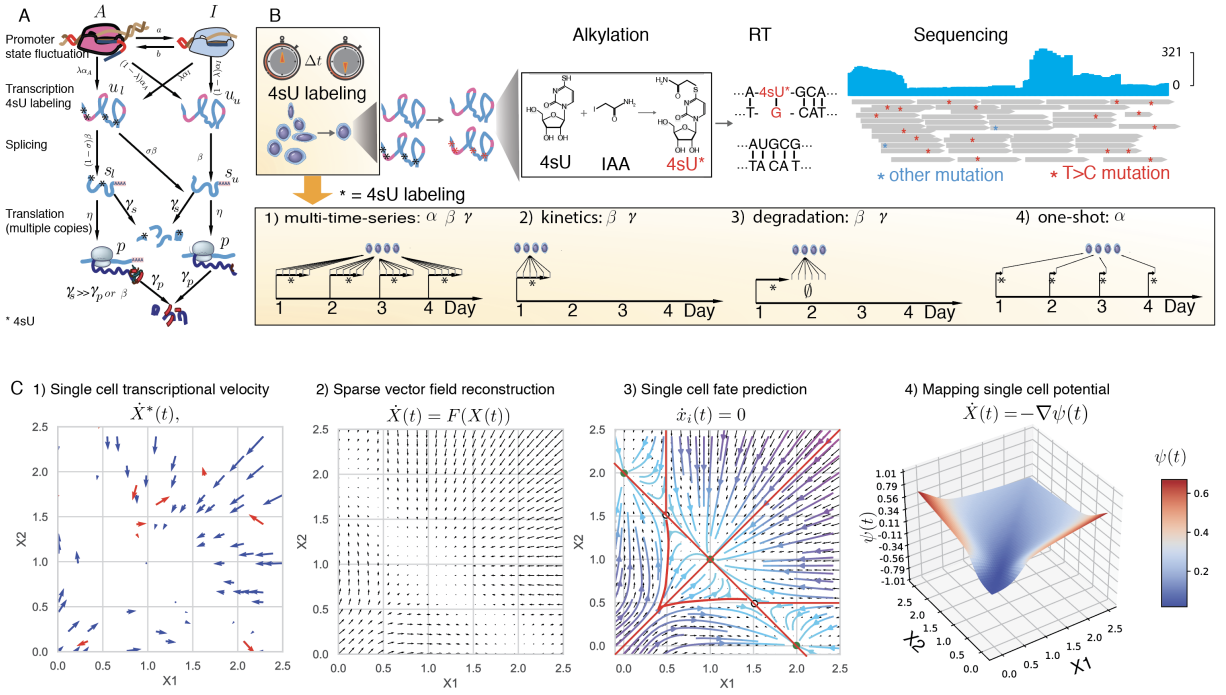
dynamo

Release 0.95.1

Xiaojie Qiu, Yan Zhang

Apr 14, 2021

CONTENTS



Understanding how gene expression in single cells progress over time is vital for revealing the mechanisms governing cell fate transitions. RNA velocity, which infers immediate changes in gene expression by comparing levels of new (unspliced) versus mature (spliced) transcripts (La Manno et al. 2018), represents an important advance to these efforts. A key question remaining is whether it is possible to predict the most probable cell state backward or forward over arbitrary time-scales. To this end, we introduce an inclusive model (termed Dynamo) capable of predicting cell states over extended time periods, that incorporates promoter state switching, transcription, splicing, translation and RNA/protein degradation by taking advantage of scRNA-seq and the co-assay of transcriptome and proteome. We also implement scSLAM-seq by extending SLAM-seq to plate-based scRNA-seq (Hendriks et al. 2018; Erhard et al. 2019; Cao, Zhou, et al. 2019) and augment the model by explicitly incorporating the metabolic labelling of nascent RNA. We show that through careful design of labelling experiments and an efficient mathematical framework, the entire kinetic behavior of a cell from this model can be robustly and accurately inferred. Aided by the improved framework, we show that it is possible to analytically reconstruct the transcriptomic vector field from sparse and noisy vector samples generated by single cell experiments. The analytically reconstructed vector further enables global mapping of potential landscapes that reflects the relative stability of a given cell state, and the minimal transition time and most probable paths between any cell states in the state space This work thus foreshadows the possibility of predicting long-term trajectories of cells during a dynamic process instead of short time velocity estimates. Our methods are implemented as an open source tool, dynamo.

CHAPTER

ONE

DISCUSSION

Please use github issue tracker to report coding related issues of dynamo. For community discussion of novel usage cases, analysis tips and biological interpretations of dynamo, please join our public slack workspace: dynamo-discussion (Only a working email address is required from the slack side).

CHAPTER

TWO

CONTRIBUTION

If you want to contribute to the development of dynamo, please check out CONTRIBUTION instruction: Contribution

2.1 10 minutes to dynamo

Welcome to dynamo!

Dynamo is a computational framework that includes an inclusive model of expression dynamics with scSLAM-seq / multiomics, vector field reconstruction and potential landscape mapping.

2.1.1 Why dynamo

Dynamo currently provides a complete solution (see below) to analyze expression dynamics of conventional scRNA-seq or time-resolved metabolic labeling based scRNA-seq. It aspires to become the leading tools in continuously integrating the most exciting developments in machine learning, systems biology, information theory, stochastic physics, etc. to model, understand and interpret datasets generated from various cutting-edge single cell genomics techniques (developments of dynamo 2/3 is under way). We hope those models, understandings and interpretations not only facilitate your research but may also eventually lead to new biological discovery. Dynamo has a strong community so you will feel supported no matter you are a new-comer of computational biology or a veteran researcher who wants to contribute to dynamo's development.

2.1.2 How to install

Dynamo requires Python 3.6 or later.

Dynamo now has been released to PyPi, you can install the PyPi version via:

```
pip install dynamo-release
```

To install the newest version of dynamo, you can git clone our repo and then pip install:

```
git clone https://github.com/aristoteleo/dynamo-release.git
pip install dynamo-release/ --user
```

Note that `--user` flag is used to install the package to your home directory, in case you don't have the root privilege.

Alternatively, you can install dynamo when you are in the dynamo-release folder by directly using python's setup install:

```
git clone https://github.com/aristoteleo/dynamo-release.git
cd dynamo-release/
python setup.py install --user
```

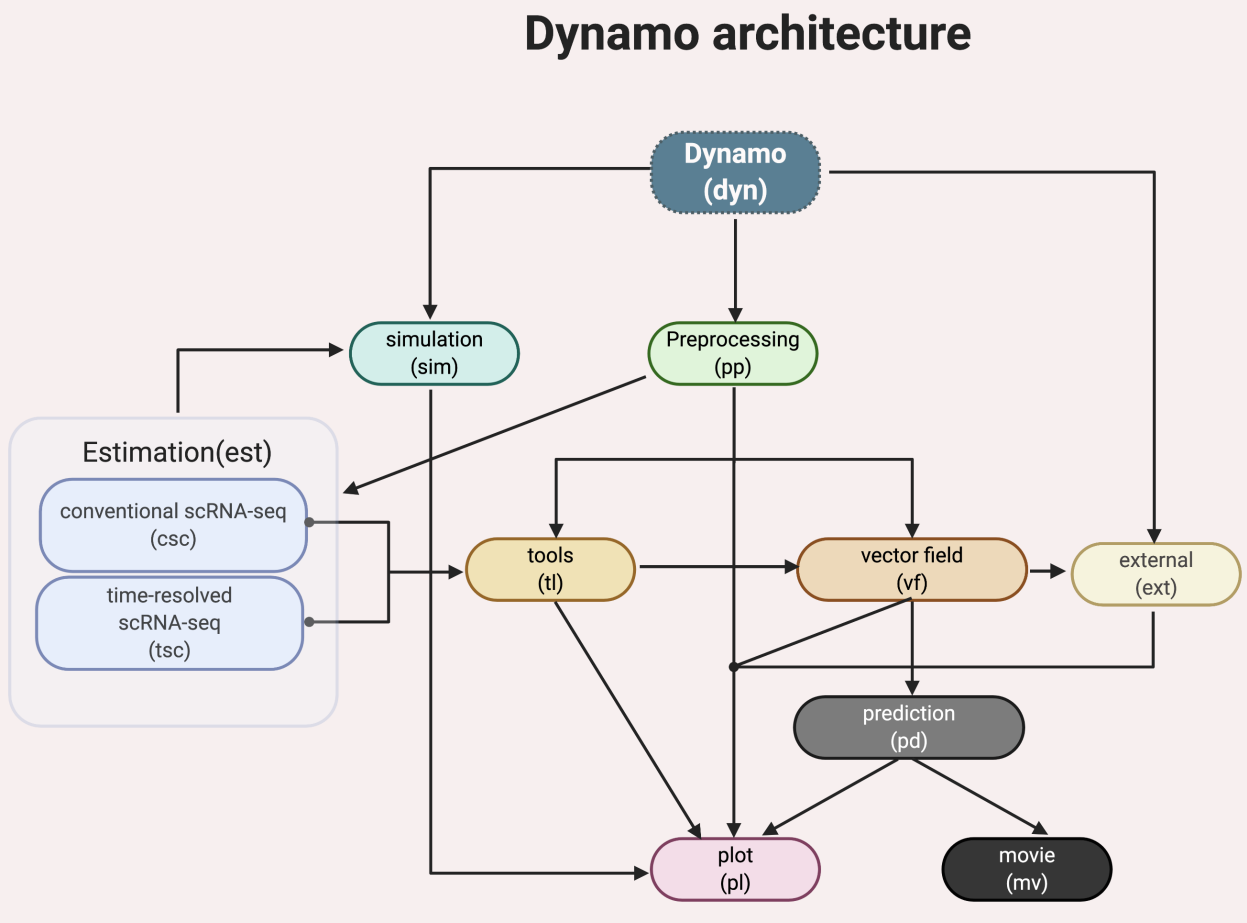
from source, using the following script:

```
pip install git+https://github.com:aristoteleo/dynamo-release
```

In order to ensure dynamo run properly, your python environment needs to satisfy dynamo's dependencies. We provide a helper function for you to check the versions of dynamo's all dependencies.

```
import dynamo as dyn
dyn.get_all_dependencies_version()
```

2.1.3 Architecture of dynamo



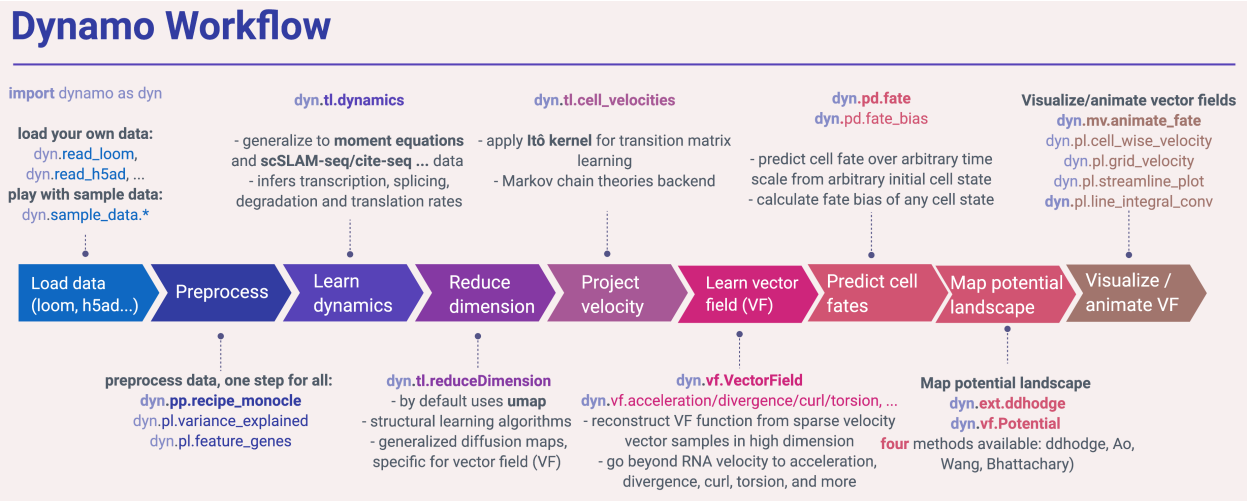
Dynamo has a few standard modules like most other single cell analysis toolkits (Scanpy, Monocle or Seurat), for example, data loading (`dyn.read*`), preprocessing (`dyn.pp.*`), tool analysis (`dyn.tl.*`), and plotting (`dyn.pl.*`). Modules specific to dynamo include:

- **a comprehensive estimation framework (`dyn.est.*`) of expression dynamics that includes:**
  - conventional single cell RNA-seq (scRNA-seq) modeling (`dyn.est.csc.*`) for **standard RNA velocity estimation** and more;

  - time-resolved metabolic labeling based single cell RNA-seq (scRNA-seq) modeling (`dyn.est.tsc.*`) for **labeling based RNA velocity estimation** and more;

- vector field reconstruction and vector calculus (`dyn.vf.*`);
- cell fate prediction (`dyn.pd.*`);
- create movie of cell fate predictions (`dyn.mv.*`);
- stochastic simulation of various metabolic labeling experiments (`dyn.sim.*`);
- integration with external tools built by us or others (`dyn.ext.*`);
- and more.

2.1.4 Typical workflow



A typical workflow in dynamo is similar to most of other single cell analysis toolkits (Scanpy, Monocle or Seurat), including steps like importing dynamo (`import dynamo as dyn`), loading data (`dyn.read*`), preprocessing (`dyn.pp.*`), tool analysis (`dyn.tl.*`) and plotting (`dyn.pl.*`). To get the best of dynamo though, you need to use the `dyn.vf.*`, `dyn.pd.*` and `dyn.mv.*` modules.

Import dynamo as:

```
import dynamo as dyn
```

We provide a few nice visualization defaults for different purpose:

```
dyn.configuration.set_figure_params('dynamo', background='white') # jupter notebooks
dyn.configuration.set_figure_params('dynamo', background='black') # presentation
dyn.configuration.set_pub_style() # manuscript
```

Load data

Dynamo relies on anndata for data IO. You can read your own data via `read`, `read_loom`, `read_h5ad`, `read_h5` or `load_NASC_seq`, etc:

```
adata = dyn.read(filename)
```

Dynamo also comes with a few builtin sample datasets so you can familiarize with dynamo before analyzing your own dataset. For example, you can load the Dentate Gyrus example dataset:

```
adata = dyn.sample_data.DentateGyrus()
```

There are many sample datasets available. You can check other available datasets via `dyn.sample_data.*`.

To process the scSLAM-seq data, please refer to the NASC-seq analysis pipeline. We are also working on a command line tool for this and will release it in due time. For processing splicing data, you can either use the velocyto command line interface or the bustool from Pachter lab.

Preprocess data

After loading data, you are ready to performs some preprocessing. You can run the `recipe_monocle` function that uses similar but generalized strategy from Monocle 3 to normalize all datasets in different layers (the spliced and unspliced or new, i.e. metabolic labelled, and total mRNAs or others), followed by feature selection and PCA dimension reduction.

```
dyn.pp.recipe_monocle(adata)
```

Learn dynamics

Next you will want to estimate the kinetic parameters of expression dynamics and then learn the velocity values for all genes that pass some filters (selected feature genes, by default) across cells. The `dyn.tl.dynamics` does all the hard work for you:

```
dyn.tl.dynamics(adata)
```

implicitly calls `dyn.tl.moments` first

```
dyn.tl.moments(adata)
```

which calculates the first, second moments (and sometimes covariance between different layers) of the expression data. First / second moments are basically mean and uncentered variance of gene expression, which are calculated based on local smoothing via a nearest neighbours graph, constructed in the reduced PCA space from the spliced or total mRNA expression of single cells.

And it then performs the following steps:

- checks the data you have and determine the experimental type automatically, either the conventional scRNA-seq, `kinetics`, `degradation` or `one-shot` single-cell metabolic labelling experiment or the `CITE-seq` or `REAP-seq` co-assay, etc.
- learns the velocity for each feature gene using either the original deterministic model based on a steady-state assumption from the seminal RNA velocity work or a few new methods, including the `stochastic` (default) or `negative binomial method` for conventional scRNA-seq or `kinetic`, `degradation` or `one-shot` models for metabolic labeling based scRNA-seq.

Those later methods are based on moment equations. All those methods use all or part of the output from `dyn.tl.moments(adata)`.

Kinetic estimation of the conventional scRNA-seq and metabolic labeling based scRNA-seq is often tricky and has a lot pitfalls. Sometimes you may even observed undesired backward vector flow. You can evaluate the confidence of gene-wise velocity via:

```
dyn.tl.gene_wise_confidence(adata, group='group', lineage_dict={'Progenitor': [
↪'terminal_cell_state']})
```

and filter those low confidence genes for downstream *Velocity vectors* analysis, etc (See more details in FAQ).

Dimension reduction

By default, we use `umap` algorithm for dimension reduction.

```
dyn.tl.reduceDimension(adata)
```

If the requested reduced dimension is already existed, dynamo won't touch it unless you set `enforce=True`.

```
dyn.tl.reduceDimension(adata, basis='umap', enforce=True)
```

Velocity vectors

We need to project the velocity vector onto low dimensional embedding for later visualization. To get there, we can either use the default `correlation/cosine kernel` or the novel Itô kernel from us.

```
dyn.tl.cell_velocities(adata)
```

The above function projects and evaluates velocity vectors on `umap` space but you can also operate them on other basis, for example `pca` space:

```
dyn.tl.cell_velocities(adata, basis='pca')
```

You can check the confidence of cell-wise velocity to understand how reliable the recovered velocity is across cells via:

```
dyn.tl.cell_wise_confidence(adata)
```

Obviously dynamo doesn't stop here. The really exciting part of dynamo lays in the fact that it learns a `functional form of vector field` in the full transcriptomic space which can be then used to predict cell fate and map single cell potential landscape.

Vector field reconstruction

In classical physics, including fluidics and aerodynamics, velocity and acceleration vector fields are used as fundamental tools to describe motion or external force of objects, respectively. In analogy, RNA velocity or protein accelerations estimated from single cells can be regarded as sparse samples in the velocity (La Manno et al. 2018) or acceleration vector field (Gorin, Svensson, and Pachter 2019) that defined on the gene expression space.

In general, a vector field can be defined as a vector-valued function f that maps any points (or cells' expression state) x in a domain with D dimension (or the gene expression system with D transcripts / proteins) to a vector y (for example, the velocity or acceleration for different genes or proteins), that is f(x) = y.

To formally define the problem of velocity vector field learning, we consider a set of measured cells with pairs of current and estimated future expression states. The difference between the predicted future state and current state for each cell corresponds to the velocity vector. We note that the measured single-cell velocity (conventional RNA velocity) is sampled from a smooth, differentiable vector field f that maps from xi to yi on the entire domain. Normally, single cell velocity measurements are results of biased, noisy and sparse sampling of the entire state space, thus the goal of velocity vector field reconstruction is to robustly learn a mapping function f that outputs yj given any point xj on the domain based on the observed data with certain smoothness constraints (Jiayi Ma et al. 2013). Under ideal scenario, the mapping function f should recover the true velocity vector field on the entire domain and predict the true dynamics in regions of expression space that are not sampled. To reconstruct vector field function in dynamo, you can simply use the following function to do all the heavy-lifting:

```
dyn.vf.VectorField(adata)
```

By default, it learns the vector field in the *pca* space but you can of course learn it in any space or even the original gene expression space.

Characterize vector field topology

Since we learn the vector field function of the data, we can then characterize the topology of the full vector field space. For example, we are able to identify

- the fixed points (attractor/saddles, etc.) which may corresponds to terminal cell types or progenitors;
- nullcline, separatrices of a recovered dynamic system, which may formally define the dynamical behaviour or the boundary of cell types in gene expression space.

Again, you only need to simply run the following function to get all those information.

```
dyn.vf.topography(adata, basis='umap')
```

Map potential landscape

The concept of potential landscape is widely appreciated across various biological disciplines, for example the adaptive landscape in population genetics, protein-folding funnel landscape in biochemistry, epigenetic landscape in developmental biology. In the context of cell fate transition, for example, differentiation, carcinogenesis, etc, a potential landscape will not only offers an intuitive description of the global dynamics of the biological process but also provides key insights to understand the multi-stability and transition rate between different cell types as well as to quantify the optimal path of cell fate transition.

Because the classical definition of potential function in physics requires gradient systems (no `curl` or cycling dynamics), which is often not applicable to open biological system. In dynamo we provided several ways to quantify the potential of single cells by decomposing the vector field into gradient, curl parts, etc. The recommended method is built on the Hodge decomposition on simplicial complexes (a sparse directional graph) constructed based on the learned vector field function that provides fruitful analogy of gradient, curl and harmonic (cyclic) flows on manifold:

```
dyn.ext.ddhodge(adata)
```

In addition, we and others proposed various strategies to decompose the `stochastic differential equations` into either the gradient or the curl component from first principles. We then can use the gradient part to define the potential.

Although an analytical decomposition on the reconstructed vector field is challenging, we are able to use a numerical algorithm we recently developed for our purpose. This approach uses a least action method under the A-type stochastic integration (Shi et al. 2012) to globally map the potential landscape (x) (Tang et al. 2017) by taking the vector field function f(x) as input.

```
dyn.vf.Potential(adata)
```

Visualization

In two or three dimensions, a streamline plot can be used to visualize the paths of cells will follow if released in different regions of the gene expression state space under a steady flow field. Although we currently do not support this, for vector field that changes over time, similar methods, for example, streakline, pathline, timeline, etc. can also be used to visualize the evolution of single cell or cell populations.

In dynamo, we have three standard visual representations of vector fields, including the `cell wise`, `grid` quiver plots and the `streamline plot`. Another intuitive way to visualize the structure of vector field is the so called line integral convolution method or LIC (Cabral and Leedom 1993), which works by adding random black-and-white paint sources on the vector field and letting the flowing particles on the vector field picking up some texture to ensure points on the same streamline having similar intensity. We relies on the yt's `annotate_line_integral_convolution` function to visualize the LIC vector field reconstructed from dynamo.

```
dyn.pl.cell_wise_vectors(adata, color=colors, ncols=3)
dyn.pl.grid_vectors(adata, color=colors, ncols=3)
dyn.pl.stremline_plot(adata, color=colors, ncols=3)
dyn.pl.line_integral_conv(adata)
```

Note that `colors` here is a list or str that can be either the column name in `.obs` or `gene names`.

