quollr: An R Package for Visualizing 2-D Models from Non-linear Dimension Reductions in High Dimensional Space

by Jayani P. Gamage, Dianne Cook, Paul Harrison, Michael Lydeamore, and Thiyanga S. Talagala

Abstract Non-linear dimension reduction (NLDR) methods provide a low-dimensional representation of high-dimensional data (*p-D*) by applying a non-linear transformation. However, the complexity of the transformations and data structures can create wildly different representations depending on the method and (hyper-)parameter choices. It is difficult to determine whether any of these representations are accurate, which one is the best, or whether they have missed important structures. The R package **quollr** has been developed as a new visual tool to determine which method and which (hyper-)parameter choices provide the most accurate representation of high-dimensional data. The `scurve` data from the package is used to illustrate the algorithm. Single-cell RNA sequencing (scRNA-seq) data from mouse limb muscles are used to demonstrate the usability of the package.

1 Introduction

Non-linear dimension reduction (NLDR) techniques, such as t-distributed stochastic neighbor embedding (tSNE) (Maaten and Hinton, 2008), uniform manifold approximation and projection (UMAP) (McInnes et al., 2018), potential of heat-diffusion for affinity-based trajectory embedding (PHATE) algorithm (Moon et al., 2019), large-scale dimensionality reduction Using triplets (TriMAP) (Amid and Warmuth, 2019), and pairwise controlled manifold approximation (PaCMAP) (Wang et al., 2021), create wildly different representations depending on the selected method and (hyper-)parameter choices. It is difficult to determine whether any of these representations are accurate, which one is the best, or whether they have missed important structures.

This paper presents the R package, `quollr`, which is useful for understanding how NLDR warps high-dimensional space and fits the data. Starting with an NLDR layout, our approach is to create a wireframe model representation, that can be lifted and displayed in the high-dimensional space (Figure 1).

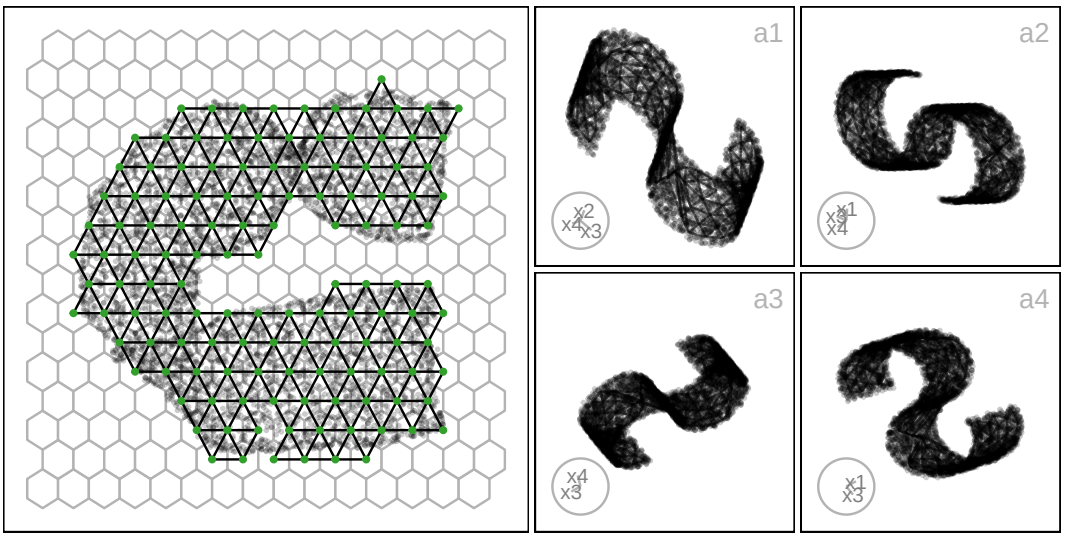


Figure 1: algorithm

The paper is organized as follows. The next section introduces the implementation of the `quollr` package on CRAN, including a demonstration of the package's key functions and visualization capabilities. In the application section, we illustrate the algorithm's functionality for studying a clustering data structure. Finally, we conclude the paper with a brief summary and discuss potential opportunities for using our algorithm.

2 Algorithm

Our algorithm includes the following steps: (1) scaling the NLDR data, (2) computing configurations of a hexagon grid, (3) binning the data, (4) obtaining the centroids of each bin, (5) indicating neighboring bins with line segments that connect the centroids, and (6) lifting the model into high dimensions (Figure 2). A detailed description of the algorithm can be found in **cite the methodology paper**.

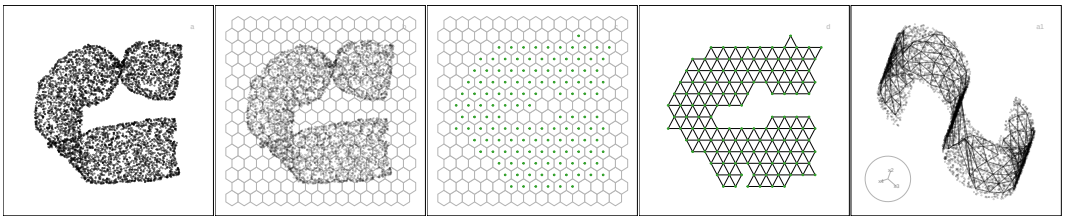


Figure 2: Key steps for constructing the model on the UMAP layout: (a) NLDR data, (b) hexagon bins, (c) bin centroids, (d) triangulated centroids, and (e) lifting the model into high dimensions. The 'Scurve' data is shown.

3 Implementation

Installation

The development version can be installed from GitHub:

```
devtools::install_github("JayaniLakshika/quollr")
```

Usage

The following demonstration of the package's functionality assumes `quollr` has been loaded. We also want to load the built-in data sets `scurve` and `scurve_umap`.

`scurve` is a 7-D simulated dataset. It is constructed by simulating 5000 observations from $\theta \sim U(-3\pi/2, 3\pi/2)$, $X_1 = \sin(\theta)$, $X_2 \sim U(0,2)$ (adding thickness to the S), $X_3 = \text{sign}(\theta) \times (\cos(\theta) - 1)$. The remaining variables X_4, X_5, X_6, X_7 are all uniform error, with small variance. `scurve_umap` is the UMAP 2-D embedding for `scurve` data with `n_neighbors` is 46 and `min_dist` is 0.9. Each data set contains a unique ID column that maps `scurve` and `scurve_umap`.

Main function

The mains steps for the algorithm can be executed by the main function `fit_highd_model()`, or can be run separately for more flexibility.

This function requires several parameters: the high-dimensional data (`highd_data`), the emdedding data (`nldr_data`), the number of bins along the x-axis (`bin1`), the buffer amount as a proportion of data (`q`), and benchmark value to extract high density hexagons (`benchmark_highdens`). The function returns an object that includes the scaled NLDR object (`nldr_obj`), the hexagonal object (`hb_obj`), the fitted model in both 2-D (`model_2d`), and p-D (`model_highd`), and triangular mesh (`trimesh_data`).

```
fit_highd_model(
  highd_data = scurve,
  nldr_data = scurve_umap,
  bin1 = 15,
  q = 0.1,
  benchmark_highdens = 5)
```

Constructing the 2-D Model

Constructing the 2-D model primarily involves (i) scaling the NLDR data, (ii) binning the data, (iii) obtaining bin centroids, (iv) connecting centroids with line segments to indicate neighbors, and (v) Remove low-density hexagons.

