Spatial Transcriptomics

Comprehensive Analysis for Stereo-seq Data

COURSE OVERVIEW

Lesson 1: Preprocessing & Clustering

- Spatial transcriptomics & Stereo-seq introduction
- Data loading and initial assessment with Scanpy
- Adjusted QC thresholds for sparse data
- Normalization and feature selection
- Spatial smoothing – improve signal quality across neighborhood
- Dimensionality reduction (PCA, UMAP)
- Unsupervised clustering
- Marker gene identification
- Cell type & Spatial-aware clustering with BANKSY

Lesson 2: Cell Type Identification

- Clustering & intersection of marker genes with signature genes
- Reference based cell type identification (Tangram, CoDi, Seurat,cell2locatio)
- Ensemble of cell type annotation algorithms
- Verification of cell type annotation using marker genes retention

Lesson 3: Spatial Analysis (our strength)

- Moran's I spatial autocorrelation - spatially variable genes
- Identifying spatially coherent gene modules with Hotspot
- Co-occurrence and neighborhood analysis (squidpy)
- Spatial domains and Cell communities

Lesson 4: Comparative Analysis

- Sample integration with Harmony
- Differential abundance testing
- Comparing cell type proportions between conditions
- Treated vs. Untreated comparison

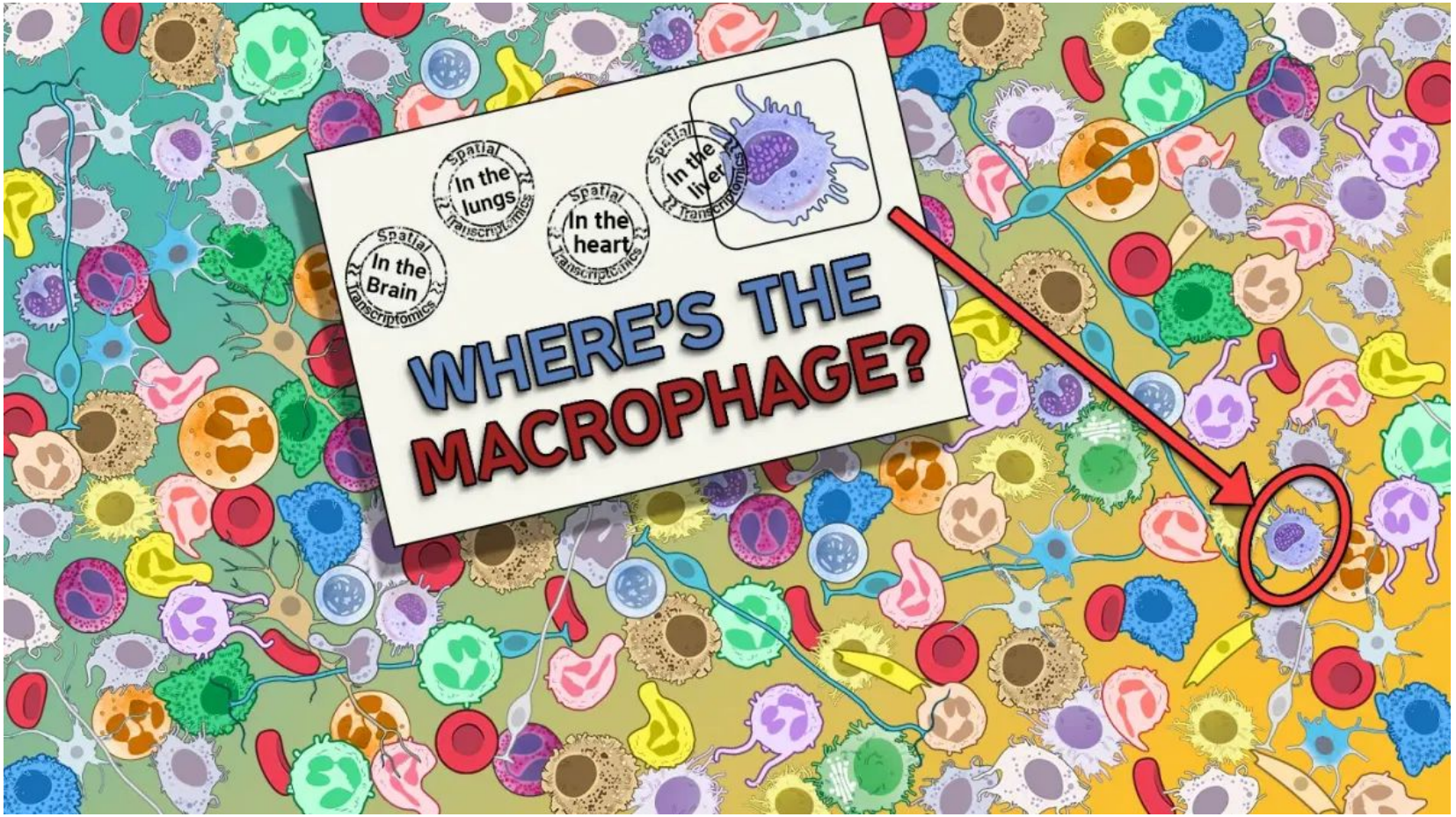
vladimirkovacevic/st-course

*RNA velocity, Pseudotime analysis, Gene Regulatory Networks, Cell-cell communication - Poor quality with sparse data

Lesson 1 - Overview

- Introduction to spatial transcriptomics
- Scanpy: Basic processing workflow
- Banksy: Spatially-aware clustering

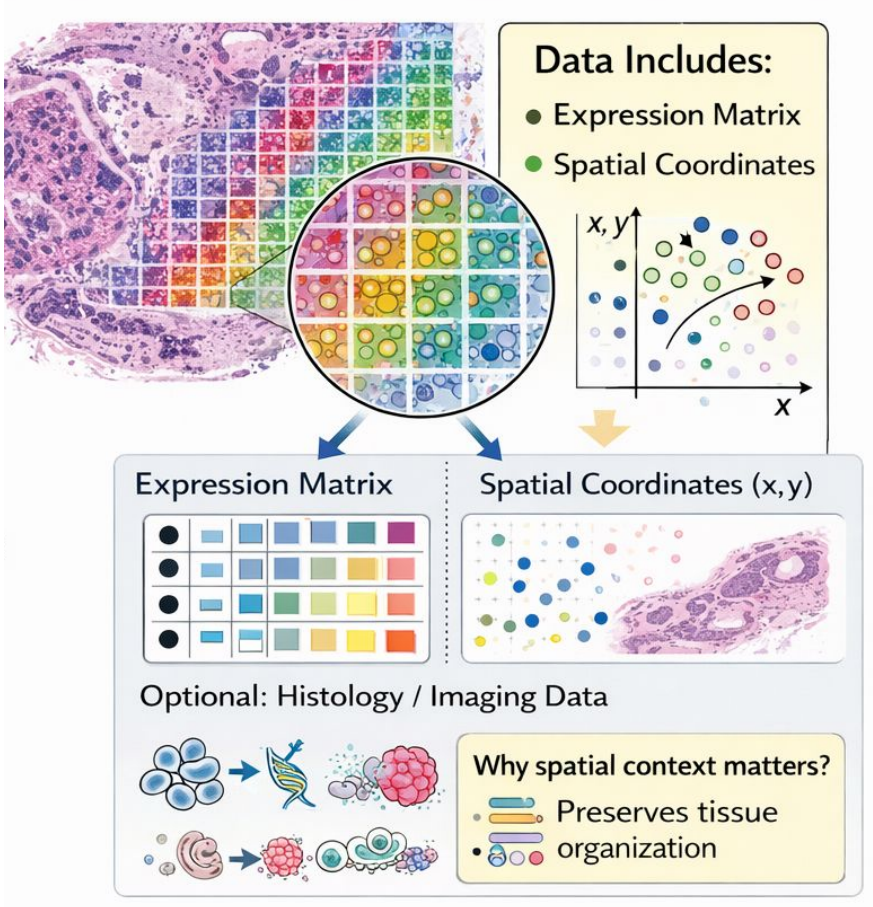
Spatial transcriptomics



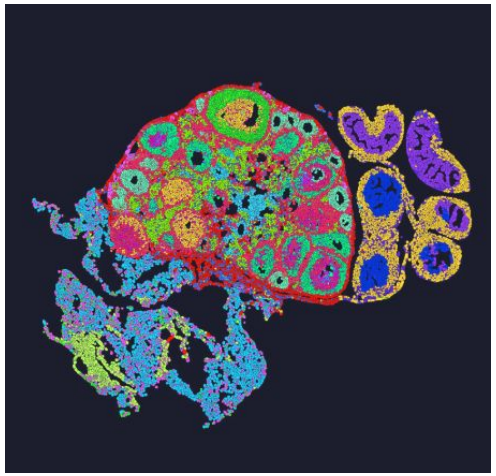
*https://sangerinstitute.blog/2025/10/22/what-is-spatial-transcriptomics/

Spatial transcriptomics

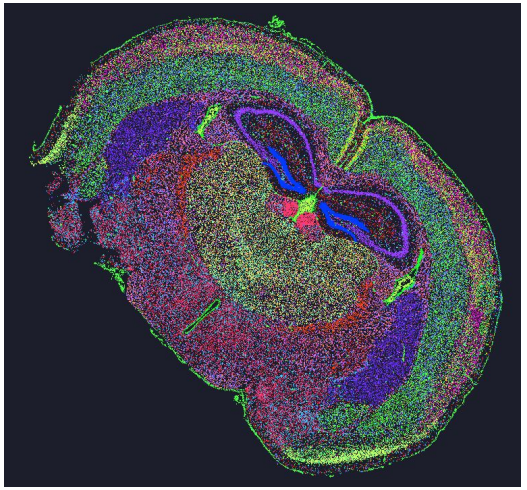
- Measures **gene expression with spatial context** in tissues
- Captures **where genes are expressed**, not just how much
- Enables study of:
  - **Tissue architecture** (layers, compartments)
  - **Cell-cell interactions**
  - **Spatial heterogeneity** (e.g., tumors, brain)