To visualize the topological structure of the reconstructed 2D vector fields, we provide the `dyn.pl.topography` function in dynamo.

```
dyn.vf.VectorField(adata, basis='umap')
dyn.pl.topography(adata)
```

Plotting functions in dynamo are designed to be extremely flexible. For example, you can combine different types of dynamo plots together (when you visualize only one item for each plot function)

```
import matplotlib.pyplot as plt
fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True, figsize=(12,␣
↪10))
f1_axes
f1_axes[0, 0] = dyn.pl.cell_wise_vectors(adata, color='umap_ddhodge_potential',␣
↪pointsize=0.1, alpha = 0.7, ax=f1_axes[0, 0], quiver_length=6, quiver_size=6, save_
↪show_or_return='return')
f1_axes[0, 1] = dyn.pl.grid_vectors(adata, color='speed_umap', ax=f1_axes[0, 1],␣
↪quiver_length=12, quiver_size=12, save_show_or_return='return')
f1_axes[1, 0] = dyn.pl.streamline_plot(adata, color='divergence_pca', ax=f1_axes[1,␣
↪0], save_show_or_return='return')
f1_axes[1, 1] = dyn.pl.topography(adata, color='acceleration_umap', ax=f1_axes[1, 1],␣
↪save_show_or_return='return')
plt.show()
```

The above creates a 2x2 plot that puts *cell_wise_vectors*, *grid_vectors*, *streamline_plot* and *topography* plots together.

2.1.5 Compatibility

Dynamo is fully compatible with velocyto, scanpy and scvelo. So you can use your loom or annadata object as input for dynamo. The velocity vector samples estimated from either velocyto or scvelo can be also directly used to reconstruct the functional form of vector field and to map the potential landscape in the entire expression space. Mapping Vector Field of Single Cells

2.2 API

Import dynamo as:

```
import dynamo as dyn
```

2.2.1 Data IO

(See more at anndata-docs)

read(filename[, backed, as_sparse, ...])	Read *.h5ad*-formatted hdf5 file.
read_h5ad(filename[, backed, as_sparse, ...])	Read *.h5ad*-formatted hdf5 file.
read_loom(filename, *[, sparse, cleanup, ...])	Read *.loom*-formatted hdf5 file.

dynamo.read

dynamo.**read**(*filename*, *backed=None*, ***, *as_sparse=()*, *as_sparse_fmt=<class 'scipy.sparse.csr.csr_matrix'>*, *chunk_size=6000*)

Read *.h5ad*-formatted hdf5 file.

Parameters

- **filename** (Union[str, Path]) – File name of data file.
- **backed** (Union[Literal['r', 'r+'], bool, None]) – If *'r'*, load AnnData in *backed* mode instead of fully loading it into memory (*memory* mode). If you want to modify backed attributes of the AnnData object, you need to choose *'r+'*.
- **as_sparse** (Sequence[str]) – If an array was saved as dense, passing its name here will read it as a sparse_matrix, by chunk of size *chunk_size*.
- **as_sparse_fmt** (Type[spmatrix]) – Sparse format class to read elements from *as_sparse* in as.
- **chunk_size** (int) – Used only when loading sparse dataset that is stored as dense. Loading iterates through chunks of the dataset of this row size until it reads the whole dataset. Higher size means higher memory consumption and higher (to a point) loading speed.

Return type AnnData

dynamo.read_h5ad

dynamo.**read_h5ad**(*filename*, *backed=None*, ***, *as_sparse=()*, *as_sparse_fmt=<class 'scipy.sparse.csr.csr_matrix'>*, *chunk_size=6000*)

Read *.h5ad*-formatted hdf5 file.

Parameters

- **filename** (Union[str, Path]) – File name of data file.
- **backed** (Union[Literal['r', 'r+'], bool, None]) – If *'r'*, load AnnData in *backed* mode instead of fully loading it into memory (*memory* mode). If you want to modify backed attributes of the AnnData object, you need to choose *'r+'*.
- **as_sparse** (Sequence[str]) – If an array was saved as dense, passing its name here will read it as a sparse_matrix, by chunk of size *chunk_size*.
- **as_sparse_fmt** (Type[spmatrix]) – Sparse format class to read elements from *as_sparse* in as.
- **chunk_size** (int) – Used only when loading sparse dataset that is stored as dense. Loading iterates through chunks of the dataset of this row size until it reads the whole dataset. Higher size means higher memory consumption and higher (to a point) loading speed.

Return type AnnData

dynamo.read_loom

dynamo.**read_loom**(*filename*, ***, *sparse=True*, *cleanup=False*, *X_name='spliced'*, *obs_names='CellID'*, *obsm_names=None*, *var_names='Gene'*, *varm_names=None*, *dtype='float32'*, *obsm_mapping=mappingproxy({})*, *varm_mapping=mappingproxy({})*, ***kwargs*)

Read *.loom*-formatted hdf5 file.

This reads the whole file into memory.

Beware that you have to explicitly state when you want to read the file as sparse data.

Parameters

- **filename** (PathLike) – The filename.
- **sparse** (bool) – Whether to read the data matrix as sparse.
- **cleanup** (bool) – Whether to collapse all obs/var fields that only store one unique value into *.uns['loom-.']*.
- **X_name** (str) – Loompy key with which the data matrix X is initialized.
- **obs_names** (str) – Loompy key where the observation/cell names are stored.
- **obsm_mapping** (Mapping[str, Iterable[str]]) – Loompy keys which will be constructed into observation matrices
- **var_names** (str) – Loompy key where the variable/gene names are stored.
- **varm_mapping** (Mapping[str, Iterable[str]]) – Loompy keys which will be constructed into variable matrices
- ****kwargs** – Arguments to loompy.connect

Example

```
pbmc = anndata.read_loom(
    "pbmc.loom",
    sparse=True,
    X_name="lognorm",
    obs_names="cell_names",
    var_names="gene_names",
    obsm_mapping={
        "X_umap": ["umap_1", "umap_2"]
    }
)
```

Return type AnnData

2.2.2 Preprocessing (pp)

pp.recipe_monocle(adata[, normalized, ...])	This function is partly based on Monocle R package (https://github.com/cole-trapnell-lab/monocle3).
pp.cell_cycle_scores(adata[, layer, ...])	Call cell cycle positions for cells within the population.

2.2.3 Estimation (est)

Note: Classes in **est** are internally to **Tools**. See our estimation classes here: estimation

2.2.4 Tools (tl)

kNN and moments of expressions

tl.neighbors(adata[, X_data, genes, basis, ...])	Function to search nearest neighbors of the adata object.
tl.mnn(adata[, n_pca_components, ...])	Function to calculate mutual nearest neighbor graph across specific data layers.
tl.moments(adata[, genes, group, ...])	Calculate kNN based first and second moments (including uncentered covariance) for

Kinetic parameters and RNA/protein velocity

tl.dynamics(adata[, tkey, t_label_keys, ...])	Inclusive model of expression dynamics considers splicing, metabolic labeling and protein translation.

Dimension reduction

tl.reduceDimension(adata[, X_data, genes, ...])	Compute a low dimension reduction projection of an annodata object first with PCA, followed by non-linear dimension reduction methods

continues on next page

Table 5 – continued from previous page

tl.DDRTree(X, maxIter, sigma, gamma[, eps, ...])	This function is a pure Python implementation of the DDRTree algorithm.
tl.psl(Y[, sG, dist, K, C, param_gamma, d, ...])	This function is a pure Python implementation of the PSL algorithm.

Clustering

tl.hdbscan(adata[, X_data, genes, layer, ...])	Apply hdbscan to cluster cells in the space defined by basis.
tl.cluster_field(adata[, basis, ...])	Cluster cells based on vector field features.

Velocity projection

tl.cell_velocities(adata[, ekey, vkey, X, ...])	Compute transition probability and project high dimension velocity vector to existing low dimension embedding.
tl.confident_cell_velocities(adata, group, ...)	Confidently compute transition probability and project high dimension velocity vector to existing low dimension embeddings using progeintors and mature cell groups priors.

Velocity metrics

tl.cell_wise_confidence(adata[, X_data, ...])	Calculate the cell-wise velocity confidence metric.
tl.gene_wise_confidence(adata, group[, ...])	Diagnostic measure to identify genes contributed to "wrong" directionality of the vector flow.

Markov chain

tl.generalized_diffusion_map(adata, **kwargs)	Apply the diffusion map algorithm on the transition matrix build from Itô kernel.
tl.stationary_distribution(adata[, method, ...])	Compute stationary distribution of cells using the transition matrix.
tl.diffusion(M[, P0, steps, backward])	Find the state distribution of a Markov process.
tl.expected_return_time(M[, backward])	Find the expected returning time.

Markers and differential expressions

tl.moran_i(adata[, X_data, genes, layer, ...])	Identify genes with strong spatial autocorrelation with Moran's I test.
tl.find_group_markers(adata, group[, genes, ...])	Find marker genes for each group of cells based on gene expression or velocity values as specified by the layer.
tl.two_groups_degs(adata, genes, layer, ...)	Find marker genes between two groups of cells based on gene expression or velocity values as specified by the layer.
tl.top_n_markers(adata[, with_moran_i, ...])	Filter cluster deg (Moran's I test) results and retrieve top markers for each cluster.

continues on next page

Table 10 – continued from previous page

tl.glm_degs(adata[, X_data, genes, layer, ...])	Differential genes expression tests using generalized linear regressions.

Converter

tl.converter(data_in[, from_type, to_type, dir])	convert adata to loom object - we may save_fig to a temp directory automatically - we may write a on-the-fly converter which doesn't involve saving and reading files

2.2.5 Vector field (vf)

Vector field reconstruction

Note: Vector field class is internally to *vf.VectorField*. See our vector field classes here: vector field

vf.SparseVFC(X, Y, Grid[, M, a, beta, ecr, ...])	Apply sparseVFC (vector field consensus) algorithm to learn a functional form of the vector field from random samples with outlier on the entire space robustly and efficiently.
vf.VectorField(adata[, basis, layer, dims, ...])	Learn a function of high dimensional vector field from sparse single cell samples in the entire space robustly.

Vector field topology

vf.topography(adata[, basis, layer, X, ...])	Map the topography of the single cell vector field in (first) two dimensions.

Beyond RNA velocity

vf.speed(adata[, basis, VecFld, method])	Calculate the speed for each cell with the reconstructed vector field function.
vf.divergence(adata[, cell_idx, sampling, ...])	Calculate divergence for each cell with the reconstructed vector field function.
vf.curl(adata[, basis, vector_field_class])	Calculate Curl for each cell with the reconstructed vector field function.
vf.acceleration(adata[, basis, ...])	Calculate acceleration for each cell with the reconstructed vector field function.
vf.curvature(adata[, basis, vector_field_class])	Calculate curvature for each cell with the reconstructed vector field function.
vf.torsion(adata[, basis, vector_field_class])	Calculate torsion for each cell with the reconstructed vector field function.

Acceleration field

vf.cell_accelerations(adata[, vf_basis, ...])	Compute RNA acceleration field via reconstructed vector field and project it to low dimensional embeddings.

Vector field ranking

vf.rank_speed_genes(adata[, group, genes, vkey])	Rank gene's absolute, positive, negative speed by different cell groups.
vf.rank_divergence_genes(adata[, group, ...])	Rank gene's absolute, positive, negative divergence by different cell groups.
vf.rank_acceleration_genes(adata[, group, ...])	Rank gene's absolute, positive, negative acceleration by different cell groups.
vf.rank_curvature_genes(adata[, group, ...])	Rank gene's absolute, positive, negative curvature by different cell groups.

Single cell potential: three approaches

vf.gen_fixed_points(func, auto_func, ...[, ...])	Calculate the fixed points of (learned) vector field function .
vf.gen_gradient(dim, N, Function, ...)	Calculate the gradient of the (learned) vector field function for the least action path (LAP) symbolically
vf.IntGrad(points, Function, DiffusionMatrix, dt)	Calculate the action of the path based on the (reconstructed) vector field function and diffusion matrix (Eq.
vf.DiffusionMatrix(x)	Diffusion matrix can be variable dependent
vf.action(n_points, tmax, point_start, ...)	It calculates the minimized action value given an intial path, ODE, and diffusion matrix.
vf.Potential(adata[, DiffMat, method])	Function to map out the pseudo-potential landscape.
vf.path_integral(VecFnc, x_lim, y_lim, ...)	A deterministic map of Waddington's epigenetic landscape for cell fate specification Sudin Bhattacharya, Qiang Zhang and Melvin E.
vf.alignment(numPaths, numTimeSteps, ...[, ...])	Align potential values so all path-potentials end up at same global min and then generate potential surface with interpolation on a grid.
vf.Wang_action(X_input, F, D, dim, N[, lamada_])	Calculate action by path integral by Wang's method.
vf.Wang_LAP(F, n_points, point_start, point_end)	Calculating least action path based methods from Jin Wang and colleagues (http://www.pnas.org/cgi/doi/10.1073/pnas.1017017108)
vf.transition_rate(X_input, F[, D, lambda_])	Calculate the rate to convert from one cell state to another cell state by taking the optimal path.
vf.MFPT(X_input, F[, D, lambda_])	Calculate the MFPT (mean first passage time) to convert from one cell state to another cell state by taking the optimal path.
vf.Ao_pot_map(vecFunc, X[, D])	Mapping potential landscape with the algorithm developed by Ao method.
vf.solveQ(D, F[, debug])	Function to calculate Q matrix by a least square method

Stochastic processes

vf.diffusionMatrix(adata[, X_data, V_data, ...])	"Calculate the diffusion matrix from the estimated velocity vector and the reconstructed vector field.

Vector field graph

vf.vfGraph(*args, **kwds)	A class for manipulating the graph creating from the transition matrix, built from the (reconstructed) vector field.

2.2.6 Prediction (pd)

pd.fate(adata, init_cells[, init_states, ...])	Predict the historical and future cell transcriptomic states over arbitrary time scales.
pd.fate_bias(adata, group[, basis, inds, ...])	Calculate the lineage (fate) bias of states whose trajectory are predicted.
pd.state_graph(adata, group[, approx, ...])	Estimate the transition probability between cell types using method of vector field integrations.

dynamo.pd.fate

dynamo.pd.**fate**(*adata*, *init_cells*, *init_states=None*, *basis=None*, *layer='X'*, *dims=None*, *genes=None*, *t_end=None*, *direction='both'*, *interpolation_num=250*, *average=False*, *sampling='arc_length'*, *VecFld_true=None*, *inverse_transform=False*, *scale=1*, *cores=1*, ***kwargs*)

Predict the historical and future cell transcriptomic states over arbitrary time scales.

This is achieved by integrating the reconstructed vector field function from one or a set of initial cell state(s). Note that this function is designed so that there is only one trajectory (based on averaged cell states if multiple initial states are provided) will be returned. *dyn.tl._fate* can be used to calculate multiple cell states.

Parameters

- **adata** (AnnData) – AnnData object that contains the reconstructed vector field function in the *uns* attribute.
- **init_cells** (*list* (default: None)) – Cell name or indices of the initial cell states for the historical or future cell state prediction with numerical integration. If the names in init_cells are not find in the adata.obs_name, it will be treated as cell indices and must be integers.
- **init_states** (*numpy.ndarray* or None (default: None)) – Initial cell states for the historical or future cell state prediction with numerical integration.
- **basis** (*str* or None (default: *None*)) – The embedding data to use for predicting cell fate. If *basis* is either *umap* or *pca*, the reconstructed trajectory will be projected back to high dimensional space via the *inverse_transform* function.
- **layer** (*str* or None (default: 'X')) – Which layer of the data will be used for predicting cell fate with the reconstructed vector field function. The layer once provided, will override the *basis* argument and then predicting cell fate in high dimensional space.
- **genes** (*list* or None (default: None)) – The gene names whose gene expression will be used for predicting cell fate. By default (when genes is set to None), the genes used for velocity embedding (var.use_for_velocity) will be used for vector field reconstruction. Note that the genes to be used need to have velocity calculated and corresponds to those used in the *dyn.tl.VectorField* function.
- **t_end** (*float* (default None)) – The length of the time period from which to predict cell state forward or backward over time. This is used by the odeint function.

- **direction** (*string* (default: both)) – The direction to predict the cell fate. One of the *forward*, *backward* or *both* string.
- **interpolation_num** (*int* (default: 100)) – The number of uniformly interpolated time points.
- **average** (*str* or *bool* (default: *False*) {'origin', 'trajectory'}) – The method to calculate the average cell state at each time step, can be one of *origin* or *trajectory*. If *origin* used, the average expression state from the init_cells will be calculated and the fate prediction is based on this state. If *trajectory* used, the average expression states of all cells predicted from the vector field function at each time point will be used. If *average* is *False*, no averaging will be applied.
- **sampling** (*str* (default: *arc_length*)) – Methods to sample points along the integration path, one of *{'arc_length', 'logspace', 'uniform_indices'}*. If *logspace*, we will sample time points linearly on log space. If *uniform_indices*, the sorted unique set of all time points from all cell states' fate prediction will be used and then evenly sampled up to *interpolation_num* time points. If *arc_length*, we will sample the integration path with uniform arc length.
- **VecFld_true** (*function*) – The true ODE function, useful when the data is generated through simulation. Replace VecFld arugment when this has been set.
- **inverse_transform** (*bool* (default: *False*)) – Whether to inverse transform the low dimensional vector field prediction back to high dimensional space.
- **scale** (*float* (default: *1*)) – The value that will be used to scale the predicted velocity value from the reconstructed vector field function.
- **cores** (*int* (default: 1):) – Number of cores to calculate path integral for predicting cell fate. If cores is set to be > 1, multiprocessing will be used to parallel the fate prediction.
- **kwargs** – Additional parameters that will be passed into the fate function.

Returns **adata** – AnnData object that is updated with the dictionary Fate (includes *t* and *prediction* keys) in uns attribute.

Return type AnnData

dynamo.pd.fate_bias

dynamo.pd.**fate_bias**(*adata*, *group*, *basis='umap'*, *inds=None*, *speed_percentile=5*, *dist_threshold=None*, *source_groups=None*, *metric='euclidean'*, *metric_kwds=None*, *cores=1*, *seed=19491001*, ***kwargs*)

Calculate the lineage (fate) bias of states whose trajectory are predicted.

Fate bias is currently calculated as the percentage of points along the predicted cell fate trajectory whose distance to their 0-th nearest neighbors on the data are close enough (determined by median 1-st nearest neighbors of all observed cells and the dist_threshold) to any cell from each group specified by *group* key. The details is described as following:

Cell fate predicted by our vector field method sometimes end up in regions that are not sampled with cells. We thus developed a heuristic method to iteratively walk backward the integration path to assign cell fate. We first identify the regions with small velocity in the tail of the integration path (determined by *speed_percentile*), then we check whether the distance of 0-th nearest points on the observed data to all those points are far away from the observed data (determined by *dist_threshold*). If they are not all close to data, we then walk backwards along the trajectory by one time step until the distance of any currently visited integration path's data points' 0-th nearest points to the observed cells is close enough. In order to calculate the cell fate probability, we diffuse one step further of the identified nearest neighbors from the integration to identify more nearest observed cells, especially

those from terminal cell types in case nearby cells first identified are all close to some random progenitor cells. Then we use group information of those observed cells to define the fate probability.

fate_bias **calculate a confidence score for the calculated fate probability with a simple metric, defined as** $1 - (sum(distances > dist_threshold * median_dist) + walk_back_steps)/(len(indices) + walk_back_steps)$

The *distance* is currently visited integration path's data points' 0-th nearest points to the observed cells. *median_dist* is median distance of their 1-st nearest cell distance of all observed cells. *walk_back_steps* is the steps walked backward along the integration path until all currently visited integration points's 0-th nearest points to the observed cells satisfy the distance threshold. *indices* are the time indices of integration points that is regarded as the regions with *small velocity* (note when walking backward, those corresponding points are not necessarily have small velocity anymore).

Parameters

- **adata** (AnnData) – AnnData object that contains the predicted fate trajectories in the *uns* attribute.
- **group** (*str*) – The column key that corresponds to the cell type or other group information for quantifying the bias of cell state.
- **basis** (*str* or None (default: *None*)) – The embedding data space where cell fates were predicted and cell fates bias will be quantified.
- **list or float or None (default** (*inds*) – The indices of the time steps that will be used for calculating fate bias. If inds is None, the last a few steps of the fate prediction based on the *sink_speed_percentile* will be use. If inds is the float (between 0 and 1), it will be regarded as a percentage, and the last percentage of steps will be used for fate bias calculation. Otherwise inds need to be a list of integers of the time steps.
- **speed_percentile** (*float* (default: *5*)) – The percentile of speed that will be used to determine the terminal cells (or sink region on the prediction path where speed is smaller than this speed percentile).
- **dist_threshold** (*float* or *None* (default: *None*)) – A multiplier of the median nearest cell distance on the embedding to determine cells that are outside the sampled domain of cells. If the mean distance of identified "terminal cells" is above this number, we will look backward along the trajectory (by minimize all indices by 1) until it finds cells satisfy this threshold. By default it is set to be 1 to ensure only considering points that are very close to observed data points.
- **source_groups** (*list* or *None* (default: *None*)) – The groups that corresponds to progenitor groups. They has to have at least one intersection with the groups from the *group* column. If group is not *None*, any identified "source_groups" cells that happen to be in those groups will be ignored and the probability of cell fate of those cells will be reassigned to the group that has the highest fate probability among other non source_groups group cells.
- **metric** (*str* or callable, default='euclidean') – The distance metric to use for the tree. The default metric is , and with p=2 is equivalent to the standard Euclidean metric. See the documentation of DistanceMetric for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.
- **metric_kwds** (*dict, default=None*) – Additional keyword arguments for the metric function.
- **cores** (*int* (default: 1)) – The number of parallel jobs to run for neighbors search. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors.