Scaling the data The algorithm starts by scaling the NLDR data to a standard range using the `gen_scaled_data()` function. This function standardizes the data so that the first embedding ranges from 0 to 1, while the second embedding scales from 0 to the maximum value of the second embedding. The output includes the scaled NLDR data along with the original limits of the embeddings.

```
scurve_umap_obj <- gen_scaled_data(nldr_data = scurve_umap)

scurve_umap_obj
```

```
#> $scaled_nldr
#> # A tibble: 5,000 x 3
#>     emb1  emb2    ID
#>    <dbl> <dbl> <int>
#>  1 0.707 0.839     1
#>  2 0.231 0.401     2
#>  3 0.232 0.215     3
#>  4 0.790 0.564     4
#>  5 0.761 0.551     5
#>  6 0.445 0.721     6
#>  7 0.900 0.137     7
#>  8 0.247 0.392     8
#>  9 0.325 0.542     9
#> 10 0.278 0.231    10
#> # i 4,990 more rows
#>
#> $lim1
#> [1] -14.42166  13.32655
#>
#> $lim2
#> [1] -12.43687  12.32455
```

Computing hexagon grid configurations The configurations of a hexagonal grid are determined by the number of bins and the bin width in each direction. The function `calc_bins_y()` is used for this purpose. This function accepts an object containing scaled NLDR data in the first and second columns, along with numeric vectors that represent the limits of the original NLDR data, the number of bins along the x-axis (`bin1`), and the buffer amount as a proportion (`q`).

```
bin_configs <- calc_bins_y(
  nldr_obj = scurve_umap_obj,
  bin1 = 15,
  q = 0.1)

bin_configs
```

```
#> $bin2
#> [1] 16
#>
#> $a1
#> [1] 0.08326136
#>
#> $a2
#> [1] 0.07210645
```

Binning the data Points are allocated to bins based on the nearest centroid. The hexagonal binning algorithm can be executed using the `hex_binning()` function, or its components can be run separately for added flexibility. The parameters used within `hex_binning()` include an object containing scaled NLDR data in the first and second columns, along with numeric vectors that represent the limits of the original NLDR data (`nldr_obj`), the number of bins along the x-axis (`bin1`), and the buffer amount as a proportion of the data (`q`). The output is an object of the `hex_bin_obj` class, which contains the bin widths in each direction (`a1`, `a2`), the number of bins in each direction (`bins`), the coordinates of the hexagonal grid starting point (`start_point`), the details of bin centroids (`centroids`), the coordinates of bins (`hex_poly`), NLDR components with their corresponding hexagon IDs (`data_hb_id`), hex bins with their corresponding standardized counts (`std_cts`), the total number of bins (`tot_bins`), the number of non-empty bins (`non_bins`), and the points within each hexagon (`pts_bins`).

```
hb_obj <- hex_binning(
  nldr_obj = scurve_umap_obj,
  bin1 = 15,
  q = 0.1)
```

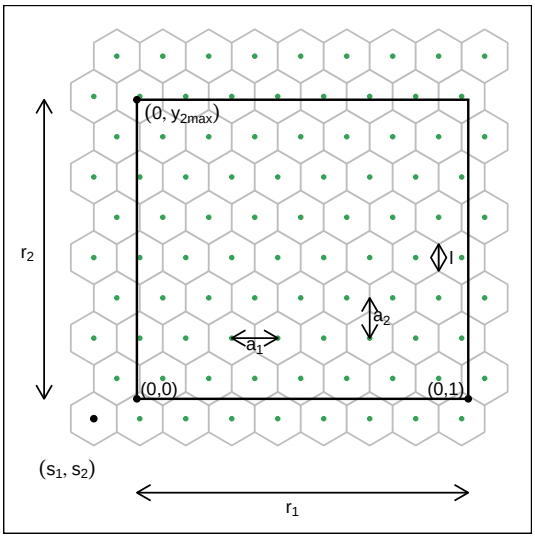


Figure 3: The components of the hexagon grid illustrating notation.

If the hexagonal binning process is run separately, it involves several steps: (i) generating all possible centroids in a hexagonal grid, (ii) creating the coordinates of the hexagons, (iii) assigning data points to their respective hexagons, (iv) computing the standardized number of points within each hexagon, and (v) mapping the points to their corresponding hexagonal bins.

Generating all possible centroids in a hexagonal grid The `gen_centroids()` function calculates the centroids of a hexagonal grid.

The coordinate limits of the embedding (`lim1` and `lim2`) are used to compute the aspect ratio between the two axes, which informs vertical spacing. The function then calls `calc_bins_y()`, a helper function that determines the appropriate number of hexagonal rows (`bin2`) and the width of each hexagon (`a1`) given the specified number of bins along the x-axis (`bin1`) and buffer (`q`).

Then, the centroids are computed iteratively. The x-coordinates for centroids in odd-numbered rows are initialized as a sequence spaced by the hexagon width. Even-numbered rows are staggered by half this width to achieve a hexagonal tiling effect. Vertical spacing (`a2`) is derived by $\sqrt{3}/2 \times a_1$.

The y-coordinates for each row are similarly calculated, and paired with the x-coordinates based on whether the total number of rows is even or odd. In the case of an odd number of rows, the final row uses only the odd-row x-coordinates to maintain the alternating pattern.

Finally, a tibble is returned containing a unique hexagon ID (`hexID`) along with the corresponding x and y centroid coordinates (`c_x`, `c_y`), which define the layout of the hexagonal grid over the $2-\mathrm{D}$ space.

```
all_centroids_df <- gen_centroids(
  nldr_obj = scurve_umap_obj,
  bin1 = 15,
  q = 0.1
  )

all_centroids_df
```

```
#> # A tibble: 240 x 3
#>    hexID     c_x     c_y
#>    <int>   <dbl>   <dbl>
#>  1     1 -0.1    -0.0892
#>  2     2 -0.0167 -0.0892
#>  3     3  0.0665 -0.0892
```

```
#>  4     4  0.150  -0.0892
#>  5     5  0.233  -0.0892
#>  6     6  0.316  -0.0892
#>  7     7  0.400  -0.0892
#>  8     8  0.483  -0.0892
#>  9     9  0.566  -0.0892
#> 10    10  0.649  -0.0892
#> # i 230 more rows
```

Creating the coordinates of the hexagons Following the generation of hexagonal centroids, the `gen_hex_coord()` function constructs the coordinates of each hexagonal bin by defining its six polygonal vertices. These coordinates are used to visualize the hexagonal tessellation.

Each hexagon is defined relative to its centroid (C_x, C_y), with six vertices positioned equidistantly around the center. The function first verifies the presence of the required hexagon width parameter `a1`. This width determines the horizontal spacing.

Two derived constants are calculated to define the relative distances to the vertices. The horizontal and vertical offset is defined as $dx = \frac{a_1}{2}$, and $dy = \frac{a_1}{\sqrt{3}}$ repectively. A vertical spacing factor $v = \frac{a_1}{2\sqrt{3}}$ refines vertical placement in staggered rows.

With these values, the function determines fixed offsets in the x and y directions for all six vertices relative to the centroid. These offsets form two vectors (`x_add_factor` and `y_add_factor`) corresponding to the six compass directions used to define the polygon shape: top, top-left, bottom-left, bottom, bottom-right, and top-right.

