Juscan.jl

lihuax(LiZehua)

April 11, 2025

Contents

Part I

Home

Juscan.jl is a Julia implementation of Scanpy, tailored for single-cell data analysis. Currently in its development (dev) version, the library aims to deliver high-performance, flexible, and extensible tools for preprocessing, dimensionality reduction, clustering, and visualization of single-cell datasets.

Chapter 1

Features

- **Single-Cell Data Analysis** Provides functionalities for data normalization, log transformation, dimensionality reduction (e.g., PCA), and graph-based clustering (e.g., Leiden algorithm).
- **High Performance Computing** Leverages Julia's computational strengths to efficiently handle large-scale single-cell datasets.
- **Modular and Extensible** Designed with a modular architecture, allowing users to easily customize and extend functionality to suit diverse analytical requirements.

Chapter 2

Development Status

Juscan.jl is under active development. Some features are still being refined and may change over time. We welcome feedback, bug reports, and contributions from the community.

Chapter 3

Installation

3.1 Installing the Development Version via Julia Package Manager

You can install the development version using Julia's package manager:

```
using Pkg
Pkg.add(url="https://github.com/zehua0417/Juscan.jl")
```

3.2 Cloning the Repository for Local Development

```
git clone https://github.com/zehua0417/Juscan.jl.git
cd Juscan.jl
julia --project -e 'using Pkg; Pkg.instantiate()'
```

Chapter 4

Quick Start Example

Here's a simple example demonstrating basic single-cell data preprocessing and analysis with Juscan.jl:

```
using Juscan

# Load the dataset
adata = Juscan.readh5ad("data/data.h5ad")

# Quality control
Juscan.Pp.filter_cells!(adata, min_genes=100, max_genes=8000, max_counts=140000)
Juscan.Pp.filter_genes!(adata, min_cells=3)
adata.var.mt = startswith.(adata.var_names, "MT-")
Juscan.Pp.calculate_qc_metrics!(adata, qc_vars=["mt"])

# Normalization and log transformation
adata.layers["normalized"] = deepcopy(adata.X)
Juscan.Pp.normalize_total!(adata, target_sum=1000, layer="normalized")
Juscan.Tl.logp1_transform!(adata, layer="normalized", key_added="normalized_logp1")

# Highly variable genes and PCA
Juscan.Tl.highly_variable_genes!(adata, n_top_genes=2000, layer="normalized_logp1")
Juscan.Tl.pca!(adata; layer="normalized_logp1", key_added="pca", n_pcs=15)

# Clustering and UMAP visualization
Juscan.Tl.clustering!(adata, method="km", use_pca=15, cluster_K=4)
Juscan.Tl.umap!(adata; key_added="umap", n_pcs=15, use_pca="pca")
Juscan.Pl.plot_umap(adata, color_by="clusters_0.5")
```

Chapter 5

Contributing

We welcome contributions of all kinds:

- **Bug Reports & Suggestions**: Please open an issue on GitHub.
- **Code or Documentation Updates**.
- **Discussions**: Share your feedback and experiences to help improve the project.

Chapter 6

License

Juscan.jl is licensed under the MIT License.

Chapter 7

Contact

If you have any questions or suggestions, please open an issue on GitHub or contact us at my e-mail.

Chapter 8

Acknowledgments

We would like to extend our sincere gratitude to the open source community for their continuous support and inspiration. In particular, we thank:

- Scanpy for its groundbreaking approach to single-cell analysis,
- AnnData for providing a robust data structure for annotated data,
- Muon.jl for its innovative multi-omic analysis tools, and
- Automatic Single-cell Toolbox (ASCT)An automated single-cell data analysis toolbox inspired by Seurat v4 in R.
- scVI.jl for offering valuable insights into variational inference methods.

Their contributions and ideas have been instrumental in shaping the development and direction of Juscan.jl.

Thank you for your interest in Juscan.jl. We hope it will empower your single-cell data analysis projects!

Part II

Tutorial

Chapter 9

Quick Start

9.1 Quick Start Guide for Juscan.jl

This guide walks you through a basic single-cell RNA-seq analysis pipeline using `Juscan.jl`. The dataset used in this example comes from doi:10.6084/m9.figshare.22716739.v1.

Prerequisites

Make sure you have installed `Juscan.jl`, `Muon.jl`, and their dependencies:

```
using Pkg
Pkg.activate(".")
Pkg.add(url="https://github.com/zehua0417/Juscan.jl")
```

Load Data

```
using Juscan
using Muon
using DataFrames

adata = Juscan.readh5ad("data/data.h5ad")
```

Quality Control

```
Juscan.Pp.filter_cells!(adata, min_genes=100)
Juscan.Pp.filter_genes!(adata, min_cells=3)
Juscan.Pp.filter_cells!(adata, max_genes=8000)
Juscan.Pp.filter_cells!(adata, max_counts=140000)

adata.var.mt = startswith.(adata.var_names, "MT-")
adata.var.ribo = startswith.(adata.var_names, "RPS") .| startswith.(adata.var_names, "RPL")
adata.var.hb = occursin.(r"^HB[^P]", adata.var_names)

Juscan.Pp.calculate_qc_metrics!(adata, qc_vars=["mt", "ribo", "hb"])
```

Visualize QC Metrics

```
Juscan.Pl.violin(
  adata,
  ["pct_counts_mt", "n_genes_by_counts", "total_counts"];
  width=300,
  height=800,
  fill_alpha=0.7,
  savefig="/home/lihuax/Pictures/Juscan/qc_violin.png",
)

Juscan.Pl.scatter(
  adata,
  "total_counts",
  "n_genes_by_counts",
  color_key="pct_counts_mt",
  width=800,
  height=800,
  colormap_name="magma",
  savefig="/home/lihuax/Pictures/Juscan/qc_scatter.png",
)
```

Normalization

```
adata.layers["normalized"] = deepcopy(adata.X)
Juscan.Pp.normalize_total!(adata, target_sum=1000, layer="normalized")
Juscan.Tl.logp1_transform!(adata, layer="normalized", key_added="normalized_logp1")
adata.layers["normalized_logp1"] = Float64.(adata.layers["normalized_logp1"])
```

Highly Variable Genes

```
Juscan.Tl.highly_variable_genes!(adata, n_top_genes=2000, layer="normalized_logp1")
Juscan.Pl.hvg_scatter(adata, savefig="/home/lihuax/Pictures/Juscan/hvg_scatter.png")
```

Dimensionality Reduction

```
Juscan.Tl.pca!(adata; key_added="pca", n_pcs=15)
Juscan.Tl.pca!(adata; layer="normalized_logp1", key_added="pca", n_pcs=15)
Juscan.Pl.plot_variance_ratio(adata, savefig="/home/lihuax/Pictures/Juscan/variance_ratio.png")

Juscan.Tl.subset_to_hvg!(adata; layer="normalized_logp1", n_top_genes=2000)
Juscan.Pp.filter_cells!(adata, min_counts=3000)
Juscan.Pp.filter_cells!(adata, max_counts=10000)
```

□ Clustering & UMAP

```
Juscan.Tl.clustering!(adata, method="km", use_pca=15, cluster_K=4, dist="Euclidean")
Juscan.Tl.umap!(
  adata;
  key_added="umap",
  use_pca="pca",
  n_pcs=15,
  min_dist=0.5,
  n_neighbors=50,
)
fig = Juscan.Pl.plot_umap(adata, color_by="clusters_0.5")
```

This pipeline demonstrates the essential steps of scRNA-seq analysis using `Juscan.jl`. For more detailed usage, please refer to the official documentation.