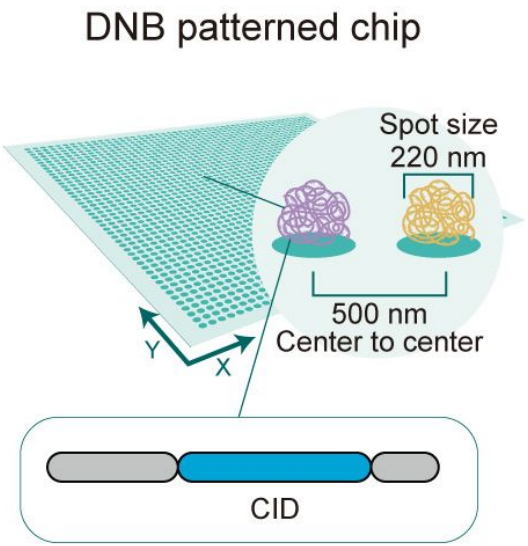
Stereo-Seq



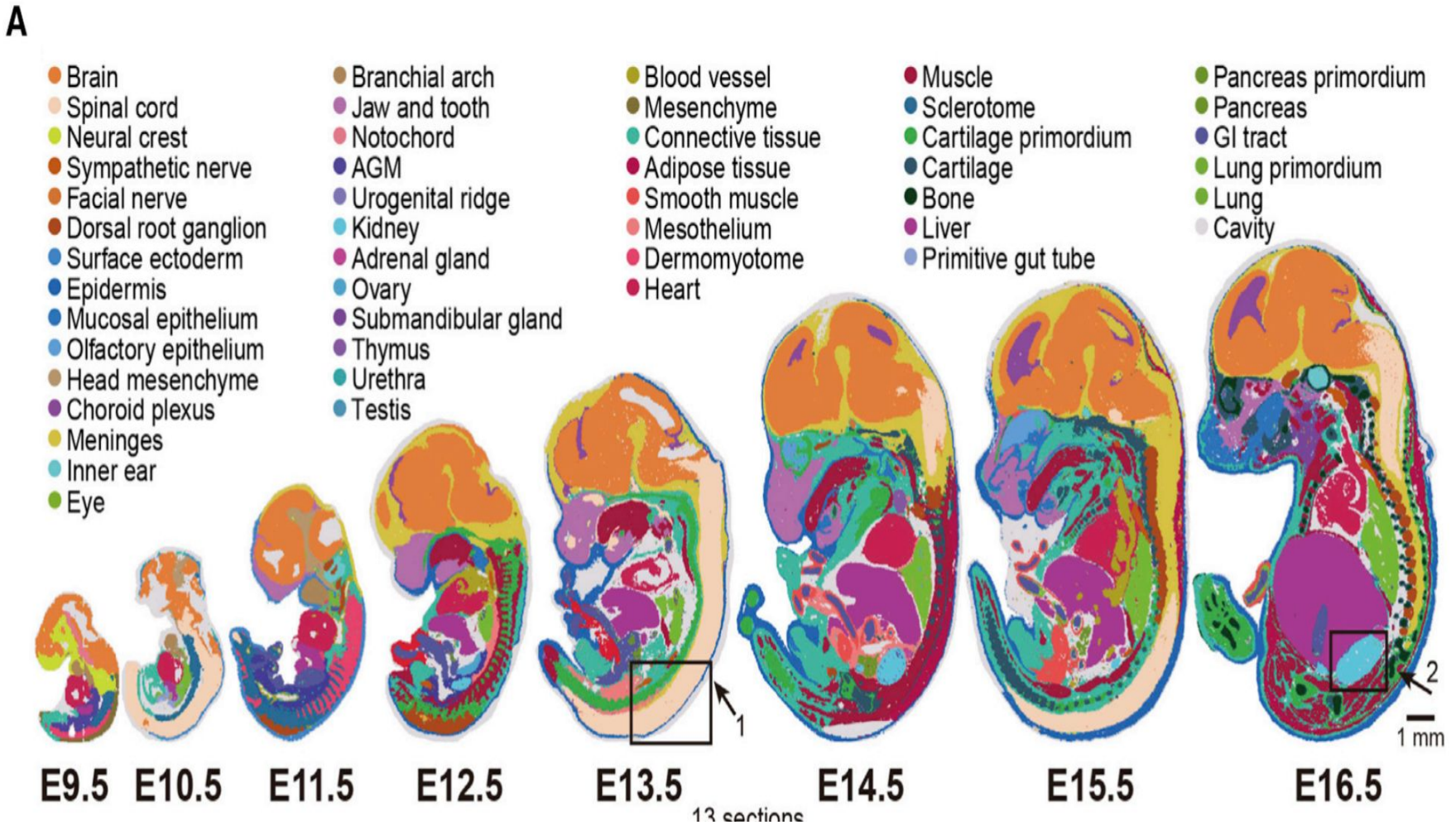
Mouse ovary single-cell clustering results (cellbin)



Mouse brain single-cell clustering results (cellbin)



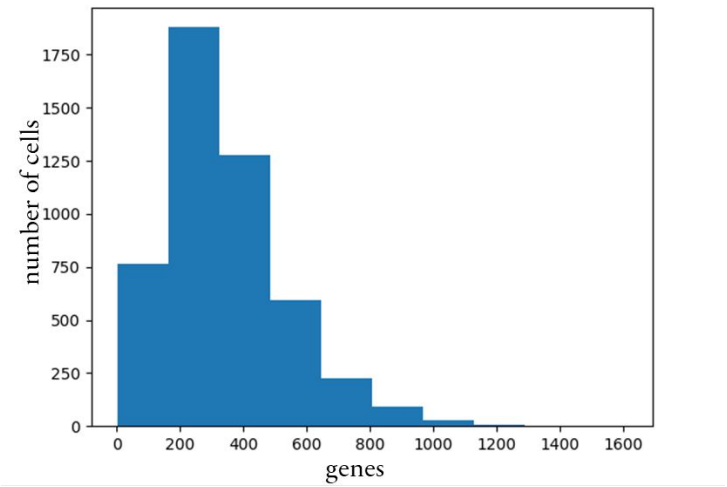
Applications: Spatiotemporal transcriptomic atlas of mouse organogenesis



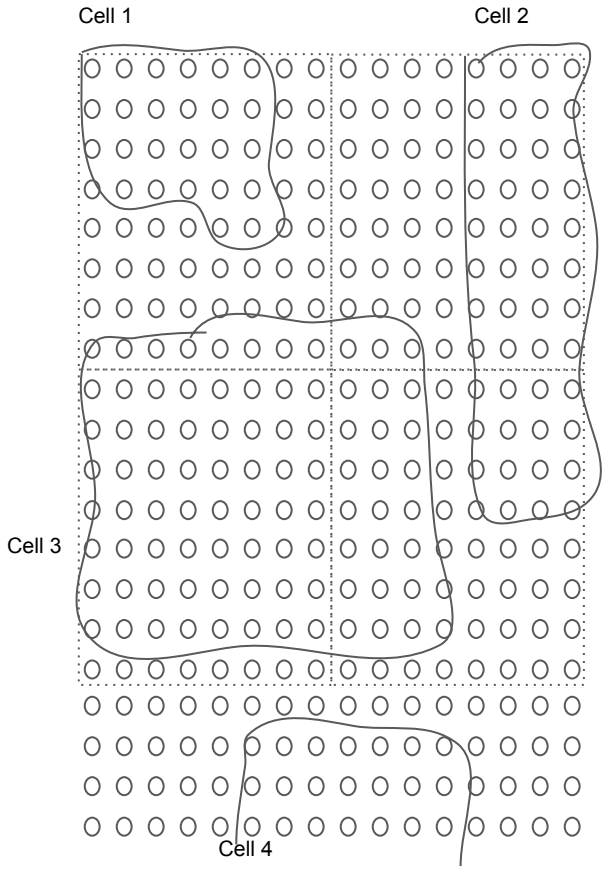
Source paper. Mosta database

Spatial transcriptomics challenges

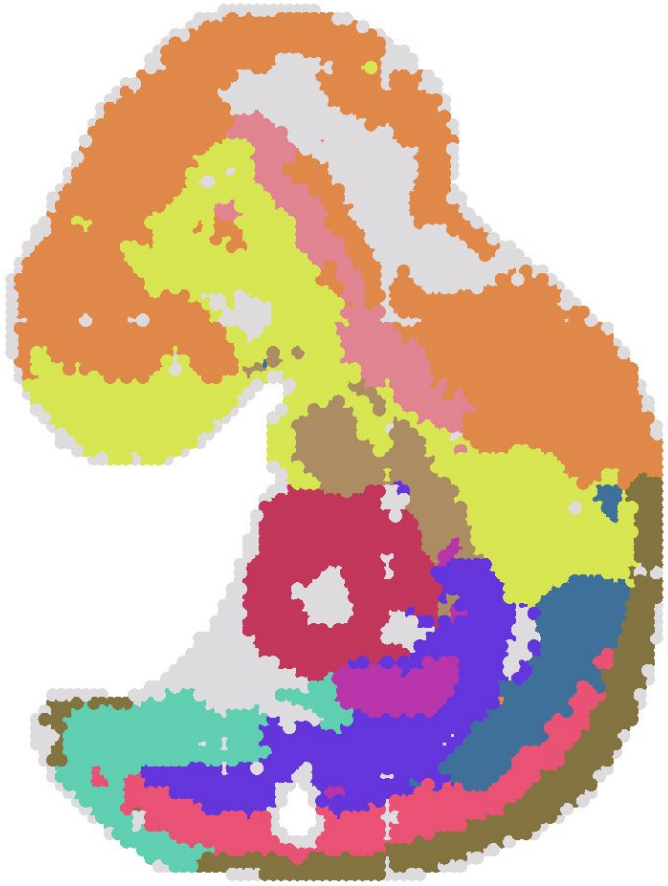
Gene resolution (dropout)