For each centroid, six vertices are computed and assigned a polygon ID (`hex_poly_id`) corresponding to the centroid's `hexID`. These points are aggregated into a tibble that includes the polygon ID (`hex_poly_id`) and the respective x (`x`) and y (`y`) coordinates for all hexagon corners.

```
all_hex_coord <- gen_hex_coord(
  centroids_data = all_centroids_df,
  a1 = bin_configs$a1
  )

all_hex_coord
```

```
#> # A tibble: 1,440 x 3
#>    hex_poly_id       x       y
#>          <int>   <dbl>   <dbl>
#>  1           1 -0.1    -0.0412
#>  2           1 -0.142  -0.0652
#>  3           1 -0.142  -0.113
#>  4           1 -0.1    -0.137
#>  5           1 -0.0584 -0.113
#>  6           1 -0.0584 -0.0652
#>  7           2 -0.0167 -0.0412
#>  8           2 -0.0584 -0.0652
#>  9           2 -0.0584 -0.113
#> 10           2 -0.0167 -0.137
#> # i 1,430 more rows
```

Assigning data points to their respective hexagons After generating the centroids that define the hexagonal grid, the next step is to assign each point in the NLDR embedding to its nearest hexagonal bin. The `assign_data()` function performs this assignment by calculating the $2-$D Euclidean distance between each point in the $2-$D embedding and all hexagon centroids.

First, the function extracts the first two dimensions of the scaled NLDR embedding, which represent the $2-$D layout. It then selects the corresponding x and y coordinates of each hexagon's centroid.

Both the embedding coordinates and the centroid coordinates are converted to matrices to facilitate distance computations. The function uses the `proxy::dist()` method to compute a pairwise Euclidean distance matrix between all NLDR points and all centroids. For each NLDR point, the function identifies the index of the centroid with the smallest distance representing the closest hexagon—and assigns the corresponding hexagon ID (`hexID`) to the point.

The result is a data frame of the scaled $2-D$ embedding with an additional `hexID` column, indicating the hexagonal bin to which each point belongs.

```
umap_hex_id <- assign_data(
  nldr_obj = scurve_umap_obj,
  centroids_data = all_centroids_df
  )

umap_hex_id
```

```
#> # A tibble: 5,000 x 4
#>     emb1  emb2    ID hexID
#>    <dbl> <dbl> <int> <int>
#>  1 0.707 0.839     1   205
#>  2 0.231 0.401     2   109
#>  3 0.232 0.215     3    65
#>  4 0.790 0.564     4   146
#>  5 0.761 0.551     5   146
#>  6 0.445 0.721     6   172
#>  7 0.900 0.137     7    58
#>  8 0.247 0.392     8   110
#>  9 0.325 0.542     9   141
#> 10 0.278 0.231    10    80
#> # i 4,990 more rows
```

Computing the standardized number of points within each hexagon The `compute_std_counts()` function calculates both the raw and standardized counts for each hexagon.

The function begins by grouping the data by `hexID` and counting the number of NLDR points falling within each bin. These raw counts are stored as `bin_counts`. To enable comparisons across bins with varying densities, the function then standardizes these counts by dividing each bin's count by the maximum count across all bins. This yields a normalized metric, `std_counts`, ranging from 0 to 1.

```
std_df <- compute_std_counts(
  scaled_nldr_hexid = umap_hex_id
  )

std_df
```

```
#> # A tibble: 142 x 3
#>    hexID bin_counts std_counts
#>    <int>      <int>      <dbl>
#>  1    21         12     0.16
#>  2    22         17     0.227
#>  3    23         22     0.293
#>  4    24         10     0.133
#>  5    25          7     0.0933
#>  6    26         12     0.16
#>  7    27          1     0.0133
#>  8    36         42     0.56
#>  9    37         42     0.56
#> 10    38         44     0.587
#> # i 132 more rows
```

Mapping the points to their corresponding hexagonal bins The `find_pts()` function extracts the list of data point identifiers (`ID`) assigned to each hexagon in the NLDR space.

The function first groups the input data by `hexID`, which represents the hexagon label associated with each point in the $2-D$ layout. Within each group, it collects the `ID`s into a list, resulting in a summary data frame where each row corresponds to a single hexagon. The resulting column, `pts_list`, contains all point identifiers associated with that hexagon.

```
pts_df <- find_pts(
  scaled_nldr_hexid = umap_hex_id
```

```
  )

pts_df

#> # A tibble: 142 x 2
#>    hexID pts_list
#>    <int> <list>
#>  1    21 <int [12]>
#>  2    22 <int [17]>
#>  3    23 <int [22]>
#>  4    24 <int [10]>
#>  5    25 <int [7]>
#>  6    26 <int [12]>
#>  7    27 <int [1]>
#>  8    36 <int [42]>
#>  9    37 <int [42]>
#> 10    38 <int [44]>
#> # i 132 more rows
```

Obtaining bin centroids The `extract_hexbin_centroids()` function combines hexagonal bin coordinates, raw and standardized counts within each hexagons.

This function begins by arranging the `counts_data` by `hexID` to ensure consistent ordering. It then performs a full join with `centroids_data`, aligning hexagon IDs between the two datasets to incorporate both hexagonal bin centroids and count metrics. After merging, the function handles missing values in the count columns: any `NA` values in `std_counts` or `bin_counts` are replaced with zeros. This ensures that hexagons with no assigned data points are retained in the output, with zero values for count-related fields. The resulting data frame contains the full set of hexagon centroids along with associated bin counts and standardized counts.

```
df_bin_centroids <- extract_hexbin_centroids(
  centroids_data = all_centroids_df,
  counts_data = hb_obj$std_cts
  )

df_bin_centroids

#> # A tibble: 240 x 5
#>    hexID     c_x     c_y bin_counts std_counts
#>    <int>   <dbl>   <dbl>      <dbl>      <dbl>
#>  1     1 -0.1    -0.0892          0          0
#>  2     2 -0.0167 -0.0892          0          0
#>  3     3  0.0665 -0.0892          0          0
#>  4     4  0.150  -0.0892          0          0
#>  5     5  0.233  -0.0892          0          0
#>  6     6  0.316  -0.0892          0          0
#>  7     7  0.400  -0.0892          0          0
#>  8     8  0.483  -0.0892          0          0
#>  9     9  0.566  -0.0892          0          0
#> 10    10  0.649  -0.0892          0          0
#> # i 230 more rows
```

Indicating neighbors by line segments connecting centroids To represent the neighborhood structure of hexagonal bins in a $2-$ D layout, we employ Delaunay triangulation on the centroids of hexagons. This geometric approach is used to infer which bins are considered neighbors.

The `tri_bin_centroids()` function generates a triangulation object from the x and y coordinates of hexagon centroids using the `tripack::tri.mesh()` function. This triangulation forms the structural basis for identifying adjacent bins.

```
tr_object <- tri_bin_centroids(
  centroids_data = df_bin_centroids
  )
```

The gen_edges() function uses this triangulation object to extract line segments between neighboring bins. It constructs a unique set of bin-to-bin connections by identifying the triangle edges and filtering duplicate or reversed links. Each edge is then annotated with its start and end coordinates, and a Euclidean distance is computed using the helper function calc_2d_dist(). Only edges within a hexagon's neighborhood radius (based on the hexagon side length a1) are retained.

```
trimesh <- gen_edges(tri_object = tr_object, a1 = hb_obj$a1)

trimesh
```

```
#> # A tibble: 653 x 8
#>     from    to  x_from  y_from    x_to    y_to from_count to_count
#>    <int> <int>   <dbl>   <dbl>   <dbl>   <dbl>      <dbl>    <dbl>
#>  1     1     2 -0.1    -0.0892 -0.0167 -0.0892          0        0
#>  2    16    17 -0.0584 -0.0171  0.0249 -0.0171          0        0
#>  3    16    32 -0.0584 -0.0171 -0.0167  0.0550          0        0
#>  4     3    17  0.0665 -0.0892  0.0249 -0.0171          0        0
#>  5    17    18  0.0249 -0.0171  0.108  -0.0171          0        0
#>  6    17    33  0.0249 -0.0171  0.0665  0.0550          0        0
#>  7    31    46 -0.1     0.0550 -0.0584  0.127           0        0
#>  8    32    47 -0.0167  0.0550  0.0249  0.127           0        0
#>  9    32    33 -0.0167  0.0550  0.0665  0.0550          0        0
#> 10     4    18  0.150  -0.0892  0.108  -0.0171          0        0
#> # i 643 more rows
```