- **seed** (*int* (default *19491001*)) – Random seed to ensure the reproducibility of each run.
- **kwargs** – Additional arguments that will be passed to each nearest neighbor search algorithm.

Returns **fate_bias** – A DataFrame that stores the fate bias for each cell state (row) to each cell group (column).

Return type *pandas.DataFrame*

dynamo.pd.state_graph

dynamo.pd.**state_graph**(*adata*, *group*, *approx=True*, *basis='umap'*, *layer=None*, *arc_sample=False*, *sample_num=100*)

Estimate the transition probability between cell types using method of vector field integrations.

Parameters

- **adata** (AnnData) – AnnData object that will be used to calculate a cell type (group) transition graph.
- **group** (*str*) – The attribute to group cells (column names in the adata.obs).
- **approx** (*bool* (default: False)) – Whether to use streamplot to get the integration lines from each cell.
- **basis** (*str* or None (default: *umap*)) – The embedding data to use for predicting cell fate. If *basis* is either *umap* or *pca*, the reconstructed trajectory will be projected back to high dimensional space via the *inverse_transform* function.
- **layer** (*str* or None (default: *None*)) – Which layer of the data will be used for predicting cell fate with the reconstructed vector field function. The layer once provided, will override the *basis* argument and then predicting cell fate in high dimensional space.
- **sample_num** (*int* (default: 100)) – The number of cells to sample in each group that will be used for calculating the transitoin graph between cell groups. This is required for facilitating the calculation.

Returns

- An updated adata object that is added with the *group* + *'_graph'* key, including the transition graph
- *and the average transition time.*

2.2.7 Plotting (pl)

Preprocessing

pl.basic_stats(adata[, group, figsize, ...])	Plot the basic statics (nGenes, nCounts and pMito) of each category of adata.
pl.show_fraction(adata[, genes, group, ...])	Plot the fraction of each category of data used in the velocity estimation.
pl.feature_genes(adata[, layer, mode, ...])	Plot selected feature genes on top of the mean vs.
pl.variance_explained(adata[, threshold, ...])	Plot the accumulative variance explained by the principal components.
pl.exp_by_groups(adata, genes[, layer, ...])	Plot the (labeled) expression values of genes across different groups (time points).

Cell cycle staging

pl.cell_cycle_scores(adata[, cells, ...])	Plot a heatmap of cells ordered by cell cycle position

Scatter base

pl.scatters(adata[, basis, x, y, color, ...])	Plot an embedding as points.

Phase diagram: conventional scRNA-seq

pl.phase_portraits(adata, genes[, x, y, ...])	Draw the phase portrait, expression values , velocity on the low dimensional embedding.

Kinetic models: labeling based scRNA-seq

pl.dynamics(adata, vkey[, unit, ...])	Plot the data and fitting of different metabolic labeling experiments.

Kinetics

pl.kinetic_curves(adata, genes[, mode, ...])	Plot the gene expression dynamics over time (pseudotime or inferred real time) as kinetic curves.
pl.kinetic_heatmap(adata, genes[, mode, ...])	Plot the gene expression dynamics over time (pseudotime or inferred real time) in a heatmap.
pl.jacobian_kinetics(adata[, source_genes, ...])	Plot the gene expression dynamics over time (pseudotime or inferred real time) in a heatmap.

Dimension reduction

pl.pca(adata, *args, **kwargs)	Scatter plot with pca basis.
pl.tsne(adata, *args, **kwargs)	Scatter plot with tsne basis.
pl.umap(adata, *args, **kwargs)	Scatter plot with umap basis.
pl.trimap(adata, *args, **kwargs)	Scatter plot with trimap basis.

Neighbor graph

pl.nneighbors(adata[, x, y, color, basis, ...])	Plot nearest neighbor graph of cells used to embed data into low dimension space.
pl.state_graph(adata, group[, basis, x, y, ...])	Plot a summarized cell type (state) transition graph.

Vector field plots: velocities and accelerations

pl.cell_wise_vectors(adata[, basis, x, y, ...])	Plot the velocity or acceleration vector of each cell.
pl.grid_vectors(adata[, basis, x, y, color, ...])	Plot the velocity or acceleration vector of each cell on a grid.
pl.streamline_plot(adata[, basis, x, y, ...])	Plot the velocity vector of each cell.

continues on next page

Table 29 – continued from previous page

pl.line_integral_conv(adata[, basis, ...])	Visualize vector field with quiver, streamline and line integral convolution (LIC), using velocity estimates on a grid from the associated data.
pl.plot_energy(adata[, basis, vecfld_dict, ...])	Plot the energy and energy change rate over each optimization iteration.

Vector field topology

pl.plot_flow_field(vecfld, x_range, y_range)	Plots the flow field with line thickness proportional to speed.
pl.plot_fixed_points(vecfld[, marker, ...])	Plot fixed points stored in the VectorField2D class.
pl.plot_nullclines(vecfld[, lw, background, ...])	Plot nullclines stored in the VectorField2D class.
pl.plot_separatrix(vecfld, x_range, y_range, t)	Plot separatrix on phase portrait.
pl.plot_traj(f, y0, t[, args, lw, ...])	Plots a trajectory on a phase portrait.
pl.topography(adata[, basis, x, y, color, ...])	Plot the streamline, fixed points (attractor / saddles), nullcline, separatrices of a recovered dynamic system for single cells.

Beyond RNA velocity

pl.speed(adata[, basis, color, frontier])	Scatter plot with cells colored by the estimated velocity speed (and other information if provided).
pl.divergence(adata[, basis, color, cmap, ...])	Scatter plot with cells colored by the estimated divergence (and other information if provided).
pl.curl(adata[, basis, color, cmap, ...])	Scatter plot with cells colored by the estimated curl (and other information if provided).
pl.curvature(adata[, basis, color, frontier])	Scatter plot with cells colored by the estimated curvature (and other information if provided).
pl.jacobian(adata[, source_genes, ...])	Scatter plot with pca basis.
pl.jacobian_heatmap(adata, cell_idx[, ...])	Plot the Jacobian matrix for each cell as a heatmap.

Potential landscape

pl.show_landscape(adata, Xgrid, Ygrid, Zgrid)	Plot the quasi-potential landscape.

Cell fate

pl.fate_bias(adata, group[, basis, ...])	Plot the lineage (fate) bias of cells states whose vector field trajectories are predicted.

Save figures

pl.save_fig([path, prefix, dpi, ext, ...])	Save a figure from pyplot.

2.2.8 Moive (mv)

mv.StreamFuncAnim(adata[, basis, dims, ...])	Animating cell fate commitment prediction via reconstructed vector field function.
mv.animate_fates(adata[, basis, dims, ...])	Animating cell fate commitment prediction via reconstructed vector field function.

dynamo.mv.StreamFuncAnim

class dynamo.mv.**StreamFuncAnim**(*adata*, *basis='umap'*, *dims=None*, *n_steps=100*, *cell_states=None*, *color='ntr'*, *ax=None*, *ln=None*, *logspace=False*)

Animating cell fate commitment prediction via reconstructed vector field function.

Animating cell fate commitment prediction via reconstructed vector field function.

This class creates necessary components to produce an animation that describes the exact speed of a set of cells at each time point, its movement in gene expression and the long range trajectory predicted by the reconstructed vector field. Thus it provides intuitive visual understanding of the RNA velocity, speed, acceleration, and cell fate commitment in action.

This function is originally inspired by https://tonysyu.github.io/animating-particles-in-a-flow.html and relies on animation module from matplotlib. Note that you may need to install *imagemagick* in order to properly show or save the animation. See for example, http://louistiao.me/posts/notebooks/save-matplotlib-animations-as-gifs/ for more details.

Parameters

- **adata** (AnnData) – AnnData object that already went through the fate prediction.
- **basis** (*str* or None (default: *None*)) – The embedding data to use for predicting cell fate. If *basis* is either *umap* or *pca*, the reconstructed trajectory will be projected back to high dimensional space via the *inverse_transform* function. space.
- **dims** (*list* or *None* (default: `None')) – The dimensions of low embedding space where cells will be drawn and it should corresponds to the space fate prediction take place.
- **n_steps** (*int* (default: *100*)) – The number of times steps (frames) fate prediction will take.
- **cell_states** (*int*, *list* or *None* (default: *None*)) – The number of cells state that will be randomly selected (if *int*), the indices of the cells states (if *list*) or all cell states which fate prediction executed (if *None*)
- **ax** (*matplotlib.Axis* (optional, default *None*)) – The matplotlib axes object that will be used as background plot of the vector field animation.
- **ln** (*tuple* or None (default: *None*)) – An iterable of artists (for example, *matplotlib.lines.Line2D*) used to draw a clear frame.
- **logspace** (*bool* (default: *False*)) – Whether or to sample time points linearly on log space. If not, the sorted unique set of all time points from all cell states' fate prediction will be used and then evenly sampled up to *n_steps* time points.

Returns

- *A class that contains .fig attribute and .update, .init_background that can be used to produce an animation*
- *of the prediction of cell fate commitment.*

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor, interpolation_
↪num=100,  direction='forward',
...     inverse_transform=False, average=False, arclen_sampling=True)
>>> instance = dyn.pl.StreamFuncAnim(adata=adata, ax=None, ln=None)
>>> anim = animation.FuncAnimation(instance.fig, instance.update, init_
↪func=instance.init_background,
...                                 frames=np.arange(100), interval=100, blit=True)
>>> from IPython.core.display import display, HTML
>>> HTML(anim.to_jshtml()) # embedding to jupyter notebook.
>>> anim.save('fate_ani.gif',writer="imagemagick") # save as gif file.
```

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor, interpolation_
↪num=100,  direction='forward',
...     inverse_transform=False, average=False, arclen_sampling=True)
>>> fig, ax = plt.subplots()
>>> ln, = ax.plot([], [], 'ro')
>>> ax.set_xlim(xlim)
>>> ax.set_ylim(ylim)
>>> instance = dyn.pl.StreamFuncAnim(adata=adata, ax=ax, ln=ln)
>>> anim = animation.FuncAnimation(fig, instance.update, init_func=instance.init_
↪background,
...                                 frames=np.arange(100), interval=100, blit=True)
>>> from IPython.core.display import display, HTML
>>> HTML(anim.to_jshtml()) # embedding to jupyter notebook.
>>> anim.save('fate_ani.gif',writer="imagemagick") # save as gif file.
```

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor, interpolation_
↪num=100,  direction='forward',
...     inverse_transform=False, average=False, arclen_sampling=True)
>>> dyn.pl.fate_animation(adata)
```

See also:: *animate_fates()*

__init__(*adata*, *basis='umap'*, *dims=None*, *n_steps=100*, *cell_states=None*, *color='ntr'*, *ax=None*, *ln=None*, *logspace=False*)

Animating cell fate commitment prediction via reconstructed vector field function.

This class creates necessary components to produce an animation that describes the exact speed of a set of cells at each time point, its movement in gene expression and the long range trajectory predicted by the reconstructed vector field. Thus it provides intuitive visual understanding of the RNA velocity, speed, acceleration, and cell fate commitment in action.

This function is originally inspired by https://tonysyu.github.io/animating-particles-in-a-flow.html and relies on animation module from matplotlib. Note that you may need to install *imagemagick* in order to properly show or save the animation. See for example, http://louistiao.me/posts/notebooks/

save-matplotlib-animations-as-gifs/ for more details.

Parameters

- **adata** (AnnData) – AnnData object that already went through the fate prediction.
- **basis** (*str* or None (default: *None*)) – The embedding data to use for predicting cell fate. If *basis* is either *umap* or *pca*, the reconstructed trajectory will be projected back to high dimensional space via the *inverse_transform* function. space.
- **dims** (*list* or *None* (default: `None')) – The dimensions of low embedding space where cells will be drawn and it should corresponds to the space fate prediction take place.
- **n_steps** (*int* (default: *100*)) – The number of times steps (frames) fate prediction will take.
- **cell_states** (*int*, *list* or *None* (default: *None*)) – The number of cells state that will be randomly selected (if *int*), the indices of the cells states (if *list*) or all cell states which fate prediction executed (if *None*)
- **ax** (*matplotlib.Axis* (optional, default *None*)) – The matplotlib axes object that will be used as background plot of the vector field animation.
- **ln** (*tuple* or None (default: *None*)) – An iterable of artists (for example, *matplotlib.lines.Line2D*) used to draw a clear frame.
- **logspace** (*bool* (default: *False*)) – Whether or to sample time points linearly on log space. If not, the sorted unique set of all time points from all cell states' fate prediction will be used and then evenly sampled up to *n_steps* time points.

Returns

- *A class that contains .fig attribute and .update, .init_background that can be used to produce an animation*
- *of the prediction of cell fate commitment.*

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor,␣
↪interpolation_num=100,  direction='forward',
...     inverse_transform=False, average=False, arclen_sampling=True)
>>> instance = dyn.pl.StreamFuncAnim(adata=adata, ax=None, ln=None)
>>> anim = animation.FuncAnimation(instance.fig, instance.update, init_
↪func=instance.init_background,
...                                 frames=np.arange(100), interval=100,␣
↪blit=True)
>>> from IPython.core.display import display, HTML
>>> HTML(anim.to_jshtml()) # embedding to jupyter notebook.
>>> anim.save('fate_ani.gif',writer="imagemagick") # save as gif file.
```

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor,␣
↪interpolation_num=100,  direction='forward',
...     inverse_transform=False, average=False, arclen_sampling=True)
>>> fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
>>> ln, = ax.plot([], [], 'ro')
>>> ax.set_xlim(xlim)
>>> ax.set_ylim(ylim)
>>> instance = dyn.pl.StreamFuncAnim(adata=adata, ax=ax, ln=ln)
>>> anim = animation.FuncAnimation(fig, instance.update, init_func=instance.
↪init_background,
...                                 frames=np.arange(100), interval=100,␣
↪blit=True)
>>> from IPython.core.display import display, HTML
>>> HTML(anim.to_jshtml()) # embedding to jupyter notebook.
>>> anim.save('fate_ani.gif',writer="imagemagick") # save as gif file.
```