Happy analyzing! □□

Part III

API

Chapter 10

Anndata Utils

The `AnnData` struct is imported from `Muon.jl`. The package provides read and write functions for `.h5ad` and `.h5mu` files, the typical H5-based format for storing Python `anndata` objects. The `AnnData` object stores datasets together with metadata, such as information on the variables (genes in scRNA-seq data) and observations (cells), as well as different kinds of annotations and transformations of the original count matrix, such as PCA or UMAP embeddings, or graphs of observations or variables.

For details on the Julia implementation in `Muon.jl`, see the documentation.

For more details on the original Python implementation of the `anndata` object, see the documentation and preprint.

- `Juscan.get_celltypes`
- `Juscan.insert_obs!`
- `Juscan.insert_var!`
- `Juscan.subset_adata`
- `Juscan.subset_adata!`

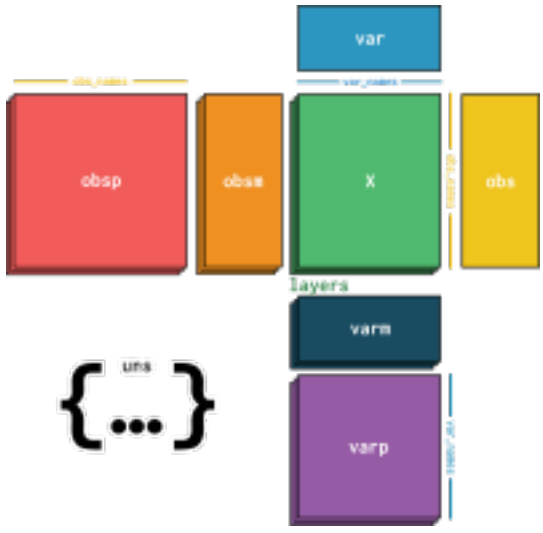

Figure 10.1: anndata

10.1 celltypes

`Juscan.get_celltypes` – Function.

```
get_celltypes(a::AnnData)
```

Tries to infer the cell types of cells in an `AnnData` object.

Returns a vector of cell type names if the cell types are stored in `adata.obs["cell_type"]`, `adata.obs["celltype"]`, `adata.obs["celltypes"]`, or `adata.obs["cell_types"]`. Otherwise, returns `nothing`.

source

10.2 subset adata

`Juscan.subset_adata` – Function.

```
subset_adata(adata::AnnData, subset_inds::Tuple, dims::Symbol=:both)
```

Subset an `AnnData` object by indices passed as a tuple of vectors of integers, UnitRanges, or vectors of Booleans. If `dims` is set to :both, the first element of `subset_inds` is used to subset cells and the second element is used to subset genes.

Arguments

- `adata`: `AnnData` object to subset
- `subset_inds`: tuple of vectors of integers, UnitRanges, or vectors of Booleans
- `dims`: dimension to subset, either `:cells`, `:genes`, or `:both`

Returns

- a copy of the `AnnData` object with the subsetted data

source

```
subset_adata(adata::AnnData, subset_inds::Union{Int, Vector{Int}, UnitRange, Vector{Bool}},
↪  dims::Symbol)
```

Subset an `AnnData` object by indices passed as a vector of integers or booleans or as a UnitRange. The `dims` argument can be set to either `:cells` or `:genes` to specify which dimension to subset.

Arguments

- `adata`: `AnnData` object to subset
- `subset_inds`: vector of integers or booleans or UnitRange
- `dims`: dimension to subset, either `:cells` or `:genes`

Returns

- a copy of the AnnData object with the subsetted data

source

Juscan.subset_adata! – Function.

```
subset_adata!(adata::AnnData, subset_inds, dims::Symbol)
```

In-place version of subset_adata, see ?subset_adata for more details.

For subset_inds, either a tuple of vectors or ranges can be passed with dims set to :both, for subsetting both cells and genes, or a single vector or range can be passed with dims set to either :cells or :genes.

Arguments

- adata: AnnData object to subset
- subset_inds: tuple of vectors of integers, UnitRanges, or vectors of Booleans or vector of integers or booleans or UnitRange
- dims: dimension to subset, either :cells, :genes, or :both

Returns

- the AnnData object with the subsetted data

source

10.3 insert

Juscan.insert_obs! – Function.

```
insert_obs!(adata::AnnData, obs::DataFrame)
```

Insert new columns into the obs DataFrame of an AnnData object.

Arguments

- adata: The AnnData object to modify.
- obs: A DataFrame containing the new columns to be inserted. The number of rows in obs must match the number of cells in adata.

Returns

- nothing: Modifies the adata.obs DataFrame in place.

Notes

- This function assumes that the number of rows in obs matches the number of cells in adata. If they do not match, an error will be thrown.

- This function modifies the adata.obs DataFrame directly. If you want to keep the original adata unchanged, make a copy before calling this function.

source

Juscan.insert_var! – Function.

```
insert_var!(adata::AnnData, var::DataFrame)
```

Insert new columns into the var DataFrame of an AnnData object.

Arguments

- adata: The AnnData object to modify.
- var: A DataFrame containing the new columns to be inserted. The number of rows in var must match the number of genes in adata.

Returns

- nothing: Modifies the adata.var DataFrame in place.

Notes

- This function assumes that the number of rows in var matches the number of genes in adata. If they do not match, an error will be thrown.
- This function modifies the adata.var DataFrame directly. If you want to keep the original adata unchanged, make a copy before calling this function.

source

Chapter 11

Preprocessing

Preprocessing is a crucial step in single-cell and multi-omics data analysis. This module provides functions for quality control, filtering, and normalization to ensure that datasets are clean and ready for downstream analyses.

Quality control (QC) metrics help identify low-quality cells and genes, such as those with extremely high or low counts, high dropout rates, or disproportionate expression levels. By computing these metrics, users can apply appropriate thresholds to retain only reliable data.