The update_trimesh_index() function re-indexes the node IDs to ensure that edge identifiers are sequentially numbered and consistent with downstream analysis.

```
trimesh <- update_trimesh_index(trimesh_data = trimesh)

trimesh
```

```
#> # A tibble: 653 x 8
#>     from    to  x_from  y_from    x_to    y_to from_count to_count
#>    <int> <int>   <dbl>   <dbl>   <dbl>   <dbl>      <dbl>    <dbl>
#>  1     1     2 -0.1    -0.0892 -0.0167 -0.0892          0        0
#>  2    16    17 -0.0584 -0.0171  0.0249 -0.0171          0        0
#>  3    16    32 -0.0584 -0.0171 -0.0167  0.0550          0        0
#>  4     3    17  0.0665 -0.0892  0.0249 -0.0171          0        0
#>  5    17    18  0.0249 -0.0171  0.108  -0.0171          0        0
#>  6    17    33  0.0249 -0.0171  0.0665  0.0550          0        0
#>  7    31    46 -0.1     0.0550 -0.0584  0.127           0        0
#>  8    32    47 -0.0167  0.0550  0.0249  0.127           0        0
#>  9    32    33 -0.0167  0.0550  0.0665  0.0550          0        0
#> 10     4    18  0.150  -0.0892  0.108  -0.0171          0        0
#> # i 643 more rows
```

Identifying and removing low-density hexagons Not all hexagons contain meaningful information. Some may have very few or no data points due to the sparsity or shape of the underlying structure. Simply removing hexagons with low counts (e.g., fewer than a fixed threshold) can lead to gaps or "holes" in the $2-D$ structure, potentially disrupting the continuity of the representation.

To address this, we propose a more nuanced method that evaluates each hexagon not only based on its own density, but also in the context of its immediate neighbors. The find_low_dens_hex() function identifies hexagonal bins with insufficient local support by calculating the average standardized count across their six neighboring bins. If this mean neighborhood density is below a user-defined threshold (e.g., 0.05), the hexagon is flagged for removal.

The find_low_dens_hex() function identifies hexagons with low point densities, considering the densities of their neighboring bins as well. The find_low_dens_hex() function relies on a helper, compute_mean_density_hex(), which iterates over all hexagons and computes the average density across neighbors based on their hexID and a defined number of bins along the x-axis (bin1). The hexagonal layout assumes a fixed grid structure, so neighbor IDs are computed by positional offsets.

```
find_low_dens_hex(
  centroids_data = df_bin_centroids,
  bin1 = 15,
  benchmark_mean_dens = 0.05
)
```

```
#>  [1]   1   2   3   4   5  11  12  13  14  15  16  17  18  19  30  31  32  33  46
#> [20]  47  61  75  76  90 105 120 135 150 151 166 181 182 196 197 198 211 212 213
#> [39] 214 215 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240
```

For simplicity, we remove low-density hexagons using a threshold of 10.

```
df_bin_centroids <- df_bin_centroids |>
  dplyr::filter(bin_counts > 10)

trimesh <- trimesh |>
  dplyr::filter(from_count > 10,
                to_count > 10)

trimesh <- update_trimesh_index(trimesh)
```

Lifting the model into high dimensions

The final step involves lifting the fitted 2-*D* model into *p*-*D* by computing the *p*-*D* mean of data points within each hexagonal bin to represent bin centroids. This transformation is performed using the `avg_highd_data()` function, which takes *p*-*D* data (`highd_data`) and embedding data with their corresponding hexagonal bin IDs as inputs (`scaled_nldr_hexid`).

```
df_bin <- avg_highd_data(
  highd_data = scurve,
  scaled_nldr_hexid = hb_obj$data_hb_id
)

df_bin
```

```
#> # A tibble: 142 x 8
#>    hexID      x1    x2    x3          x4         x5       x6         x7
#>    <int>   <dbl> <dbl> <dbl>       <dbl>      <dbl>    <dbl>      <dbl>
#>  1    21 -0.992   1.91 1.11  -0.000427   0.000624  0.00749  0.00105
#>  2    22 -0.906   1.93 1.41  -0.0000183  0.00331  -0.0204  -0.000363
#>  3    23 -0.680   1.93 1.72  -0.000810  -0.00259  -0.00449  0.00153
#>  4    24 -0.272   1.93 1.96   0.00251    0.00668  -0.0460   0.00128
#>  5    25  0.0760  1.93 2.00   0.00876    0.00447   0.00851 -0.00195
#>  6    26  0.461   1.93 1.89  -0.00478    0.00492   0.00835  0.00172
#>  7    27  0.719   1.99 1.70   0.0109    -0.00349  -0.0297  -0.00223
#>  8    36 -0.985   1.75 0.853 -0.00202    0.000397  0.00331  0.000338
#>  9    37 -0.980   1.66 1.17  -0.000374  -0.00154   0.0165   0.000126
#> 10    38 -0.821   1.64 1.56  -0.000459   0.000538 -0.0123   0.000780
#> # i 132 more rows
```

Prediction

The `predict_emb()` function is used to predict 2-*D* embedding for a new *p*-*D* data point using the fitted model. This function is useful to predict 2-*D* embedding irrespective of the NLDR technique.

In the prediction process, first, the nearest *p*-*D* model point is identified for a given new *p*-*D* data point by computing *p*-*D* Euclidean distance. Then, the corresponding 2-*D* bin centroid mapping for the identified *p*-*D* model point is determined. Finally, the coordinates of the identified 2-*D* bin centroid is used as the predicted NLDR embedding for the new *p*-*D* data point.

```
predict_emb(
  highd_data = scurve,
  model_2d = df_bin_centroids,
  model_highd = df_bin
  )
```

```
#> # A tibble: 5,000 x 4
#>    pred_emb_1 pred_emb_2    ID pred_hb_id
#>         <dbl>      <dbl> <int>      <int>
#>  1      0.691      0.848     1        205
#>  2      0.191      0.416     2        109
#>  3      0.316      0.199     3         66
#>  4      0.774      0.560     4        146
#>  5      0.774      0.560     5        146
#>  6      0.441      0.704     6        172
#>  7      0.941      0.127     7         58
#>  8      0.275      0.416     8        110
#>  9      0.358      0.560     9        141
#> 10      0.316      0.343    10         96
#> # i 4,990 more rows
```

Compute residuals and Root Mean Square Error (RMSE)

As a Goodness of fit statistics for the model, `glance()` is used to compute residuals and RMSE. These metrics are used to assess how well the fitted model will capture the underlying structure of the *p-D* data.

This function begins by renaming the columns of the input `model_highd` data frame to avoid naming conflicts during subsequent joins. It then uses the `predict_emb()` function to assign each point in the high-dimensional dataset to a corresponding bin in the 2D model, producing a prediction data frame that contains both the predicted bin assignment (`pred_hb_id`) and the original observation `ID`.