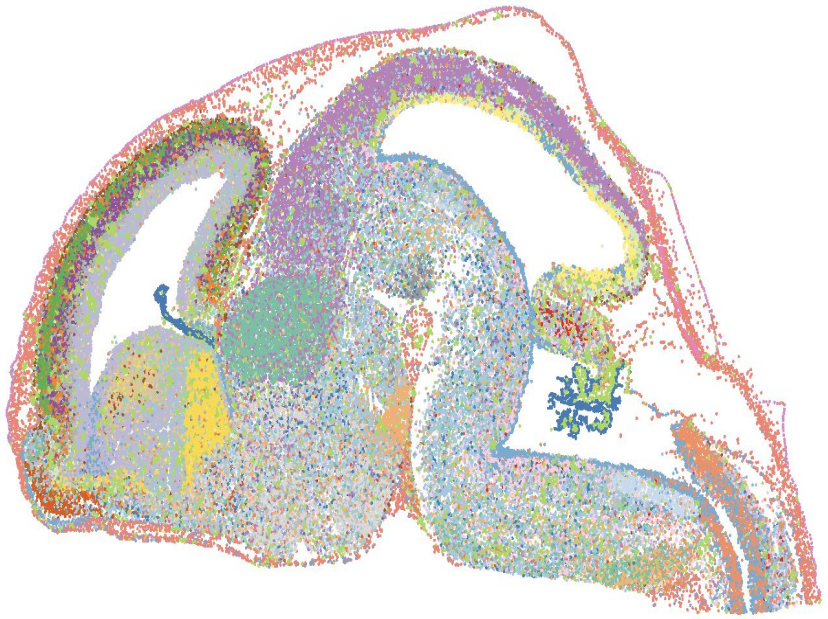


Cell border detection



High and low cell type mixture tissues

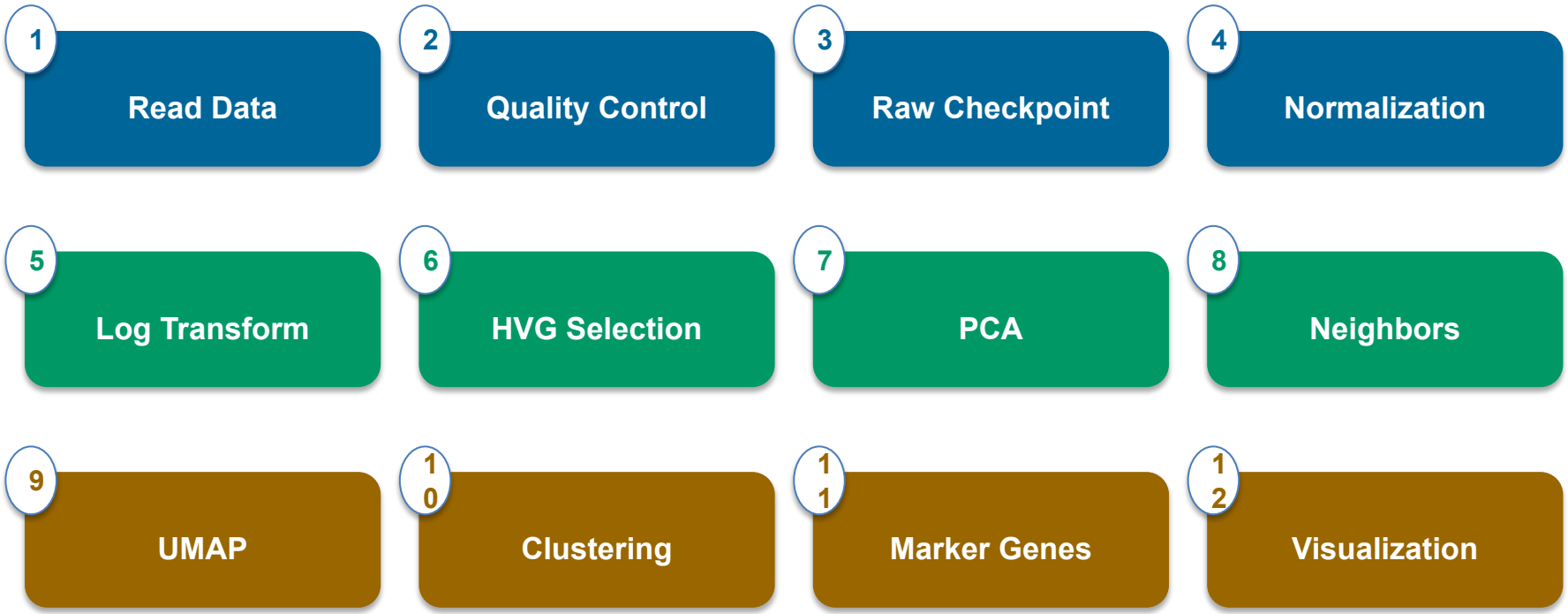




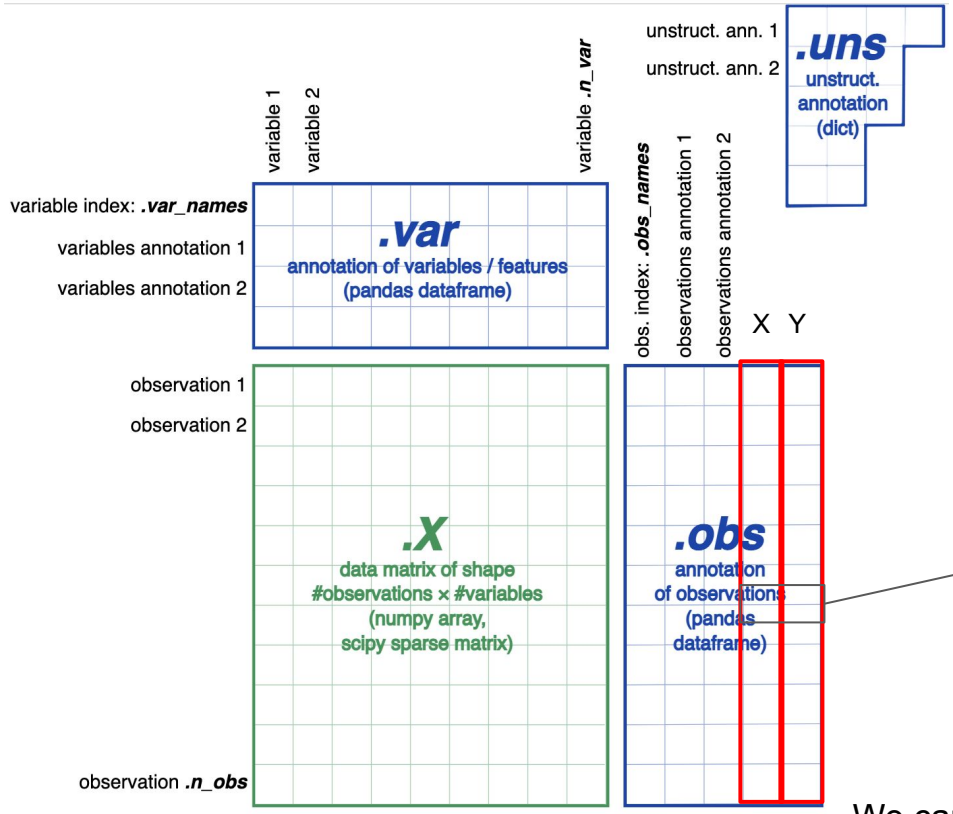
Cere Gran NeuB
216
ChP
658
Corti Glu Neu
1,461
Corti prog
1,305
CortiHippo Glu Neu
1,180
CR
515
Cranium Fibro
202
DA Neu
334
Die GNeu
4,250
DorsHb RGC
1,452
Endo
647
Epidermis
488
Ery

Scanpy: Basic processing

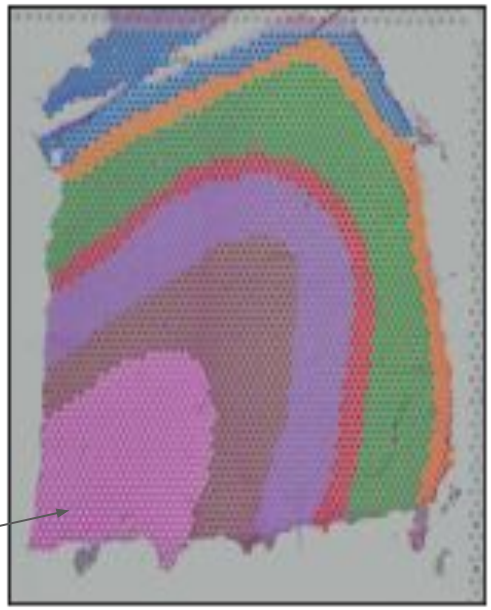
Scanpy Analysis Pipeline



Annotated data object



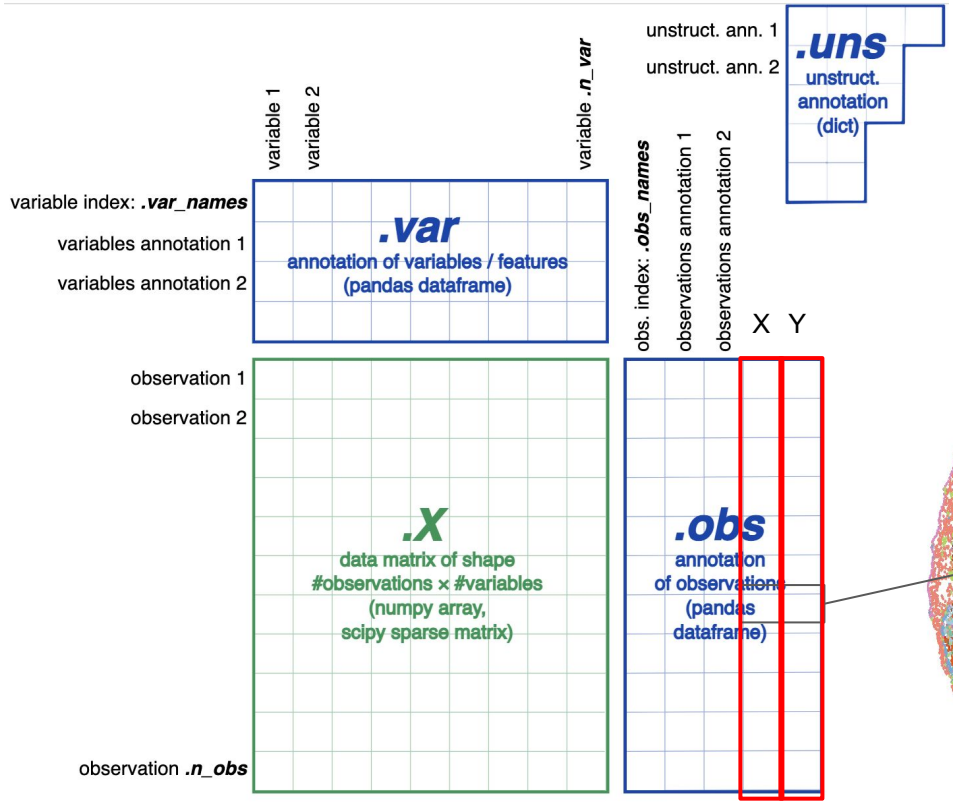
DLPFC tissue blocks



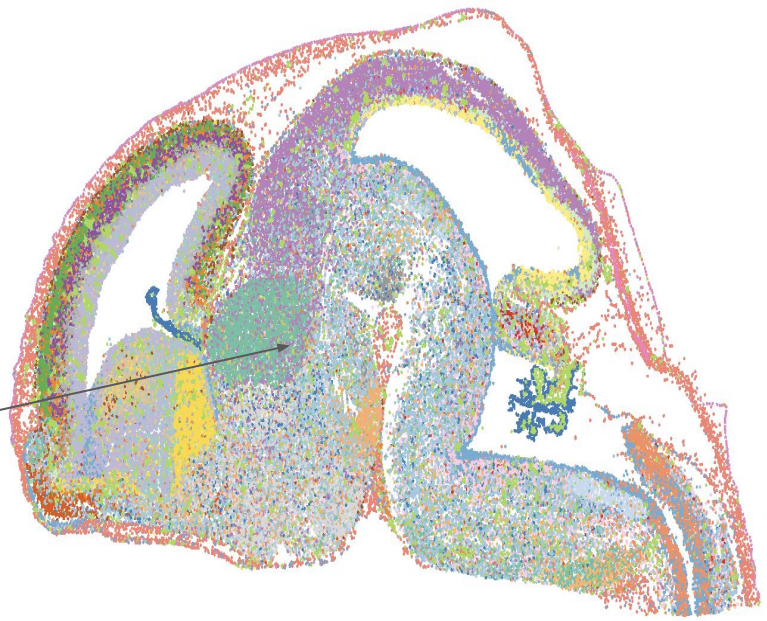
We can fully reconstruct the position of each cell in the tissue (even 3D or time series)

https://anndata-tutorials.readthedocs.io/en/latest/getting-started.html

Annotated data object



Mouse embryo brain, day 16



Cere Gran NeuB 216
ChP 658
Corti Glu Neu 1,461
Corti prog 1,305
CortiHippo Glu Neu 1,180
CR 515
Cranium Fibro 202
DA Neu 334
Die GNeu 4,250
DorsHb RGC 1,452
Endo 647
Epidermis 488
Ery