Filtering enables users to remove unwanted cells or genes based on QC criteria, ensuring that low-quality features do not affect downstream analyses.

Normalization methods adjust for differences in sequencing depth and technical variation, allowing meaningful comparisons across cells and conditions.

These preprocessing functions ensure that data is well-structured and suitable for clustering, dimensionality reduction, and other analytical tasks.

- `Juscan.Pp.calculate_qc_metrics`
- `Juscan.Pp.calculate_qc_metrics!`
- `Juscan.Pp.describe_obs`
- `Juscan.Pp.describe_obs!`
- `Juscan.Pp.describe_var`
- `Juscan.Pp.describe_var!`
- `Juscan.Pp.filter_cells`
- `Juscan.Pp.filter_cells!`
- `Juscan.Pp.filter_genes`
- `Juscan.Pp.filter_genes!`
- `Juscan.Pp.normalize_total`
- `Juscan.Pp.normalize_total!`

11.1 quality control metrics

`Juscan.Pp.calculate_qc_metrics` – Function.

```
calculate_qc_metrics!(adata; expr_type="counts", var_type="genes", qc_vars=String[],
                      percent_top=[50, 100, 200, 500], layer=nothing, use_raw=false,
                      use_log1p=true, parallel=nothing) -> Tuple{DataFrame, DataFrame}
```

Calculate quality control metrics and return.

Arguments

- `adata`: `Muon.AnnData` Annotated data matrix containing single-cell or multi-omics data.
- `expr_type`: `String` (default: `"counts"`) The type of expression data to use, such as raw counts or normalized values.
- `var_type`: `String` (default: `"genes"`) Specifies the variable type, e.g., `"genes"` for gene expression.
- `qc_vars`: `Union{Vector{String}, String}` (default: `String[]`) A list of quality control variables to consider. If a single string is provided, it will be converted to a vector.
- `percent_top`: `Union{Vector{Int}, Nothing}` (default: `[50, 100, 200, 500]`) A list of top feature (e.g., gene) counts to compute the percentage of total counts for each cell.
- `layer`: `Union{String, Nothing}` (default: `nothing`) Specifies the layer in `adata` to use. If `nothing`, the default matrix is used.
- `use_raw`: `Bool` (default: `false`) Whether to use the raw counts matrix (`adata.raw`) instead of the processed data.
- `use_log1p`: `Bool` (default: `true`) Whether to apply `log1p` transformation to the data before computing quality control metrics.
- `parallel`: `Union{Bool, Nothing}` (default: `nothing`) **(Deprecated)** This argument is ignored but retained for backward compatibility.

Returns

- A tuple of two `DataFrames.DataFrame`:
  1. The first `DataFrame` contains per-cell quality control metrics.
  2. The second `DataFrame` contains per-gene quality control metrics.

source

`Juscan.Pp.calculate_qc_metrics!` – Function.

```
calculate_qc_metrics!(adata; expr_type="counts", var_type="genes", qc_vars=String[],
                      percent_top=[50, 100, 200, 500], layer=nothing, use_raw=false,
                      use_log1p=true, parallel=nothing) -> Nothing
```

Calculate and store quality control (QC) metrics directly in the `obs` and `var` attributes of `adata`.

Arguments

- adata: Muon.AnnData Annotated data matrix containing single-cell or multi-omics data. The QC metrics will be stored in adata.obs (per-cell) and adata.var (per-gene).
- expr_type: String (default: "counts") The type of expression data to use, such as raw counts or normalized values.
- var_type: String (default: "genes") Specifies the variable type, e.g., "genes" for gene expression.
- qc_vars: Union{Vector{String}, String} (default: String[]) A list of quality control variables to consider. If a single string is provided, it will be converted to a vector.
- percent_top: Union{Vector{Int}, Nothing} (default: [50, 100, 200, 500]) A list of top feature (e.g., gene) counts to compute the percentage of total counts for each cell.
- layer: Union{String, Nothing} (default: nothing) Specifies the layer in adata to use. If nothing, the default matrix is used.
- use_raw: Bool (default: false) Whether to use the raw counts matrix (adata.raw) instead of the processed data.
- use_log1p: Bool (default: true) Whether to apply log1p transformation to the data before computing quality control metrics.
- parallel: Union{Bool, Nothing} (default: nothing) **(Deprecated)** This argument is ignored but retained for backward compatibility.

Returns

- Nothing: This function modifies adata in place by adding QC metrics to adata.obs and adata.var.

Notes

- Unlike calculate_qc_metrics, which returns QC metrics as DataFrame objects, this function directly updates adata without returning any values.

source

sub functions

Juscan.Pp.describe_obs – Function.

```
describe_obs(adata; expr_type="counts", var_type="genes", qc_vars=String[],
             percent_top=[50, 100, 200, 500], layer=nothing, use_raw=false,
             use_log1p=true, X=nothing, parallel=nothing) -> DataFrames.DataFrame
```

Compute per-cell quality control (QC) metrics from the expression matrix and return them as a DataFrame.

Arguments

- adata: Muon.AnnData Annotated data matrix containing single-cell or multi-omics data.
- expr_type: String (default: "counts") The type of expression data to use, such as raw counts or normalized values.
- var_type: String (default: "genes") Specifies the variable type, e.g., "genes" for gene expression.
- qc_vars: Union{Vector{String}, String} (default: String[]) A list of quality control variables to consider. If a single string is provided, it will be converted to a vector.

- percent_top: Union{Vector{Int}, Nothing} (default: [50, 100, 200, 500]) A list of top feature (e.g., gene) counts to compute the percentage of total counts for each cell.
- layer: Union{String, Nothing} (default: nothing) Specifies the layer in adata to use. If nothing, the default matrix is used.
- use_raw: Bool (default: false) Whether to use the raw counts matrix (adata.raw) instead of the processed data.
- use_log1p: Bool (default: true) Whether to apply log1p transformation to the data before computing quality control metrics.
- X: Union{Nothing, SparseMatrixCSC} (default: nothing) The expression matrix to use. If nothing, the matrix is determined by _choose_mtx_rep.
- parallel: Union{Bool, Nothing} (default: nothing) **(Deprecated)** This argument is ignored but retained for backward compatibility.

Returns

- DataFrames.DataFrame: A DataFrame containing per-cell QC metrics, including:
  - "n_$(var_type)_by_$(expr_type)": Number of detected features per cell.
  - "log1p_n_$(var_type)_by_$(expr_type)": Log-transformed number of features per cell.
  - "total_$(expr_type)": Total expression count per cell.
  - "log1p_total_$(expr_type)": Log-transformed total expression count per cell.
  - "pct_$(expr_type)_in_top_N_$(var_type)": Percentage of expression from the top N features.
  - "total_$(expr_type)_$(qc_var)": Total expression for each QC variable per cell.
  - "log1p_total_$(expr_type)_$(qc_var)": Log-transformed total expression for each QC variable per cell.
  - "pct_$(expr_type)_$(qc_var)": Percentage of total expression contributed by each QC variable.