The function joins this prediction output with both the high-dimensional model (to get mean bin coordinates in the original space) and the original high-dimensional data (to retrieve true coordinates). It then calculates squared differences between the true and predicted high-dimensional coordinates for each dimension, storing these as `error_square_x1`, `error_square_x2`, ..., up to the dimensionality of the data.

From these per-dimension errors, the function computes absolute error which is the sum of absolute differences across all dimensions and observations and the RMSE which is the average of the total squared error per point.

These metrics are returned in a tibble as `Error` (absolute error) and `RMSE` (root mean squared error).

```
glance(
  highd_data = scurve,
  model_2d = df_bin_centroids,
  model_highd = df_bin
  )
```

```
#> # A tibble: 1 x 2
#>   Error  RMSE
#>   <dbl> <dbl>
#> 1 1481. 0.180
```

Furthermore, `augment()` accepts 2-*D* and *p-D* model points, and the *p-D* data and adds information about each observation in the data set.

The function starts by renaming columns in the `model_highd` data frame to avoid conflicts. It then predicts the high-dimensional bin assignments for each point using the `predict_emb()` function, mapping each observation to its nearest $2-$ D bin (`pred_hb_id`). This prediction is joined with the mean high-dimensional coordinates of each bin from the model and with the original high-dimensional data.

Next, the function computes residuals between each original coordinate (`x1`, `x2`, ..., `xp`) and the corresponding modeled coordinate (`model_high_d_x1`, ..., `model_high_d_xp`) across all dimensions. It calculates both squared errors and absolute errors per dimension. These are used to compute two aggregate diagnostic measures per point. First, the `row_wise_total_error` which is the total squared error across all dimensions, and the `row_wise_abs_error` which is the total absolute error across all dimensions.

The final output is a data frame that combines the original IDs, high-dimensional coordinates, predicted bin IDs, modeled coordinates, residuals, row wise total error, absolute error for the fitted values, and row wise total absolute error for each observation. The augmented dataset is always returned as a `tibble::tibble` with the same number of rows as the passed dataset.

```
model_error <- augment(
  highd_data = scurve,
  model_2d = df_bin_centroids,
  model_highd = df_bin
  )

model_error
```

```
#> # A tibble: 5,000 x 32
#>       ID      x1     x2       x3       x4       x5       x6        x7 pred_hb_id
#>    <int>   <dbl>  <dbl>    <dbl>    <dbl>    <dbl>    <dbl>     <dbl>      <int>
#>  1     1 -0.120  1.64   -1.99     0.0104   0.0125   0.0923  -0.00128         205
#>  2     2 -0.0492 1.51    0.00121 -0.0177   0.00726 -0.0362  -0.00535         109
#>  3     3 -0.774  1.30    0.367   -0.00173  0.0156  -0.0962   0.00335          66
#>  4     4 -0.606  0.246  -1.80    -0.00897 -0.0187  -0.0716   0.00126         146
#>  5     5 -0.478  0.0177 -1.88     0.00848  0.00533  0.0998   0.000677        146
#>  6     6  0.818  0.927  -1.58    -0.00318 -0.00980  0.0989   0.00696         172
#>  7     7  0.910  1.40    1.42     0.00699 -0.0182  -0.0710   0.00966          58
#>  8     8 -0.0691 1.59    0.00239  0.0127  -0.0130   0.0396  -0.000185        110
#>  9     9  0.859  1.59   -0.488   -0.0119   0.00421 -0.00440 -0.00595         141
#> 10    10 -0.727  1.62    0.314    0.00251  0.0177  -0.0755  -0.00369          96
#> # i 4,990 more rows
#> # i 23 more variables: model_high_d_x1 <dbl>, model_high_d_x2 <dbl>,
#> #   model_high_d_x3 <dbl>, model_high_d_x4 <dbl>, model_high_d_x5 <dbl>,
#> #   model_high_d_x6 <dbl>, model_high_d_x7 <dbl>, error_square_x1 <dbl>,
#> #   error_square_x2 <dbl>, error_square_x3 <dbl>, error_square_x4 <dbl>,
#> #   error_square_x5 <dbl>, error_square_x6 <dbl>, error_square_x7 <dbl>,
#> #   row_wise_total_error <dbl>, abs_error_x1 <dbl>, abs_error_x2 <dbl>, ...
```

Visualizations

The package offers several $2-$D visualizations, including:

- A full hexagonal grid,
- A hexagonal grid that matches the data,
- A full grid based on centroid triangulation,
- A centroid triangulation grid that aligns with the data,
- A triangular mesh for any provided set of points.

The generated p-D model, overlaid with the data, can also be visualized using `show_langevitour`. Additionally, it features a function for visualizing the $2-$D projection of the fitted model overlaid on the data, called `plot_proj`.

Furthermore, there are two interactive plots, `show_link_plots` and `show_error_link_plots`, which are designed to help diagnose the model.

Each visualization can be generated using its respective function, as described in this section.

Hexagonal grid The `geom_hexgrid()` function introduces a custom `ggplot2` layer designed for visualizing hexagonal grid on a provided set of bin centroids.

To display the complete grid, users should supply all available bin centroids.

```
full_hexgrid <- ggplot() +
  geom_hexgrid(
    data = hb_obj$centroids,
    aes(x = c_x, y = c_y)
    )
```

If the goal is to plot only the subset of hexagons that correspond to bins containing data points, then only the centroids associated with those bins should be passed.

```
data_hexgrid <- ggplot() +
  geom_hexgrid(
    data = df_bin_centroids,
    aes(x = c_x, y = c_y)
    )
```

Triangular mesh The `geom_trimesh()` function introduces a custom `ggplot2` layer designed for visualizing 2-*D* wireframe on a provided set of bin centroids.

To display the complete wireframe, users should supply all available bin centroids.

```
full_triangulation_grid <- ggplot() +
  geom_trimesh(
    data = hb_obj$centroids,
    aes(x = c_x, y = c_y)
    )
```

If the goal is to plot only the subset of hexagons that correspond to bins containing data points, then only the centroids associated with those bins should be passed.

```
data_triangulation_grid <- ggplot() +
  geom_trimesh(
    data = df_bin_centroids,
    aes(x = c_x, y = c_y)
    )
```

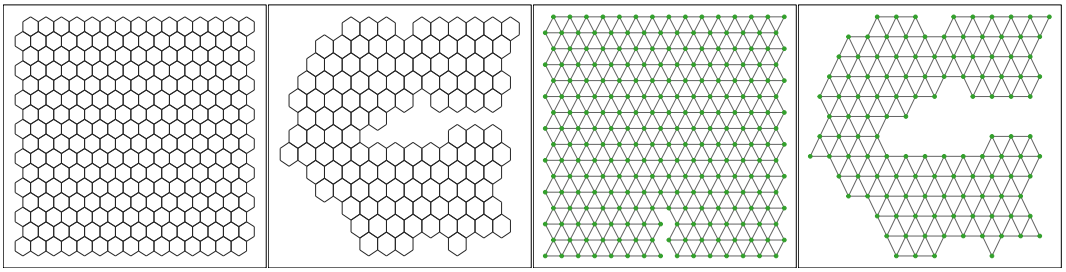


Figure 4: The outputs of 'geom_hexgrid' and 'geom_trimesh' include: (a) a complete hexagonal grid, (b) a hexagonal grid that corresponds with the data, (c) a full grid based on centroid triangulation, and (d) a centroid triangulation grid that aligns with the data.

