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) (**?**), uniform manifold approximation and projection (UMAP) (**?**), potential of heat-diffusion for affinity-based trajectory embedding (PHATE) algorithm (**?**), large-scale dimensionality reduction Using triplets (TriMAP) (**?**), and pairwise controlled manifold approximation (PaCMAP) (**?**), can create hugely 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 2-*D* wireframe model representation, that can be lifted and displayed in the high-dimensional (*p-D*) space (Figure 1).

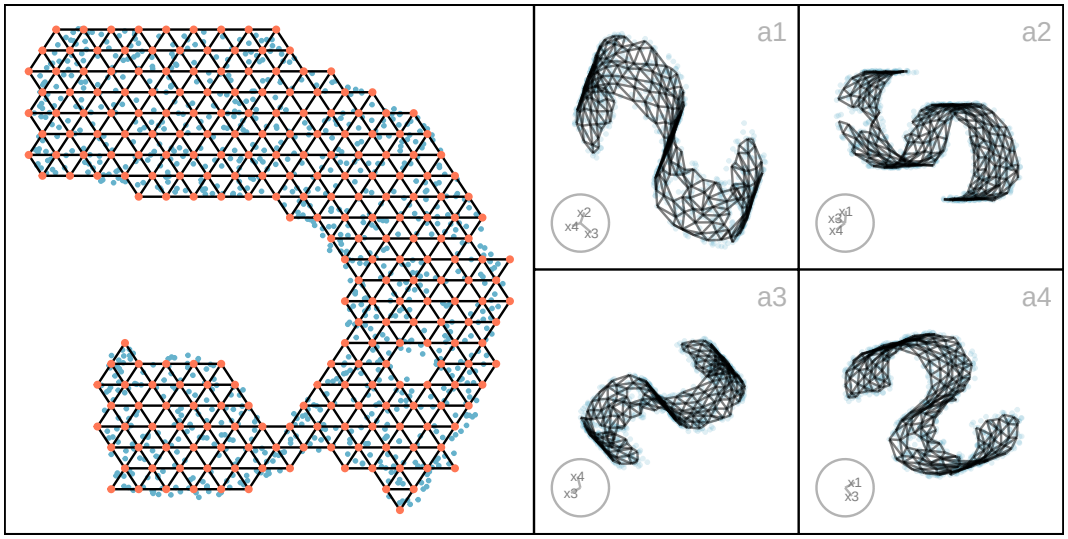

Figure 1: Wireframe model representation of the NLDR layout, lifted and displayed in high-dimensional space. The left panel shows the NLDR layout with a triangular mesh overlay, forming the wireframe structure. This mesh can be lifted into higher dimensions and projected to examine how the geometric structure of the data is preserved. Panels (a1–a4) display different 2-*D*projections of the lifted wireframe, where the underlying curved sheet structure of the data is more clearly visible. The triangulated mesh highlights how local neighborhoods in the layout correspond to relationships in the high-dimensional space, enabling diagnostics of distortion and preservation across dimensions.

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 **?**.

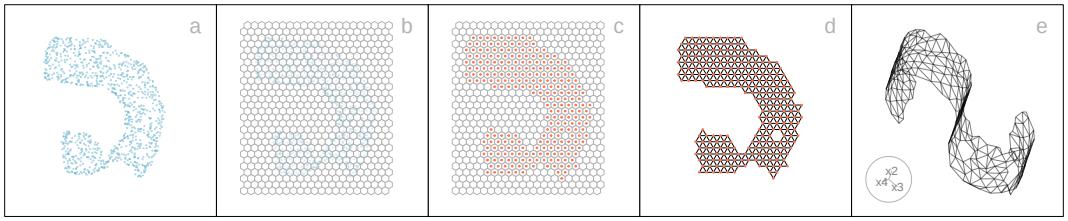

Figure 2: Key steps for constructing the model on the UMAP layout: (a) NLDR data, (b) hexagon bins, (c) bin centroids, (d) triangulated centroids, and (e) lifting the model into high dimensions. The 'Scurve' data is shown.

3 Implementation

Installation

The development version can be installed from GitHub:

```
pak::pak("JayaniLakshika/quollr")
```

Usage

The following demonstration of the package's functionality assumes `quollr` has been loaded. The built-in data sets `scurve` and `scurve_umap` are also used throughout.

`scurve` is a 7-D simulated dataset. It is constructed by simulating 1000 observations from

$$\begin{aligned} \theta &\sim U(-3\pi/2, 3\pi/2), \\ X_1 &= \sin(\theta), \\ X_2 &\sim U(0,2), \text{adding thickness to the S,} \\ X_3 &= \text{sign}(\theta) \times (\cos(\theta) - 1). \end{aligned}$$

The remaining variables X_4, X_5, X_6, and X_7 are included as low-magnitude noise dimensions, generated from uniform distributions with narrow ranges around zero: $X_4, X_5 \sim U(-0.02, 0.02)$, $X_6 \sim U(-0.1, 0.1)$, and $X_7 \sim U(-0.01, 0.01)$. These small variances ensure that the added dimensions contribute noise without obscuring the main geometric structure. `scurve_umap` is the UMAP 2-D embedding for `scurve` data with `n_neighbors`$=$ 46 and `min_dist`$=$ 0.9. Each data set contains a unique ID column that maps the 7-D point in `scurve` to the corresponding 2-D `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 (`b1`), the buffer amount as a proportion of data (`q`), and benchmark value to extract high density hexagons (`hd_thresh`). The function returns an object of class `highd_vis_model` containing the scaled NLDR object (`nldr_scaled_obj`) with three elements: the first is the scaled NLDR data (`scaled_nldr`), and the second and third are the limits of the original