We can fully reconstruct the position of each cell in the tissue (even 3D or time series)

https://anndata-tutorials.readthedocs.io/en/latest/getting-started.html

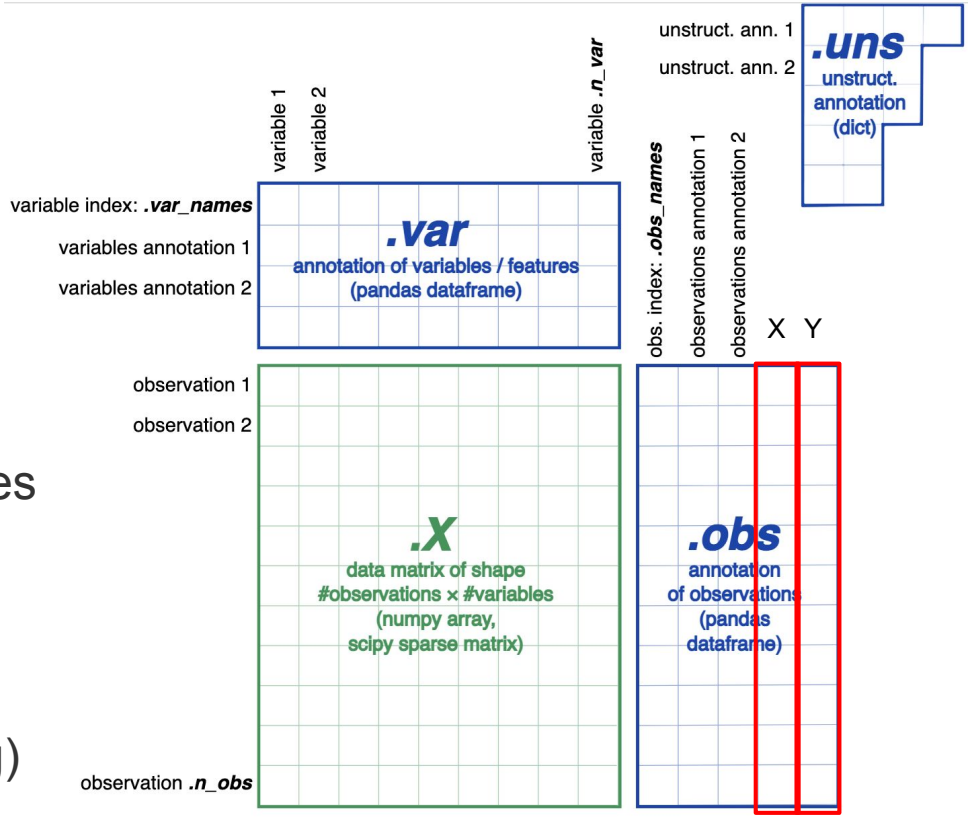
Step 1: Read Data

Load spatial transcriptomics data from GEF file

```
import scanpy as sc

# Read H5AD file
adata = sc.read_h5ad(
    filename='./data.h5ad'
)



# Seurat .rds -> AnnData (.h5ad) using SeuratDisk
```

Understanding the AnnData Object

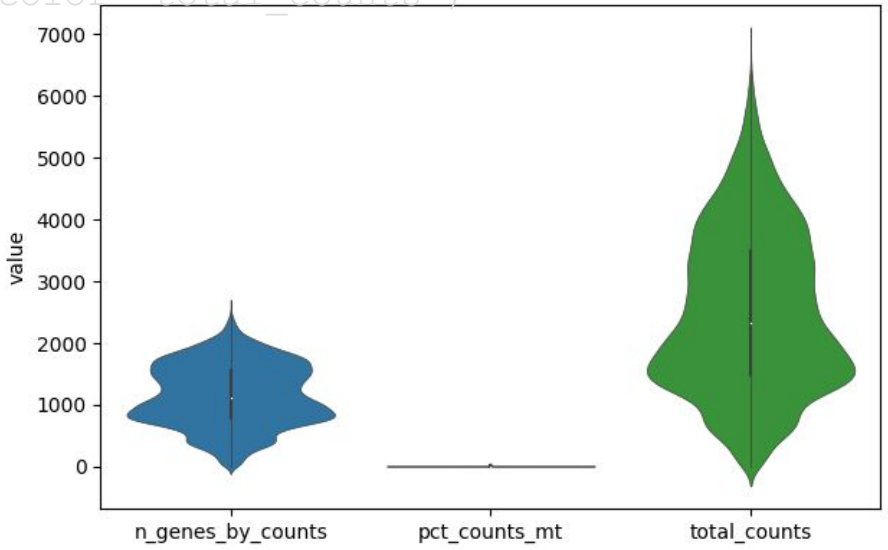
- AnnData: Core data structure in Scanpy
- adata.shape: (n_cells, n_genes) dimensions
- adata.obs: Cell/bin metadata
- adata.var: Gene information
- adata.X: Expression count matrix (sparse or dense)
- adata.obsm['spatial']: Spatial coordinates (x, y)
- sc.tl / sc.pp: Analysis tools (tools / preprocessing)
- sc.pl: Plotting functions (scanpy.plotting)



Step 2: Quality Control

Calculate and visualize quality control metrics

```
# Calculate QC metrics
sc.pp.calculate_qc_metrics(adata)

# View QC distributions
sc.pl.violin(adata,
['n_genes_by_counts', 'total_counts'])

# View spatial distribution of QC
metrics
sc.pl.spatial(adata,
color='total_counts')
```



- total_counts: Total UMI counts per cell/bin
- n_genes_by_counts: Number of genes detected per cell/bin
- pct_counts_mt: Percentage of mitochondrial genes
- Used to identify low-quality cells or empty bins
- Filter cells with too few genes/counts
- Filter cells with high mitochondrial content
- QC ensures downstream analysis quality

QC Visualization Outputs

Violin Plot

Shows distribution of QC metrics across all cells. Identifies outliers and quality thresholds.

Spatial Scatter

Maps total_counts and n_genes spatially. Reveals tissue structure and potential artifacts.

Histogram

Distribution of gene counts. Helps set filtering thresholds.

Step 3: Raw Data Checkpoint

Store raw counts before normalization

```
  ├── X         ← counts
  ├── var
  ├── obs
  ├── spatial coords

adata.layers["counts"] = adata.copy() # the undo button you have to build
# Save raw counts for later use in:
# - Marker gene analysis
# - Differential expression
# - Data recovery if needed

data
  ├── exp_matrix         ← will be modified
  ├── layers[“counts”]
  │                  ├── X   ← frozen copy
  │                  └── var
  ├── obs             ← shared
  ├── obsm            ← shared
```

Step 4: Normalization

```
# Normalize total counts per cell
sc.pp.normalize_total(adata,
target_sum=1e4)

# Log transformation
sc.pp.log1p(adata)

# Alternative: scTransform normalization
# SCTransform: use R/Seurat or scanpy
external
```

What raw counts represent?

Raw counts in a Stereo-seq file represent deduplicated UMIs per gene per bin (not the number of reads from the FASTQ file)

Normalization: Scales each bin to the same total so expression is comparable across spots:

1. Compute total counts in the bin
2. Scale all genes in that bin so the sum equals target_sum (e.g., 10,000)

Log1p: Compresses high counts and stabilizes variance (reduce influence of highly expressed genes) so clustering and PCA are not dominated by a few genes.

Step 4: Normalization - sc_transform

```
# SCTransform: use R/Seurat
```

SCTransform is a variance-stabilizing normalization method based on a **regularized negative binomial regression**, originally designed for **single-cell RNA-seq**.

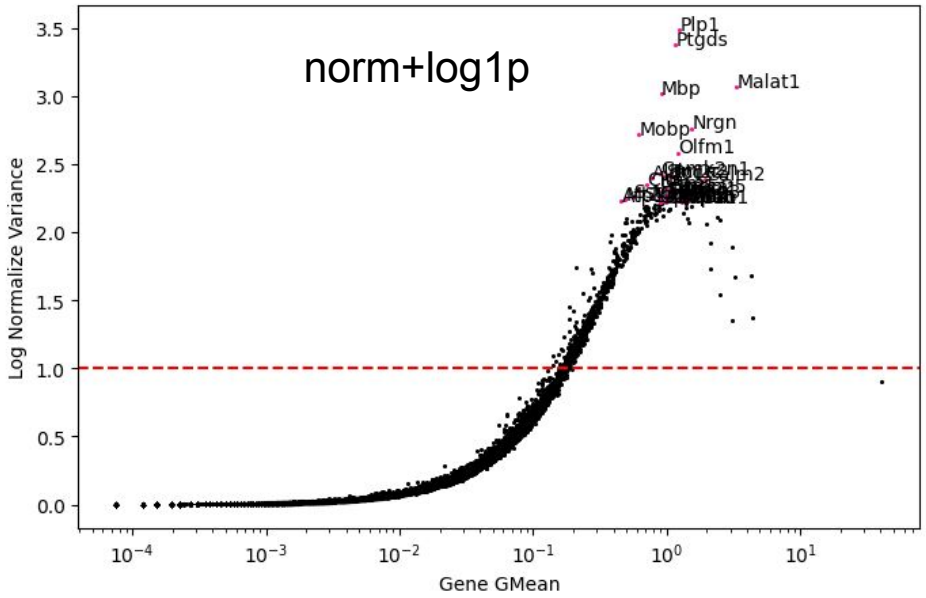
Could create issues with sparse ST data:

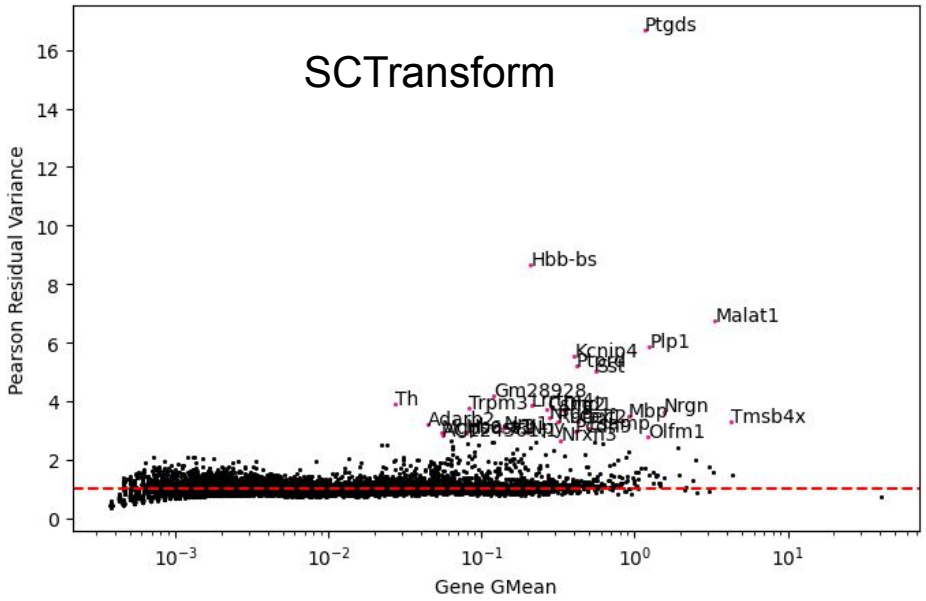
- Spatial variation is biological signal. SCTransform may interpret spatial gradients as technical effects
- Neighbor-aware methods break BANKSY / GraphST expect count-like or log-normalized data
- SCTransform produces Pearson residuals - Over-correction, can erase tissue domains
- Especially dangerous for low-count bins

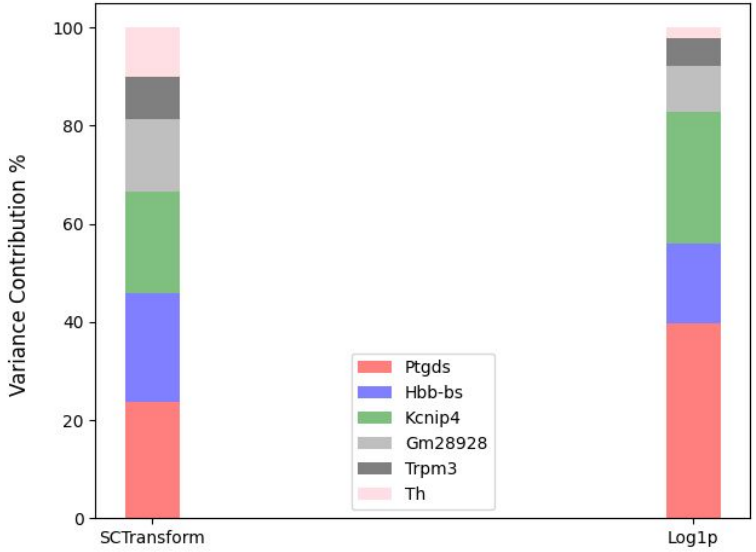
Still:

- It could provide new insights about some genes
- Can improve clustering
- Do some analysis with norm+log1p some with SCTransform

Step 4: Normalization - sc_transform







Step 4: Normalization - sc_transform

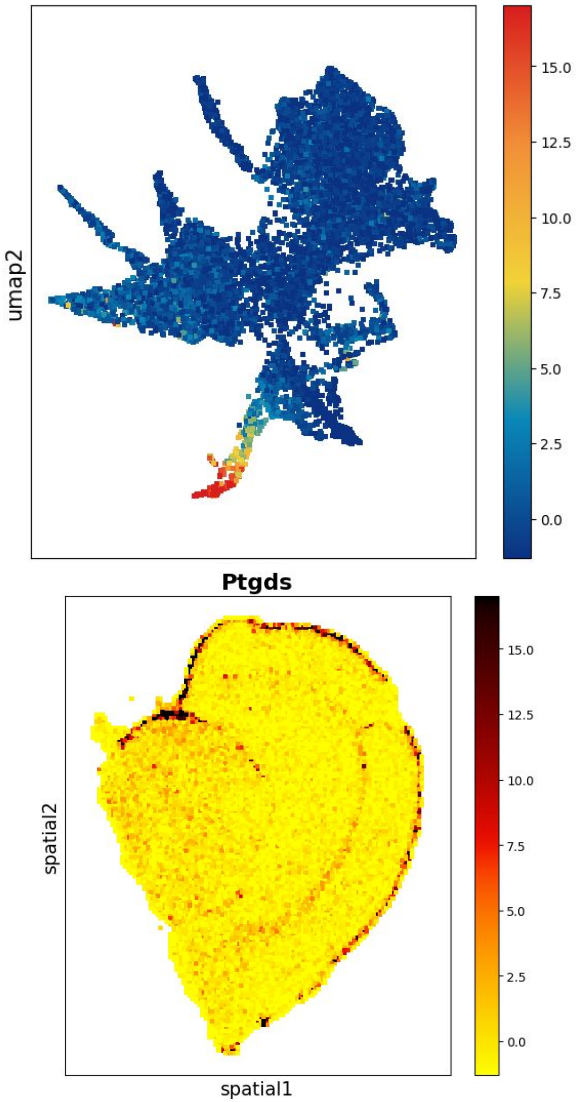
UMAP SCTransform

UMAP norm+log1p

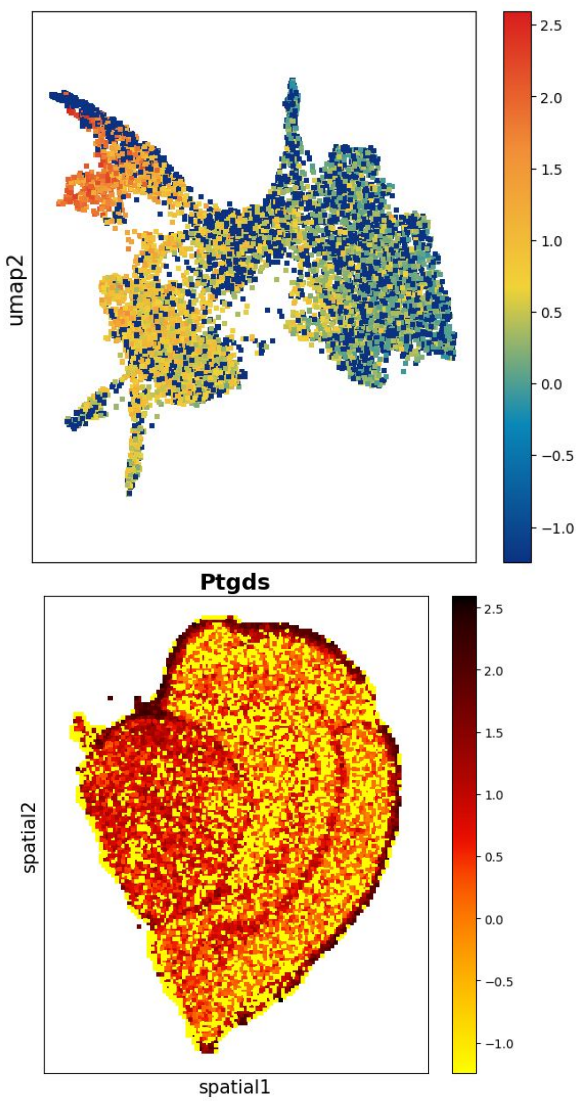
SCTransform may contribute to obtaining more separable clusters

Step 4: Normalization - sc_transform

Ptgds SCTransform



Ptgds norm+log1p



SCTransform may contribute to obtaining more evident gene expression distribution

Step 5: Identify Highly Variable Genes

Select informative genes for downstream analysis

```
# Find highly variable genes
sc.pp.highly_variable_genes(
    adata,
    flavor="seurat_v3", # models variance using regularized negative
           #   binomial regression - more stable under zero inflation
    n_top_genes=3000,
    span=0.6 # Controls smoothing for mean-variance trend
)
# Increase n_top_genes and span if data is extremely sparse

# Remove mitochondrial and ribosomal genes from HVGs
adata.var["mt"] = adata.var_names.str.startswith("MT-")
adata.var["ribo"] = adata.var_names.str.startswith(("RPL", "RPS"))
adata.var.loc[adata.var["mt"] | adata.var["ribo"], "highly_variable"]
= False

# View HVG statistics
hvg = adata.var['highly_variable']
print(f"Number of HVGs: {hvg.sum()}")
```

Highly Variable Genes (HVG)

- HVGs capture biological variation between cells
- Filter out genes with low expression (min_mean)
- Filter out ubiquitously expressed genes (max_mean)
- Select genes with high dispersion (min_disp) - variance-to-mean ratio of a gene across bins, measuring how much it varies relative to its average expression.
- Reduces noise from non-informative genes
- Speeds up downstream computations
- Typically select 3000-5000 HVGs for sparse ST data
- HVGs drive PCA and clustering results

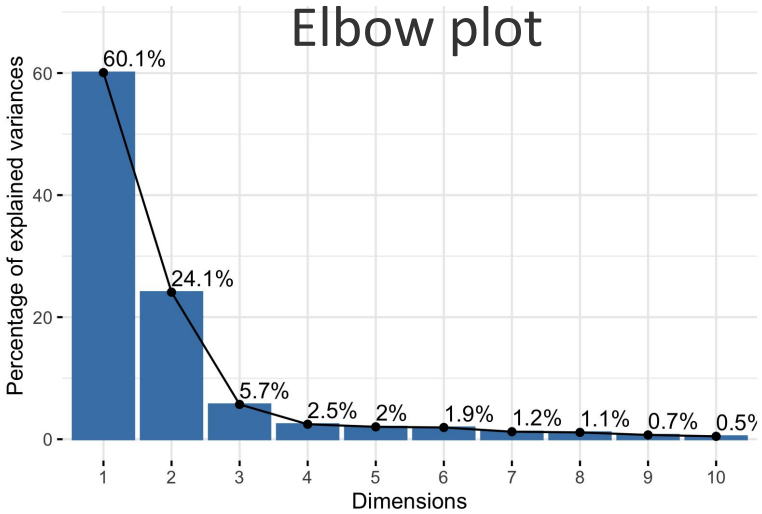
Step 6: Principal Component Analysis (PCA)

Reduce dimensionality while preserving variance

```
# Run PCA on highly variable genes
sc.tl.pca(
    adata,
    n_comps=50,
    use_highly_variable=True,
    svd_solver="arpack",  #  More numerically stable on sparse matrices
    random_state=0
)


# View variance explained
sc.pl.pca_variance_ratio(adata)
```

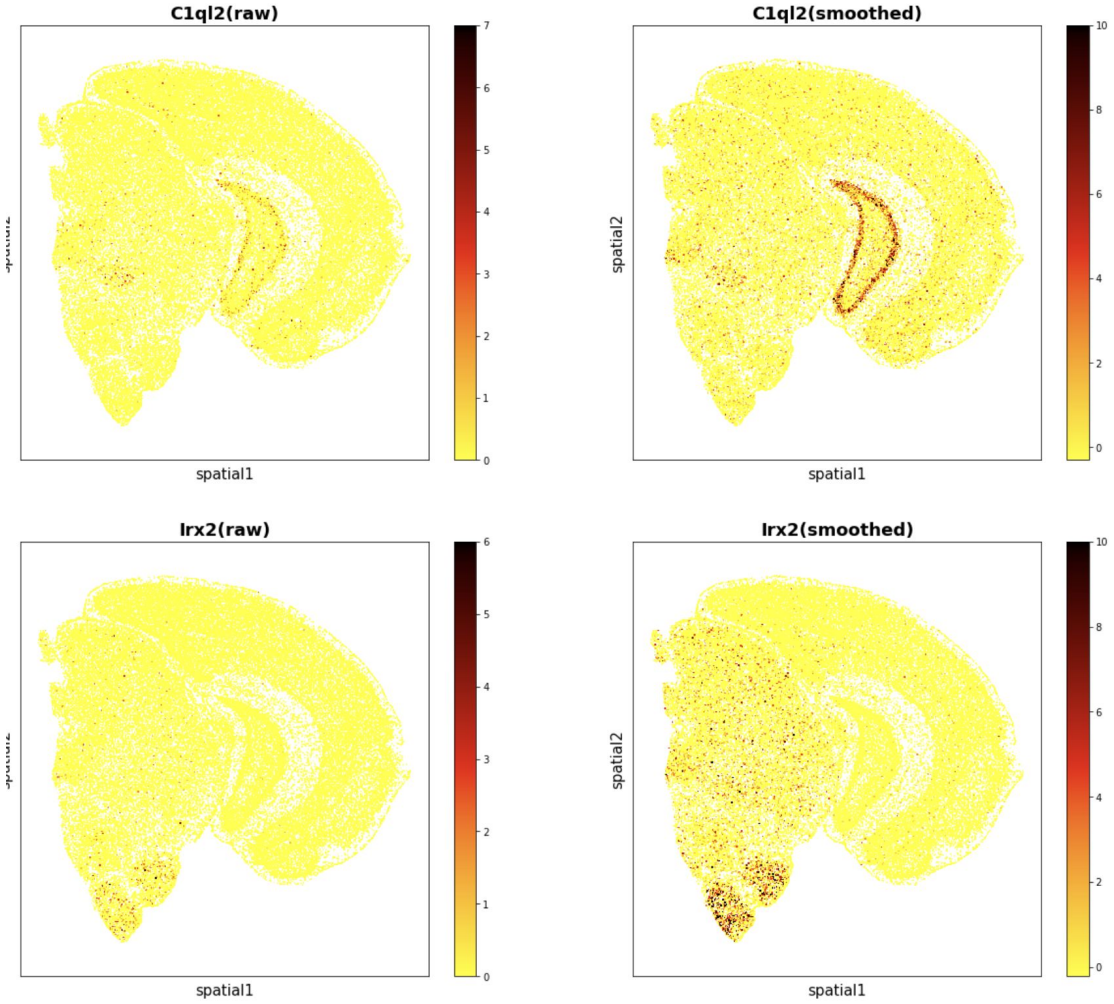
Understanding PCA

- Reduces high-dimensional gene space to principal components
- Each PC captures maximum remaining variance
- Typically use top 20-50 PCs for downstream analysis
- Elbow plot helps choose optimal number of PCs
- PCs used for neighbor graph and clustering
- Denoises data by removing low-variance dimensions
- PCA does not remove batch effects by design, but projection and truncation can obscure them
- Low explained variance is concerning if:
  - Clusters are unstable
  - Results change wildly with PCA number