Notes

- This function computes per-cell QC metrics based on expression data. For per-gene metrics, see describe_var.

source

Juscan.Pp.describe_obs! – Function.

```
describe_obs!(adata; expr_type="counts", var_type="genes", qc_vars=String[],
              percent_top=[50, 100, 200, 500], layer=nothing, use_raw=false,
              use_log1p=true, X=nothing, parallel=nothing) -> Nothing
```

Compute per-cell quality control (QC) metrics and store them in adata.obs.

Arguments

- adata: Muon.AnnData Annotated data matrix containing single-cell or multi-omics data.
- expr_type: String (default: "counts") The type of expression data to use, such as raw counts or normalized values.

- var_type: String (default: "genes") Specifies the variable type, e.g., "genes" for gene expression.
- qc_vars: Union{Vector{String}, String} (default: String[]) A list of quality control variables to consider. If a single string is provided, it will be converted to a vector.
- percent_top: Union{Vector{Int}, Nothing} (default: [50, 100, 200, 500]) A list of top feature (e.g., gene) counts to compute the percentage of total counts for each cell.
- layer: Union{String, Nothing} (default: nothing) Specifies the layer in adata to use. If nothing, the default matrix is used.
- use_raw: Bool (default: false) Whether to use the raw counts matrix (adata.raw) instead of the processed data.
- use_log1p: Bool (default: true) Whether to apply log1p transformation to the data before computing quality control metrics.
- X: Union{Nothing, SparseMatrixCSC} (default: nothing) The expression matrix to use. If nothing, the matrix is determined by _choose_mtx_rep.
- parallel: Union{Bool, Nothing} (default: nothing) **(Deprecated)** This argument is ignored but retained for backward compatibility.

Returns

- Nothing: This function modifies adata in place by adding per-cell QC metrics to adata.obs.

Notes

- This function is similar to describe_obs but modifies adata directly. It computes QC metrics using describe_obs and stores the results in adata.obs.
  - For per-gene QC metrics, see describe_var!.

source

Juscan.Pp.describe_var – Function.

```
describe_var(adata; expr_type="counts", var_type="genes", layer=nothing,
             use_raw=false, use_log1p=true, X=nothing) -> DataFrame
```

Compute per-gene quality control (QC) metrics and return as a DataFrame.

Arguments

- adata: Muon.AnnData Annotated data matrix containing single-cell or multi-omics data.
- expr_type: String (default: "counts") The type of expression data to use, such as raw counts or normalized values.
- var_type: String (default: "genes") Specifies the variable type, e.g., "genes" for gene expression.
- layer: Union{String, Nothing} (default: nothing) Specifies the layer in adata to use. If nothing, the default matrix is used.
- use_raw: Bool (default: false) Whether to use the raw counts matrix (adata.raw) instead of the processed data.
- use_log1p: Bool (default: true) Whether to apply log1p transformation to the data before computing quality control metrics.

- X: `Union{Nothing, SparseMatrixCSC}` (default: `nothing`) The expression matrix to use. If `nothing`, the matrix is determined by `_choose_mtx_rep`.

Returns

- `DataFrames.DataFrame`: A DataFrame where each row represents a gene (or variable), containing the following QC metrics:
  - `"n_cells_by_<expr_type>"`: Number of cells in which each gene is detected.
  - `"mean_<expr_type>"`: Mean expression level of each gene.
  - `"log1p_mean_<expr_type>"`: Log-transformed mean expression level.
  - `"pct_dropout_by_<expr_type>"`: Percentage of cells in which each gene is not detected.
  - `"total_<expr_type>"`: Total expression level of each gene.
  - `"log1p_total_<expr_type>"`: Log-transformed total expression level.

Notes

- This function calculates quality control metrics per gene (or other variables).
  - To store these metrics directly in `adata.var`, use `describe_var!`.
  - To compute per-cell QC metrics, see `describe_obs`.

source

`Juscan.Pp.describe_var!` – Function.

```
describe_var!(adata; expr_type="counts", var_type="genes", layer=nothing,
              use_raw=false, use_log1p=true, X=nothing) -> Nothing
```

Compute per-gene quality control (QC) metrics and store them in `adata.var`.

Arguments

- `adata`: `Muon.AnnData` Annotated data matrix containing single-cell or multi-omics data.
- `expr_type`: `String` (default: `"counts"`) The type of expression data to use, such as raw counts or normalized values.
- `var_type`: `String` (default: `"genes"`) Specifies the variable type, e.g., `"genes"` for gene expression.
- `layer`: `Union{String, Nothing}` (default: `nothing`) Specifies the layer in `adata` to use. If `nothing`, the default matrix is used.
- `use_raw`: `Bool` (default: `false`) Whether to use the raw counts matrix (`adata.raw`) instead of the processed data.
- `use_log1p`: `Bool` (default: `true`) Whether to apply `log1p` transformation to the data before computing quality control metrics.
- X: `Union{Nothing, SparseMatrixCSC}` (default: `nothing`) The expression matrix to use. If `nothing`, the matrix is determined by `_choose_mtx_rep`.

Returns

- `Nothing`: This function modifies `adata` in place by adding per-gene QC metrics to `adata.var`.

Notes

- This function is similar to describe_var but modifies adata directly. It computes QC metrics using describe_var and stores the results in adata.var.
  - For per-cell QC metrics, see describe_obs!.

source

11.2 filter cells and genes

filter cells

Juscan.Pp.filter_cells – Function.

```
filter_cells(
  data::Muon.AnnData;
  min_counts=nothing, min_genes=nothing,
  max_counts=nothing, max_genes=nothing,
  copy=false) -> Union{Tuple{BitVector, Vector}, Muon.AnnData}
```

Filters cells based on the given threshold criteria and returns either a subset mask and count vector or a new filtered Muon.AnnData object.

Arguments

- data::Muon.AnnData: The input single-cell data.
- min_counts::Union{Int, Nothing}: Minimum total counts per cell.
- min_genes::Union{Int, Nothing}: Minimum number of genes expressed per cell.
- max_counts::Union{Int, Nothing}: Maximum total counts per cell.
- max_genes::Union{Int, Nothing}: Maximum number of genes expressed per cell.
- copy::Bool: If true, returns a filtered copy; otherwise, returns a mask and count vector.

Returns

- If copy == false, returns a tuple (cell_subset::BitVector, number_per_cell::Vector).
- If copy == true, returns a filtered Muon.AnnData object.

source

```
filter_cells(
  data::AbstractMatrix;
  min_counts=nothing, min_genes=nothing,
  max_counts=nothing, max_genes=nothing
) -> Tuple{BitVector, Vector}
```

Filters cells from a count matrix based on the given threshold criteria.

