORE

laplacian_scores(*feature_arr: ndarray[Any, dtype[float64]]*, *distance_matrix: ndarray[Any, dtype[float64]]*, *epsilon: float*, *permutations: int*, *covariates: Optional[ndarray[Any, dtype[float64]]]*, *return_random_laplacians: bool*) → dict[str, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]

Parameters

- **feature_arr** (*ndarray[Any, dtype[float64]]*) – An array of shape (N, num_features), where N is the number of nodes in the graph, and num_features is the number of features. Each column represents a feature on N elements. Columns should be preprocessed to remove constant features.
- **distance_matrix** (*ndarray[Any, dtype[float64]]*) – vectorform distance matrix
- **epsilon** (*float*) – connect nodes of graph if their distance is less than epsilon
- **permutations** (*int*) – Generate *permutations* many random permutations σ of the set of nodes of G, and compute the laplacian scores of the features $f \circ \sigma$ for each permutation σ. These additional laplacian scores are used to perform a non-parametric permutation test, returning a p-value representing the chance that the Laplacian would be equally as high for a randomly selected permutation of the feature.
- **covariates** (*Optional[ndarray[Any, dtype[float64]]]*) – (optional) array of shape (N, num_covariates), or simply (N,), where N is the number of nodes in the graph, and num_covariates is the number of covariates
- **return_random_laplacians** (*bool*) – if True, the output dictionary will contain all of the generated laplacians. This will likely be the largest object in the dictionary.

Returns

A pair of dictionaries *(feature_data, other)*. All values in feature_data are of shape (num_features,).

- feature_data['feature_laplacians'] := the laplacian scores of f, shape (num_features,)
- feature_data['laplacian_p_values'] := the p-values from the permutation test, shape (num_features,)
- feature_data['laplacian_q_values'] := the q-values from the permutation test, shape (num_features,)
- (Optional, if covariates is not None) (for i in range(1, covariates.shape[0])) feature_data['beta_i'] := the p-value that beta_i is not zero for that feature; see p. 228, 'Applied Linear Statistical Models', Nachtsheim, Kutner, Neter, Li. Shape (num_features,)

- (Optional, if covariates is not None) feature_data['regression_coefficients_fstat_p_values'] := the p-value that not all beta_i are zero, using the F-statistic, see p. 226, 'Applied Linear Statistical Models', Nachtsheim, Kutner, Neter, Li. Shape (num_features,)
- (Optional, if covariates is not None) feature_data['laplacian_p_values_post_regression'] := the p-value of the residual laplacian of the feature once the covariates have been regressed out.
- (Optional, if covariates is not None) feature_data['laplacian_q_values_post_regression'] := the q-values from the permutation test, shape (num_features,)
- (Optional, if covariates is not None) other['covariate_laplacians'] := the laplacian scores of the covariates, shape (num_covariates,) (if a matrix of covariates was supplied, else this entry will be absent)
- (Optional, if *return_random_laplacians* is True) other['random_feature_laplacians'] := the matrix of randomly generated feature laplacians, shape (permutations,num_features).
- (Optional, if *covariates* is not None and *return_random_laplacians* is True) other['random_covariate_laplacians'] := the matrix of randomly generated covariate laplacians, shape (permutations, num_covariates)

Return type

dict[str, numpy.ndarray[*Any*, numpy.dtype[numpy.float64]]]

CHAPTER

SIXTEEN

AVERAGE CELL SHAPES

avg_shape(*obj_names: list[str]*, *gw_dist_dict: dict[tuple[str, str], float]*, *iodms: dict[str, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*, *gw_coupling_mat_dict: dict[tuple[str, str], scipy.sparse._coo.coo_matrix]*)

Compute capped and uncapped average distance matrices. In both cases the distance matrix is rescaled so that the minimal distance between two points is 1. The "capped" distance matrix has a max distance of 2.

Parameters

- **obj_names** (*list[str]*) – Keys for the gw_dist_dict and iodms.
- **gw_dist_dict** (*dict[tuple[str, str], float]*) – Dictionary mapping ordered pairs (cellA_name, cellB_name) to Gromov-Wasserstein distances.
- **iodms** (*dict[str, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) – (intra-object distance matrices) - Maps object names to intra-object distance matrices. Matrices are assumed to be given in vector form rather than squareform.
- **gw_coupling_mat_dict** (*dict[tuple[str, str], scipy.sparse._coo.coo_matrix]*) – Dictionary mapping ordered pairs (cellA_name, cellB_name) to Gromov-Wasserstein coupling matrices from cellA to cellB.

avg_shape_spt(*obj_names: list[str]*, *gw_dist_dict: dict[tuple[str, str], float]*, *iodms: dict[str, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*, *gw_coupling_mat_dict: dict[tuple[str, str], scipy.sparse._coo.coo_matrix]*, *k: int*)

Given a set of cells together with their intracell distance matrices and the (precomputed) pairwise GW coupling matrices between cells, construct a morphological "average" of cells in the cluster. This function:

- aligns all cells in the cluster with each other using the coupling matrices
- takes a "local average" of all intracell distance matrices, forming a distance matrix which models the average local connectivity structure of the neurons
- draws a minimum spanning tree through the intracell distance graph, allowing us to visualize this average morphology

Parameters

- **obj_names** (*list[str]*) – Keys for the gw_dist_dict and iodms; unique identifiers for the cells.
- **gw_dist_dict** (*dict[tuple[str, str], float]*) – Dictionary mapping ordered pairs (cellA_name, cellB_name) to Gromov-Wasserstein distances between them.
- **iodms** (*dict[str, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) – (intra-object distance matrices) - Maps object names to intra-object distance matrices. Matrices are assumed to be given in vector form rather than squareform.

- **k** (*int*) – how many neighbors in the nearest-neighbors graph.
- **gw_coupling_mat_dict** (*dict[tuple[str, str], scipy.sparse._coo.coo_matrix]*) –

Gw_coupling_mat_dict

Dictionary mapping ordered pairs (cellA_name, cellB_name) to Gromov-Wasserstein coupling matrices from cellA to cellB.

CHAPTER

SEVENTEEN

CLUSTERING

leiden_clustering(*gw_mat: ndarray[Any, dtype[float64]]*, *nn: int = 5*, *resolution: Optional[float] = None*, *seed: Optional[int] = None*) → ndarray[Any, dtype[int64]]

Compute clustering of cells based on GW distance, using Leiden clustering on a nearest-neighbors graph

Parameters

- **gw_mat** (*ndarray[Any, dtype[float64]]*) – NxN distance matrix of GW distance between cells
- **nn** (*int*) – number of neighbors in nearest-neighbors graph
- **resolution** (*Optional[float]*) – If None, use modularity to get optimal partition. If float, get partition at set resolution.
- **seed** (*Optional[int]*) – Seed for the random number generator. Uses a random seed if nothing is specified.

Returns

numpy array of cluster assignment for each cell

Return type

ndarray[*Any*, *dtype*[*int64*]]

louvain_clustering(*gw_mat: ndarray[Any, dtype[float64]]*, *nn: int*) → ndarray[Any, dtype[int64]]

Compute clustering of cells based on GW distance, using Louvain clustering on a nearest-neighbors graph

Parameters

- **gw_mat** (*ndarray[Any, dtype[float64]]*) – NxN distance matrix of GW distance between cells
- **nn** (*int*) – number of neighbors in nearest-neighbors graph

Returns

numpy array of shape (num_cells,) the cluster assignment for each cell

Return type

ndarray[*Any*, *dtype*[*int64*]]

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

P

Q

R

S

T

W