Gaussian Smoothing

- Gaussian smoothing is a kernel-based smoothing operation that modifies an expression matrix by averaging values locally using weights defined by a Gaussian function
- For each cell/bin, nearby neighbors are identified (e.g., via a k-nearest neighbor graph)
- The expression value of a cell/bin is then replaced with a weighted average of its neighborhood, reducing random local fluctuations



Gaussian Smoothing

- Key parameters for Gaussian smoothing:
  - n_neighbors: size of local smoothing neighborhood
  - smooth_threshold: influences the Gaussian variance (controls smoothing strength).
- Benefits
  - Noise reduction: Suppresses stochastic gene expression noise, enhancing signal stability
  - Biologically plausible patterns: Produces smoother spatial expression distributions that helps **clustering** and **visualization** by reducing spurious variability
  - Weighted averaging preserves gradual transitions better than uniform averaging
- Not optimal when fine structural boundaries must remain sharp: excessive smoothing can blur true edges

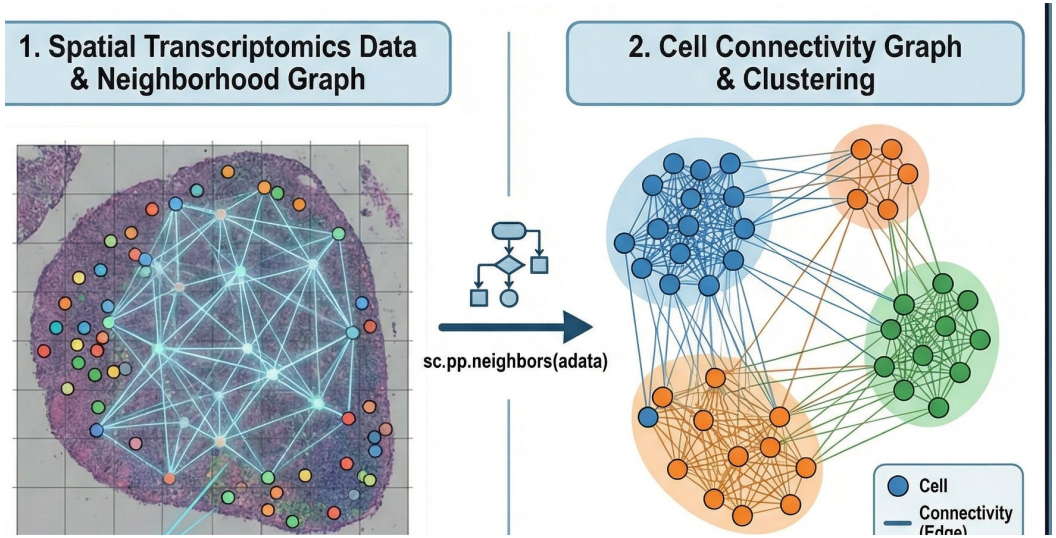
Step 7: Compute Neighbor Graph

Build cell connectivity graph for clustering

```
# Compute k-nearest neighbors
sc.pp.neighbors(adata,
    metric="euclidean",
    n_pcs=50
)

# For spatial analysis, also compute:
# sq.gr.spatial_neighbors(adata)  # squidpy
```

Expression graph = **cell state**
Spatial graph = **microenvironment**

- Connects each cell to its k nearest neighbors
- Distance computed in PCA space
- n_pcs: Number of PCs to use (typically 50)
- Creates a graph structure for clustering
- Foundation for Leiden/Louvain algorithms
- spatial_neighbors: Uses actual spatial coordinates
- n_jobs: Parallel processing for speed



Step 8: UMAP Embedding

Create 2D visualization of cell relationships

```
# Compute UMAP for visualization
sc.tl.umap(
    adata,
    min_dist=0.4, # Prevents over-fragmentation caused by dropout
    # min_dist=0.25 Emphasize discrete domains
    spread=1.0, # Keeps embedding readable without exaggeration
    n_components=2,
    random_state=0
)


# Visualize UMAP embedding
sc.pl.umap(adata)
```

Common mistakes in ST UMAP
❌ min_dist=0.1 (scRNA default)
→ fake micro-clusters
❌ Running UMAP without spatially aware neighbors
→ structure unrelated to tissue layout
❌ Re-running UMAP without fixing random_state
→ irreproducible embeddings

Understanding UMAP

- UMAP: Uniform Manifold Approximation and Projection
- Creates 2D representation preserving local structure
- Cells with similar expression are placed nearby
- Better at preserving global structure than t-SNE
- init_pos='spectral': Initialize with spectral embedding:
  - stability
  - convergence
  - preservation of large-scale structure
- Used for visualization, NOT for clustering
- Clusters appear as distinct groups in UMAP space

Step 9: Leiden Clustering

Identify cell clusters/communities

```
# Run Leiden clustering
sc.tl.leiden(
    adata,
    key_added="leiden",
    resolution=0.4, # Larger resolution - more clusters
    random_state=0
)

# Alternative: Louvain clustering
# sc.tl.louvain(adata)

# Check cluster sizes
print(adata.obs['leiden'].value_counts())
```

Clustering Algorithms in Scanpy

• Leiden Algorithm (Recommended):
- Community detection method
- Improved version of Louvain
- Guarantees connected clusters
- resolution: Controls granularity
• Higher resolution = more clusters
• Other Options:
- Louvain: Classic community detection
- Phenograph: Similar to Leiden
- Significantly faster for large data

Visualizing Clusters

Visualize cluster assignments in UMAP and spatial coordinates

```
# UMAP colored by cluster
sc.pl.umap(adata, color='leiden')

# Spatial distribution of clusters
sc.pl.spatial(adata, color='leiden')
```

Step 10: Find Marker Genes

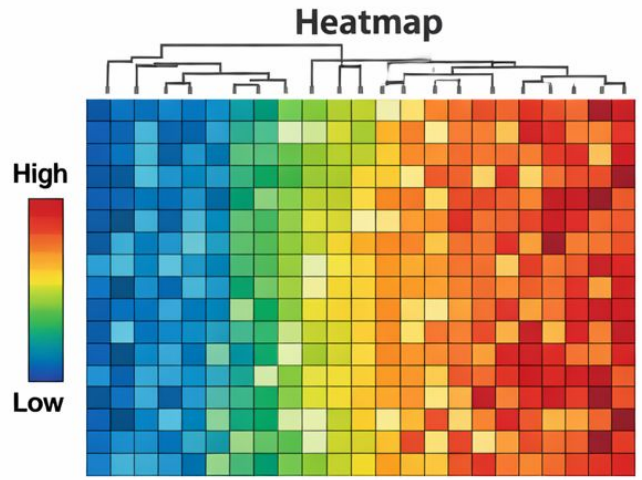
Identify genes that define each cluster

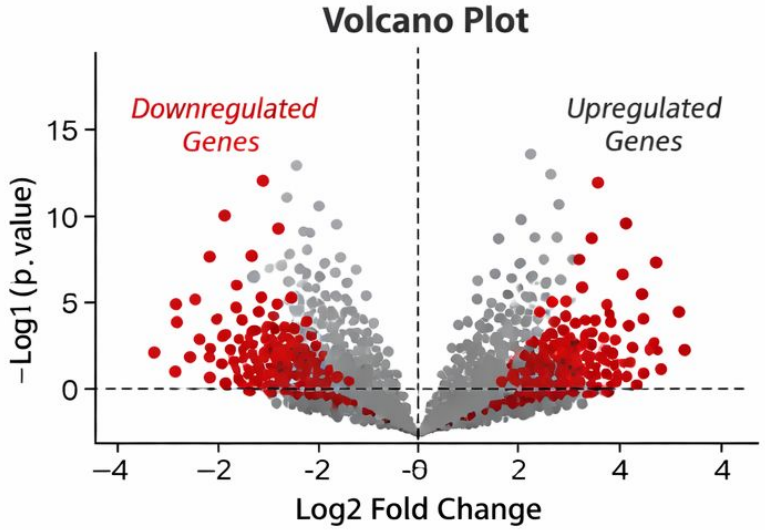
```
sc.pp.normalize_total(adata, target_sum=1e4)
sc.pp.log1p(adata)

# Find marker genes for each cluster
sc.tl.rank_genes_groups(
    adata,
    groupby="leiden",
    method="wilcoxon", # Non-parametric - robust to dropout
    corr_method="benjamini-hochberg",
    n_genes=50,
    use_raw=False
)


# View top markers
sc.get.rank_genes_groups_df(adata,

    n_genes=5
)
```

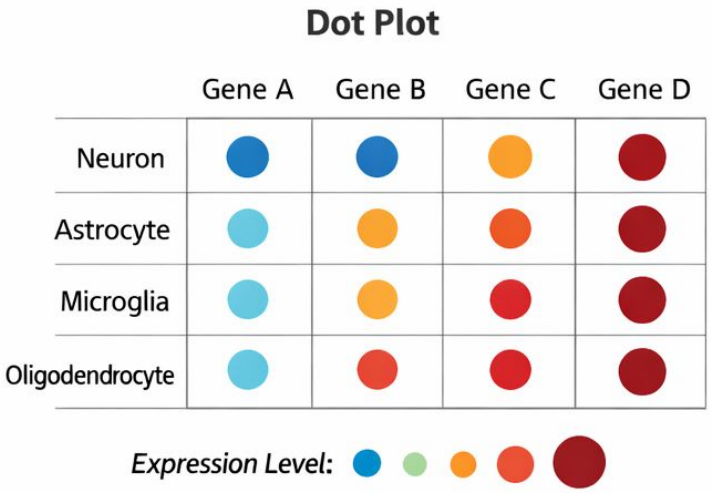
Marker Gene Identification

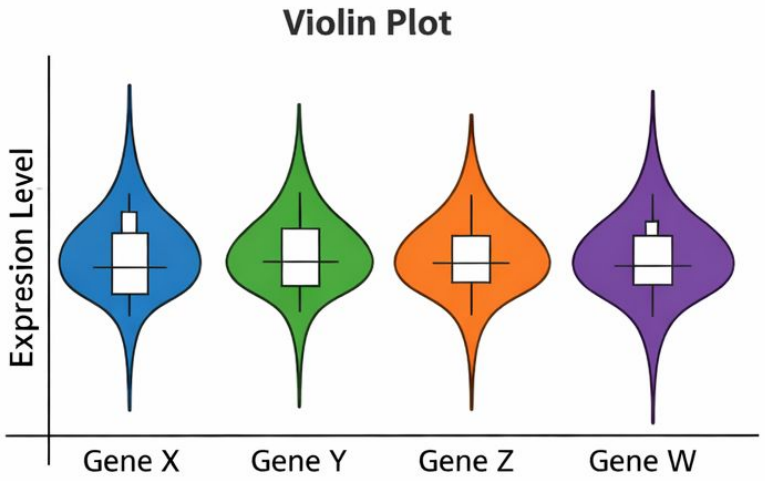
- Compares expression between one cluster vs. rest
- method='t-test': Parametric test for large samples
- method='wilcoxon_test': Non-parametric alternative - no need for normal distribution
- use_raw=False: Use norm+log1p data
- use_highly_variable=True: Only test HVGs
- Results include: log fold change, p-value, scores
- Top markers help identify cell types