Arguments

- `data::AbstractMatrix`: Gene expression count matrix with cells as rows.
- `min_counts::Union{Int, Nothing}`: Minimum total counts per cell.
- `min_genes::Union{Int, Nothing}`: Minimum number of genes expressed per cell.
- `max_counts::Union{Int, Nothing}`: Maximum total counts per cell.
- `max_genes::Union{Int, Nothing}`: Maximum number of genes expressed per cell.

Returns

- `cell_subset::BitVector`: A mask indicating cells that pass the filter.
- `number_per_cell::Vector`: A vector of counts or expressed gene numbers per cell.

source

`Juscan.Pp.filter_cells!` – Function.

```
filter_cells!(
  data::Muon.AnnData;
  min_counts=nothing, min_genes=nothing,
  max_counts=nothing, max_genes=nothing
) -> Nothing
```

Filters cells in-place in a `Muon.AnnData` object.

Arguments

- `data::Muon.AnnData`: The input single-cell data.
- `min_counts::Union{Int, Nothing}`: Minimum total counts per cell.
- `min_genes::Union{Int, Nothing}`: Minimum number of genes expressed per cell.
- `max_counts::Union{Int, Nothing}`: Maximum total counts per cell.
- `max_genes::Union{Int, Nothing}`: Maximum number of genes expressed per cell.

Returns

- `Nothing`. The filtering is applied in-place.

source

filter genes

`Juscan.Pp.filter_genes` – Function.

```
filter_genes(
  data::Muon.AnnData;
  min_counts=nothing, min_cells=nothing,
  max_counts=nothing, max_cells=nothing,
  copy=false
) -> Union{Tuple{BitVector, Vector}, Muon.AnnData}
```

Filters genes based on the given threshold criteria and returns either a subset mask and count vector or a new filtered `Muon.AnnData` object.

Arguments

- `data::Muon.AnnData`: The input single-cell data.
- `min_counts::Union{Int, Nothing}`: Minimum total counts per gene.
- `min_cells::Union{Int, Nothing}`: Minimum number of cells expressing the gene.
- `max_counts::Union{Int, Nothing}`: Maximum total counts per gene.
- `max_cells::Union{Int, Nothing}`: Maximum number of cells expressing the gene.
- `copy::Bool`: If `true`, returns a filtered copy; otherwise, returns a mask and count vector.

Returns

- If `copy == false`, returns a tuple `(gene_subset::BitVector, number_per_gene::Vector)`.
- If `copy == true`, returns a filtered `Muon.AnnData` object.

source

```
filter_genes(
  data::AbstractMatrix;
  min_counts=nothing, min_cells=nothing,
  max_counts=nothing, max_cells=nothing
) -> Tuple{BitVector, Vector}
```

Filters genes from a count matrix based on the given threshold criteria.

Arguments

- `data::AbstractMatrix`: Gene expression count matrix with genes as columns.
- `min_counts::Union{Int, Nothing}`: Minimum total counts per gene.
- `min_cells::Union{Int, Nothing}`: Minimum number of cells expressing the gene.
- `max_counts::Union{Int, Nothing}`: Maximum total counts per gene.
- `max_cells::Union{Int, Nothing}`: Maximum number of cells expressing the gene.

Returns

- `gene_subset::BitVector`: A mask indicating genes that pass the filter.
- `number_per_gene::Vector`: A vector of counts or expressed cell numbers per gene.

source

`Juscan.Pp.filter_genes!` – Function.

```
filter_genes!(
  data::Muon.AnnData;
  min_counts=nothing, min_cells=nothing,
  max_counts=nothing, max_cells=nothing
) -> Nothing
```

Filters genes in-place in a `Muon.AnnData` object.

Arguments

- `data::Muon.AnnData`: The input single-cell data.
- `min_counts::Union{Int, Nothing}`: Minimum total counts per gene.
- `min_cells::Union{Int, Nothing}`: Minimum number of cells expressing the gene.
- `max_counts::Union{Int, Nothing}`: Maximum total counts per gene.
- `max_cells::Union{Int, Nothing}`: Maximum number of cells expressing the gene.

Returns

- `Nothing`. The filtering is applied in-place.

source

11.3 normalization

`Juscan.Pp.normalize_total` – Function.

```
normalize_total(
    adata::AnnData;
    target_sum::Union{Real, Nothing}=nothing,
    exclude_highly_expressed::Bool=false,
    max_fraction::Float64=0.05,
    key_added::Union{String, Nothing}=nothing,
    layer::Union{String, Nothing}=nothing,
    layers::Union{String, Vector{String}, Nothing}=nothing,
    layer_norm::Union{String, Nothing}=nothing,
    copy::Bool=false
) -> Union{AnnData, Dict{String, Any}}
```

Normalize total counts per cell to a target sum.

Arguments

- `adata::AnnData`: The single-cell dataset to be normalized.
- `target_sum::Union{Real, Nothing}=nothing`: Target total count per cell. If `nothing`, normalization is performed to the median of counts per cell.
- `exclude_highly_expressed::Bool=false`: If `true`, highly expressed genes are excluded from the normalization factor calculation.
- `max_fraction::Float64=0.05`: A gene is considered highly expressed if it accounts for more than `max_fraction` of the total counts in any cell.
- `key_added::Union{String, Nothing}=nothing`: If provided, the computed normalization factors are stored in `adata.obs[key_added]`.
- `layer::Union{String, Nothing}=nothing`: The layer to normalize. If `nothing`, the main count matrix (`adata.X`) is used.
- `layers::Union{String, Vector{String}, Nothing}=nothing`: Specifies multiple layers to normalize. If `"all"`, all layers in `adata.layers` are normalized.

- `layer_norm::Union{String, Nothing}=nothing`: Defines the normalization reference for additional layers. Can be `"after"`, `"X"`, or `nothing`.
- `copy::Bool=false`: If `true`, returns a copy of `adata` with normalized counts; otherwise, modifies `adata` in place.

Returns

- If `copy=true`, returns a new `AnnData` object with normalized counts.
- If `copy=false`, modifies `adata` in place and returns a dictionary containing:
  - `"X"`: The normalized count matrix.
  - `"norm_factor"`: The computed normalization factors.
  - Additional layers if `layers` is specified.

Notes

- If `exclude_highly_expressed=true`, genes that exceed `max_fraction` in any cell are ignored when computing normalization factors.
- Cells with zero counts will generate a warning.
- The function supports layer-wise normalization if `layers` is specified.