```
>>> from matplotlib import animation
>>> progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']
>>> fate_progenitor = progenitor
>>> info_genes = adata.var_names[adata.var.use_for_velocity]
>>> dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor,␣
↪interpolation_num=100,  direction='forward',
...    inverse_transform=False, average=False, arclen_sampling=True)
>>> dyn.pl.fate_animation(adata)
```

See also:: *animate_fates()*

Methods

__init__(adata[, basis, dims, n_steps, ...])	Animating cell fate commitment prediction via reconstructed vector field function.
init_background()	
update(frame)	Update locations of "particles" in flow on each frame frame.

dynamo.mv.animate_fates

dynamo.mv.**animate_fates**(*adata*, *basis='umap'*, *dims=None*, *n_steps=100*, *cell_states=None*, *color='ntr'*, *ax=None*, *ln=None*, *logspace=False*, *interval=100*, *blit=True*, *save_show_or_return='show'*, *save_kwargs={}*, ***kwargs*)

Animating cell fate commitment prediction via reconstructed vector field function.

This class creates necessary components to produce an animation that describes the exact speed of a set of cells at each time point, its movement in gene expression and the long range trajectory predicted by the reconstructed vector field. Thus it provides intuitive visual understanding of the RNA velocity, speed, acceleration, and cell fate commitment in action.

This function is originally inspired by https://tonysyu.github.io/animating-particles-in-a-flow.html and relies on animation module from matplotlib. Note that you may need to install *imagemagick* in order to properly show or save the animation. See for example, http://louistiao.me/posts/notebooks/save-matplotlib-animations-as-gifs/ for more details.

Parameters

- **adata** (AnnData) – AnnData object that already went through the fate prediction.
- **basis** (*str* or None (default: *None*)) – The embedding data to use for predicting cell fate.

If *basis* is either *umap* or *pca*, the reconstructed trajectory will be projected back to high dimensional space via the *inverse_transform* function. space.

- **dims** (*list* or *None* (default: `None')) – The dimensions of low embedding space where cells will be drawn and it should corresponds to the space fate prediction take place.
- **n_steps** (*int* (default: *100*)) – The number of times steps (frames) fate prediction will take.
- **cell_states** (*int*, *list* or *None* (default: *None*)) – The number of cells state that will be randomly selected (if *int*), the indices of the cells states (if *list*) or all cell states which fate prediction executed (if *None*)
- **ax** (*matplotlib.Axis* (optional, default *None*)) – The matplotlib axes object that will be used as background plot of the vector field animation.
- **ln** (*tuple* or None (default: *None*)) – An iterable of artists (for example, *matplotlib.lines.Line2D*) used to draw a clear frame.
- **logspace** (*bool* (default: *False*)) – Whether or to sample time points linearly on log space. If not, the sorted unique set of all time points from all cell states' fate prediction will be used and then evenly sampled up to *n_steps* time points.
- **interval** (*float* (default: *200*)) – Delay between frames in milliseconds.
- **blit** (*bool* (default: *False*)) – Whether blitting is used to optimize drawing. Note: when using blitting, any animated artists will be drawn according to their zorder; however, they will be drawn on top of any previous artists, regardless of their zorder.
- **save_show_or_return** (*str* {'save', 'show', 'return'} (default: *save*)) – Whether to save, show or return the figure. By default a gif will be used.
- **save_kwargs** (*dict* (default: *{}*)) – A dictionary that will passed to the anim.save. By default it is an empty dictionary and the save_fig function will use the {"filename": 'fate_ani.gif', "writer": "imagemagick"} as its parameters. Otherwise you can provide a dictionary that properly modify those keys according to your needs. see https://matplotlib.org/api/_as_gen/matplotlib.animation.Animation.save.html for more details.
- **kwargs** – Additional arguments passed to animation.FuncAnimation.
- **Returns** –
- **-------** – Nothing but produce an animation that will be embedded to jupyter notebook or saved to disk.
- **1** (*Examples*) –
- **----------** –
- **from matplotlib import animation** (*>>>*) –
- **progenitor = adata.obs_names[adata.obs.clusters == 'cluster_1']** (*>>>*) –
- **fate_progenitor = progenitor** (*>>>*) –
- **info_genes = adata.var_names[adata.var.use_for_velocity]** (*>>>*) –
- **dyn.pd.fate(adata, basis='umap', init_cells=fate_progenitor, interpolation_num=100, direction='forward',** (*>>>*) –
- **inverse_transform=False, average=False, arclen_sampling=True)** (*..*) –

- **dyn.pl.fate_animation(adata)** (>>>) – See also:: *StreamFuncAnim()*

2.2.9 Simulation (sim)

—

Simple ODE vector field simulation

sim.two_genes_motif(x[, t, a1, a2, b1, b2, ...])	The ODE model for the famous Pu.1-Gata.1 like network motif with self-activation and mutual inhibition.
sim.neurogenesis(x[, t, mature_mu, n, k, a, ...])	The ODE model for the neurogenesis system that used in benchmarking Monocle 2, Scribe and dynamo (here), original from Xiaojie Qiu, et.
sim.toggle(ab[, t, beta, gamma, n])	Right hand side (rhs) for toggle ODEs.
sim.Ying_model(x[, t])	network used in the potential landscape paper from Ying, et.

Gillespie simulation

sim.Gillespie([a, b, la, aa, ai, si, be, ...])	A simulator of RNA dynamics that includes RNA bursting, transcription, metabolic labeling, splicing, transcription, RNA/protein degradation
sim.Simulator([motif, clip])	Simulate the gene expression dynamics via deterministic ODE model
sim.state_space_sampler(ode, dim[, clip, ...])	Sample N points from the dim dimension gene expression space while restricting the values to be between min_val and max_val.
sim.evaluate(reference, prediction[, metric])	Function to evaluate the vector field related reference quantities vs.

2.2.10 External (ext)

ext.ddhodge(adata[, X_data, layer, basis, ...])	Modeling Latent Flow Structure using Hodge Decomposition based on the creation of sparse diffusion graph from the reconstructed vector field function.
ext.scribe(adata[, genes, TFs, Targets, ...])	Apply Scribe to calculate causal network from spliced/unspliced, metabolic labeling based and other "real" time series datasets.
ext.mutual_inform(adata, genes, layer_x, layer_y)	Calculate mutual information (as well as pearson correlation) of genes between two different layers.
ext.scifate_glmnet(adata[, ...])	Reconstruction of regulatory network (Cao, et. al, Nature Biotechnology, 2020) from TFs to other target

dynamo.ext.ddhodge

dynamo.ext.**ddhodge**(*adata*, *X_data=None*, *layer=None*, *basis='pca'*, *n=30*, *VecFld=None*, *adjmethod='graphize_vecfld'*, *distance_free=False*, *n_downsamples=5000*, *up_sampling=True*, *sampling_method='velocity'*, *seed=19491001*, *enforce=False*, *cores=1*)

Modeling Latent Flow Structure using Hodge Decomposition based on the creation of sparse diffusion graph from the reconstructed vector field function. This method is relevant to the curl-free/divergence-free vector field reconstruction.

Parameters

- **adata** (AnnData) – an Annodata object.
- **X_data** (*np.ndarray* (default: *None*)) – The user supplied expression (embedding) data that will be used for graph hodege decomposition directly.
- **layer** (*str* or None (default: None)) – Which layer of the data will be used for graph Hodge decomposition.
- **basis** (*str* (default: *pca*)) – Which basis of the data will be used for graph Hodge decomposition.
- **n** (*int* (default: *10*)) – Number of nearest neighbors when the nearest neighbor graph is not included.
- **VecFld** (*dictionary* or None (default: None)) – The reconstructed vector field function.
- **adjmethod** (*str* (default: *graphize_vecfld*)) – The method to build the ajacency matrix that will be used to create the sparse diffusion graph, can be either "naive" or "graphize_vecfld". If "naive" used, the transition_matrix that created during vector field projection will be used; if "graphize_vecfld" used, a method that guarantees the preservance of divergence will be used.
- **n_downsamples** (*int* (default: *5000*)) – Number of cells to downsample to if the cell number is large than this value. Three downsampling methods are available, see *sampling_method*.
- **up_sampling** (*bool* (default: *True*)) – Whether to assign calculated potential, curl and divergence to cells not sampled based on values from their nearest sampled cells.
- **sampling_method** (*str* (default: *random*)) – Methods to downsample datasets to facilitate calculation. Can be one of {*random*, *velocity*, *trn*}, each corresponds to random sampling, velocity magnitude based and topology representing network based sampling.
- **seed** (int or 1-d array_like, optional (default: *0*)) – Seed for RandomState. Must be convertible to 32 bit unsigned integers. Used in sampling control points. Default is to be 0 for ensure consistency between different runs.
- **enforce** (*bool* (default: *False*)) – Whether to enforce the calculation of adjacency matrix for estimating potential, curl, divergence for each cell.
- **cores** (*int* (default: 1):) – Number of cores to run the graphize_vecfld function. If cores is set to be > 1, multiprocessing will be used to parallel the graphize_vecfld calculation.

Returns

adata –

***AnnData* object that is updated with the *ddhodge* key in the *obsp* attribute which to adjacency matrix that** corresponds to the sparse diffusion graph. Two columns *potential* and *divergence* corresponds to the potential and divergence for each cell will also be added.

Return type AnnData

dynamo.ext.scribe

dynamo.ext.**scribe**(*adata*, *genes=None*, *TFs=None*, *Targets=None*, *gene_filter_rate=0.1*, *cell_filter_UMI=10000*, *motif_ref='https://www.dropbox.com/s/s8em539ojl55kgf/motifAnnotations_hgnc.csv?dl=1'*, *nt_layers=['X_new', 'X_total']*, *normalize=True*, *do_CLR=True*, *drop_zero_cells=True*, *TF_link_ENCODE_ref='https://www.dropbox.com/s/bjuope41pte7mf4/df_gene_TF_l*

Apply Scribe to calculate causal network from spliced/unspliced, metabolic labeling based and other "real" time series datasets. Note that this function can be applied to both of the metabolic labeling based single-cell assays with newly synthesized and total RNA as well as the regular single cell assays with both the unspliced and spliced transcripts. Furthermore, you can also replace the either the new or unspliced RNA with dynamo estimated cell-wise velocity, transcription, splicing and degradation rates for each gene (similarly, replacing the expression values of transcription factors with RNA binding, ribosome, epigenetics or epitranscriptomic factors, etc.) to infer the total regulatory effects, transcription, splicing and post-transcriptional regulation of different factors.

Parameters

- **adata** (AnnData.) – adata object that includes both newly synthesized and total gene expression of cells. Alternatively, the object should include both unspliced and spliced gene expression of cells.
- **genes** (*List* (default: None)) – The list of gene names that will be used for casual network inference. By default, it is *None* and thus will use all genes.
- **TFs** (*List* or *None* (default: None)) – The list of transcription factors that will be used for casual network inference. When it is *None* gene list included in the file linked by *motif_ref* will be used.
- **Targets** (*List* or *None* (default: None)) – The list of target genes that will be used for casual network inference. When it is *None* gene list not included in the file linked by *motif_ref* will be used.
- **gene_filter_rate** (*float* (default: 0.1)) – minimum percentage of expressed cells for gene filtering.
- **cell_filter_UMI** (*int* (default: 10000)) – minimum number of UMIs for cell filtering.
- **motif_ref** (*str* (default: 'https://www.dropbox.com/s/bjuope41pte7mf4/df_gene_TF_link_ENCODE.csv?dl=1')) – It provides the list of TFs gene names and is used to parse the data to get the list of TFs and Targets for the causal network inference from those TFs to Targets. But currently the motif based filtering is not implemented. By default it is a dropbox link that store the data from us. Other motif reference can bed downloaded from RcisTarget: https://resources.aertslab.org/cistarget/. For human motif matrix, it can be downloaded from June's shared folder: https://shendure-web.gs.washington.edu/content/members/cao1025/public/nobackup/sci_fate/data/hg19-tss-centered-10kb-7species.mc9nr.feather
- **nt_layers** (*List* (Default: ['X_new', 'X_total'])) – The two keys for layers that will be used for the network inference. Note that the layers can be changed flexibly. See the description of this function above. The first key corresponds to the transcriptome of the next time point, for example unspliced RNAs (or estimated velocitym, see Fig 6 of the Scribe preprint: https://www.biorxiv.org/content/10.1101/426981v1) from RNA velocity, old RNA from scSLAM-seq data, etc. The second key corresponds to the transcriptome of the initial time point, for example spliced RNAs from RNA velocity, old RNA from scSLAM-seq data.

- **drop_zero_cells** (*bool* (Default: False)) – Whether to drop cells that with zero expression for either the potential regulator or potential target. This can signify the relationship between potential regulators and targets, speed up the calculation, but at the risk of ignoring strong inhibition effects from certain regulators to targets.
- **do_CLR** (*bool* (Default: True)) – Whether to perform context likelihood relatedness analysis on the reconstructed causal network
- **TF_link_ENCODE_ref** (*str* (default: 'https://www.dropbox.com/s/s8em539ojl55kgf/motifAnnotations_hgnc.csv?dl=1')) – The path to the TF chip-seq data. By default it is a dropbox link from us that stores the data. Other data can be downloaded from: https://amp.pharm.mssm.edu/Harmonizome/dataset/ENCODE+Transcription+Factor+Targets.

Returns

Return type An updated adata object with a new key *causal_net* in .uns attribute, which stores the inferred causal network.

dynamo.ext.mutual_inform

dynamo.ext.**mutual_inform**(*adata*, *genes*, *layer_x*, *layer_y*, *cores=1*)

Calculate mutual information (as well as pearson correlation) of genes between two different layers.

Parameters

- **adata** (AnnData.) – adata object that will be used for mutual information calculation.
- **genes** (*List* (default: None)) – Gene names from the adata object that will be used for mutual information calculation.
- **layer_x** – The first key of the layer from the adata object that will be used for mutual information calculation.
- **layer_y** – The second key of the layer from the adata object that will be used for mutual information calculation.
- **cores** (*int* (default: 1)) – Number of cores to run the MI calculation. If cores is set to be > 1, multiprocessing will be used to parallel the calculation.

Returns

- An updated adata object that updated with a new columns (*mi*, *pearson*) in .var contains the mutual information
- *of input genes.*

dynamo.ext.scifate_glmnet

dynamo.ext.**scifate_glmnet**(*adata*, *gene_filter_rate=0.1*, *cell_filter_UMI=10000*, *core_n_lasso=1*, *core_n_filtering=1*, *motif_ref='https://www.dropbox.com/s/s8em539ojl55kgf/motifAnnotations_hgnc.csv?dl=1'*, *TF_link_ENCODE_ref='https://www.dropbox.com/s/bjuope41pte7mf4/df_gene_TF_link_ENCO*
nt_layers=['X_new', 'X_total'])

Reconstruction of regulatory network (Cao, et. al, Nature Biotechnology, 2020) from TFs to other target genes via LASSO regression between the total expression of known transcription factors and the newly synthesized RNA of potential targets. The inferred regulatory relationships between TF and targets are further filtered based on evidence of promoter motifs (not implemented currently) and the ENCODE chip-seq peaks. The python wrapper for the glmnet FORTRON code, glm-python (https://github.com/bbalasub1/glmnet_python) was used. More details on lasso regression with glm-python can

be found here (https://github.com/bbalasub1/glmnet_python/blob/master/test/glmnet_examples.ipynb). Note that this function can be applied to both of the metabolic labeling single-cell assays with newly synthesized and total RNA as well as the regular single cell assays with both the unspliced and spliced transcripts. Furthermore, you can also replace the either the new or unspliced RNA with dynamo estimated cell-wise velocity, transcription, splicing and degradation rates for each gene (similarly, replacing the expression values of transcription factors with RNA binding, ribosome, epigenetics or epitranscriptomic factors, etc.) to infer the tottal regulatory effects, transcription, splicing and post-transcriptional regulation of different factors. In addition, this approach will be fully integrated with Scribe (Qiu, et. al, 2020) which employs restricted directed information to determine causality by estimating the strength of information transferred from a potential regulator to its downstream target. In contrast of lasso regression, Scribe can learn both linear and non-linear causality in deterministic and stochastic systems. It also incorporates rigorous procedures to alleviate sampling bias and builds upon novel estimators and regularization techniques to facilitate inference of large-scale causal networks.

Parameters

- **adata** (AnnData.) – adata object that includes both newly synthesized and total gene expression of cells. Alternatively, the object should include both unspliced and spliced gene expression of cells.
- **gene_filter_rate** (*float* (default: 0.1)) – minimum percentage of expressed cells for gene filtering.
- **cell_filter_UMI** (*int* (default: 10000)) – minimum number of UMIs for cell filtering.
- **core_n_lasso** (*int* (default: 1)) – number of cores for lasso regression in linkage analysis. By default, it is 1 and parallel is turned off. Parallel computing can significantly speed up the computation process, especially for datasets involve many cells or genes. But for smaller datasets or genes, it could result in a reduction in speed due to the additional overhead. User discretion is advised.
- **core_n_filtering** (*int* (default: 1)) – number of cores for filtering TF-gene links. Not used currently.
- **motif_ref** (*str* (default: 'https://www.dropbox.com/s/bjuope41pte7mf4/df_gene_TF_link_ENCODE.csv?dl=1')) – The path to the TF binding motif data as described above. It provides the list of TFs gene names and is used to process adata object to generate the TF expression and target new expression matrix for glmnet based TF-target synthesis rate linkage analysis. But currently it is not used for motif based filtering. By default it is a dropbox link that store the data from us. Other motif reference can bed downloaded from RcisTarget: https://resources.aertslab.org/cistarget/. For human motif matrix, it can be downloaded from June's shared folder: https://shendure-web.gs.washington.edu/content/members/cao1025/public/nobackup/sci_fate/data/hg19-tss-centered-10kb-7species.mc9nr.feather
- **TF_link_ENCODE_ref** (*str* (default: 'https://www.dropbox.com/s/s8em539ojl55kgf/motifAnnotations_hgnc.csv?dl=1')) – The path to the TF chip-seq data. By default it is a dropbox link from us that stores the data. Other data can be downloaded from: https://amp.pharm.mssm.edu/Harmonizome/dataset/ENCODE+Transcription+Factor+Targets.
- **nt_layers** (*list([str, str])* (default: *['X_new', 'X_total']*)) – The layers that will be used for the network inference. Note that the layers can be changed flexibly. See the description of this function above.
- **that if your internet connection is slow, we recommend to download the motif_ref and TF_link_ENCODE_ref and** (*Note*) –
- **those two arguments with the local paths where the downloaded datasets are saved.** (*supplies*) –

Returns

- An updated adata object with a new key *scifate* in .uns attribute, which stores the raw lasso regression results
- *and the filter results after applying the Fisher exact test of the ChIP-seq peaks.*

2.2.11 Utilities

Package versions

get_all_dependencies_version([display])	Adapted from answer 2 in https://stackoverflow.com/questions/40428931/package-for-listing-version-of-packages-used-in-a-jupyter-notebook

dynamo.get_all_dependencies_version

dynamo.**get_all_dependencies_version**(*display=True*)

Adapted from answer 2 in https://stackoverflow.com/questions/40428931/package-for-listing-version-of-packages-used-in-a-jupyter-notebook

Clean up adata

cleanup(adata[, del_prediction])	clean up adata before saving it to a file

dynamo.cleanup

dynamo.**cleanup**(*adata*, *del_prediction=False*)

clean up adata before saving it to a file

Figures configuration

configuration.set_figure_params([dynamo, ...])	Set resolution/size, styling and format of figures.
configuration.set_pub_style([scaler])	formatting helper function that can be used to save publishable figures

dynamo.configuration.set_figure_params

dynamo.configuration.**set_figure_params**(*dynamo=True*, *background='white'*, *fontsize=8*, *figsize=6, 4*, *dpi=None*, *dpi_save=None*, *frameon=None*, *vector_friendly=True*, *color_map=None*, *format='pdf'*, *transparent=False*, *ipython_format='png2x'*)

Set resolution/size, styling and format of figures. This function is adapted from: https://github.com/theislab/scanpy/blob/f539870d7484675876281eb1c475595bf4a69bdb/scanpy/_settings.py

Parameters

- **dynamo** (*bool* (default: *True*)) – Init default values for `matplotlib.rcParams` suited for dynamo.

- **background** (*str* (default: *white*)) – The background color of the plot. By default we use the white ground which is suitable for producing figures for publication. Setting it to *black* background will be great for presentation.
- **fontsize** (*[float, float]* or None (default: *6*)) –
- **figsize** ((*float, float)* (default: *(6.5, 5)*)) – Width and height for default figure size.
- **dpi** (*int* or None (default: *None*)) – Resolution of rendered figures - this influences the size of figures in notebooks.
- **dpi_save** (*int* or None (default: *None*)) – Resolution of saved figures. This should typically be higher to achieve publication quality.
- **frameon** (*bool* or None (default: *None*)) – Add frames and axes labels to scatter plots.
- **vector_friendly** (*bool* (default: *True*)) – Plot scatter plots using *png* backend even when exporting as *pdf* or *svg*.
- **color_map** (*str* (default: *None*)) – Convenience method for setting the default color map.
- **format** (`{'png', 'pdf', 'svg', etc.} (default: 'pdf')`) – This sets the default format for saving figures: *file_format_figs*.
- **transparent** (*bool* (default: *False*)) – Save figures with transparent back ground. Sets *rcParams['savefig.transparent']*.
- **ipython_format** (list of *str* (default: 'png2x')) – Only concerns the notebook/IPython environment; see *IPython.core.display.set_matplotlib_formats* for more details.

dynamo.configuration.set_pub_style

dynamo.configuration.**set_pub_style**(*scaler=1*)

formatting helper function that can be used to save publishable figures

2.3 Class

2.3.1 Estimation

Conventional scRNA-seq (est.csc)

class csc.**ss_estimation**(*U=None*, *Ul=None*, *S=None*, *Sl=None*, *P=None*, *US=None*, *S2=None*, *conn=None*, *t=None*, *ind_for_proteins=None*, *model='stochastic'*, *est_method='gmm'*, *experiment_type='deg'*, *assumption_mRNA=None*, *assumption_protein='ss'*, *concat_data=True*, *cores=1*, ***kwargs*)

The class that estimates parameters with input data.

Parameters

- **U** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA count.
- **Ul** (ndarray or sparse *csr_matrix*) – A matrix of unspliced, labeled mRNA count.
- **S** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA count.
- **Sl** (ndarray or sparse *csr_matrix*) – A matrix of spliced, labeled mRNA count.
- **P** (ndarray or sparse *csr_matrix*) – A matrix of protein count.

- **US** (ndarray or sparse *csr_matrix*) – A matrix of second moment of unspliced/spliced gene expression count for conventional or NTR velocity.
- **S2** (ndarray or sparse *csr_matrix*) – A matrix of second moment of spliced gene expression count for conventional or NTR velocity.
- **conn** (ndarray or sparse *csr_matrix*) – The connectivity matrix that can be used to calculate first /second moment of the data.
- **t** (ss_estimation) – A vector of time points.
- **ind_for_proteins** (ndarray) – A 1-D vector of the indices in the U, Ul, S, Sl layers that corresponds to the row name in the *protein* or *X_protein* key of *.obsm* attribute.
- **experiment_type** (*str*) – labelling experiment type. Available options are: (1) 'deg': degradation experiment; (2) 'kin': synthesis experiment; (3) 'one-shot': one-shot kinetic experiment; (4) 'mix_std_stm': a mixed steady state and stimulation labeling experiment.
- **assumption_mRNA** (*str*) – Parameter estimation assumption for mRNA. Available options are: (1) 'ss': pseudo steady state; (2) None: kinetic data with no assumption.
- **assumption_protein** (*str*) – Parameter estimation assumption for protein. Available options are: (1) 'ss': pseudo steady state;
- **concat_data** (*bool (default: True)*) – Whether to concatenate data
- **cores** (*int* (default: 1)) – Number of cores to run the estimation. If cores is set to be > 1, multiprocessing will be used to parallel the parameter estimation.

Returns

- **t** (ss_estimation) – A vector of time points.
- **data** (*dict*) – A dictionary with uu, ul, su, sl, p as its keys.
- **extyp** (*str*) – labelling experiment type.
- **asspt_mRNA** (*str*) – Parameter estimation assumption for mRNA.
- **asspt_prot** (*str*) – Parameter estimation assumption for protein.
- **parameters** (*dict*) –

  A dictionary with alpha, beta, gamma, eta, delta as its keys. alpha: transcription rate beta: RNA splicing rate gamma: spliced mRNA degradation rate eta: translation rate delta: protein degradation rate

concatenate_data()
Concatenate available data into a single matrix.

See "concat_time_series_matrices" for details.

fit(*intercept=False*, *perc_left=None*, *perc_right=5*, *clusters=None*, *one_shot_method='combined'*)
Fit the input data to estimate all or a subset of the parameters

Parameters

- **intercept** (*bool*) – If using steady state assumption for fitting, then: True – the linear regression is performed with an unfixed intercept; False – the linear regression is performed with a fixed zero intercept.
- **perc_left** (*float* (default: 5)) – The percentage of samples included in the linear regression in the left tail. If set to None, then all the samples are included.
- **perc_right** (*float* (default: 5)) – The percentage of samples included in the linear regression in the right tail. If set to None, then all the samples are included.

- **clusters** (*list*) – A list of n clusters, each element is a list of indices of the samples which belong to this cluster.

fit_alpha_oneshot(*t*, *U*, *beta*, *clusters=None*)
Estimate alpha with the one-shot data.

Parameters

- **t** (*float*) – labelling duration.
- **U** (ndarray) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **beta** (ndarray) – A vector of betas for all the genes.
- **clusters** (*list*) – A list of n clusters, each element is a list of indices of the samples which belong to this cluster.

Returns **alpha** – A numpy array with the dimension of n_genes x clusters.

Return type ndarray

fit_beta_gamma_lsq(*t*, *U*, *S*)
Estimate beta and gamma with the degradation data using the least squares method.

Parameters

- **t** (ndarray) – A vector of time points.
- **U** (ndarray) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **S** (ndarray) – A matrix of spliced mRNA counts. Dimension: genes x cells.

Returns

- **beta** (ndarray) – A vector of betas for all the genes.
- **gamma** (ndarray) – A vector of gammas for all the genes.
- **u0** (*float*) – Initial value of u.
- **s0** (*float*) – Initial value of s.

fit_gamma_nosplicing_lsq(*t*, *L*)
Estimate gamma with the degradation data using the least squares method when there is no splicing data.

Parameters

- **t** (ndarray) – A vector of time points.
- **L** (ndarray) – A matrix of labeled mRNA counts. Dimension: genes x cells.

Returns

- **gamma** (ndarray) – A vector of gammas for all the genes.
- **l0** (*float*) – The estimated value for the initial spliced, labeled mRNA count.

fit_gamma_steady_state(*u*, *s*, *intercept=True*, *perc_left=None*, *perc_right=5*, *normalize=True*)
Estimate gamma using linear regression based on the steady state assumption.

Parameters

- **u** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **s** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA counts. Dimension: genes x cells.

- **intercept** (*bool*) – If using steady state assumption for fitting, then: True – the linear regression is performed with an unfixed intercept; False – the linear regresssion is performed with a fixed zero intercept.
- **perc_left** (*float*) – The percentage of samples included in the linear regression in the left tail. If set to None, then all the left samples are excluded.
- **perc_right** (*float*) – The percentage of samples included in the linear regression in the right tail. If set to None, then all the samples are included.
- **normalize** (*bool*) – Whether to first normalize the

Returns

- **k** (*float*) – The slope of the linear regression model, which is gamma under the steady state assumption.
- **b** (*float*) – The intercept of the linear regression model.
- **r2** (*float*) – Coefficient of determination or r square for the extreme data points.
- **r2** (*float*) – Coefficient of determination or r square for the extreme data points.
- **all_r2** (*float*) – Coefficient of determination or r square for all data points.

fit_gamma_stochastic(*est_method*, *u*, *s*, *us*, *ss*, *perc_left=None*, *perc_right=5*, *normalize=True*)

Estimate gamma using GMM (generalized method of moments) or negbin distrubtion based on the steady state assumption.

Parameters

- **est_method** (*str* {*gmm*, *negbin*} The estimation method to be used when using the *stochastic* model.) –

  – Available options when the *model* is 'ss' include:

  (2) 'gmm': The new generalized methods of moments from us that is based on master equations, similar to the "moment" model in the excellent scVelo package; (3) 'negbin': The new method from us that models steady state RNA expression as a negative binomial distribution, also built upon on master equations. Note that all those methods require using extreme data points (except negbin, which use all data points) for estimation. Extreme data points are defined as the data from cells whose expression of unspliced / spliced or new / total RNA, etc. are in the top or bottom, 5%, for example. *linear_regression* only considers the mean of RNA species (based on the *deterministic* ordinary different equations) while moment based methods (*gmm*, *negbin*) considers both first moment (mean) and second moment (uncentered variance) of RNA species (based on the *stochastic* master equations). The above method are all (generalized) linear regression based method. In order to return estimated parameters (including RNA half-life), it additionally returns R-squared (either just for extreme data points or all data points) as well as the log-likelihood of the fitting, which will be used for transition matrix and velocity embedding. All *est_method* uses least square to estimate optimal parameters with latin cubic sampler for initial sampling.
- **u** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **s** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA counts. Dimension: genes x cells.
- **us** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **ss** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA counts. Dimension: genes x cells.

- **perc_left** (*float*) – The percentage of samples included in the linear regression in the left tail. If set to None, then all the left samples are excluded.
- **perc_right** (*float*) – The percentage of samples included in the linear regression in the right tail. If set to None, then all the samples are included.
- **normalize** (*bool*) – Whether to first normalize the

Returns

- **k** (*float*) – The slope of the linear regression model, which is gamma under the steady state assumption.
- **b** (*float*) – The intercept of the linear regression model.
- **r2** (*float*) – Coefficient of determination or r square for the extreme data points.
- **r2** (*float*) – Coefficient of determination or r square for the extreme data points.
- **all_r2** (*float*) – Coefficient of determination or r square for all data points.

get_exist_data_names()
Get the names of all the data that are not 'None'.

get_n_genes(*key=None*, *data=None*)
Get the number of genes.

set_parameter(*name*, *value*)
Set the value for the specified parameter.

Parameters

- **name** (*string*) – The name of the parameter. E.g. 'beta'.
- **value** (ndarray) – A vector of values for the parameter to be set to.

solve_alpha_mix_std_stm(*t*, *ul*, *beta*, *clusters=None*, *alpha_time_dependent=True*)
Estimate the steady state transcription rate and analytically calculate the stimulation transcription rate given beta and steady state alpha for a mixed steady state and stimulation labeling experiment.

This approach assumes the same constant beta or gamma for both steady state or stimulation period.

Parameters

- **t** (*list* or *numpy.ndarray*) – Time period for stimulation state labeling for each cell.
- **ul** – A vector of labeled RNA amount in each cell.
- **beta** (*numpy.ndarray*) – A list of splicing rate for genes.
- **clusters** (*list*) – A list of n clusters, each element is a list of indices of the samples which belong to this cluster.
- **alpha_time_dependent** (*bool*) – Whether or not to model the simulation alpha rate as a time dependent variable.

Returns **alpha_std, alpha_stm** – The constant steady state transcription rate (alpha_std) or time-dependent or time-independent (determined by alpha_time_dependent) transcription rate (alpha_stm)

Return type *numpy.ndarray*, *numpy.ndarray*

class csc.**velocity**(*alpha=None*, *beta=None*, *gamma=None*, *eta=None*, *delta=None*, *t=None*, *estimation=None*)
The class that computes RNA/protein velocity given unknown parameters.

Parameters

- **alpha** (ndarray) – A matrix of transcription rate.
- **beta** (ndarray) – A vector of splicing rate constant for each gene.
- **gamma** (ndarray) – A vector of spliced mRNA degradation rate constant for each gene.
- **eta** (ndarray) – A vector of protein synthesis rate constant for each gene.
- **delta** (ndarray) – A vector of protein degradation rate constant for each gene.
- **t** (ndarray or None (default: None)) – A vector of the measured time points for cells
- **estimation** (ss_estimation) – An instance of the estimation class. If this not None, the parameters will be taken from this class instead of the input arguments.

get_n_cells()
Get the number of cells if the parameter alpha is given.

Returns **n_cells** – The second dimension of the alpha matrix, if alpha is given.

Return type int

get_n_genes()
Get the number of genes.

Returns **n_genes** – The first dimension of the alpha matrix, if alpha is given. Or, the length of beta, gamma, eta, or delta, if they are given.

Return type int

vel_p(*S*, *P*)
Calculate the protein velocity.

Parameters

- **S** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA counts. Dimension: genes x cells.
- **P** (ndarray or sparse *csr_matrix*) – A matrix of protein counts. Dimension: genes x cells.

Returns **V** – Each column of V is a velocity vector for the corresponding cell. Dimension: genes x cells.

Return type ndarray or sparse *csr_matrix*

vel_s(*U*, *S*)
Calculate the unspliced mRNA velocity.

Parameters

- **U** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA counts. Dimension: genes x cells.
- **S** (ndarray or sparse *csr_matrix*) – A matrix of spliced mRNA counts. Dimension: genes x cells.

Returns **V** – Each column of V is a velocity vector for the corresponding cell. Dimension: genes x cells.

Return type ndarray or sparse *csr_matrix*

vel_u(*U*)
Calculate the unspliced mRNA velocity.

Parameters **U** (ndarray or sparse *csr_matrix*) – A matrix of unspliced mRNA count. Dimension: genes x cells.

Returns **V** – Each column of V is a velocity vector for the corresponding cell. Dimension: genes x cells.

Return type `ndarray` or sparse *csr_matrix*

Time-resolved metabolic labeling based scRNA-seq (est.tsc)

Base class: a general estimation framework

class `tsc.`**kinetic_estimation**(*param_ranges*, *x0_ranges*, *simulator*)

A general parameter estimation framework for all types of time-seris data

Parameters

- **param_ranges** (`ndarray`) – A n-by-2 numpy array containing the lower and upper ranges of n parameters (and initial conditions if not fixed).
- **x0_ranges** (`ndarray`) – Lower and upper bounds for initial conditions for the integrators. To fix a parameter, set its lower and upper bounds to the same value.
- **simulator** (`utils_kinetic.Linear_ODE`) – An instance of python class which solves ODEs. It should have properties 't' (k time points, 1d numpy array), 'x0' (initial conditions for m species, 1d numpy array), and 'x' (solution, k-by-m array), as well as two functions: integrate (numerical integration), solve (analytical method).

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (`ndarray`) – A numpy array of n time points.
- **x_data** (`ndarray`) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (`numpy.ndarray`, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (`ndarray`) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

Deterministic models via analytical solution of ODEs

class tsc.**Estimation_DeterministicDeg**(*beta=None*, *gamma=None*, *x0=None*)

An estimation class for degradation (with splicing) experiments. Order of species: <unspliced>, <spliced>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k

cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_DeterministicDegNosp**(*gamma=None*, *x0=None*)

An estimation class for degradation (without splicing) experiments.

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_DeterministicKinNosp**(*alpha*, *gamma*, *x0=0*)

An estimation class for kinetics (without splicing) experiments with the deterministic model. Order of species: <unspliced>, <spliced>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)

- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_DeterministicKin**(*alpha*, *beta*, *gamma*, *x0=array([0.0, 0.0])*)

An estimation class for kinetics experiments with the deterministic model. Order of species: <unspliced>, <spliced>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)

- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

Stochastic models via matrix form of moment equations

class tsc.**Estimation_MomentDeg**(*beta=None*, *gamma=None*, *x0=None*, *include_cov=True*)

An estimation class for degradation (with splicing) experiments. Order of species: <unspliced>, <spliced>, <uu>, <ss>, <us> Order of parameters: beta, gamma

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*

- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_MomentDegNosp**(*gamma=None*, *x0=None*)

An estimation class for degradation (without splicing) experiments.

An estimation class for degradation (without splicing) experiments. Order of species: <r>, <rr>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_MomentKin**(*a*, *b*, *alpha_a*, *alpha_i*, *beta*, *gamma*, *include_cov=True*)

An estimation class for kinetics experiments. Order of species: <unspliced>, <spliced>, <uu>, <ss>, <us>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Estimation_MomentKinNosp**(*a*, *b*, *alpha_a*, *alpha_i*, *gamma*)

An estimation class for kinetics experiments. Order of species: <r>, <rr>

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)

Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

Mixture models for kinetic / degradation experiments

class tsc.**Lambda_NoSwitching**(*model1*, *model2*, *alpha=None*, *lambd=None*, *gamma=None*, *x0=None*, *beta=None*)

An estimation class with the mixture model. If beta is None, it is assumed that the data does not have the splicing process.

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)
Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2(*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)
perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

class tsc.**Mixture_KinDeg_NoSwitching**(*model1*, *model2*, *alpha=None*, *gamma=None*, *x0=None*, *beta=None*)
An estimation class with the mixture model. If beta is None, it is assumed that the data does not have the splicing process.

fit_lsq(*t*, *x_data*, *p0=None*, *n_p0=1*, *bounds=None*, *sample_method='lhs'*, *method=None*, *normalize=True*)
Fit time-seris data using least squares

Parameters

- **t** (ndarray) – A numpy array of n time points.
- **x_data** (ndarray) – A m-by-n numpy a array of m species, each having n values for the n time points.
- **p0** (numpy.ndarray, optional, default: None) – Initial guesses of parameters. If None, a random number is generated within the bounds.
- **n_p0** (*int, optional, default: 1*) – Number of initial guesses.
- **bounds** (*tuple, optional, default: None*) – Lower and upper bounds for parameters.
- **sample_method** (str, optional, default: *lhs*) – Method used for sampling initial guesses of parameters: *lhs*: latin hypercube sampling; *uniform*: uniform random sampling.
- **method** (*str or None, optional, default: None*) – Method used for solving ODEs. See options in simulator classes. If None, default method is used.
- **normalize** (*bool, optional, default: True*) – Whether or not normalize values in x_data across species, so that large values do not dominate the optimizer.

Returns

- **popt** (ndarray) – Optimal parameters.
- **cost** (*float*) – The cost function evaluated at the optimum.

test_chi2 (*t*, *x_data*, *species=None*, *method='matrix'*, *normalize=True*)

perform a Pearson's chi-square test. The statistics is computed as: sum_i (O_i - E_i)^2 / E_i, where O_i is the data and E_i is the model predication.

The data can be either 1. stratified moments: 't' is an array of k distinct time points, 'x_data' is a m-by-k matrix of data, where m is the number of species. or 2. raw data: 't' is an array of k time points for k cells, 'x_data' is a m-by-k matrix of data, where m is the number of species. Note that if the method is 'numerical', t has to monotonically increasing.

If not all species are included in the data, use 'species' to specify the species of interest.

Returns

- **p** (*float*)
- *The p-value of a one-tailed chi-square test.*
- **c2** (*float*)
- *The chi-square statistics.*
- **df** (*int*)
- *Degree of freedom.*

2.3.2 Vector field

Vector field class

class dynamo.vf.**vectorfield**(*X=None*, *V=None*, *Grid=None*, ***kwargs*)

Initialize the VectorField class.

Parameters

- **X** (*'np.ndarray' (dimension: n_obs x n_features)*) – Original data.
- **V** (*'np.ndarray' (dimension: n_obs x n_features)*) – Velocities of cells in the same order and dimension of X.
- **Grid** (*'np.ndarray'*) – The function that returns diffusion matrix which can be dependent on the variables (for example, genes)
- **M** (*'int' (default: None)*) – The number of basis functions to approximate the vector field. By default it is calculated as *min(len(X), int(1500 * np.log(len(X)) / (np.log(len(X)) + np.log(100))))*. So that any datasets with less than about 900 data points (cells) will use full data for vector field reconstruction while any dataset larger than that will at most use 1500 data points.
- **a** (*float* (default 5)) – Parameter of the model of outliers. We assume the outliers obey uniform distribution, and the volume of outlier's variation space is a.
- **beta** (*float* (default: None)) – Parameter of Gaussian Kernel, k(x, y) = exp(-beta*||x-y||^2). If None, a rule-of-thumb bandwidth will be computed automatically.
- **ecr** (*float* (default: 1e-5)) – The minimum limitation of energy change rate in the iteration process.
- **gamma** (*float* (default: 0.9)) – Percentage of inliers in the samples. This is an inital value for EM iteration, and it is not important. Default value is 0.9.
- **lambda** (*float* (default: 3)) – Represents the trade-off between the goodness of data fit and regularization.
- **minP** (*float* (default: 1e-5)) – The posterior probability Matrix P may be singular for matrix inversion. We set the minimum value of P as minP.
- **MaxIter** (*int* (default: 500)) – Maximum iteration times.
- **theta** (*float* (default 0.75)) – Define how could be an inlier. If the posterior probability of a sample is an inlier is larger than theta, then it is regarded as an inlier.
- **div_cur_free_kernels** (*bool* (default: False)) – A logic flag to determine whether the divergence-free or curl-free kernels will be used for learning the vector field.
- **sigma** (*'int'*) – Bandwidth parameter.
- **eta** (*'int'*) – Combination coefficient for the divergence-free or the curl-free kernels.
- **seed** (int or 1-d array_like, optional (default: *0*)) – Seed for RandomState. Must be convertible to 32 bit unsigned integers. Used in sampling control points. Default is to be 0 for ensure consistency between different runs.

fit(*normalize=False*, *method='SparseVFC'*, ***kwargs*)

Learn an function of vector field from sparse single cell samples in the entire space robustly. Reference: Regularized vector field learning with sparse approximation for mismatch removal, Ma, Jiayi, etc. al, Pattern Recognition

Parameters

- **normalize** (*'bool' (default: False)*) – Logic flag to determine whether to normalize the data to have zero means and unit covariance. This is often required for raw dataset (for example, raw UMI counts and RNA velocity values in high dimension). But it is normally not required for low dimensional embeddings by PCA or other non-linear dimension reduction methods.
- **method** (*'string'*) – Method that is used to reconstruct the vector field functionally. Currently only SparseVFC supported but other improved approaches are under development.

Returns **VecFld** – A dictionary which contains X, Y, beta, V, C, P, VFCIndex. Where V = f(X), P is the posterior probability and VFCIndex is the indexes of inliers which found by VFC.

Return type `dict'

get_Jacobian(*method='analytical'*, *input_vector_convention='row'*, ***kwargs*)

Get the Jacobian of the vector field function. If method is 'analytical': The analytical Jacobian will be returned and it always take row vectors as input no matter what input_vector_convention is.

If method is 'numerical': If the input_vector_convention is 'row', it means that fjac takes row vectors as input, otherwise the input should be an array of column vectors. Note that the returned Jacobian would behave exactly the same if the input is an 1d array.

The column vector convention is slightly faster than the row vector convention. So the matrix of row vector convention is converted into column vector convention under the hood.

No matter the method and input vector convention, the returned Jacobian is of the following format:

df_1/dx_1 df_1/dx_2 df_1/dx_3 ... df_2/dx_1 df_2/dx_2 df_2/dx_3 ... df_3/dx_1 df_3/dx_2 df_3/dx_3 ... ...

evaluate(*CorrectIndex*, *VFCIndex*, *siz*)

Evaluate the precision, recall, corrRate of the sparseVFC algorithm.

Parameters

- **CorrectIndex** (*'List'*) – Ground truth indexes of the correct vector field samples.
- **VFCIndex** (*'List'*) – Indexes of the correct vector field samples learned by VFC.
- **siz** (*'int'*) – Number of initial matches.

Returns

- *A tuple of precision, recall, corrRate*
- **Precision, recall, corrRate** (*Precision and recall of VFC, percentage of initial correct matches.*)
- See also:: `sparseVFC()`.

2.4 Release notes

Information to be added.

2.5 Reference

2.6 Acknowledgement

We would like to sincerely thank the developers of velocyto (La Manno Gioele and others), scanpy (Alex Wolf and others) and svelo (Volker Bergen and others) on their amazing tools which demonstrate the best practice of scientific programming in Python. Dynamo takes various technical inspiration from those packages. It also provides full compatibilities with them. Velocity estimations from either velocyto or scvelo can both be used as input in dynamo to learn the functional form of vector field and then to predict the cell fate over extended time period as well as to map global cell state potential.

2.7 Zebrafish pigmentation

This tutorial uses data from Saunders, et al (2019). Special thanks also go to Lauren for the tutorial improvement.

In this study, the authors profiled thousands of neural crest-derived cells from trunks of post-embryonic zebrafish. These cell classes include pigment cells, multipotent pigment cell progenitors, peripheral neurons, Schwann cells, chromaffin cells and others. These cells were collected during an active period of post-embryonic development, which has many similarities to fetal and neonatal development in mammals, when many of these cell types are migrating and differentiating as the animal transitions into its adult form. This study also explores the role of thyroid hormone (TH), a common endocrine factor, on the development of these different cell types.

Such developmental and other dynamical processes are especially suitable for dynamo analysis as dynamo is designed to accurately estimate direction and magnitude of expression dynamics (`RNA velocity`), predict the entire lineage trajectory of any intial cell state (`vector field`), characterize the structure (`vector field topology`) of full gene expression space, as well as fate commitment potential (`single cell potential`).

[ ]:
```
# get the latest pypi version
# to get the latest version on github and other installations approaches, see:
# https://dynamo-release.readthedocs.io/en/latest/ten_minutes_to_dynamo.html#how-to-
↪install
!pip install dynamo-release --upgrade --quiet
```

Import the package and silence some warning information (mostly `is_categorical_dtype` warning from anndata)

[1]:
```
import warnings
warnings.filterwarnings('ignore')