NLDR data (`lim1` and `lim2`); 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,
  b1 = 21,
  q = 0.1,
  hd_thresh = 0)
```

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) removing low-density hexagons.

Scaling the data The algorithm starts by scaling the NLDR data to to the range $[0, 1] \times [0, y_{2,max}]$, where $y_{2,max} = r_2/r_1$ is the ratio of ranges of embedding components. The output includes the scaled NLDR data (`scaled_nldr`) along with the original limits of the embeddings (`lim1`, `lim2`).

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

scurve_umap_obj
```

```
> $scaled_nldr
> # A tibble: 1,000 x 3
>     emb1  emb2    ID
>    <dbl> <dbl> <int>
>  1 0.277 0.913     1
>  2 0.697 0.538     2
>  3 0.779 0.399     3
>  4 0.173 0.953     4
>  5 0.218 0.983     5
>  6 0.593 1.05      6
>  7 0.180 0.210     7
>  8 0.976 0.571     8
>  9 0.803 0.829     9
> 10 0.932 0.410    10
> # i 990 more rows
>
> $lim1
> [1] -9.146398  8.552211
>
> $lim2
> [1] -10.36686  10.10691
```

Computing hexagon grid configuration The function `calc_bins_y()` determines the configuration of the hexagonal grid by computing the number of bins along the y-axis (`b2`), the hexagon width (`a1`), and height (`a2`). This function accepts (1) an object (`nldr_scaled_obj`) containing three elements: the first is the scaled NLDR data (`scaled_nldr`), and the second and third are the limits of the original NLDR data (`lim1` and `lim2`); (2) the number of bins along the x-axis (`b1`), and (3) the buffer amount as a proportion (`q`). The buffer ensures that the grid fully covers the data space by extending one hexagon width (a_1) and height (a_2) beyond the observed data in all directions. By default, $q = 0.1$, but it must be set to a value smaller than the minimum data value to avoid exceeding the data range.

```
bin_configs <- calc_bins_y(
  nldr_scaled_obj = scurve_umap_obj,
  b1 = 25,
  q = 0.1)

bin_configs
```

```
> $b2
> [1] 33
>
> $a1
> [1] 0.04952516
>
> $a2
> [1] 0.04289005
```

Binning the data Points are allocated to bins based on the nearest centroid of the hexagonal bins. The hexagonal binning algorithm can be executed using the `hex_binning()` function, or its individual components can be run separately for added flexibility. While running the process step by step would involve generating centroids, constructing hexagon coordinates, assigning points to bins, standardizing counts, and mapping the data back to hexagons, the `hex_binning()` function automates this entire workflow. The parameters used within `hex_binning()` include an object (`nldr_scaled_obj`) containing three elements: the first is the scaled NLDR data (`scaled_nldr`), and the second and third are the limits of the original NLDR data (`lim1` and `lim2`); the number of bins along the x-axis (`b1`), 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 (`b`), the number of non-empty bins (`m`), and the points within each hexagon (`pts_bins`).

```
hb_obj <- hex_binning(
  nldr_scaled_obj = scurve_umap_obj,
  b1 = 25,
  q = 0.1)
```

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 hexagons along y-axis (`b2`) and the width of each hexagon (`a1`) given the specified number of bins along the x-axis (`b1`) 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 (`vs`) is given 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 (`h`) along with the corresponding x and y centroid coordinates (`c_x`, `c_y`), which define the layout of the hexagonal grid over the 2-D space.

```
all_centroids_df <- gen_centroids(
  nldr_scaled_obj = scurve_umap_obj,
  b1 = 25,
  q = 0.1
  )

all_centroids_df
```

```
> # A tibble: 825 x 3
>        h       c_x    c_y
>    <int>     <dbl>  <dbl>
>  1     1 -0.1      -0.116
>  2     2 -0.0505   -0.116
>  3     3 -0.000950 -0.116
>  4     4  0.0486   -0.116
>  5     5  0.0981   -0.116
>  6     6  0.148    -0.116
```

```
>  7     7  0.197    -0.116
>  8     8  0.247    -0.116
>  9     9  0.296    -0.116
> 10    10  0.346    -0.116
> # i 815 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 (`hs`).

Two derived constants are calculated to define the relative distances to the vertices. The horizontal and vertical offset is defined as $dx = a_1/2$, and $dy = a_1/\sqrt{3}$ repectively. A vertical spacing factor $vf = a_1/2\sqrt{3}$ 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 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 as the centroid. These vertices are then combined into a tibble that records the polygon ID (`h`) and the respective x (`x`) and y (`y`) coordinates for all hexagon corners.

To reduce computational overhead, the geometry calculations are implemented in C++ using `gen_hex_coord_cpp()`, which returns a `tibble` of vertex coordinates.

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

head(all_hex_coord, 5)
```

```
>   h           x           y
> 1 1 -0.10000000 -0.08708678
> 2 1 -0.12476258 -0.10138346
> 3 1 -0.12476258 -0.12997683
> 4 1 -0.10000000 -0.14427351
> 5 1 -0.07523742 -0.12997683
```

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 (`h`) to the point.

The result is a `tibble` of the scaled 2-*D* embedding with an additional `h` column, indicating the hexagonal bin to which each point belongs.