Example

```
adata = AnnData(X)
normalize_total(adata; target_sum=10000)
```

source

`Juscan.Pp.normalize_total!` – Function.

```
normalize_total!(
    adata::AnnData;
    target_sum::Union{Real, Nothing}=nothing,
    exclude_highly_expressed::Bool=false,
    max_fraction::Float64=0.05,
    key_added::Union{String, Nothing}=nothing,
    layer::Union{String, Nothing}=nothing,
    layers::Union{String, Vector{String}, Nothing}=nothing
) -> Nothing
```

Normalize total counts per cell **in-place**.

Arguments

- `adata::AnnData`: The single-cell dataset to be normalized.
- `target_sum::Union{Real, Nothing}=nothing`: Target total count per cell. If `nothing`, normalization is performed to the median of counts per cell.
- `exclude_highly_expressed::Bool=false`: If `true`, highly expressed genes are excluded from the normalization factor calculation.

- `max_fraction::Float64=0.05`: A gene is considered highly expressed if it accounts for more than `max_fraction` of the total counts in any cell.
- `key_added::Union{String, Nothing}=nothing`: If provided, the computed normalization factors are stored in `adata.obs[key_added]`.
- `layer::Union{String, Nothing}=nothing`: The layer to normalize. If `nothing`, the main count matrix (`adata.X`) is used.
- `layers::Union{String, Vector{String}, Nothing}=nothing`: Specifies multiple layers to normalize. If `"all"`, all layers in `adata.layers` are normalized.

Returns

- This function **modifies** `adata` in place and does not return a new object.

Notes

- If `exclude_highly_expressed=true`, genes that exceed `max_fraction` in any cell are ignored when computing normalization factors.
- Cells with zero counts will generate a warning.
- The function supports layer-wise normalization if `layers` is specified.

Example

```
adata = AnnData(X)
normalize_total!(adata; target_sum=10000)
```

source

Chapter 12

Tools

The `tools` module provides essential utility functions to support various analytical workflows in single-cell and multi-omics data processing. These functions serve as building blocks for higher-level analysis by offering efficient, reusable operations that streamline common computational tasks.

Highly Variable Genes (HVG)

The module includes methods for identifying highly variable genes, which are crucial for downstream analyses such as clustering and dimensionality reduction. These functions help select informative genes by computing variability metrics across cells.

Principal Component Analysis (PCA)

PCA is a widely used dimensionality reduction technique that captures the most significant variations in the dataset. The `tools` module provides efficient implementations for computing PCA, enabling users to reduce data complexity while preserving important biological signals.

Clustering

The module supports clustering techniques to group cells based on gene expression patterns. These methods help uncover underlying cellular heterogeneity and identify distinct cell populations, facilitating biological interpretation of single-cell data.

By providing these fundamental tools, the `tools` module enhances data processing workflows, making it easier to perform robust, reproducible analyses across different stages of research.

- `Juscan.Tl.clustering!`
- `Juscan.Tl.highly_variable_genes`
- `Juscan.Tl.highly_variable_genes!`
- `Juscan.Tl.log_transform!`
- `Juscan.Tl.logp1_transform!`
- `Juscan.Tl.pca!`
- `Juscan.Tl.subset_to_hvg!`
- `Juscan.Tl.umap!`

12.1 highly variable genes

`Juscan.Tl.subset_to_hvg!` – Function.

```
subset_to_hvg!(adata::AnnData;
    layer::Union{String,Nothing} = nothing,
    n_top_genes::Int=2000,
    batch_key::Union{String,Nothing} = nothing,
    span::Float64=0.3,
    verbose::Bool=true
)
```

Calculates highly variable genes with `highly_variable_genes!` and subsets the `AnnData` object to the calculated HVGs. For description of input arguments, see `highly_variable_genes!`

Arguments

- `adata`: `AnnData` object

Keyword arguments

- `layer`: optional; which layer to use for calculating the HVGs. Function assumes this is a layer of counts. If `layer` is not provided, `adata.X` is used.
- `n_top_genes`: optional; desired number of highly variable genes. Default: 2000.
- `batch_key`: optional; key where to look for the batch indices in `adata.obs`. If not provided, data is treated as one batch.
- `span`: span to use in the loess fit for the mean-variance local regression. See the Loess.jl docs for details.
- `verbose`: whether or not to print info on current status

Returns

- `adata` object subset to the calculated HVGs, both in the countmatrix/layer data used for HVG calculation and in the `adata.var` dictionary.

source

`Juscan.Tl.highly_variable_genes` – Function.

```
highly_variable_genes(adata::AnnData;
    layer::Union{String,Nothing} = nothing,
    n_top_genes::Int=2000,
    batch_key::Union{String,Nothing} = nothing,
    span::Float64=0.3
    )
```

Computes highly variable genes according to the workflows on `scanpy` and Seurat v3 per batch and returns a dictionary with the information on the joint HVGs. For the in-place version, see `highly_variable_genes!`

More specifically, it is the Julia re-implementation of the corresponding scanpy function For implementation details, please check the scanpy/Seurat documentations or the source code of the lower-level _highly_variable_genes_seurat_v3 function in this package. Results are almost identical to the scanpy function. The differences have been traced back to differences in the local regression for the mean-variance relationship implemented in the Loess.jl package, that differs slightly from the corresponding Python implementation.

Arguments

- adata: AnnData object

Keyword arguments

- layer: optional; which layer to use for calculating the HVGs. Function assumes this is a layer of counts. If layer is not provided, adata.X is used.
- n_top_genes: optional; desired number of highly variable genes. Default: 2000.
- batch_key: optional; key where to look for the batch indices in adata.obs. If not provided, data is treated as one batch.
- span: span to use in the loess fit for the mean-variance local regression. See the Loess.jl docs for details.
- replace_hvgs: whether or not to replace the hvg information if there are already hvgs calculated. If false, the new values are added with a "_1" suffix. Default:true,
- verbose: whether or not to print info on current status

Returns

- a dictionary containing information on the highly variable genes, specifically containing the following keys is added:
  - highly_variable: vector of Bools indicating which genes are highly variable
  - highly_variable_rank: rank of the highly variable genes according to (corrected) variance
  - means: vector with means of each gene
  - variances: vector with variances of each gene
  - variances_norm: normalized variances of each gene
  - highly_variable_nbatches: if there are batches in the dataset, logs the number of batches in which each highly variable gene was actually detected as highly variable.

source

Juscan.Tl.highly_variable_genes! – Function.

```
highly_variable_genes!(adata::AnnData;
    layer::Union{String,Nothing} = nothing,
    n_top_genes::Int=2000,
    batch_key::Union{String,Nothing} = nothing,
    span::Float64=0.3,
    replace_hvgs::Bool=true,
    verbose::Bool=false
    )
```

Computes highly variable genes per batch according to the workflows on scanpy and Seurat v3 in-place. This is the in-place version that adds an dictionary containing information on the highly variable genes directly to the adata.var and returns the modified AnnData object. For details, see the not-in-place version ?highly_variable_genes.

source

12.2 principal component analysis

Juscan.Tl.pca! – Function.

```
pca!(adata::Muon.AnnData; layer="log_transformed", n_pcs=1000, key_added="pca", verbose=true)
```

Performs Principal Component Analysis (PCA) on the specified layer of an AnnData object and stores the result in adata.obsm.