import dynamo as dyn
```

this is like R's sessionInfo() which helps you to debug version related bugs if any.

[2]:
```
dyn.get_all_dependencies_version()
```

```
package dynamo-release umap-learn anndata cvxopt hdbscan loompy matplotlib  \
version         0.95.2      0.4.6   0.7.4  1.2.3  0.8.26  3.0.6      3.3.0

package   numba   numpy pandas pynndescent python-igraph scikit-learn  scipy  \
version  0.51.0  1.19.1  1.1.1       0.4.8          0.8.2       0.23.2  1.5.2

package seaborn setuptools statsmodels    tqdm  trimap numdifftools colorcet
version   0.9.0     49.6.0      0.11.1  4.48.2  1.0.12       0.9.39    2.0.2
```

emulate ggplot2 plotting style with white background

[3]:
```
dyn.configuration.set_figure_params('dynamo', background='white')
```

2.7.1 Load data

Dynamo comes with a few builtin sample datasets so you can familiarize with dynamo before analyzing your own dataset. You can read your own data via `read`, `read_loom`, `read_h5ad`, `read_h5` (powered by the anndata package) or load_NASC_seq, etc. Here I just load the zebrafish sample data that comes with dynamo. This dataset has 4181 cells and 16940 genes. Its `.obs` attribute also included `condition`, `batch` information from the original study (you should also store those information to your `.obs` attribute which is essentially a Pandas Dataframe, see more at anndata). `Cluster`, `Cell_type`, umap coordinates that was originally analyzed with Monocle 3 are also provided.

[4]:
```
adata = dyn.sample_data.zebrafish()
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

After loading data, you are ready to performs some preprocessing. You can run the `recipe_monocle` function that uses similar but generalized strategy from Monocle 3 to normalize all datasets in different layers (the spliced and unspliced or new, i.e. metabolic labelled, and total mRNAs or others), followed by feature selection, `log1p` transformation of the data and PCA dimension reduction. `recipe_monocle` also does a few additionl steps, which include:

- converting ensemble gene names to gene official name and set them as `.var_names` if needed.
- calculating number of expressed genes (`nGenes`), total expression values (`nCounts`), percentage of total mitochondria gene values (`pMito`) for each cell and save them to `.obs`.
- detecting your experiment type (conventional scRNA-seq or time-resolved metabolic labeling datasets) and set certain proper layers (i.e. ignore some unconventional layers provided by the users) to be size factor normalized, `log1p` transformed, etc.
- makings cell (`.obs_names`) and gene names (`.var_names`) unique.
- savings data in `.layers` as `csr` sparse matrix for the purpose of memory efficency.
- adding collapsed `new, total` and `unspliced, spliced` layers from the `uu, ul, su, sl` layers of a metabolic labeling experiment.
- calculating each cell's cell cycle stage score.
- calculating new to total ratio (`ntr`) for each gene and cell.

Note that by default, we don't filter any cells or genes for your adata object to avoid the trouble of losing your favorite genes/cells. However, if your dataset is huge, we recommend filtering them by setting `keep_filtered_cells=False, keep_filtered_genes=False` in `recipe_monocle`.

[5]:
```
dyn.pp.recipe_monocle(adata)
```

[5]:
```
AnnData object with n_obs × n_vars = 4181 × 16940
    obs: 'split_id', 'sample', 'Size_Factor', 'condition', 'Cluster', 'Cell_type',
↪'umap_1', 'umap_2', 'batch', 'nGenes', 'nCounts', 'pMito', 'use_for_pca', 'spliced_
↪Size_Factor', 'initial_spliced_cell_size', 'unspliced_Size_Factor', 'initial_
↪unspliced_cell_size', 'initial_cell_size', 'ntr'
    var: 'pass_basic_filter', 'score', 'log_m', 'log_cv', 'use_for_pca', 'ntr'
    uns: 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_
↪fit', 'feature_selection'
```

(continues on next page)

(continued from previous page)

```
    obsm: 'X_pca', 'X'
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced'
```

2.7.2 RNA velocity with parallelism

RNA velocity ($\frac{ds}{dt}$) for conventional scRNA-seq is just $\frac{ds}{dt} = \beta u - \gamma s$ (while u/s is the unspliced or spliced mRNA respectively.β is splicing rate and is generally assumed to be 1 while γ is degration rate and is what we need to estimate). To estimate gamma for conventional scRNA-seq data, we provided three approaches `deterministic`, `stochastic` and `negbin`. The first one is equivalent to velocyto's implementation or scvelo's deterministic mode while the second one scvelo's stochastic mode. Negative binomal is a novel method from us that relies on the negative binomial formulation of gene exrpession distribution at steady state. Furthermore, we support *multi-core* parallelism of gamma estimation so you can analyze your large single-cell datasets with dynamo efficiently.

`dyn.tl.dynamics` function combines gamma estimation and velocity calculation in one-shot. Furthermore, it implicitly calls `dyn.tl.moments` first, and then performs the following steps:

- checks the data you have and determines the experimental type automatically, either the conventional scRNA-seq, kinetics, degradation or one-shot single-cell metabolic labelling experiment or the CITE-seq or REAP-seq co-assay, etc.
- learns the velocity for each feature gene using either the original deterministic model based on a steady-state assumption from the seminal RNA velocity work or a few new methods, including the stochastic (default) or negative binomial method for conventional scRNA-seq or kinetic, degradation or one-shot models for metabolic labeling based scRNA-seq.

Those later methods are based on moment equations which basically considers both mean and uncentered variance of gene expression. The moment based models require calculation of the first and second moments of the expression data, which relies on the cell nearest neighbours graph, constructed in the reduced PCA space from the spliced or total mRNA expression.