***p*-*D* model visualization** To evaluate how well the *p*-*D* model captures the underlying structure of the high-dimensional data, we provide a visualization using the `show_langevitour()` function. This function renders a dynamic projection of both the high-dimensional data and the model using the `langevitour` package.

Before plotting, the data needs to be organized into a combined format through the `comb_data_model()` function. This function takes three inputs: `highd_data` (the high-dimensional observations), `model_highd` (high-dimensional summaries for each bin), and `model_2d` (the hexagonal bin centroids of the model). It returns a tidy data frame combining both the data and the model.

In this structure, the `type` variable distinguishes between original observations (`"data"`) and the bin-averaged model representation (`"model"`).

```
df_exe <- comb_data_model(
  highd_data = scurve,
  model_highd = df_bin,
  model_2d = df_bin_centroids
  )

df_exe
```

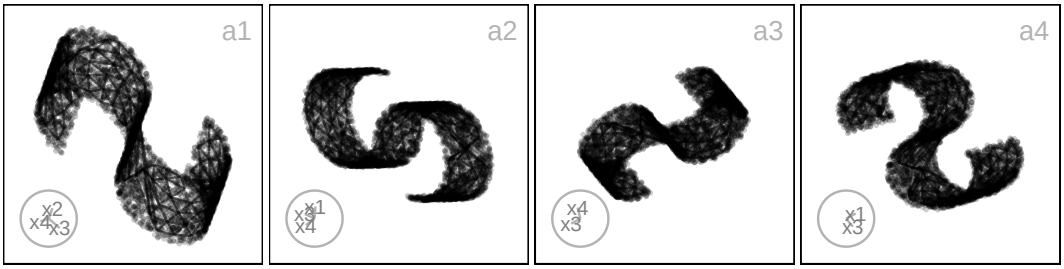
```
#> # A tibble: 5,122 x 8
#>         x1    x2    x3         x4         x5       x6        x7 type
#>      <dbl> <dbl> <dbl>      <dbl>      <dbl>    <dbl>     <dbl> <chr>
#>  1 -0.992   1.91 1.11  -0.000427   0.000624  0.00749  0.00105  model
#>  2 -0.906   1.93 1.41  -0.0000183  0.00331  -0.0204  -0.000363 model
#>  3 -0.680   1.93 1.72  -0.000810  -0.00259  -0.00449  0.00153  model
#>  4  0.461   1.93 1.89  -0.00478    0.00492   0.00835  0.00172  model
#>  5 -0.985   1.75 0.853 -0.00202    0.000397  0.00331  0.000338 model
#>  6 -0.980   1.66 1.17  -0.000374  -0.00154   0.0165   0.000126 model
```

```
#>  7 -0.821   1.64 1.56  -0.000459   0.000538 -0.0123   0.000780 model
#>  8 -0.484   1.68 1.87   0.00313    0.00241   0.00823 -0.00117  model
#>  9 -0.0991  1.70 1.99   0.00103    0.00150   0.00877 -0.000193 model
#> 10  0.295   1.74 1.95  -0.00165    0.000459  0.00330  0.000257 model
#> # i 5,112 more rows
```

The `show_langevitour()` function then renders the visualization using the `langevitour` interface, displaying both types of points in a dynamic tour. The `edge_data` input defines connections between neighboring bins (i.e., the hexagonal edges) to visualize the model's structure.

In the resulting interactive visualization black points represent the high-dimensional data, green points represent the model centroids from each bin, and the lines between model points reflect the $2-$D wireframe structure mapped back to high-dimensional space.

```
show_langevitour(
  point_data = df_exe,
  edge_data = trimesh
  )
```



Link plots There are mainly two interactive link plots can be generated.

To support interactive evaluation of how well the *p-D* model captures the structure of the high-dimensional data, we introduce `show_link_plots()`. This visualization combines two complementary views: the nonlinear dimension reduction (NLDR) representation and a dynamic tour of the model ovelaid the data in the high-dimensional space. Both views are interactively linked, enabling users to explore.

Before visualization, the input data must be prepared using the `comb_all_data_model()` function. This function combines the high-dimensional data (`highd_data`), the NLDR data (`nldr_data`), and the bin-averaged high-dimensional model representation (`model_highd`) aligned to the $2-$D bin layout (`model_2d`):

This combined dataset includes both the original observations and the bin-level model averages, labeled with a `type` variable for distinguishing between them.

```
df_exe <- comb_all_data_model(
  highd_data = scurve,
  nldr_data = scurve_umap,
  model_highd = df_bin,
  model_2d = df_bin_centroids
  )
```

The function `show_link_plots()` generates two side-by-side, interactively linked plots; a $2-$D NLDR representation, and a dynamic projection tour in the original high-dimensional space (using the `langevitour` package), displaying both the data and the model. The function takes the output from `comb_all_data_model()` (`point_data`) and `edge_data` which defines connections between neighboring bins.

These two views are linked using `crosstalk`, allowing interactive selection of points in the NLDR plot to highlight corresponding structures in the `langevitour` output.

```
show_link_plots(
  point_data = df_exe,
  edge_data = trimesh
  )
```

`show_error_link_plots()` helps to see investigate whether the model fits the points everywhere or fits better in some places, or simply mismatches the pattern.

Before visualization, the input data must be prepared using the `comb_all_data_model_error()` function. The function requires several arguments: points data which contain high-dimensional data (`highd_data`), NLDR data (`nldr_data`), high-dimensional model data (`model_highd`), $2-D$ model data (`model_2d`), and model error (`error_data`).

This combined dataset includes both the original observations and the bin-level model averages, labeled with a `type` variable for distinguishing between them.

```
df_exe <- comb_all_data_model_error(
  highd_data = scurve,
  nldr_data = scurve_umap,
  model_highd = df_bin,
  model_2d = df_bin_centroids,
  error_data = model_error
  )

df_exe
```

```
#> # A tibble: 5,122 x 12
#>         x1    x2    x3         x4         x5       x6        x7 type   emb1  emb2
#>      <dbl> <dbl> <dbl>      <dbl>      <dbl>    <dbl>     <dbl> <chr> <dbl> <dbl>
#>  1 -0.992   1.91 1.11  -0.000427   0.000624  0.00749  0.00105  model    NA    NA
#>  2 -0.906   1.93 1.41  -0.0000183  0.00331  -0.0204  -0.000363 model    NA    NA
#>  3 -0.680   1.93 1.72  -0.000810  -0.00259  -0.00449  0.00153  model    NA    NA
#>  4  0.461   1.93 1.89  -0.00478    0.00492   0.00835  0.00172  model    NA    NA
#>  5 -0.985   1.75 0.853 -0.00202    0.000397  0.00331  0.000338 model    NA    NA
#>  6 -0.980   1.66 1.17  -0.000374  -0.00154   0.0165   0.000126 model    NA    NA
#>  7 -0.821   1.64 1.56  -0.000459   0.000538 -0.0123   0.000780 model    NA    NA
#>  8 -0.484   1.68 1.87   0.00313    0.00241   0.00823 -0.00117  model    NA    NA
#>  9 -0.0991  1.70 1.99   0.00103    0.00150   0.00877 -0.000193 model    NA    NA
#> 10  0.295   1.74 1.95  -0.00165    0.000459  0.00330  0.000257 model    NA    NA
#> # i 5,112 more rows
#> # i 2 more variables: sqrt_row_wise_total_error <dbl>, density <dbl>
```

The function `show_error_link_plots()` generates three side-by-side, interactively linked plots; a error distribution, a $2-D$ NLDR representation, and a dynamic projection tour in the original high-dimensional space (using the `langevitour` package), displaying both the data and the model. The function takes the output from `comb_all_data_model_error()` (`point_data`) and `edge_data` which defines connections between neighboring bins.