If the specified layer is missing, the function will attempt to log-transform a normalized layer, or normalize and log-transform the raw counts if needed.

Arguments

- adata::Muon.AnnData: The annotated data object on which to perform PCA.

Keyword Arguments

- layer::String = "log_transformed": The data layer to use for PCA. Defaults to "log_transformed".
- n_pcs::Int = 1000: The number of principal components to compute. Automatically clipped to the smallest matrix dimension if too large.
- key_added::String = "pca": The key under which to store the PCA result in adata.obsm.
- verbose::Bool = true: Whether to print progress messages.

Returns

The modified AnnData object with PCA results stored in adata.obsm[key_added].

Notes

- This function performs automatic preprocessing if the requested layer is not present.
- The PCA is computed via SVD on standardized data.

source

Juscan.Tl.umap! – Function.

```
umap!(adata::Muon.AnnData; layer="log_transformed", use_pca=nothing, n_pcs=100,
↪  key_added="umap", verbose=true, kwargs...)
```

Computes a UMAP embedding from the data in the specified layer or PCA representation and stores the result in adata.obsm.

If PCA is requested via use_pca, it will be computed automatically if not already present.

Arguments

- `adata::Muon.AnnData`: The annotated data object on which to compute UMAP.

Keyword Arguments

- `layer::String = "log_transformed"`: The layer to use as input if `use_pca` is not specified.
- `use_pca::Union{String, Nothing} = nothing`: If specified, use this key in `adata.obsm` as PCA input. If missing, it will be computed.
- `n_pcs::Int = 100`: Number of principal components to use if PCA needs to be computed.
- `key_added::String = "umap"`: The key under which to store the UMAP embedding.
- `verbose::Bool = true`: Whether to print progress messages.
- `kwargs...`: Additional keyword arguments passed to `UMAP.UMAP_()`.

Returns

The modified `AnnData` object with UMAP results stored in:

- `adata.obsm[key_added]`: The UMAP embedding.
- `adata.obsm["knns"]`: K-nearest neighbors matrix.
- `adata.obsm["knn_dists"]`: KNN distance matrix.
- `adata.obsp["fuzzy_neighbor_graph"]`: Fuzzy graph representation.

Notes

- Automatically performs normalization and log transformation if necessary.
- Uses the `UMAP.jl` package under the hood.

source

`Juscan.Tl.log_transform!` – Function.

```
log_transform!(adata::Muon.AnnData; layer="normalized", key_added="log_transformed",
↪  verbose=false)
```

Applies a log transformation to the specified data layer of an `AnnData` object and stores the result in `adata.layers`.

If the specified layer is missing, the function defaults to applying a log(1 + x) transformation to `adata.X`.

Arguments

- `adata::Muon.AnnData`: The data object to transform.

Keyword Arguments

- `layer::String = "normalized"`: The layer to transform. Must exist in `adata.layers`.
- `key_added::String = "log_transformed"`: The key to store the result under in `adata.layers`.
- `verbose::Bool = false`: Whether to print messages during the process.

Returns

The modified AnnData object with the log-transformed data added to `adata.layers[key_added]`.

Notes

- The transformation is log(x + ϵ), where ϵ is a small constant to avoid log(0).
- For default fallback behavior, see `logp1_transform!()`.

source

`Juscan.Tl.logp1_transform!` – Function.

```
logp1_transform!(adata::Muon.AnnData; layer=nothing, key_added="log1_transformed",
↪  verbose=false)
```

Applies a log(1 + x) transformation to the specified layer or the main data matrix `adata.X` in an AnnData object. The result is stored in `adata.layers[key_added]`.

Arguments

- `adata::Muon.AnnData`: The annotated data object to transform.

Keyword Arguments

- `layer::Union{String, Nothing} = nothing`: The name of the data layer to transform. If `nothing` or the layer is missing, uses `adata.X`.
- `key_added::AbstractString = "log1_transformed"`: The name under which to store the transformed result in `adata.layers`.
- `verbose::Bool = false`: Whether to print transformation messages.

Returns

The modified AnnData object with the log(1 + x) transformed data stored in `adata.layers[key_added]`.

Notes

- This transformation is commonly used to stabilize variance and reduce the effect of outliers.
- Compared to `log_transform!`, this version adds 1 to the data before taking the logarithm, making it more robust to zero entries.

source

12.3 clustering

`Juscan.Tl.clustering!` – Function.

```
clustering!(data::Muon.AnnData; kwargs...)
```

Perform clustering on a `Muon.AnnData` object and store the results in place.

This function supports modularity-based graph clustering (`"mc"`) and K-means clustering (`"km"`). By default, modularity clustering is applied using a shared nearest neighbor (SNN) graph constructed from PCA-reduced data.

Arguments

- `data::Muon.AnnData`: The annotated data matrix to cluster. The object will be modified in place.

Keyword Arguments

- `method::AbstractString = "mc"`: Clustering method. Options are `"mc"` (modularity clustering) or `"km"` (K-means).
- `reduction::Union{AbstractString, Symbol} = :auto`: Dimensionality reduction method to use. Accepts `"pca"` or `"harmony"`. When `:auto`, it defaults to `"pca"` (support for `"harmony"` is planned).
- `use_pca::Union{AbstractString, Integer} = "pca_cut"`: Number of principal components to use, or the key in `obsm` specifying a PCA representation.
- `tree_K::Integer = 20`: Number of neighbors to use when building the SNN graph. Relevant only for `"mc"` clustering.
- `resolution::Union{Symbol, Real, AbstractRange} = :auto`: Resolution(s) for modularity optimization. `:auto` uses `0.2:0.1:2.0`.
- `cluster_K::Union{Nothing, Integer} = nothing`: Number of clusters for K-means. If `nothing`, it will be auto-determined.
- `cluster_K_max::Union{Nothing, Integer} = 30`: Maximum number of clusters to try for automatic K-means clustering.
- `dist::AbstractString = "Euclidean"`: Distance metric for K-means, e.g., `"Euclidean"`.
- `network::AbstractString = "SNN"`: Type of graph network to construct. Currently supports `"SNN"`.
- `random_starts_number::Integer = 10`: Number of random initializations for clustering (modularity clustering only).
- `iter_number::Integer = 10`: Maximum number of iterations for the clustering optimization.
- `prune::AbstractFloat = 1/15`: Pruning factor for the graph. Must be between 0 and 1.
- `seed::Integer = -1`: Random seed. Use a negative number to skip setting the seed.