[6]:
```
dyn.tl.dynamics(adata, model='stochastic', cores=3)
# or dyn.tl.dynamics(adata, model='deterministic')
# or dyn.tl.dynamics(adata, model='stochastic', est_method='negbin')
```

[6]:
```
AnnData object with n_obs × n_vars = 4181 × 16940
    obs: 'split_id', 'sample', 'Size_Factor', 'condition', 'Cluster', 'Cell_type',
↪'umap_1', 'umap_2', 'batch', 'nGenes', 'nCounts', 'pMito', 'use_for_pca', 'spliced_
↪Size_Factor', 'initial_spliced_cell_size', 'unspliced_Size_Factor', 'initial_
↪unspliced_cell_size', 'initial_cell_size', 'ntr'
    var: 'pass_basic_filter', 'score', 'log_m', 'log_cv', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics'
    uns: 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_
↪fit', 'feature_selection', 'dynamics'
    obsm: 'X_pca', 'X'
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced', 'M_u', 'M_uu', 'M_s',
↪'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con'
```

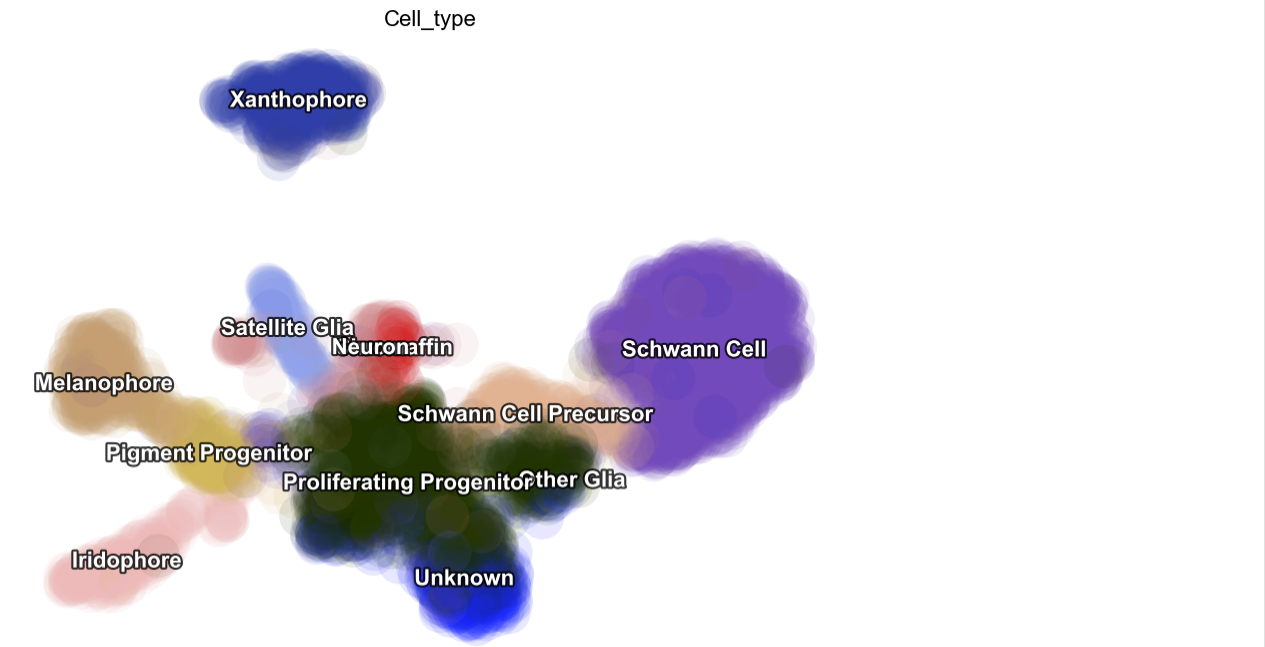
Next we perform dimension reduction (by default, UMAP) and visualize the UMAP embedding of cells. The provided `Cell_type` information is also used to color cells. To get cluster/cell type information for your own data, dynamo also provides facilities to perform clustering and marker gene detection. By default we use HDBSCAN for clustering. HDBSCAN package was developed also by Leland McInnes, the developer of **UMAP**. You may clustering your single

cells in UMAP space (set `basis='umap'` instead of the default `pca` in HDBSCAN). See more discussion aboout this here.

For marker gene detection, please check functions in **Markers and differential expressions** section in our API. A more detailed tutorial designated for this will be released soon.

[7]:
```
dyn.tl.reduceDimension(adata)

dyn.pl.umap(adata, color='Cell_type')
```

```
<Figure size 600x400 with 0 Axes>
```



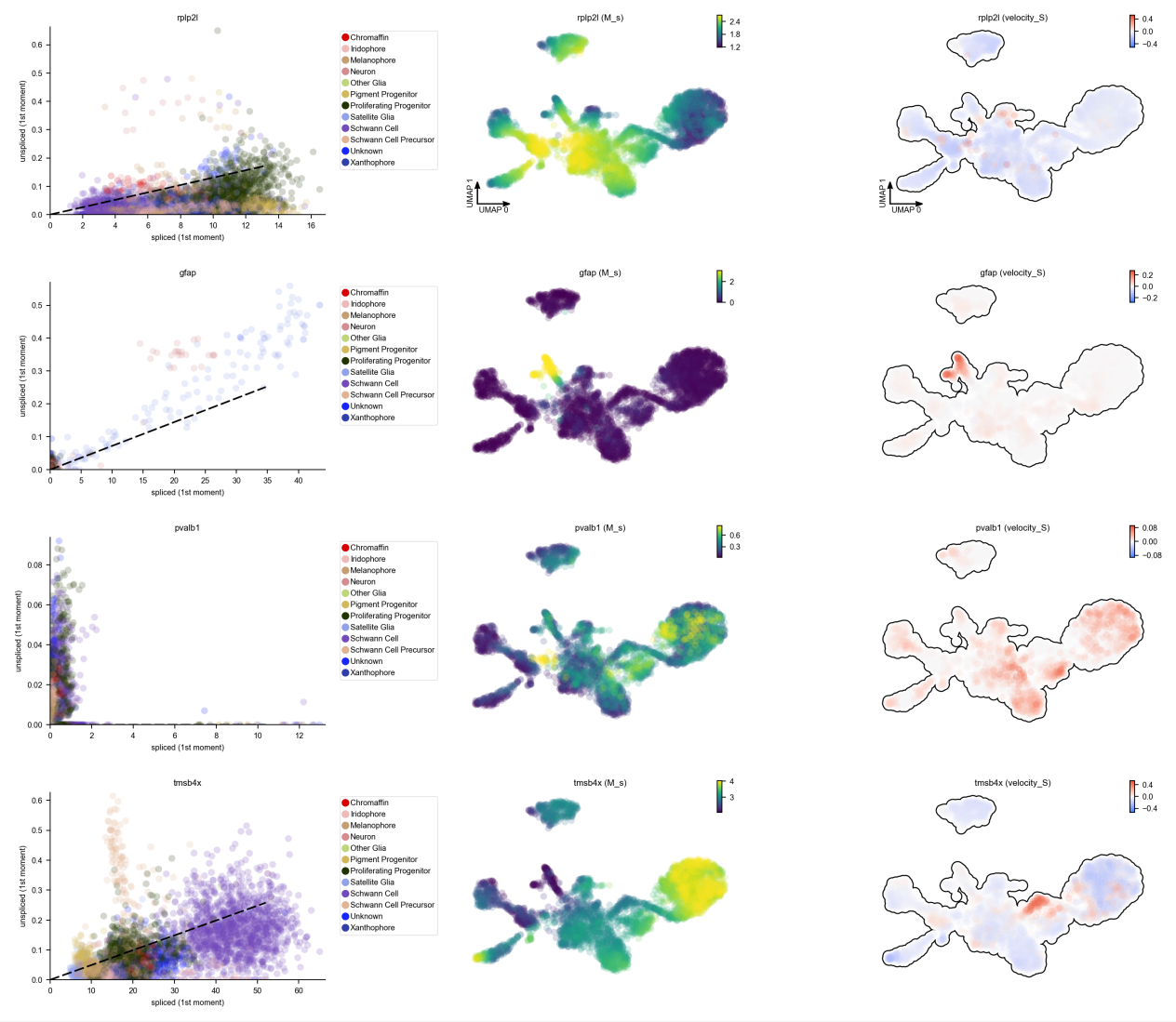
Kinetic estimation of the conventional scRNA-seq and metabolic labeling based scRNA-seq is often tricky and has a lot pitfalls. Sometimes you may even observed undesired backward vector flow. You can evaluate the confidence of gene-wise velocity via:

```
dyn.tl.gene_wise_confidence(adata, group='group', lineage_dict={'Progenitor': [
↪'terminal_cell_state']})
```

Here `group` is the column for the group informations for cells in the `.obs`. `lineage_dict` is a dictionary indicates broad lineage information in which key points to the progenitor group while values (a list) are the possible terminal cell groups, all from the `group` column.

In the following, let us have a look at the phase diagram of some genes that have velocity calculated. You will see the `pvalb1` gene has a strange phase diagram with a few cells have high spliced expression values but extremely low unspliced expression values. Those kind of phase space may points to inproper intron capture of those genes during the library prepartion or sequencing and they should never be used for velocity projection and vector field analysis. A tutorial with details for identifying those genes, evaluating the confidence of velocity estimation and then correcting (briefly mentioned below) the RNA velocity results will be released soon.

[8]:
```
dyn.pl.phase_portraits(adata, genes=adata.var_names[adata.var.use_for_dynamics][:4],␣
↪figsize=(6, 4), color='Cell_type')
```



2.7.3 Velocity projection

In order to visualize the velocity vectors, we need to project the high dimensional velocity vector of cells to lower dimension (although dynamo also enables you to visualize raw gene-pair velocity vectors, see below). The projection involves calculating a transition matrix between cells for local averaging of velocity vectors in low dimension. There are three methods to calculate the transition matrix, either `kmc`, `cosine`, `pearson`. `kmc` is our new approach to learn the transition matrix via diffusion approximation or an Itô kernel. `cosine` or `pearson` are the methods used in the original velocyto or the scvelo implementation. Kernels that are based on the `reconstructed vector field` in high dimension is also possible and maybe more suitable because of its and robustness and smoothness. We will show you how to do that in another tutorial soon!

[9]:
```
dyn.tl.cell_velocities(adata, method='pearson', other_kernels_dict={'transform': 'sqrt
↪'})
```

```
calculating transition matrix via pearson kernel with sqrt transform.: 100%|| 4181/
↪4181 [00:10<00:00, 409.52it/s]
projecting velocity vector to low dimensional embedding...: 100%|| 4181/4181 [00:01
↪<00:00, 3947.53it/s]
```

[9]:
```
AnnData object with n_obs × n_vars = 4181 × 16940
    obs: 'split_id', 'sample', 'Size_Factor', 'condition', 'Cluster', 'Cell_type',
↪'umap_1', 'umap_2', 'batch', 'nGenes', 'nCounts', 'pMito', 'use_for_pca', 'spliced_
↪Size_Factor', 'initial_spliced_cell_size', 'unspliced_Size_Factor', 'initial_
↪unspliced_cell_size', 'initial_cell_size', 'ntr'
    var: 'pass_basic_filter', 'score', 'log_m', 'log_cv', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics', 'use_for_transition'
    uns: 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_
↪fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit', 'grid_velocity_umap'
    obsm: 'X_pca', 'X', 'X_umap', 'velocity_umap'
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced', 'M_u', 'M_uu', 'M_s',
↪'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con', 'connectivities', 'distances', 'pearson_transition_matrix'
```

You can check the confidence of cell-wise velocity to understand how reliable the recovered velocity is across cells or even `correct` velocty based on some prior:

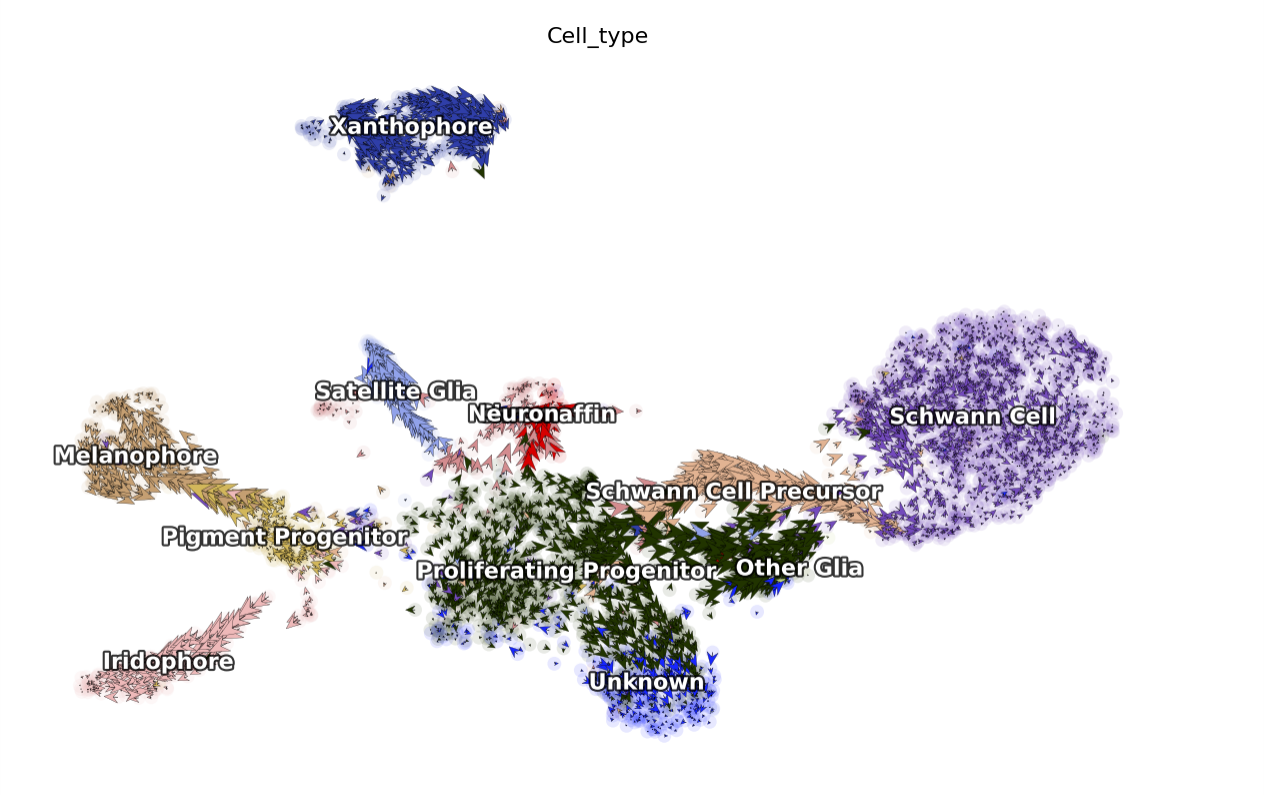
```
dyn.tl.cell_wise_confidence(adata, basis='pca')
dyn.tl.confident_cell_velocities(adata, group='group', lineage_dict={'Progenitor': [
↪'terminal_cell_state']},)
```

There are three methods implemented for calculating the cell wise velocity confidence metric. By default it uses `jaccard` index, which measures how well each velocity vector meets the geometric constraints defined by the local neighborhood structure. Jaccard index is calculated as the fraction of the number of the intersected set of nearest neighbors from each cell at current expression state (X) and that from the future expression state (X + V) over the number of the union of these two sets. The `cosine` or `correlation` method is similar to that used by scVelo.

Next let us visualize the projected RNA velocity. We can see that the recovered RNA velocity flow shows a nice transition from `proliferating progenitors` to `pigment progenitors` which then bifurcate into either `melanophore` or `iridopore` on the left. In the middle, the `proliferating progenitors` bifurcate upward into either `chromaffin`, `neuron` or `satellite glia cells`. On the right, the `proliferation progenitors` bifurcate into either `Schwann cell precursor` which then become `Schwann cells` or `other glia`. In the bottom, some `proliferating progenitors`choose to become an `unkown` cell lineage. In addition, the `xanthophore` cells are seem to be an outlier group on the top, indicating it has a different lineage path comparing to `melanophore` or `iridophore` pigment cells. The transcriptional discontinuity from multipotent progenitors to `xanthophore` cells may also imply its lineage trajectory is more rapid comparing to that of `melanophore` or `iridophore` pigment cells.

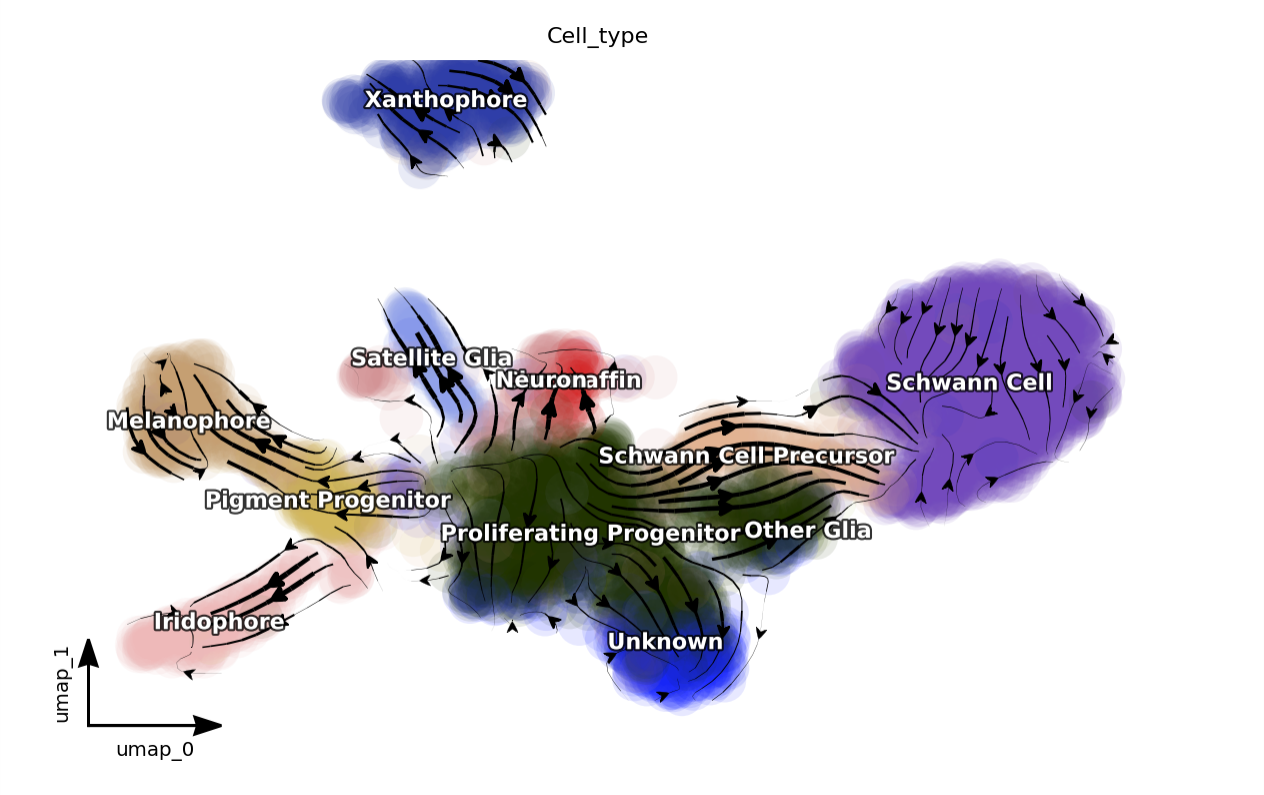
[10]:
```
dyn.pl.cell_wise_vectors(adata, color=['Cell_type'], basis='umap', show_legend='on␣
↪data', quiver_length=6, quiver_size=6, pointsize=0.1, show_arrowed_spines=False)
```

```
<Figure size 600x400 with 0 Axes>
```



[11]:
```
dyn.pl.streamline_plot(adata, color=['Cell_type'], basis='umap', show_legend='on data
↪', show_arrowed_spines=True)
```

```
<Figure size 600x400 with 0 Axes>
```



Note that, if you pass `x='gene_a', y='gene_b'` to `cell_wise_vectors`, `grid_vectors` or `streamline_plot`, you can visualize the raw gene-pair velocity flows. `gene_a` and `gene_b` need to have velocity calculated (or `use_for_dynamics` in `.var` for those genes are `True`)

2.7.4 Reconstruct vector field

In classical physics, including fluidics and aerodynamics, velocity and acceleration vector fields are used as fundamental tools to describe motion or external force of objects, respectively. In analogy, RNA velocity or protein accelerations estimated from single cells can be regarded as sparse samples in the velocity (La Manno et al. 2018) or acceleration vector field (Gorin, Svensson, and Pachter 2019) that defined on the gene expression space.

In general, a vector field can be defined as a vector-valued function f that maps any points (or cells' expression state) x in a domain with D dimension (or the gene expression system with D transcripts / proteins) to a vector y (for example, the velocity or acceleration for different genes or proteins), that is f(x) = y.

To formally define the problem of velocity vector field learning, we consider a set of measured cells with pairs of current and estimated future expression states. The difference between the predicted future state and current state for each cell corresponds to the velocity. We suppose that the measured single-cell velocity is sampled from a smooth, differentiable vector field f that maps from xi to yi on the entire domain. Normally, single cell velocity measurements are results of biased, noisy and sparse sampling of the entire state space, thus the goal of velocity vector field reconstruction is to robustly learn a mapping function f that outputs yj given any point xj on the domain based on the observed data with certain smoothness constraints (Jiayi Ma et al. 2013). Under ideal scenario, the mapping function f should recover the true velocity vector field on the entire domain and predict the true dynamics in regions of expression space that are not sampled. To reconstruct vector field function in dynamo, you can simply use the following function to do all the heavy-lifting:

[12]:
```
# you can set `verbose = 1/2/3` to obtain different levels of running information of␣
↪vector field reconstruction
dyn.vf.VectorField(adata, basis='umap', M=1000, pot_curl_div=True)
```

```
Constructing diffusion graph from reconstructed vector field: 4181it [03:22, 20.61it/
↪s]
Calculating 2-D curl: 100%|| 4181/4181 [00:00<00:00, 11763.55it/s]
Calculating divergence: 100%|| 4181/4181 [00:00<00:00, 10544.14it/s]
```

Vector field recunstruction is blazingly efficient and scale linearly with the cell number and dimensions. So you can do vector field reconstruction for hundred thousands of cells in PCA space on a matter of minutes. How good your vector field reconstruction is? We have several metrics to quantify that and we will provide a detailed tutorial on that in a couple of days. The easiest way, though, is to check the energy / energy change rate to see whether they are decreasing and gradually stabiling during the vector field learning process:

```
dyn.pl.plot_energy(adata)
```

2.7.5 Characterize vector field topology

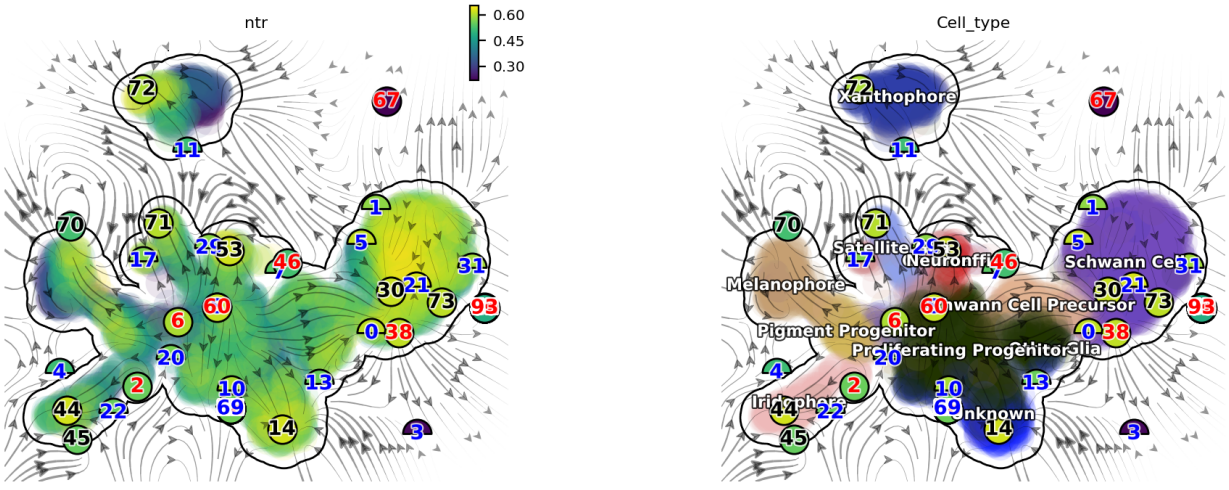
Since we learn the vector field function of the data, we can then characterize the topology of the full vector field space. For example, we are able to identify

- the fixed points (attractor/saddles, etc.) which may corresponds to terminal cell types or progenitors;
- nullcline, separatrices of a recovered dynamic system, which may formally define the dynamical behaviour or the boundary of cell types in gene expression space.