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

head(umap_hex_id, 5)
```

```
> # A tibble: 5 x 4
>    emb1  emb2    ID     h
>   <dbl> <dbl> <int> <int>
> 1 0.277 0.913     1   609
> 2 0.697 0.538     2   392
> 3 0.779 0.399     3   319
> 4 0.173 0.953     4   631
> 5 0.218 0.983     5   657
```

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

The function begins by grouping the data by hexagon ID (`h`) and counting the number of NLDR points falling within each bin. These raw counts are stored as `n_h`. 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 standardized bin counts, `w_h`, ranging from 0 to 1.

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

head(std_df, 5)
```

```
> # A tibble: 5 x 3
>       h   n_h   w_h
>   <int> <int> <dbl>
> 1    82     1 0.001
> 2    83     4 0.004
> 3    84     1 0.001
> 4    85     3 0.003
> 5    86     1 0.001
```

Mapping the points to their corresponding hexagonal bins The `group_hex_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 `h`, which represents the hexagon ID associated with each point in the 2-*D* layout. Within each group, it collects the `ID`s into a list, resulting in a summary where each row corresponds to a single hexagon. The resulting column, `pts_list`, contains all point identifiers associated with that hexagon.

```
pts_df <- group_hex_pts(
  scaled_nldr_hexid = umap_hex_id
  )

head(pts_df, 5)
```

```
> # A tibble: 5 x 2
>       h pts_list
>   <int> <list>
> 1    82 <int [1]>
> 2    83 <int [4]>
> 3    84 <int [1]>
> 4    85 <int [3]>
> 5    86 <int [1]>
```

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

This function begins by arranging the `counts_data` by `h` to ensure consistent ordering. It then performs a full join with `centroids_data`, aligning hexagon IDs (`h`) between the two datasets to incorporate both hexagonal bin centroids (`h`) and count metrics. After merging, the function handles missing values in the count columns: any `NA` values in `w_h` or `n_h` 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 contains the full set of hexagon centroids along with associated bin counts (`n_h`) and standardized counts (`w_h`).

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

head(df_bin_centroids, 5)
```

```
>   h           c_x        c_y n_h w_h
> 1 1 -0.1000000000 -0.1156801   0   0
> 2 2 -0.0504748374 -0.1156801   0   0
> 3 3 -0.0009496749 -0.1156801   0   0
> 4 4  0.0485754877 -0.1156801   0   0
> 5 5  0.0981006503 -0.1156801   0   0
```

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()`.

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

head(trimesh, 5)
```

```
> # A tibble: 5 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      -0.116  -0.0505 -0.116           0        0
> 2    26    27 -0.0752   -0.0728 -0.0257 -0.0728          0        0
> 3     3    27 -0.000950 -0.116  -0.0257 -0.0728          0        0
> 4    26    52 -0.0752   -0.0728 -0.0505 -0.0299          0        0
> 5    27    28 -0.0257   -0.0728  0.0238 -0.0728          0        0
```

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)

head(trimesh, 5)
```

```
> # A tibble: 5 x 10
>    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      -0.116  -0.0505 -0.116           0        0
> 2    26    27 -0.0752   -0.0728 -0.0257 -0.0728          0        0
> 3     3    27 -0.000950 -0.116  -0.0257 -0.0728          0        0
> 4    26    52 -0.0752   -0.0728 -0.0505 -0.0299          0        0
> 5    27    28 -0.0257   -0.0728  0.0238 -0.0728          0        0
> # i 2 more variables: from_reindexed <int>, to_reindexed <int>
```

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 relies on a helper, `compute_mean_density_hex()`, which iterates over all hexagons and computes the average density across neighbors based on their hexagon ID (`h`) and a defined number of bins along the x-axis (`b1`). The hexagonal layout assumes a fixed grid structure, so neighbor IDs are computed by positional offsets.

```
find_low_dens_hex(
  model_2d = df_bin_centroids,
  b1 = 25,
  md_thresh = 0.05
)
```

```
>   [1]  82  83  84  85  86  93  94 107 108 109 110 111 112 113 118 119 120 131
>  [19] 132 133 134 135 136 137 138 141 142 143 144 145 156 157 158 159 160 161
>  [37] 162 163 164 165 166 167 168 169 170 171 181 182 183 184 185 186 187 188
>  [55] 189 190 191 192 193 194 195 196 206 207 209 210 211 212 213 214 215 216
>  [73] 217 218 219 220 221 222 231 232 233 234 235 236 237 239 240 241 242 243
>  [91] 244 245 246 256 257 258 259 260 261 266 267 268 269 271 272 281 282 283
> [109] 284 285 291 292 293 294 295 296 297 307 317 318 319 320 321 322 323 341
> [127] 342 343 344 345 346 347 367 368 369 371 372 373 391 392 393 394 395 396
> [145] 397 398 416 417 418 419 420 421 422 423 440 441 442 443 444 446 447 465
> [163] 466 467 468 469 470 471 472 489 490 491 492 493 494 495 496 497 509 510
> [181] 511 512 513 514 515 516 517 518 519 520 521 522 528 530 531 532 533 534
> [199] 535 536 537 538 539 540 541 542 543 544 545 546 553 554 555 556 557 558
> [217] 559 560 561 562 563 564 565 566 567 568 569 570 578 579 580 581 582 583
> [235] 584 585 586 587 588 589 590 591 592 593 594 595 603 604 605 606 607 608
> [253] 609 610 611 612 613 614 615 616 617 618 619 620 628 629 630 631 633 634
> [271] 635 636 637 638 640 641 642 643 644 653 654 655 656 657 658 659 660 661
> [289] 662 663 664 665 666 667 678 679 680 681 682 683 685 686 687 688 689 690
> [307] 691 704 705 706 707 708 709 710 711 712 713 714 729 730 731 732 733 734
> [325] 735 736 737 738 755 757
```

For simplicity, we remove low-density hexagons using a threshold of 0.