Returns

Nothing. The clustering results are stored in the `obs` field of the input `AnnData` object, typically under a key like `"clusters"`.

Example

```
using Juscan

clustering!(adata; method="mc", use_pca="X_pca", resolution=1.0)
```

Notes

- `"harmony"` support is planned but not yet implemented.
- The `"mc"` method builds a neighbor graph and performs community detection; `"km"` performs K-means clustering in reduced space.
- The `method` parameter only supports `"mc"` and `"km"`; invalid inputs will return a warning.

source

Chapter 13

Plots

The `plots` module provides a set of visualization tools designed to make single-cell data exploration intuitive and informative. These functions enable users to create publication-ready figures for quality control, feature exploration, dimensionality reduction, and clustering results.

13.1 Violin and Scatter Plots

Violin and scatter plots are essential for examining cell-level metrics, such as gene counts, total expression, or mitochondrial content. The `violin` function offers compact summaries of distributions, while `scatter` allows for flexible 2D comparisons between features, optionally colored by additional metadata.

13.2 Highly Variable Genes Visualization

The `hvg_scatter` function provides a dual-panel visualization of gene variability metrics, distinguishing highly variable genes (HVGs) from background genes. This plot is especially useful for evaluating gene selection prior to dimensionality reduction.

13.3 Dimensionality Reduction Plots

The `plot_variance_ratio` function visualizes the explained variance of each principal component, helping users decide how many PCs to retain. `plot_umap` projects cells into a low-dimensional embedding using UMAP, colored by user-specified labels to reveal structure and cluster separation.

13.4 Color Palettes

Color plays a vital role in visual clarity. Juscan.jl includes palette expansion and colormap utilities to generate well-balanced, customizable color schemes. These utilities ensure consistency across all plots.

By combining visual elegance with analytical depth, the `plots` module empowers users to communicate insights clearly and effectively.

- `Juscan.Pl.expand_palette`
- `Juscan.Pl.get_continuous_colormap`
- `Juscan.Pl.hvg_scatter`

- Juscan.Pl.plot_umap
- Juscan.Pl.plot_variance_ratio
- Juscan.Pl.scatter
- Juscan.Pl.violin

13.5 QC and Feature Plots

`Juscan.Pl.violin` – Function.

```
violin(adata::AnnData, keys; kwargs...)
```

Draws violin plots for one or more features stored in `adata.obs`.

Arguments

- `adata::AnnData`: Annotated data object.
- `keys::Union{String, Vector{String}}`: One or more feature names from `adata.obs` to plot.

Keyword Arguments

- `width::Real=600`: Width of each subplot.
- `height::Real=400`: Height of the plot.
- `jitter::Union{Bool, Real}=0.5`: Jitter width or `true` to apply default jitter.
- `dot_size::Real=2`: Size of jittered dots.
- `title::String="violin plot"`: Title of the full figure.
- `palette_name::String="friendly"`: Color palette name.
- `fill_alpha::Real=1.0`: Transparency of the violin fill.
- `downsample_strategy::String="evenly"`: Strategy for downsampling color palette.
- `savefig::Union{Bool, String}=false`: Whether to save the figure, or path to save.

Returns

- A `CairoMakie.Figure` or `true` if `savefig=true`.

source

`Juscan.Pl.scatter` – Function.

```
scatter(adata::AnnData, x_key, y_key; kwargs...)
```

Generates a scatter plot from two features in `adata.obs`, optionally colored by a third.

Arguments

- `adata::AnnData`: Annotated data object.

- x_key::String: Feature name for x-axis.
- y_key::String: Feature name for y-axis.

Keyword Arguments

- color_key::Union{String, Nothing}=nothing: Feature used for point colors.
- title::String="scatter plot": Plot title.
- width::Real=600: Width of the plot.
- height::Real=400: Height of the plot.
- colormap_name::String="viridis": Colormap for color_key.
- ncolors::Int=265: Number of colors in colormap.
- downsample_strategy::String="evenly": Strategy for colormap downsampling.
- savefig::Union{Bool, String}=false: Whether to save the figure.

Returns

- A Figure or true if savefig=true.

source

13.6 HVG Visualization

Juscan.Pl.hvg_scatter – Function.

```
hvg_scatter(adata::AnnData; savefig=false)
```

Plots mean expression vs dispersion for genes, highlighting highly variable genes (HVGs).

Arguments

- adata::AnnData: Annotated data object with highly_variable, means, variances, and variances_norm in adata.var.

Keyword Arguments

- savefig::Union{Bool, String}=false: Whether to save the figure.

Returns

- A Figure or true if savefig=true.

source

13.7 Dimensionality Reduction

Juscan.Pl.plot_variance_ratio – Function.

```
plot_variance_ratio(adata::AnnData; key="pca_variance", n=50, savefig=false)
```

Plots the explained variance ratio for the top principal components (PCA).

Arguments

- `adata::AnnData`: Annotated data object.
- `key::AbstractString="pca_variance"`: Key in `adata.uns` for PCA singular values.
- `n::Real=50`: Number of PCs to plot.

Keyword Arguments

- `savefig::Union{Bool, String}=false`: Whether to save the figure.

Returns

- A `Figure` or `true` if `savefig=true`.

source

`Juscan.Pl.plot_umap` – Function.

```
plot_umap(adata; kwargs...)
```

Plots UMAP embedding colored by a categorical label in `adata.obs`.

Keyword Arguments

- `color_by::String="clusters_0.6"`: Observation field to color points.
- `key::String="umap"`: Key in `adata.obsm` containing UMAP coordinates.
- `palette_name::String="rainbow"`: Color palette name.
- `width::Real=800`: Width of the plot.
- `height::Real=600`: Height of the plot.
- `downsample_strategy::String="evenly"`: Color downsampling strategy.
- `savefig::Union{Bool, String}=false`: Save to file if `true` or path given.

Returns

- A `Figure` or `true` if `savefig=true`.

Throws

- Error if `umap_coords` is not found in `adata.obsm`.

source

13.8 Color Utilities

`Juscan.Pl.expand_palette` – Function.

```
expand_palette(base_colors::Vector{<:Colorant}, n::Int)
```

Return n interpolated colors based on the `base_colors` using `cgrad`.

source

`Juscan.Pl.get_continuous_colormap` – Function.

```
get_continuous_colormap(name::String, n::Int=265)
```

Get n evenly spaced colors from a predefined continuous colormap in `ColorSchemes.jl`. Available names: :viridis, :inferno, :plasma, :magma, :turbo, :cividis, etc.

source