Note that we use the name of `topography` instead of `topology` in `tools` or `plot` modules because we figured out the 2D full vector field plot (instead of just domains with cells as those visualized by `streamline_plot` function) with those fixed points, nullclines, etc. looks like a topography plot. Enlighten us if you have a better idea. And see also more discussion here.

When we recostruct a 2 D vector field (which is the case above), we automatically characterize the vector field topology. Let us take a look a the fixed points identified by dynamo for this system.

[13]:
```
dyn.pl.topography(adata, basis='umap', background='white', color=['ntr', 'Cell_type'],
↪ streamline_color='black', show_legend='on data', frontier=True)
```



There are a lot of fixed points identified by dynamo. Some of them are less confident than others and we use the filled color of each node to represent the confidence. The shape of node also has some meaning. Half circles are saddle points while full circle are stable fixed points (the eigenvalue of the jacobian matrix at those places are all negative based on the reconstructed vector field). The color of digits in each node is related to the type of fixed points:

- black: absorbing fixed points;
- red: emitting fixed points;
- blue: unstable fixed points.

We notice that, interesting, `node 6` corresponds an emitting fixed point which makes sense as it is located in the domain of progenitor cell state; on the other hand, `nodes 70, 44, 14 and 72` are absorbing fixed points, and each corresponds to the `melanophore`, `iridophore`, `unknown` or the `xanthophore` terminal cell type state. Lastly, `nodes 20 and 29` are unstable fixed points (saddle points), each corresponds to the bifurcation point of the iridophore and melanophore lineages or that of the neuron and satellite glia lineages.

So overall this topology analysis did a pretty good job!

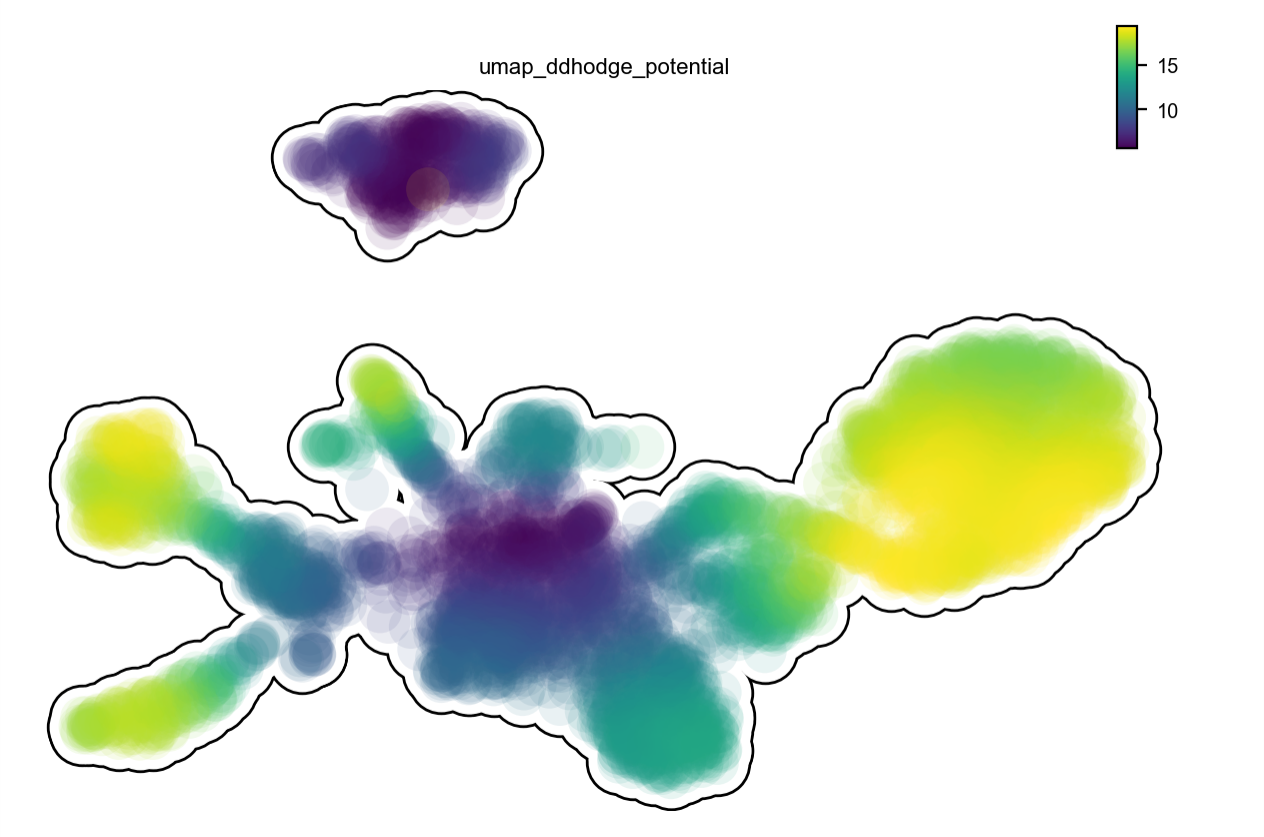
The concept of potential landscape is widely appreciated across various biological disciplines, for example the adaptive landscape in population genetics, protein-folding funnel landscape in biochemistry, epigenetic landscape in developmental biology. In the context of cell fate transition, for example, differentiation, carcinogenesis, etc, a potential landscape not only offers an intuitive description of the global dynamics of the biological process but also provides key insights to understand the multi-stability and transition rate between different cell types as well as to quantify the optimal path of cell fate transition.

The classical definition of potential function in physics requires gradient systems (no curl/cycling part), it thus is often not applicable to open biological system. In dynamo we provided several ways to quantify the potential of single cells by decomposing the vector field into gradient, curl parts, etc and use the gradient part to define potential. The recommended method is built on the Hodge decomposition on simplicial complexes (a sparse directional graph) constructed based on the learned vector field function that provides fruitful analogy of gradient, curl and harmonic (cyclic) flows on manifold.

Single cell potential (In fact, it is the negative of potential here for the purpose to match up with the common usuage of `pseudotime` so that small values correspond to the progenitor state while large values terminal cell states.) estimated by dynamo can be regarded as a replacement of pseudotime. Since dynamo utilizes velocity which consists of direction and magnitude of cell dynamics, potential should be more relevant to real time and intrinsically directional (so you don't need to orient the trajectory).

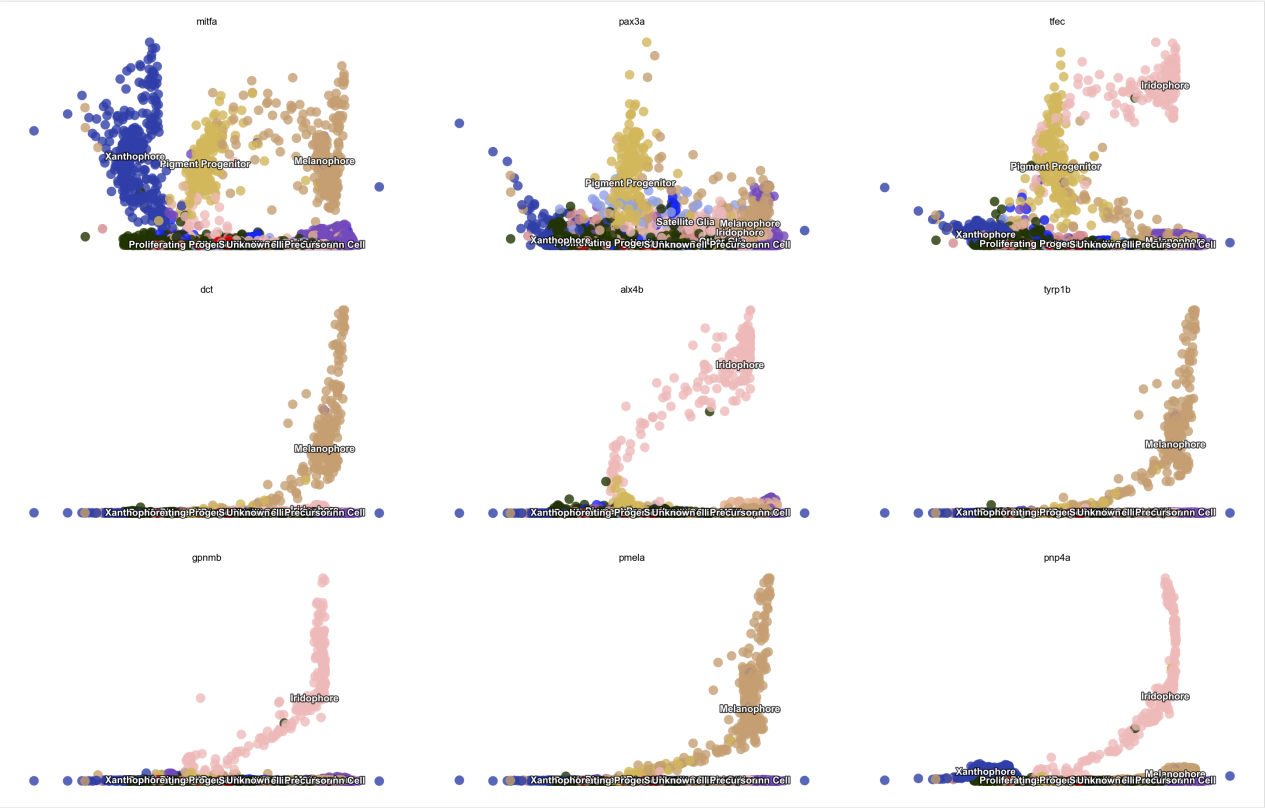
[14]:
```
dyn.pl.umap(adata,  color='umap_ddhodge_potential', frontier=True)
```

```
<Figure size 600x400 with 0 Axes>
```



Here we can check a few genes from `figure 3 (si 5)` of Saunders, et al (2019) to see their expression dynamics over time. As expected, we can see that `mitfa` expression declined only marginally with melanophore differentiation yet decreased markedly with a transition from progenitor to iridophore as expected (Curran et al., 2010). `pax3a` was expressed in pigment progenitors and decreased across pseudotime in melanophores, whereas expression of `tfec`, a transcription factor expressed in iridophores (Lister et al., 2011), increased over pseudotime. Melanin synthesis enzyme genes, `dct` and `tyrp1b`, as well as `pmel`, encoding a melanosome-associated transmembrane protein, all increased over pseudotime in melanophores. In iridophores, `gpnmb` and `pnp4a` showed elevated expression late in pseudotime, as expected (Curran et al., 2010; Higdon et al., 2013).

[15]:
```
import numpy as np

fig3_si5 = ['mitfa', 'pax3a', 'tfec', 'dct', 'alx4b', 'tyrp1b', 'gpnmb', 'pmela',
↪'pnp4a']

dyn.pl.scatters(adata, x=np.repeat('umap_ddhodge_potential', 9), pointsize=0.25,␣
↪alpha=0.8, y=fig3_si5, layer='X_spliced', color='Cell_type',
                ncols=3, background='white', figsize=(7, 4))
```



2.7.6 Beyond RNA velocity

Here let us take a glimpse on how dynamo can go beyond RNA velocity analysis by taking advantage of the analytical vector field function it learns. Here we will first project the RNA velocity to pca space and then reconstruct the vector field function in the PCA space. We then followed by calculating `curl` (`curl` is calculated in 2 dimensional UMAP space by default as it is only defined in 2/3 dimension), `divergence`, `acceleration` and `curvature`. Those calculations are incredibly efficient (on the order of seconds for ten thousands of cells in 30 PCs) as they are calculated analytically based on the reconstructed vector field function.

- **curl**: a quantity to characterize the infinitesimal rotation of a cell state based on the reconstructed vector field.
  - in 2D, curl is a value; in 3D curl, is a matrix.
  - if rotation is clockwise, 2D curl has negative value and vice versa
  - combinbing with expression of cell cycle markers, curl analysis can help us to reveal whether a cell is going through a strong cell cycle process.
- **divergence**: a quantity to characterize local "outgoingness" of a cell – the extent to which there is more of the field vectors exiting an infinitesimal region of space than entering it.
  - positive values means cells is going out to become other cells or cell's movement to other cell is speeded up and vice versa.
  - divergence analysis can be used to reveal progenitor (source) or a terminal cell state (sink).
- **acceleration**: the derivative of velocity vector.
  - if cell speeds up (normally happen when cells exit cell cycle and start to commit), the acceleration will be positive and vice versa.

  - RNA acceleration is a vector like RNA velocity vector so you can actually plot acceleration field like velocity field (that is why we name our vector flow related plotting functions `cell_wise_vectors`, `grid_vectors` to support plotting both `velocity` and `acceleration field` (see below)).
  - Here the norm of the acceleration for all PC components in each cells will be calculated and visualized (like the **speed/magnitude** of the velocity vector).

- **curvature**: a quantity to characterize the curviness a cell's vector field trajectory.
  - if a progenitor develops into multiple lineages, some of those paths will have curvature (it is like making a turn on a crossroad while driving a car).
  - genes strongly contribute to the curvature correspond to regulatory genes steering the cell fate

[16]:
```
dyn.tl.cell_velocities(adata, basis='pca')
dyn.vf.VectorField(adata, basis='pca')
dyn.vf.speed(adata, basis='pca')
dyn.vf.curl(adata, basis='umap')
dyn.vf.divergence(adata, basis='pca')
dyn.vf.acceleration(adata, basis='pca')
dyn.vf.curvature(adata, basis='pca')
```

```
projecting velocity vector to low dimensional embedding...:  16%|        | 651/4181␣
↪[00:00<00:01, 3228.03it/s]
```

```
Using existing pearson_transition_matrix found in .obsp.
```

```
projecting velocity vector to low dimensional embedding...: 100%|| 4181/4181 [00:01
↪<00:00, 3505.59it/s]
Calculating 2-D curl: 100%|| 4181/4181 [00:00<00:00, 11446.05it/s]
Calculating divergence: 100%|| 4181/4181 [00:00<00:00, 6807.54it/s]
Calculating acceleration: 100%|| 4181/4181 [00:00<00:00, 124851.45it/s]
Calculating acceleration: 100%|| 4181/4181 [00:00<00:00, 166839.99it/s]
Calculating curvature: 100%|| 4181/4181 [00:00<00:00, 48393.99it/s]
```

2.7.7 Integrative analysis

We can integrate those above quantities to fully characterize the regulatory mechanism during zebrafish pigmentation.

A separate tutorial is needed to fully explore these analyses, but let's take a quick look at the results. We can see that:

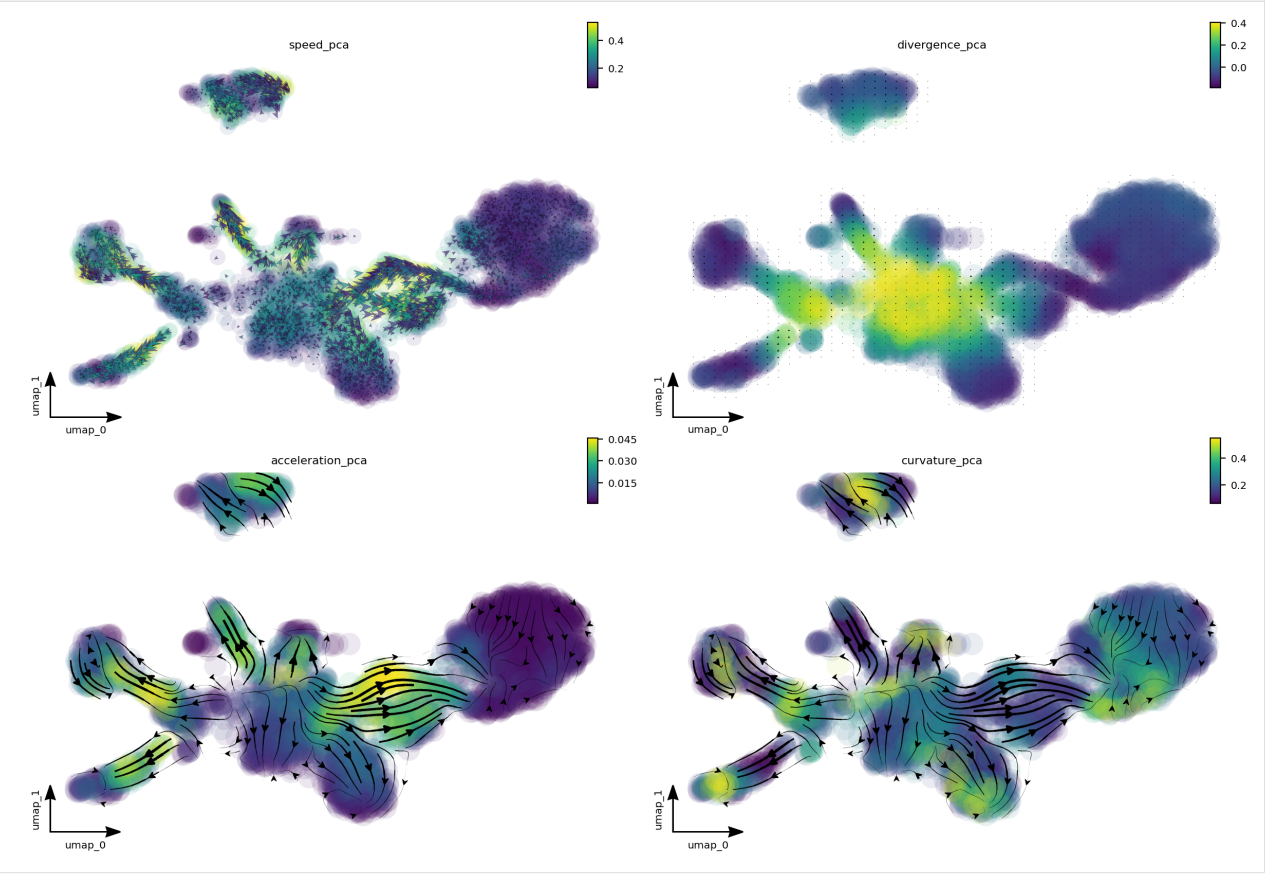
- from cell speed and acceleration, progenitors generally have low speed as it is like a metastable cell state. However transition of pigment progenitors and proliferating progenitors speeds up after committing to a particular lineage, for example, iridophore/melanophore/shawnn cell lineage, etc.
- from cell divergence, those progenitors (pigment progenitors and proliferating progenitors) functions like a source with high divergence while melanophore/iridophores/chromaffin/schawn cells as well as other cell types functions like a sink with significantly lower divergence.
- from cell curvature, when cell makes cell fate decisions (at the bifurcation point of iridophore and melanophore lineages or that of the neuron and satellite glia lineages), strong curvature is apparent. Curvature is also artificially strong when velocity is noisy.