```
df_bin_centroids <- df_bin_centroids |>
  dplyr::filter(n_h > 0)

trimesh <- trimesh |>
  dplyr::filter(from_count > 0,
                to_count > 0)

trimesh <- update_trimesh_index(trimesh)
```

Lifting the model into high dimensions

The final step involves lifting the fitted 2-*D* model into *p*-*D*. This is done by modelling a point in *p*-*D* as the *p*-*D* mean of data points in the 2-*D* centroid. This 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
)

head(df_bin, 5)
```

```
> # A tibble: 5 x 8
>       h    x1     x2    x3       x4       x5      x6       x7
>   <int> <dbl>  <dbl> <dbl>    <dbl>    <dbl>   <dbl>    <dbl>
> 1    82 0.888 0.102   1.46  0.0145  -0.0118   0.0303 -0.00707
> 2    83 0.744 0.123   1.66 -0.00950  0.00499 -0.0208 -0.00425
> 3    84 0.591 0.0447  1.81 -0.0168   0.0173  -0.0485 -0.00112
> 4    85 0.371 0.0937  1.93 -0.00109  0.00840  0.0268 -0.00286
> 5    86 0.255 0.0447  1.97 -0.00862 -0.00733 -0.0826  0.00229
```

Prediction

The `predict_emb()` function is used to predict a point in a 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 the 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.

To accelerate this process, the nearest-neighbor search is implemented in C++ using `Rcpp` via the internal function `compute_highd_dist()`.

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

head(predict_data, 5)
```

```
> # A tibble: 5 x 4
>   pred_emb_1 pred_emb_2    ID pred_h
>        <dbl>      <dbl> <int>  <int>
> 1      0.296      0.914     1    609
> 2      0.717      0.528     2    392
> 3      0.791      0.399     3    319
> 4      0.172      0.957     4    631
> 5      0.197      0.999     5    657
```

It is worth noting that while `predict_emb()` provides a general approach that works across methods, some NLDR techniques have their own built-in prediction mechanisms. For example, UMAP (**?**) supports direct prediction of embeddings for new data once a model is fitted.

Compute residuals and Root Mean Square Error (RMSE)

Residuals and root mean squared error (RMSE) are used as goodness of fit metrics for the model. These metrics can be computed using the `glance()` function, which provides a tidy output for evaluation.

The function requires both the fitted model object returned by `fit_highd_model()` and *p*-*D* data to begin. The *p*-*D* model output (`model_highd`) is first renamed to avoid naming conflicts during subsequent data joins. It then uses the `predict_emb()` function to assign each point in the *p*-*D* dataset to a corresponding hexagon bin in the 2-*D* model, producing a prediction data frame that contains both the predicted bin assignment (`pred_h`) and the original observation `ID`.

The function joins this prediction output with both the *p*-*D* model and the *p*-*D* data (to retrieve true coordinates). It then calculates squared differences between the original and predicted *p*-*D* 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). The computation of total absolute error and RMSE is performed in C++ for efficiency using the internal `compute_errors()` function.

```
glance(
  model_object = scurve_model_obj,
  highd_data = scurve
  )
```

```
> # A tibble: 1 x 2
>   Error  RMSE
>   <dbl> <dbl>
> 1  196. 0.116
```

Furthermore, `augment()` requires both the fitted model object returned by `fit_highd_model()` and *p-D* data to begin. It extends the fitted model by adding prediction results and error diagnostics to the original *p-D* data.

The function starts with the same process as is used in the `glance()` function to produce a predicted point in *p-D* for each point in the *p-D* dataset.

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 data, 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(
  model_object = scurve_model_obj,
  highd_data = scurve
  )

head(model_error, 5)
```

```
> # A tibble: 5 x 32
>      ID      x1    x2       x3       x4       x5      x6       x7 pred_h
>   <int>   <dbl> <dbl>    <dbl>    <dbl>    <dbl>   <dbl>    <dbl>  <int>
> 1     1 -0.120  0.819 -1.99     0.0114   0.00351  0.0334  0.00638    427
> 2     2 -0.0492 0.166  0.00121  0.0115  -0.0166  -0.0297  0.00509    287
> 3     3 -0.774  0.651  0.367   -0.0172   0.00600  0.0211  0.00303    226
> 4     4 -0.606  0.952 -1.80     0.0157  -0.00978 -0.0590 -0.00754    446
> 5     5 -0.478  1.10  -1.88    -0.00423  0.00495 -0.0482 -0.00982    468
> # 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>,
> #   abs_error_x3 <dbl>, abs_error_x4 <dbl>, abs_error_x5 <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)
    )
```

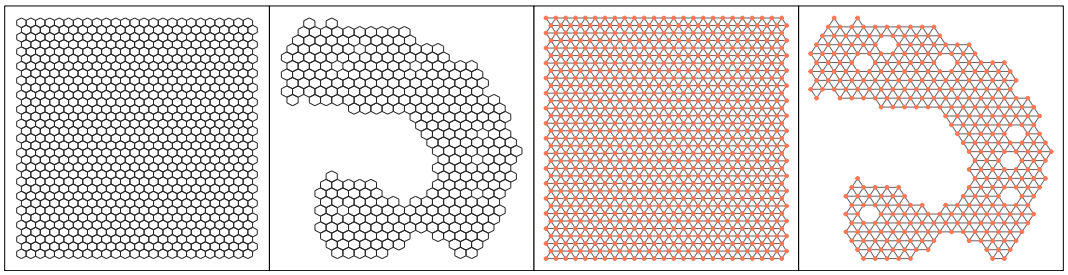

Figure 3: The outputs of 'geom_hexgrid' and 'geom_trimesh' include: (a) a complete hexagonal grid, (b) a hexagonal grid that corresponds with the data, (c) a full grid based on centroid triangulation, and (d) a centroid triangulation grid that aligns with the data.