These two views are linked using `crosstalk`, allowing interactive selection of points in the NLDR plot to highlight corresponding structures in the high-dimensional projection. This setup facilitates the diagnosis of local distortion, structural artifacts, and model fit quality.

These three views are linked using `crosstalk`, allowing interactive selection of points in error plot and the NLDR plot to highlight corresponding structures in the `langevitour` output.

```
show_error_link_plots(
  point_data = df_exe,
  edge_data = trimesh
  )
```

4 Application

Single-cell RNA sequencing (scRNA-seq) is a popular and powerful technology that allows you to profile the whole transcriptome of a large number of individual cells (Andrews et al., 2021).

Clustering of single-cell data is used to identify groups of cells with similar expression profiles. NLDR often used to summarise the discovered clusters, and help to understand the results. The purpose of this example is to *illustrate how to use our method to help decide on an appropriate NLDR layout that accurately represents the data*.

Limb muscle cells of mice in Consortium et al. (2018) are examined. There are 1067 single cells, with 14997 gene expressions. Following their pre-processing, tSNE was performed using ten principal

components. Figure 5 (b) is the reproduction of the published plot. The question is whether this accurately represents the cluster structure in the data. Our method can help here, and also help to provide a better $2-$ D layout, as needed.

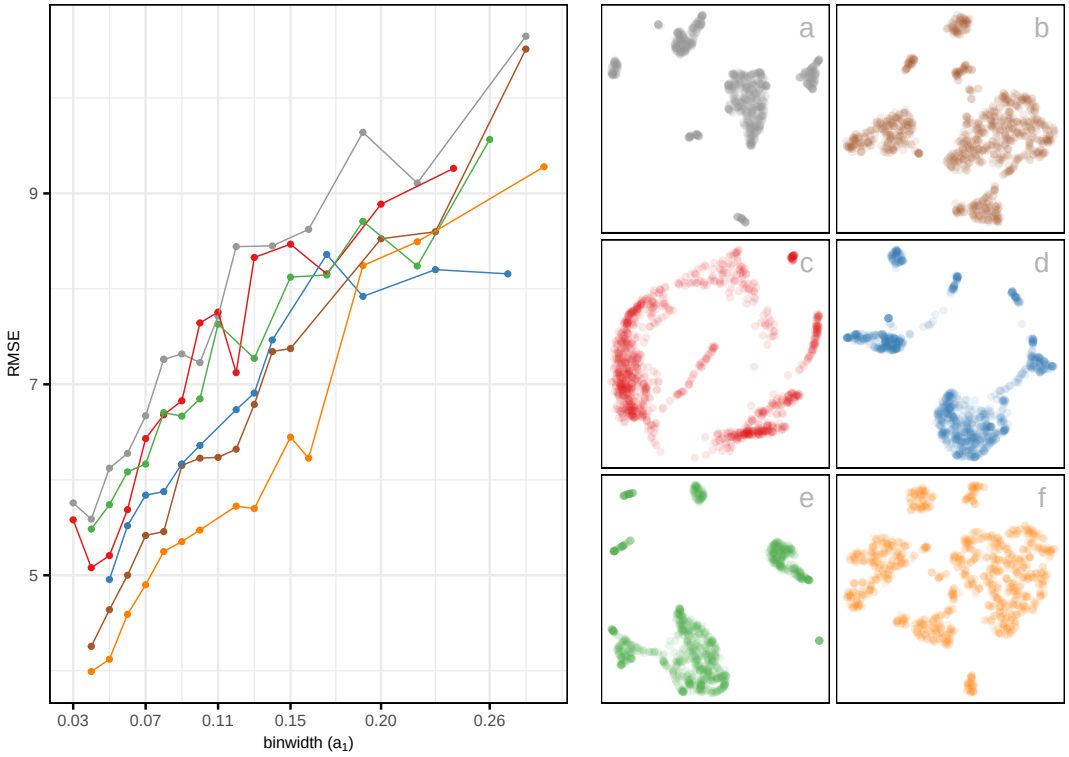


Figure 5: Assessing which of the 6 NLDR layouts on the limb muscle data is the better representation using RMSE for varying binwidth (a_1). Colour used for the lines and points in the left plot and in the scatterplots represents NLDR layout (a-f). Layout d is perform well at large binwidth (where the binwidth is not enough to capture the data struture) and poorly as bin width decreases. Layout f is the best choice.

5 Discussion

This paper presents the R package `quollr` to develop a way to take the fitted model, as represented by the positions of points in 2-D, and turn it into a high-dimensional wireframe to overlay on the data, viewing it with a tour.

The paper includes a clustering example to illustrate how `quollr` is useful to assess which NLDR technique and which (hyper)parameter choice gives the most accurate representation. In addition, how to select parameters for hexagonal binning and fitting model are explained.

Possible future improvements would be...

This new tool provides an effective start point for automatically creating regular hexagons and help to evaluate which NLDR technique and which hyperparameter choice gives the most accurate representation of p-D data.

6 Acknowledgements

This article is created using **knitr** (Xie, 2015) and **rmarkdown** (Xie et al., 2018) in R with the `rjtools::rjournal_article` template. These R packages were used for this work: `cli` (Csárdi, 2025), `dplyr` (Wickham, 2023), `ggplot2` (Wickham, 2016), `interp` (>= 1.1-6) (Gebhardt et al., 2024), `langevitour` (Harrison, 2023), `proxy`(Meyer and Buchta, 2022), `stats` (R Core Team, 2025), `tibble` (Müller and Wickham, 2023), `tidyselect` (Henry and Wickham, 2024), `crosstalk` (Cheng and Sievert, 2023), `plotly` (Sievert, 2020), `kableExtra` (Zhu, 2024), `patchwork` (Pedersen, 2024), and `readr` (Wickham et al., 2024).

The source code for reproducing this paper can be found at: https://github.com/JayaniLakshika/paper-quollr.