Marker Gene Visualization



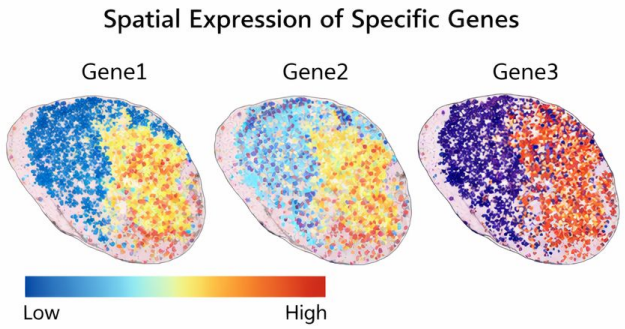


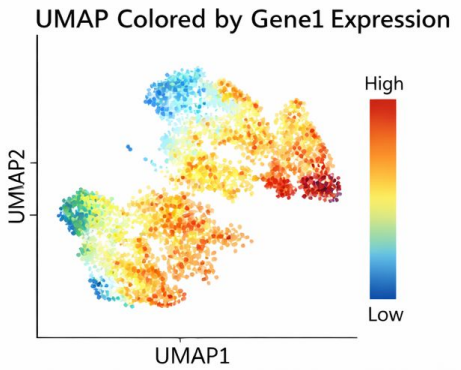


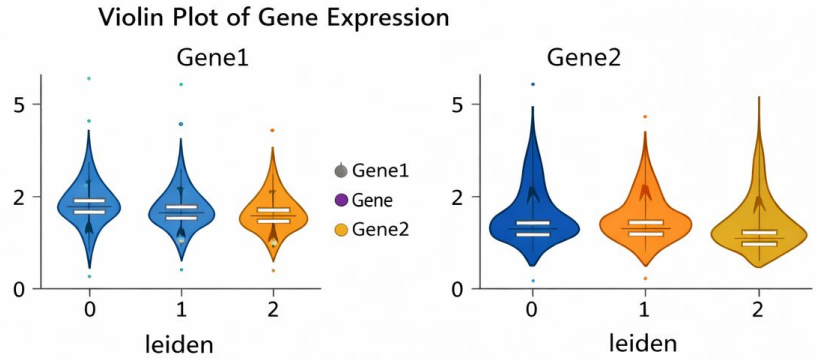


Visualizing Specific Genes

```
# Spatial expression of specific
genes
sc.pl.spatial(adata,
    color=['Gene1', 'Gene2', 'Gene3']
)
# UMAP colored by gene expression
sc.pl.umap(adata, color='Gene1')

# Violin plot of gene expression
sc.pl.violin(adata,
    keys=['Gene1', 'Gene2'],
    groupby='leiden'
)
```







Complete Workflow Summary

Complete analysis in ~15 lines of code

```
import scanpy as sc
# 1. Read data
adata = sc.read_h5ad('data.h5ad')

# 2. QC and preprocessing
sc.pp.calculate_qc_metrics(adata)
adata.raw = adata.copy()
sc.pp.normalize_total(adata, target_sum=1e4)
sc.pp.log1p(adata)

# 3. Feature selection and reduction
sc.pp.highly_variable_genes(adata)
sc.tl.pca(adata, n_comps=20, use_highly_variable=True)

# 4. Clustering
sc.pp.neighbors(adata, n_pcs=30)
sc.tl.umap(adata)
sc.tl.leiden(adata)

# 5. Marker genes
sc.tl.rank_genes_groups(adata, groupby='leiden')
adata.write('adata.h5ad')
```

Best Practices

- Always perform QC before analysis
- Save raw counts before normalization
- Choose HVG parameters based on data
- Check elbow plot for optimal PC number
- Leiden generally preferred over Louvain (Sci Rep)
- Use norm+log1p data for marker gene analysis
- Validate clusters with known markers

Common Issues & Solutions

• Memory errors:

- Reduce bin_size
- Use fewer PCs

• Poor clustering:

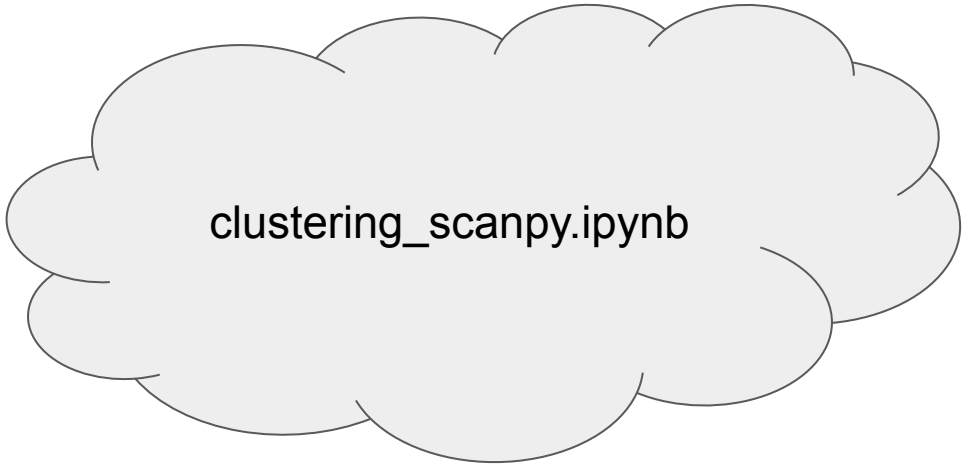
- Adjust resolution parameter
- Check QC filtering
- Try different normalization

• Slow performance:

- Reduce n_neighbors

• No clear structure:

- Check data quality
- Try different bin_size
- Verify tissue region

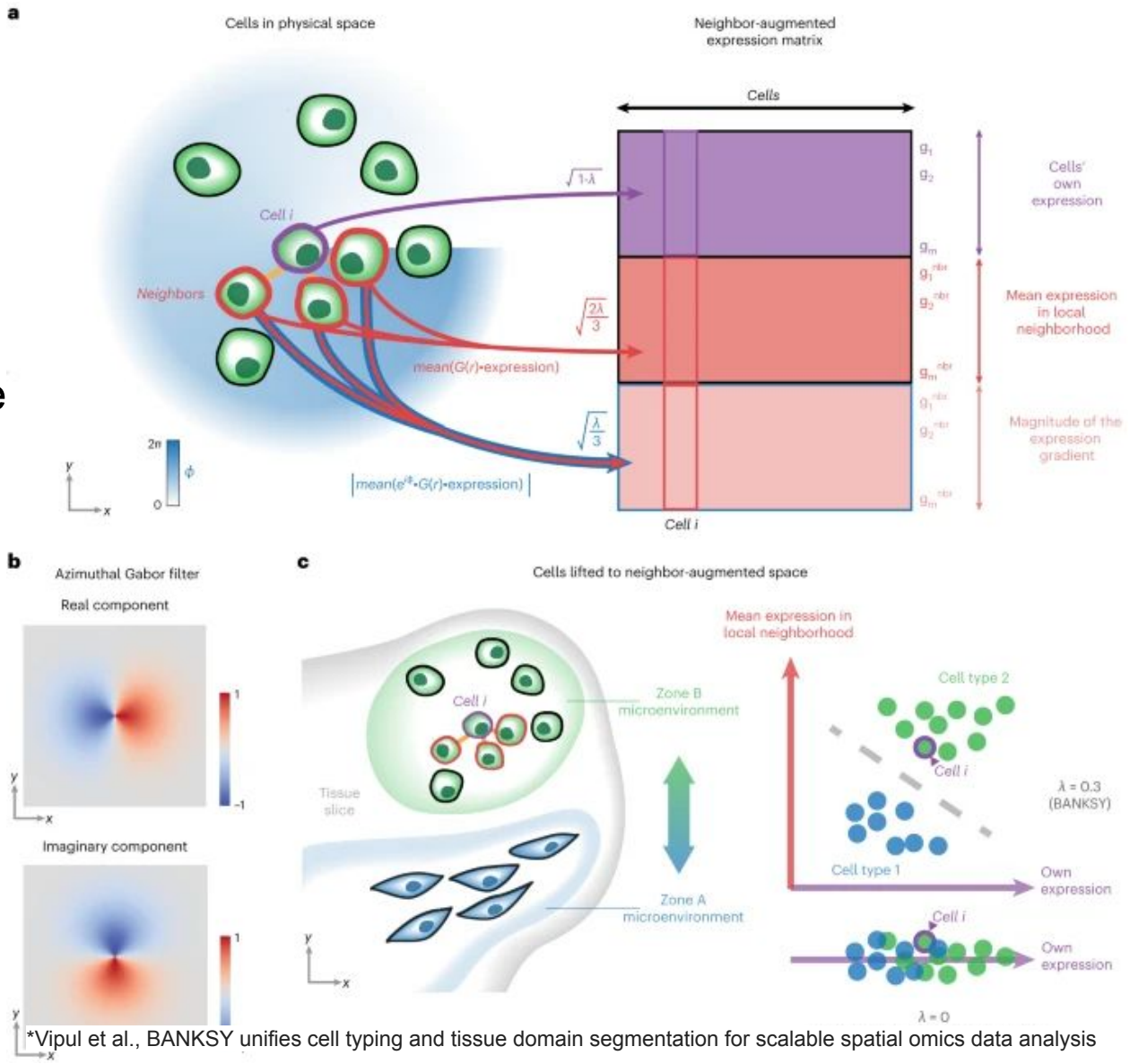


BANKSY: Spatially-aware Clustering

Building Aggregates with a Neighborhood Kernel and Spatial Yardstick

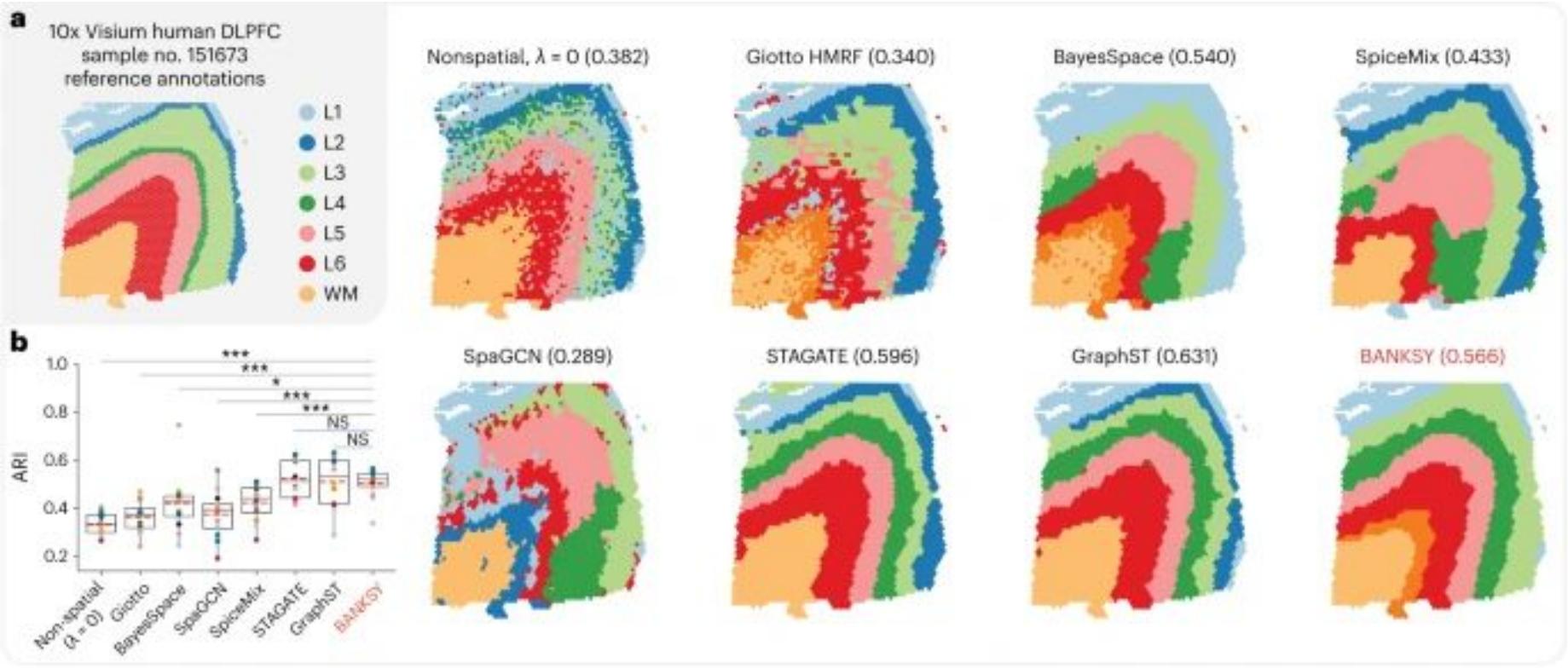
What is BANKSY?