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

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
  )
```

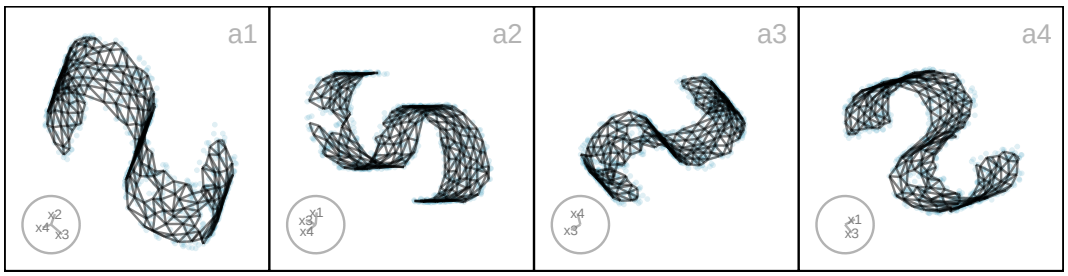

Figure 4: 2-*D* projections of the lifted high-dimensional wireframe model from the 'Scurve' UMAP layout. Each panel (a1–a4) shows the model (black) overlaid on 'Scurve' data (in purple) in different projections. These views illustrate how the lifted wireframe model captures the structure of the 'Scurve' data. The two twists visible in the UMAP layout can also be seen in the lifted model.

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.

```
nldrdt_link <- show_link_plots(
  point_data = df_exe,
  edge_data = trimesh,
  point_colour = clr_choice
  )

class(nldrdt_link) <- c(class(nldrdt_link), "htmlwidget")

nldrdt_link
```

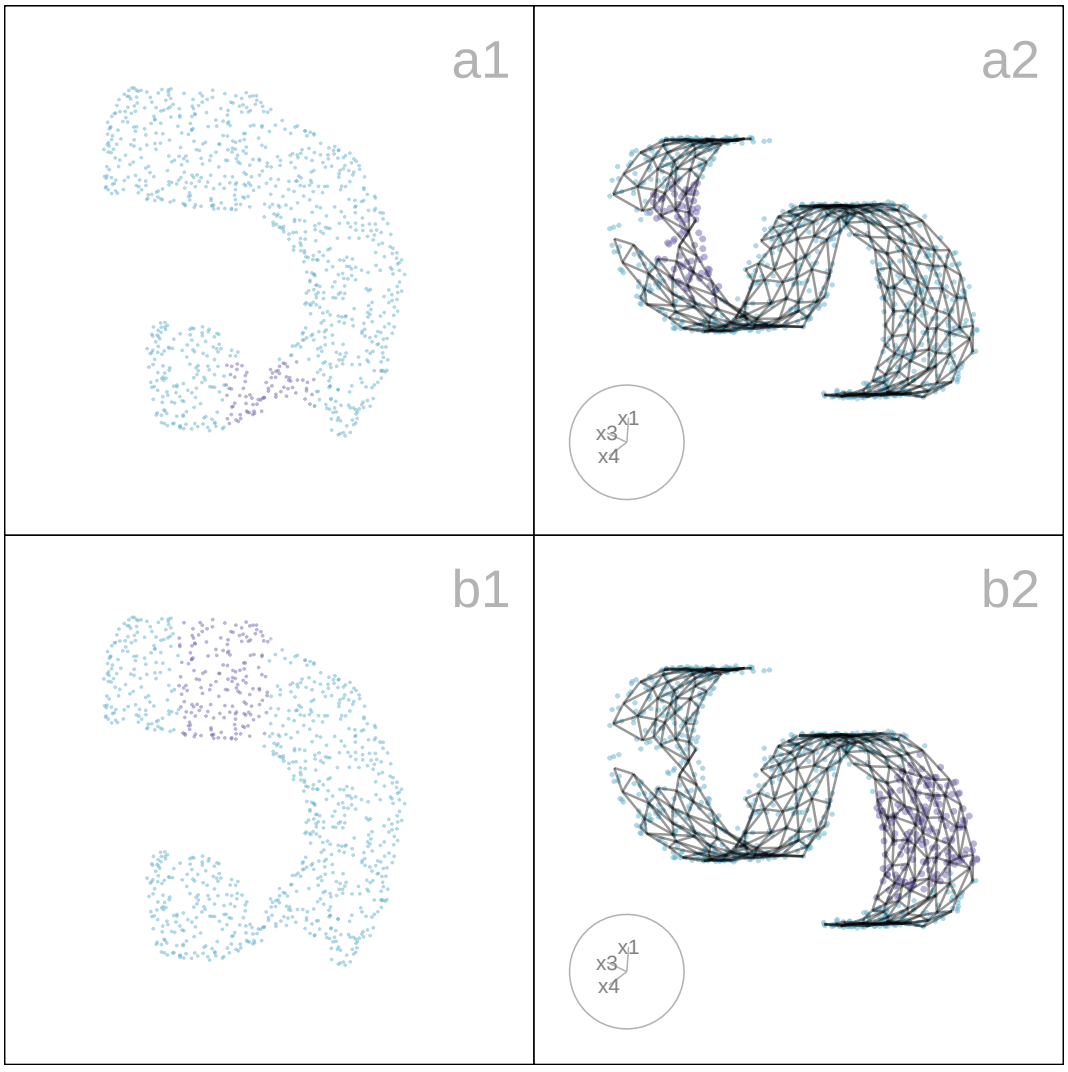

Figure 5: Exploring the correspondence between UMAP layout and 'Scurve' structure in 7-*D*. Two sets of plots are linked: UMAP layout (a1, b1) and projection of 7-*D* model and data (a2, b2). The purple points indicate the selected subsets, which differ between rows. In (a1), the lower bridge of the 'Scurve' is highlighted, which corresponds in (a2) to points spanning across both arms of the high-dimensional structure. In (b1), a different region near the upper arm of the 'Scurve' is selected, and in (b2) these points map onto one side of the curved manifold in 7-*D* projection. While the UMAP layout suggests distinct local clusters, the linked tour views reveal how these selections trace continuous structures in the 7-*D* space, highlighting distortions introduced by UMAP.

`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
  )
```