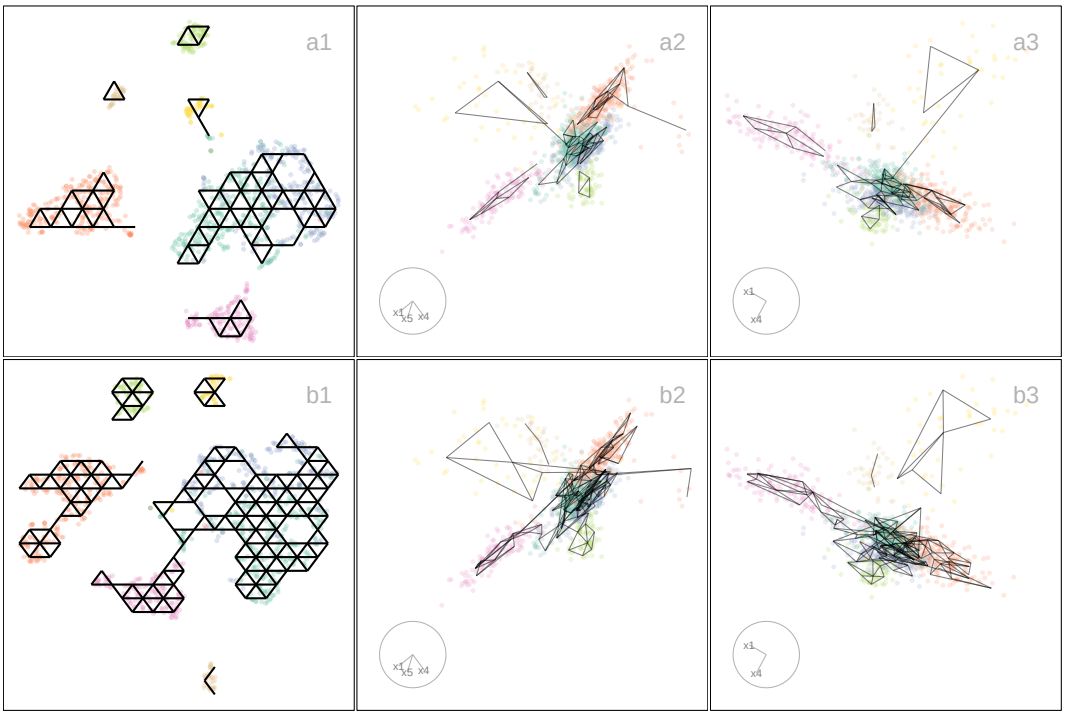


Figure 6: Compare the published $2-D$ layout (Figure reffig:limb-rmse b) and the $2-D$ layout selected (Figure reffig:limb-rmse f) by RMSE plot (Figure reffig:limb-rmse) from the tSNE, UMAP, PHATE, TriMAP, and PaCMAP with different (hyper)parameters. The Limb muscle data ($n = 1067$) has seven close different shaped clusters in 10-D.

Bibliography

E. Amid and M. K. Warmuth. Trimap: Large-scale dimensionality reduction using triplets. *ArXiv*, abs/1910.00204, 2019. URL https://api.semanticscholar.org/CorpusID:203610264. [p1]

T. S. Andrews, V. Y. Kiselev, D. McCarthy, and M. Hemberg. Tutorial: guidelines for the computational analysis of single-cell rna sequencing data. *Nature Protocols*, 16(1):1–9, 2021. doi: 10.1038/s41596-020-00409-w. URL https://doi.org/10.1038/s41596-020-00409-w. [p14]

J. Cheng and C. Sievert. *crosstalk: Inter-Widget Interactivity for HTML Widgets*, 2023. URL https://CRAN.R-project.org/package=crosstalk. R package version 1.2.1. [p15]

T. M. Consortium, O. coordination Schaum Nicholas 1 Karkanias Jim 2 Neff Norma F. 2 May Andrew P. 2 Quake Stephen R. quake@ stanford. edu 2 3 f Wyss-Coray Tony twc@ stanford. edu 4 5 6 g Darmanis Spyros spyros. darmanis@ czbiohub. org 2 h, L. coordination Batson Joshua 2 Botvinnik Olga 2 Chen Michelle B. 3 Chen Steven 2 Green Foad 2 Jones Robert C. 3 Maynard Ashley 2 Penland Lolita 2 Pisco Angela Oliveira 2 Sit Rene V. 2 Stanley Geoffrey M. 3 Webber James T. 2 Zanini Fabio 3, and C. data analysis Batson Joshua 2 Botvinnik Olga 2 Castro Paola 2 Croote Derek 3 Darmanis Spyros 2 DeRisi Joseph L. 2 27 Karkanias Jim 2 Pisco Angela Oliveira 2 Stanley Geoffrey M. 3 Webber James T. 2 Zanini Fabio 3. Single-cell transcriptomics of 20 mouse organs creates a tabula muris. *Nature*, 562(7727):367–372, 2018. [p14]

G. Csárdi. *cli: Helpers for Developing Command Line Interfaces*, 2025. URL https://CRAN.R-project.org/package=cli. R package version 3.6.4. [p15]

A. Gebhardt, R. Bivand, and D. Sinclair. *interp: Interpolation Methods*, 2024. URL https://CRAN.R-project.org/package=interp. R package version 1.1-6. [p15]

P. Harrison. langevitour: Smooth interactive touring of high dimensions, demonstrated with scrna-seq data. *The R Journal*, 15(2):206–219, 2023. doi: 10.32614/RJ-2023-046. [p15]

L. Henry and H. Wickham. *tidyselect: Select from a Set of Strings*, 2024. URL https://CRAN.R-project.org/package=tidyselect. R package version 1.2.1. [p15]

L. V. D. Maaten and G. E. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9: 2579–2605, 2008. [p1]

L. McInnes, J. Healy, N. Saul, and L. Großberger. UMAP: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018. doi: 10.21105/joss.00861. URL https://doi.org/10.21105/joss.00861. [p1]

D. Meyer and C. Buchta. *proxy: Distance and Similarity Measures*, 2022. URL https://CRAN.R-project.org/package=proxy. R package version 0.4-27. [p15]

K. R. Moon, D. van Dijk, Z. Wang, S. A. Gigante, D. B. Burkhardt, W. S. Chen, K. Yim, A. van den Elzen, M. J. Hirn, R. R. Coifman, N. B. Ivanova, G. Wolf, and S. Krishnaswamy. Visualizing structure and transitions in high-dimensional biological data. *Nature Biotechnology*, 37:1482–1492, 2019. [p1]

K. Müller and H. Wickham. *tibble: Simple Data Frames*, 2023. URL https://CRAN.R-project.org/package=tibble. R package version 3.2.1. [p15]

T. L. Pedersen. *patchwork: The Composer of Plots*, 2024. URL https://CRAN.R-project.org/package=patchwork. R package version 1.3.0. [p15]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2025. URL https://www.R-project.org/. [p15]

C. Sievert. *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC, 2020. ISBN 9781138331457. URL https://plotly-r.com. [p15]

Y. Wang, H. Huang, C. Rudin, and Y. Shaposhnik. Understanding how dimension reduction tools work: An empirical approach to deciphering t-sne, umap, trimap, and pacmap for data visualization. *Journal of Machine Learning Research*, 22(201):1–73, 2021. URL http://jmlr.org/papers/v22/20-1061.html. [p1]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org. [p15]

H. Wickham. *conflicted: An Alternative Conflict Resolution Strategy*, 2023. URL https://CRAN.R-project.org/package=conflicted. R package version 1.2.0. [p15]

H. Wickham, J. Hester, and J. Bryan. *readr: Read Rectangular Text Data*, 2024. URL https://CRAN.R-project.org/package=readr. R package version 2.1.5. [p15]

Y. Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL https://yihui.name/knitr/. ISBN 978-1498716963. [p15]

Y. Xie, J. Allaire, and G. Grolemund. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida, 2018. URL https://bookdown.org/yihui/rmarkdown. ISBN 978-1138359338. [p15]

H. Zhu. *kableExtra: Construct Complex Table with 'kable' and Pipe Syntax*, 2024. URL https://CRAN.R-project.org/package=kableExtra. R package version 1.4.0. [p15]

Jayani P. Gamage
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
https://jayanilakshika.netlify.app/
ORCiD: 0000-0002-6265-6481
jayani.piyadigamage@monash.edu

Dianne Cook
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
http://www.dicook.org/
ORCiD: 0000-0002-3813-7155
dicook@monash.edu

Paul Harrison
Monash University
MGBP, BDInstitute, VIC 3800 Australia

ORCiD: 0000-0002-3980-268X
paul.harrison@monash.edu

Michael Lydeamore
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
ORCiD: 0000-0001-6515-827X
michael.lydeamore@monash.edu

Thiyanga S. Talagala
University of Sri Jayewardenepura
Department of Statistics, Gangodawila, Nugegoda 10100 Sri Lanka
https://thiyanga.netlify.app/
ORCiD: 0000-0002-0656-9789
ttalagala@sjp.ac.lk