- Graph-based clustering by integrating gene expression and spatial proximity
- Neighborhood-aware (Bayesian) feature aggregation on a spatial graph
- Final clustering via Leiden or Louvain (community detection)



*Vipul et al., BANKSY unifies cell typing and tissue domain segmentation for scalable spatial omics data analysis

Banksy results



*Vipul et al., BANKSY unifies cell typing and tissue domain segmentation for scalable spatial omics data analysis

Banksy - under the hood

How BANKSY works (step-by-step)

1. **Build a spatial graph**
   - Nodes = cells / spots
   - Edges connect nearby neighbors (kNN or radius-based)

2. **Compute neighborhood-aggregated features.** For each cell:
   - Self-expression
   - Neighborhood-averaged expression: Compute azimuthal Gabor responses from spatial coordinates - preserves direction of the tissue

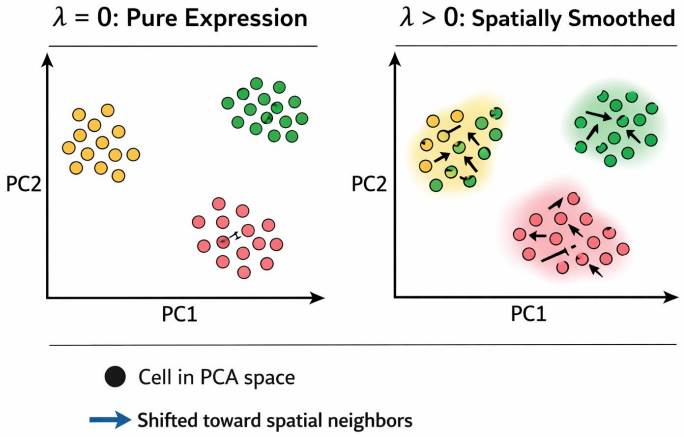
Bayesian smoothing

$X_{banksy} = (1 - \lambda) \cdot X_{self} + \lambda \cdot X_{neighbors}$ (posterior estimate)

   - Expression values are stabilized by borrowing information from neighbors
   - Reduces dropout and technical noise

1. **Dimensionality reduction**
   - PCA(X_{banksy}) on the BANKSY-transformed matrix

2. **Clustering**
   - Build a kNN graph in PCA space
   - Apply **Leiden (default)** or **Louvain**
   - Output: spatially coherent clusters



Core Idea

- Nearby cells should inform each other's expression profiles
  - Explicitly blends molecular similarity and spatial proximity
  - Clusters spatially enhanced expression, not raw counts
- Good use case: Tissues with clear spatial domains
  - Noisy or low-UMI spatial transcriptomics data
  - When biologically interpretable regions are desired

BANKSY

Building Aggregates with a Neighborhood Kernel and Spatial Yardstick

Three Ways to Run the Analysis

Method 1

banksy.py

Python Docker Wrapper

Method 2

banksy.R

Standalone R Script

Method 3

docker_run.sh

Bash + Docker

Method 1: banksy.py

Full-Featured Python Wrapper with Docker Backend

What It Does

Validates Docker availability: checks daemon, pulls image if needed

Validates input file: confirms .h5ad exists, checks file size

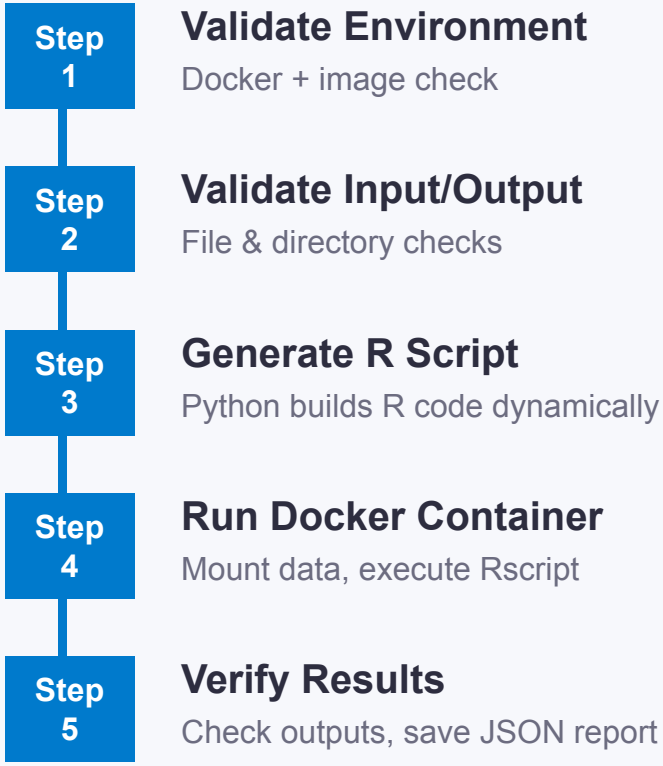
Auto-generates R script: builds BANKSY R code from Python parameters

Runs Docker container: mounts data, executes analysis, streams logs

Verifies outputs: checks all expected files were produced

Saves run metadata: JSON log with parameters, timing, and results

Architecture



Docker Image: almahmoud/bioc2024-banksy:manual

Key File: banksy/banksy.py (1,245 lines)

Method 1: banksy.py

Usage, Parameters & Output

Basic Usage

```
python banksy.py --input
data/sample.h5ad \
--output results/banksy
-lambda 0.8 --resolution 1.0 -v
```

Key Parameters

--input, -i: Path to input .h5ad file (required)

--output, -o: Output directory (auto-created)

--lambda: Spatial weight: 0.2=cell-typing, 0.8=domain [0.0, 0.2]

--k-geom: Neighborhood sizes [15, 30]

--resolution: Leiden clustering resolution [0.8]

--n-pcs: Principal components [20]

--verbose, -v: Enable debug logging

--timeout: Max execution seconds [3600]

Generated Outputs

- **banksy_results.rds**
  Full SpatialExperiment R object
- **_clusters.csv**
  Cluster IDs, QC metrics, coordinates
- **_summary.csv**
  Analysis parameters and cluster counts
- **_spatial.png**
  Spatial cluster maps per lambda
- **_UMAP.png**
  UMAP embeddings per lambda
- **banksy.log**
  Complete execution log
- **banksy_run_info.json**

Advantages

Best error handling and validation

Real-time log streaming to console

JSON metadata for pipeline integration

Method 2: banksy.R

Standalone R Script with Full CLI Interface

What It Does

Self-contained R script: no Python or wrapper dependencies needed

Full CLI argument parsing: built-in --help, flags, and validation

Runs directly with Rscript: works locally, in Docker, or on HPC

Structured logging: timestamped messages with INFO/WARNING/ERROR levels

Configurable QC: adaptive thresholds with user-tunable quantile cutoffs

Optional components: skip UMAP or plots with flags to save time

Analysis Pipeline

1. **Parse CLI args**
   Robust argument parsing with validation
2. **Load h5ad**
   rhdf5-based reading (sparse & dense support)
3. **Quality Control**
   Adaptive quantile-based cell/gene filtering
4. **Normalize**
   Library size normalization + log1p transform
5. **BANKSY**
   computeBanksy -> PCA -> UMAP -> clusterBanksy
6. **Save & Plot**
   RDS, CSV, spatial plots, UMAP plots

Required R packages: Banksy, SpatialExperiment, scuttle, scater, rhdf5, Matrix, ggplot2, data.table

Method 2: banksy.R

Usage, Parameters & Examples

Basic Usage

```
# Cell-typing (lambda=0.2)

Rscript banksy.R --input sample.h5ad \

--output results --lambda 0.2
```

Generated Outputs

***_banksy_results.rds:** Full R analysis object

***_clusters.csv:** Cell IDs, clusters, QC, coordinates

***_summary.csv:** Run parameters and cluster counts

***_spatial.png / *_UMAP.png:** Visualizations

.banksy_complete: Completion timestamp marker

Full Parameter Set

--input, -i: Input .h5ad file path (required)

--output, -o: Output directory (required)

--sample-name: Custom sample name [filename stem]

--lambda: Comma-separated lambda values [0.2,0.8]

--k-geom: Comma-separated k values [15,30]

--resolution, -r: Clustering resolution [0.8]

--n-pcs, -p: Number of PCs [20]

--qc-lower / --qc-upper: QC quantile thresholds [0.01/0.99]

--skip-plots / --skip-umap: Skip visualization steps

--seed: Random seed for reproducibility [42]

Method 3: docker_run.sh

Lightweight Bash Wrapper for Docker Execution

What It Does

Minimal bash script: ~80 lines, simple and transparent

Validates arguments: checks input file and R script exist

Creates output directory: mkdir -p for the output path

Mounts 3 volumes: input (ro), output (rw), R script (ro)

Runs Docker: executes Rscript inside the BANKSY container

Resolves paths: converts relative to absolute for Docker mounts

Docker Volume Mounts

Host: <input.h5ad> - > **/data/input.h5ad**

Mode: Read-only

Host: <output_dir> - > **/output**

Mode: Read-write

Host: <script.R> - > **/scripts/run_banksy.R**

Mode: Read-only

Method 3: docker_run.sh

Usage & Script Anatomy

```
docker run --rm \
--platform linux/amd64 \
  -v "$INPUT_H5AD:/data/input.h5ad:ro" \
  -v "$OUTPUT_DIR:/output" \
  -v "$BANKSY_SCRIPT:/scripts/run_banksy.R:ro" \
  almahmoud/bioc2024-banksy:manual \
  Rscript /scripts/run_banksy.R


# Flags explained:
#   --rm           Remove container after exit
#   --platform     Force amd64 (for Apple Silicon)
#   -v host:cont   Bind mount host path to container
#   :ro            Read-only mount
```

Comparison: Which Method to Use?

Feature Comparison Across All Three Approaches

Feature	banksy.py	banksy.R	docker_run.sh
Language	Python (calls Docker)	Pure R	Bash (calls Docker)
Docker Required	Yes	No (optional)	Yes
CLI Arguments	argparse (--flag)	Built-in (--flag)	Positional ($1 $2 $3)
R Script	Auto-generated	Self-contained	User-provided
Error Handling	Comprehensive	tryCatch + logging	set -euo pipefail
Validation	Docker + file + output	File + packages	File existence only
Logging	Python logger + file	Timestamped console	None (R script output)
Output Verification	Yes (JSON report)	Completion marker	No
Customization	CLI flags	CLI flags (most options)	Edit R script directly
Best For	Production pipelines	R users / HPC	Quick one-off runs

Recommendation: banksy.py for automated pipelines | banksy.R for interactive R workflows | docker_run.sh for simple, quick runs