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.

```
errornldrdt_link <- show_error_link_plots(
  point_data = df_exe,
  edge_data = trimesh,
  point_colour = clr_choice
)

class(errornldrdt_link) <- c(class(errornldrdt_link), "htmlwidget")

errornldrdt_link
```

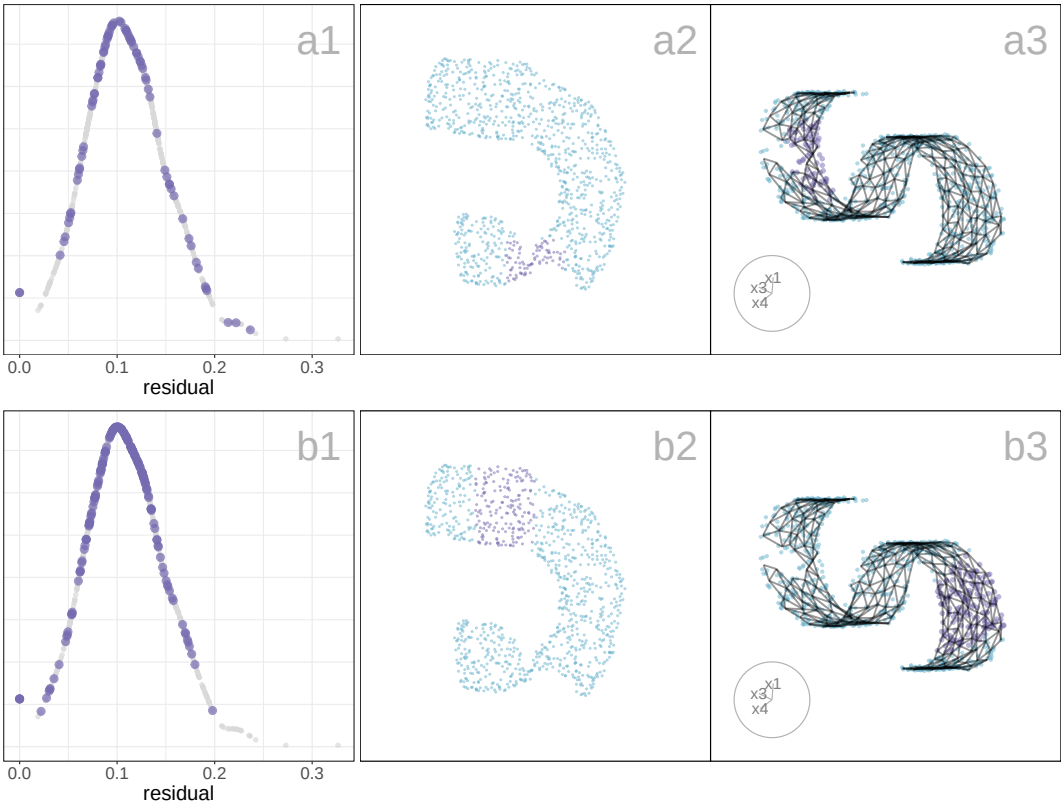

Figure 6: Exploring residuals in relation to UMAP layouts using a 7-*D* 'Scurve' model. Three views are linked: distribution of residuals (a1, b1), UMAP layout (a2, b2), and projection of the 7-*D* model with data (a3, b3). The purple points highlight selected subsets of the data, which differ across rows. In the top row (a1–a3), points with higher residuals (a1) are selected, corresponding to the sparse bridging region in the UMAP layout (a2) and the less dense end of the 'Scurve' in the high-dimensional projection (a3). In the bottom row (b1–b3), points with lower residuals (b1) are highlighted, which map to one side of the dense region in the NLDR layout (b2) and to a thicker band of the 'Scurve' in the projection (b3). This comparison illustrates how residuals can help diagnose distortions in UMAP, with high-residual points often concentrated in sparse or stretched regions of the structure.

4 Computational efficiency and optimization

Several core computations within `quollr` are optimized using compiled C++ code via the `Rcpp` and `RcppArmadillo` packages. While the user interacts with high-level R functions, performance-critical steps such as nearest-neighbor searches (`compute_highd_dist()`), error metrics (`compute_errors()`), 2-*D* distance calculations (`calc_2d_dist_cpp()`), and generation of hexagon coordinates (`gen_hex_coord_cpp()`) are handled internally in C++. This design provides significant speedups when analyzing large datasets while maintaining a user-friendly R interface. These C++ functions are not exported but are bundled within the package and fully accessible for inspection in the source code.

5 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 (**?**).

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 **?** are examined. There are 1067 single cells, with 14997 gene expressions. Following their pre-processing, different NLDR methods were performed using ten principal components. Figure 7 (b) is the reproduction of the published plot. The question is whether this accurately represents the cluster structure in the data. Our method help to provide a better 2-*D* layout.

```
design <- gen_design(n_right = 6, ncol_right = 2)

plot_rmse_layouts(plots = list(error_plot_limb, nldr1,
                               nldr2, nldr3, nldr4,
                               nldr5, nldr6), design = design)
```

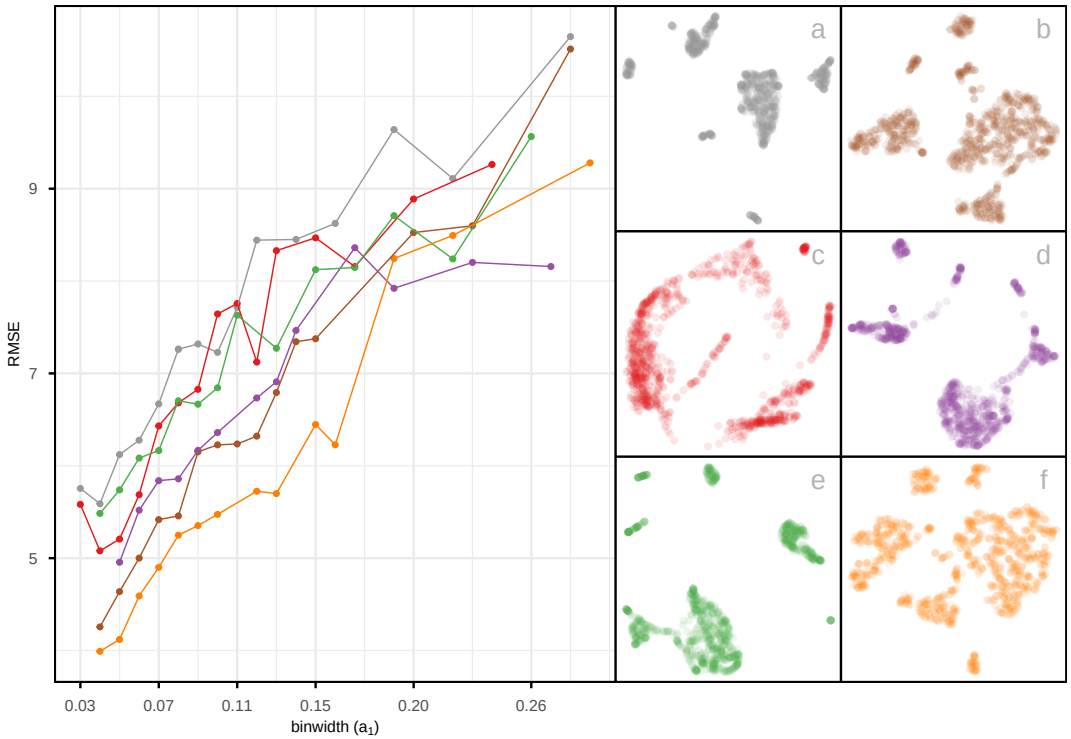

Figure 7: Assessing which of the 6 NLDR layouts on the limb muscle data is the better representation using RMSE for varying binwidth (a_1). Colour used for the lines and points in the left plot and in the scatterplots represents NLDR layout (a-f). Layout d is perform well at large binwidth (where the binwidth is not enough to capture the data struture) and poorly as bin width decreases. Layout f is the best choice.

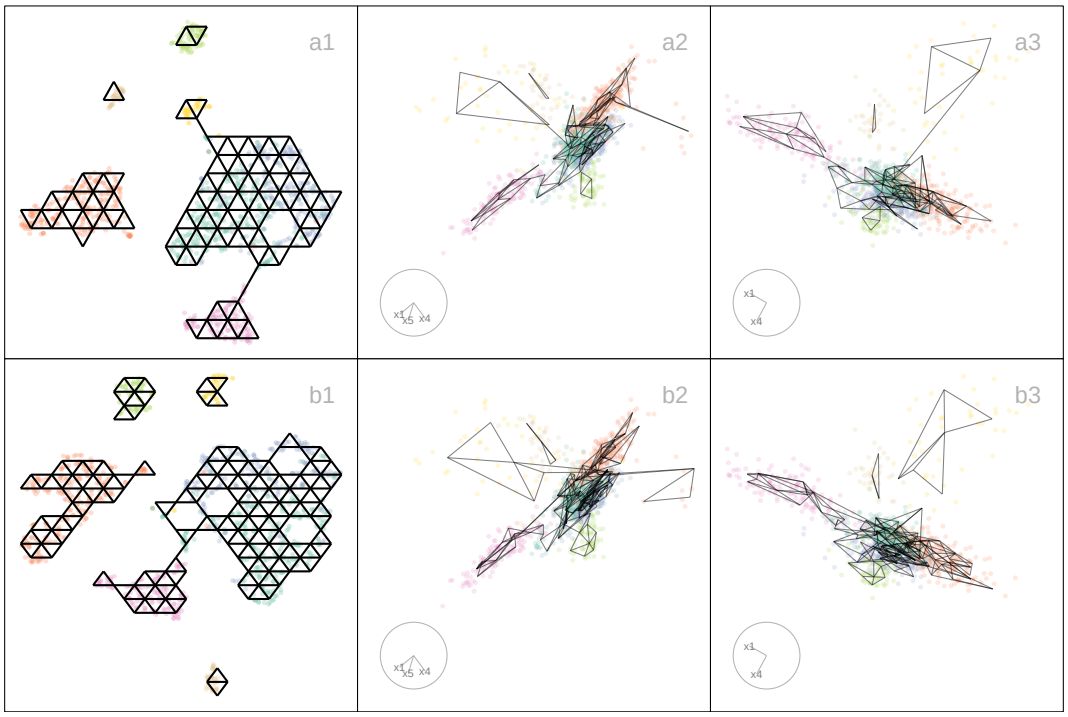

Figure 8: Compare the published $2-$ D layout (Figure 7b) and the $2-$ D layout selected (Figure 7f) by RMSE plot (Figure 7) from the tSNE, UMAP, PHATE, TriMAP, and PaCMAP with different hyper-parameters. The Limb muscle data ($n = 1067$) has seven close different shaped clusters in 10-D.

6 Discussion

The `quollr` package introduces a new framework for interpreting NLDR outputs by fitting a geometric wireframe model in 2-D and lifting it into high-dimensional space. This lifted model provides a direct way to assess how well a 2-D layout, produced by methods such as tSNE, UMAP, PHATE, TriMAP, or PaCMAP, preserves the structure of the original high-dimensional data. The approach offers both numerical and visual diagnostics to support the selection of NLDR methods and tuning hyper-parameters that produce the most accurate 2-D representations.

In contrast to the common practice of visually inspecting scatterplots for clusters or patterns, `quollr` provides a quantitative route for evaluation. It enables the computation of RMSE and residuals between the original high-dimensional data and the lifted model, offering interpretable diagnostics. These diagnostics are complemented by interactive linked plots and high-dimensional dynamic visualizations using the `langevitour` package, allowing users to inspect where the model fits well and where it does not.

To support efficient computation, particularly for large-scale datasets, several core functions in `quollr` are implemented in C++ using `Rcpp` and `RcppArmadillo`. These include functions for computing Euclidean distances in high-dimensional and 2-D space, identifying nearest centroids, calculating residual errors, and generating polygonal coordinates of hexagons. For instance, `compute_highd_dist()` accelerates nearest neighbor lookup in high-dimensional space, `compute_errors()` calculates RMSE and total absolute error efficiently, and `calc_2d_dist_cpp()` speeds up distance calculations in 2-D. Additionally, `gen_hex_coord_cpp()` constructs the coordinates for hexagonal bins based on their centroids with minimal overhead. These optimizations result in substantial performance gains compared to native R implementations, making the package responsive even when used in interactive contexts or on large datasets such as single-cell transcriptomic profiles.

The modular structure of the package is designed to support both flexibility and reproducibility. Users can access individual functions to control each step of the pipeline such as scaling, binning, and triangulation or use the main function `fit_highd_model()` for end-to-end model construction. The diagnostics can be used not only to compare NLDR methods but also to tune binning parameters, assess layout stability, and detect local distortions in the embedding.

There are several avenues for future development. While hexagonal binning provides a regular structure conducive to modeling, alternative spatial discretizations (e.g., adaptive binning or density-aware tessellations) could be explored to better capture varying data densities. Expanding support for additional distance metrics in the lifting and prediction steps may improve performance across

Table 1: Summary of notation for describing new methodology.

Notation	Description	Variable
\$n, p, k\$	number of observations, variables, embedding dimension, respectively	NA
$\mathbfit{X}, \mathbfit{x}$	p-dimensional data (population, sample)	NA
$\mathbfit{y}$	k-dimensional layout	NA
P	orthonormal basis, generating a $d\text{-}dimensional$ linear projection of p-dimensional data	NA
T	true model	NA
g	functional mapping from \pD{} to \kD{}, especially as prescribed by NLDR	NA
$\mathbfit{\theta}$	(Hyper-) parameters for NLDR method	NA
r	ranges of the embedding components	NA
$C^{(j)}$	j-dimensional bin centers	(c_x, c_y)
(b_1, b_2)	number of bins in each direction	(b1, b2) = bins
(a_1, a_2)	binwidths, distance between centroids in each direction	a1, a2
$(s_1, \ s_2)$	starting coordinates of the hexagonal grid	start_point
q	buffer to ensure hexgrid covers data, proportion of data range, 0-1	q
m	number of non-empty bins	m
b	number of hexagons in the grid	b
h	hexagonal id	NA
l	side length	NA
A	area	NA
n_h	bin count	NA
w_h	standardised bin counts	NA
NA	threshold to extract high density hexagons	hd_thresh

different domains. Additionally, statistical inference tools could be introduced to assess the stability and robustness of the fitted model, which would enhance interpretability and confidence in the outcomes.

7 Acknowledgements

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

This article is created using **knitr** (**?**) and **rmarkdown** (**?**) in R with the `rjtools::rjournal_article` template. These R packages were used for this work: `cli` (**?**), `dplyr` (**?**), `ggplot2` (**?**), `interp` (>= 1.1-6) (**?**), `langevitour` (**?**), `proxy`(**?**), `stats` (**?**), `tibble` (**?**), `tidyselect` (**?**), `crosstalk` (**?**), `plotly` (**?**), `kableExtra` (**?**), `patchwork` (**?**), and `readr` (**?**).

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