[17]:
```
import matplotlib.pyplot as plt

fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True, figsize=(12,␣
↪8))
f1_axes
f1_axes[0, 0] = dyn.pl.cell_wise_vectors(adata, color='speed_pca', pointsize=0.5,␣
↪alpha = 0.7, ax=f1_axes[0, 0], quiver_length=6, quiver_size=6, save_show_or_return=
↪'return')
```

(continues on next page)

(continued from previous page)

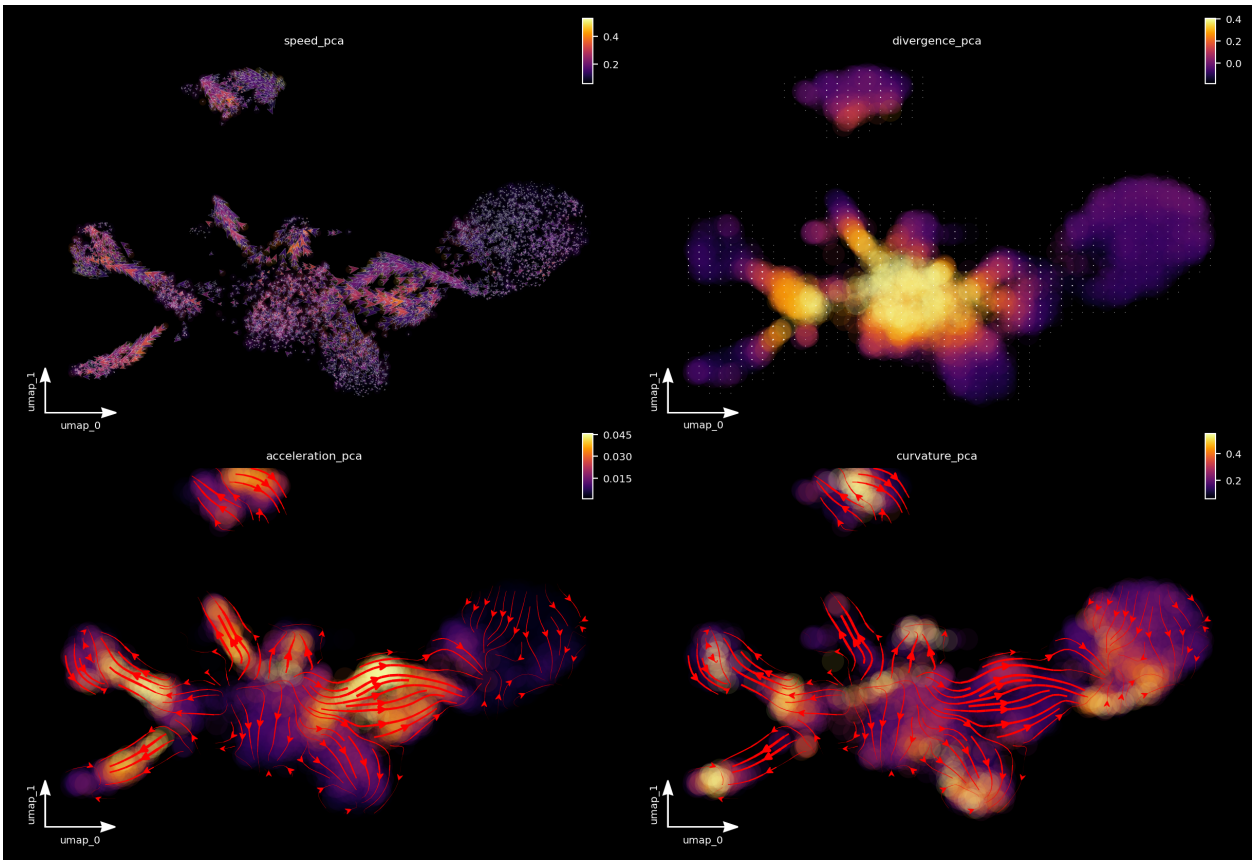
```
f1_axes[0, 1] = dyn.pl.grid_vectors(adata, color='divergence_pca', ax=f1_axes[0, 1],␣
↪quiver_length=12, quiver_size=12, save_show_or_return='return')
f1_axes[1, 0] = dyn.pl.streamline_plot(adata, color='acceleration_pca', ax=f1_axes[1,␣
↪0], save_show_or_return='return')
f1_axes[1, 1] = dyn.pl.streamline_plot(adata, color='curvature_pca', ax=f1_axes[1, 1],
↪ save_show_or_return='return')
plt.show()
```



Emulate ggplot2 plotting styple with black background, get ready for a cool presentation!!!

[18]:
```
dyn.configuration.set_figure_params('dynamo', background='black')
```

[19]:
```
fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True, figsize=(12,␣
↪8))
f1_axes
f1_axes[0, 0] = dyn.pl.cell_wise_vectors(adata, color='speed_pca', pointsize=0.1,␣
↪alpha = 0.7, ax=f1_axes[0, 0], quiver_length=6, quiver_size=6, save_show_or_return=
↪'return', background='black')
f1_axes[0, 1] = dyn.pl.grid_vectors(adata, color='divergence_pca', ax=f1_axes[0, 1],␣
↪quiver_length=12, quiver_size=12, save_show_or_return='return', background='black')
f1_axes[1, 0] = dyn.pl.streamline_plot(adata, color='acceleration_pca', ax=f1_axes[1,␣
↪0], save_show_or_return='return', background='black')
f1_axes[1, 1] = dyn.pl.streamline_plot(adata, color='curvature_pca', ax=f1_axes[1, 1],
↪ save_show_or_return='return', background='black')
plt.show()
```



2.7.8 Animate fate transition

Before we go, let us have some fun with animating cell fate commitment predictions via reconstructed vector field function. This cool application hopefully will also convince you that vector field reconstruction can enable some amazing analysis that is hardly imaginable before. With those and many other possibilities in single cell genomics, the prospect of biology to finally become a discipline as qualitative as physics and mathematics has never been so promising!

To animate cell fate prediction, we need to first select some progenitor cells as initial cell states.

[20]:
```
progenitor = adata.obs_names[adata.obs.Cell_type.isin(['Proliferating Progenitor',
↪'Pigment Progenitor'])]
len(progenitor)
```

[20]:
```
1194
```

Then, we need to predict the cell fate trajectory via integrating with the vector field function, starting from those initial cell states.

[21]:
```
dyn.pd.fate(adata, basis='umap', init_cells=progenitor, interpolation_num=100, ␣
↪direction='forward',
   inverse_transform=False, average=False, cores=3)
```

[21]:
```
AnnData object with n_obs × n_vars = 4181 × 16940
    obs: 'split_id', 'sample', 'Size_Factor', 'condition', 'Cluster', 'Cell_type',
↪'umap_1', 'umap_2', 'batch', 'nGenes', 'nCounts', 'pMito', 'use_for_pca', 'spliced_
↪Size_Factor', 'initial_spliced_cell_size', 'unspliced_Size_Factor', 'initial_
↪unspliced_cell_size', 'initial_cell_size', 'ntr', 'umap_ddhodge_div', 'umap_ddhodge_
↪potential', 'curl_umap', 'divergence_umap', 'speed_pca', 'divergence_pca',
↪'acceleration_pca', 'curvature_pca'
```

(continues on next page)

(continued from previous page)

```
    var: 'pass_basic_filter', 'score', 'log_m', 'log_cv', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics', 'use_for_transition'
    uns: 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_
↪fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit', 'grid_velocity_umap
↪', 'VecFld_umap', 'VecFld', 'grid_velocity_pca', 'VecFld_pca', 'curvature_pca',
↪'fate_umap'
    obsm: 'X_pca', 'X', 'X_umap', 'velocity_umap', 'velocity_umap_SparseVFC', 'X_umap_
↪SparseVFC', 'velocity_pca', 'velocity_pca_SparseVFC', 'X_pca_SparseVFC',
↪'acceleration_pca'
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced', 'M_u', 'M_uu', 'M_s',
↪'M_us', 'M_ss', 'velocity_S', 'acceleration'
    obsp: 'moments_con', 'connectivities', 'distances', 'pearson_transition_matrix',
↪'umap_ddhodge'
```

Furthermore, we need to prepare a `matplotlib axes` as the background of the animation and then the animated components from each frame will be plotted on its top. Here I use the `topography` plot as the background but you can use other plots if you like.

[22]:
```
%%capture
fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='Cell_type', ax=ax, save_show_or_return='return')
```

The `dyn.mv.*` module provides functionalities to create necessary components to produce an animation that describes the estimated speed of a set of cells at each time point, its movement in gene expression space and the long range trajectory predicted by the reconstructed vector field functions. Thus it provides intuitive visual understanding of the RNA velocity, speed, acceleration, and cell fate commitment in action!!

[23]:
```
%%capture
instance = dyn.mv.StreamFuncAnim(adata=adata, color='Cell_type', ax=ax)
```

Lastly, let us embed the animation into our notebook.

Note that here I have to set `animation.embed_limit rc parameter` to a big value (in MB) to ensure all frames of the animation will be embedded in this notebook.

[24]:
```
import matplotlib
matplotlib.rcParams['animation.embed_limit'] = 2**128 # Ensure all frames will be␣
↪embedded.

from matplotlib import animation
import numpy as np

anim = animation.FuncAnimation(instance.fig, instance.update, init_func=instance.init_
↪background,
                               frames=np.arange(100), interval=100, blit=True)
from IPython.core.display import display, HTML
HTML(anim.to_jshtml()) # embedding to jupyter notebook.
```

[24]:
```
<IPython.core.display.HTML object>
```

Alternatively, we can directly save the animation as an gif file with the `dyn.mv.animate_fates` function, using something like the following:

```
dyn.mv.animate_fates(adata, color='Cell_type', basis='umap', n_steps=100, fig=fig,␣
↪ax=ax,
                     save_show_or_return='save', logspace=True, max_time=None)
```

[25]:
```
%%capture
fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='Cell_type', ax=ax, save_show_or_return='return')
dyn.mv.animate_fates(adata, color='Cell_type', basis='umap', n_steps=100, fig=fig,␣
↪ax=ax,
                     save_show_or_return='save', logspace=True, max_time=None)
```

2.8 Pancreatic endocrinogenesis

This tutorial uses raw data from scvelo package. Special thanks go to the scvelo team!

[ ]:
```
# get the latest version from pypi
# for other installations approaches, see https://dynamo-release.readthedocs.io/en/
↪latest/ten_minutes_to_dynamo.html#how-to-install
!pip install dynamo-release --upgrade --quiet
```

[1]:
```
# from IPython.core.display import display, HTML
# display(HTML("<style>.container { width:90% !important; }</style>"))
# %matplotlib inline
import warnings
warnings.filterwarnings('ignore')

import dynamo as dyn
```

this is like R's sessionInfo()

[2]:
```
dyn.get_all_dependencies_version()
```

```
package dynamo-release umap-learn anndata cvxopt hdbscan loompy matplotlib  \
version         0.95.2      0.4.6   0.7.4  1.2.3  0.8.26  3.0.6      3.3.0

package   numba   numpy pandas pynndescent python-igraph scikit-learn  scipy  \
version  0.51.0  1.19.1  1.1.1       0.4.8          0.8.2       0.23.2  1.5.2

package seaborn setuptools statsmodels    tqdm  trimap numdifftools colorcet
version   0.9.0      49.6.0       0.11.1  4.48.2  1.0.12       0.9.39    2.0.2
```

[3]:
```
# run dynamo to get RNA velocity

dyn.configuration.set_figure_params('dynamo', background='white')

adata = dyn.sample_data.pancreatic_endocrinogenesis()

dyn.pp.recipe_monocle(adata, n_top_genes=1000, fg_kwargs={'shared_count': 20})

dyn.tl.dynamics(adata, model='stochastic')

dyn.tl.reduceDimension(adata, n_pca_components=30)

dyn.tl.cell_velocities(adata, method='pearson', other_kernels_dict={'transform': 'sqrt
↪'})
```

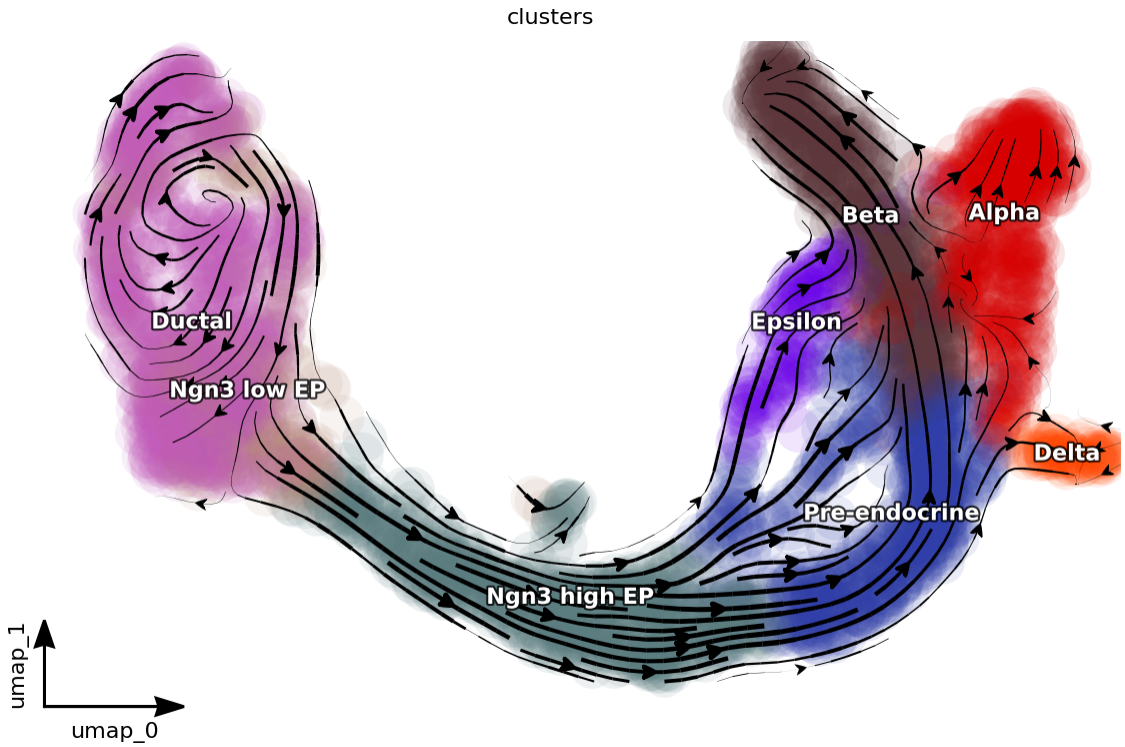
(continues on next page)

(continued from previous page)

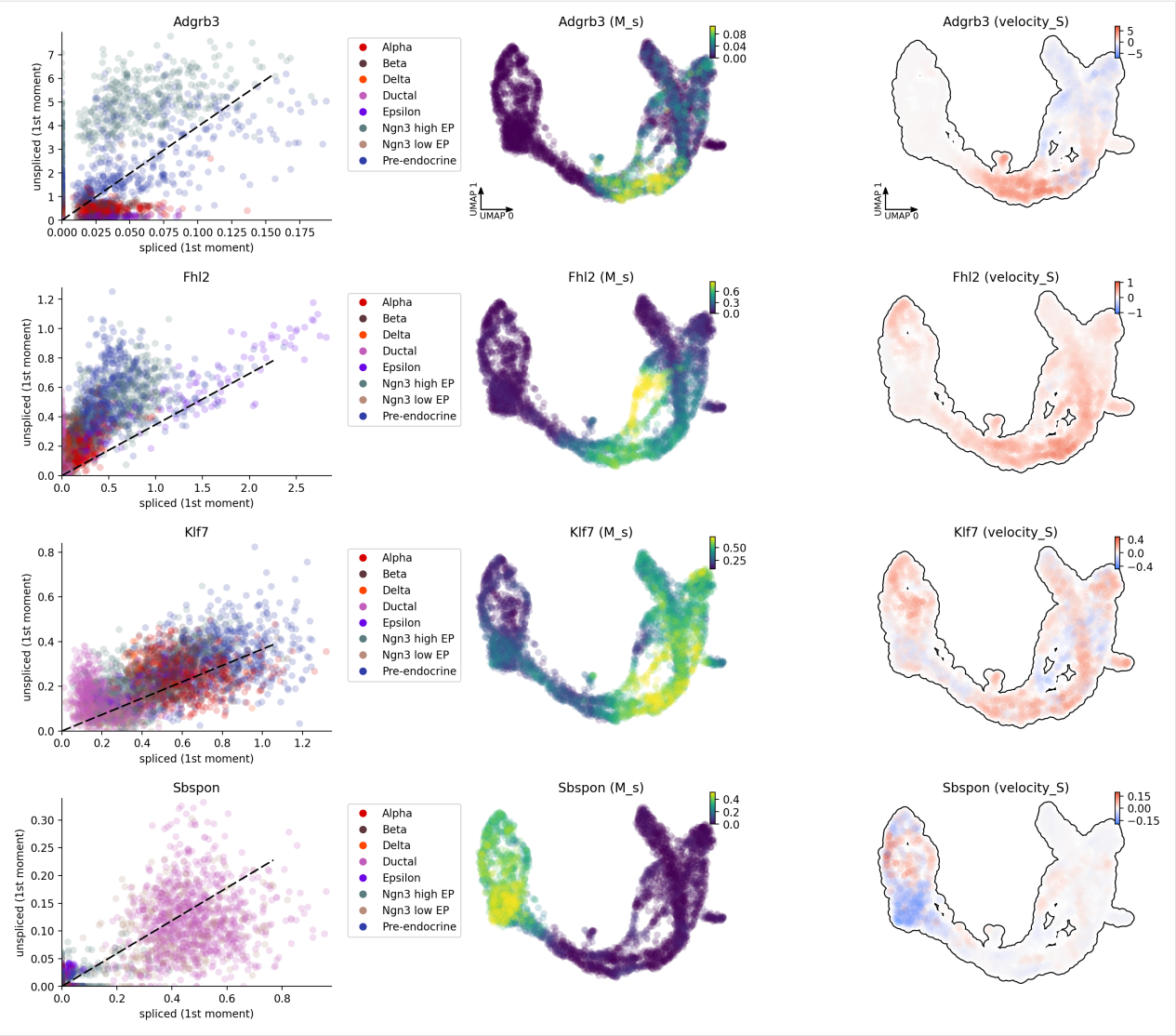
```
dyn.pl.streamline_plot(adata, color=['clusters'], basis='umap', show_legend='on data',
↪ show_arrowed_spines=True)
```

```
estimating gamma: 100%|| 1000/1000 [00:27<00:00, 36.67it/s]
calculating transition matrix via pearson kernel with sqrt transform.: 100%|| 3696/
↪3696 [00:04<00:00, 799.84it/s]
projecting velocity vector to low dimensional embedding...: 100%|| 3696/3696 [00:00
↪<00:00, 4314.30it/s]
```

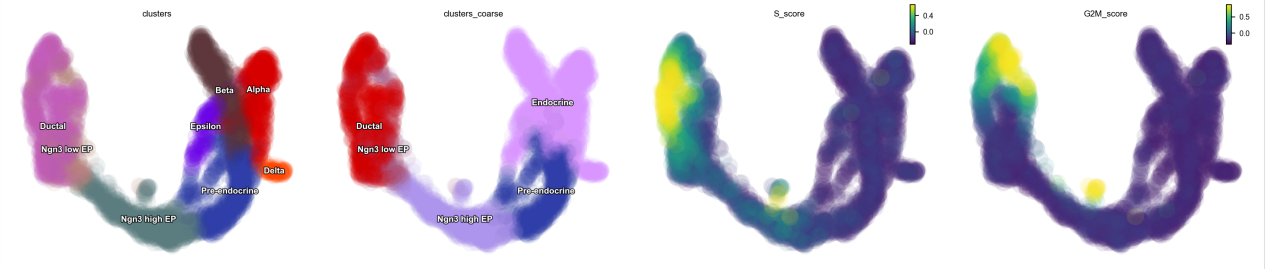
```
<Figure size 600x400 with 0 Axes>
```



[4]:
```
dyn.pl.phase_portraits(adata, genes=adata.var_names[adata.var.use_for_dynamics][:4],␣
↪figsize=(6, 4), color='clusters')
```

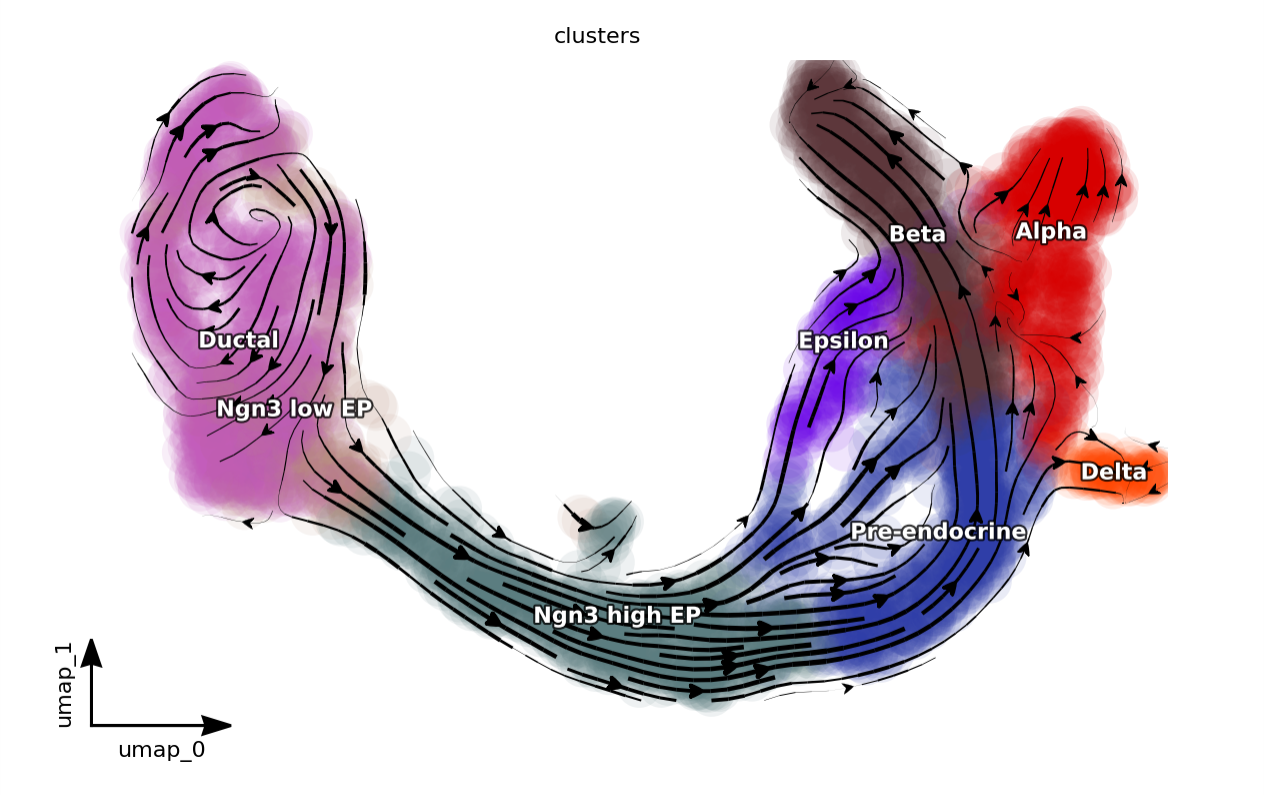


[5]:
```
dyn.pl.umap(adata, color=['clusters', 'clusters_coarse', "S_score", "G2M_score"],␣
↪ncols=4, alpha=0.1)
```



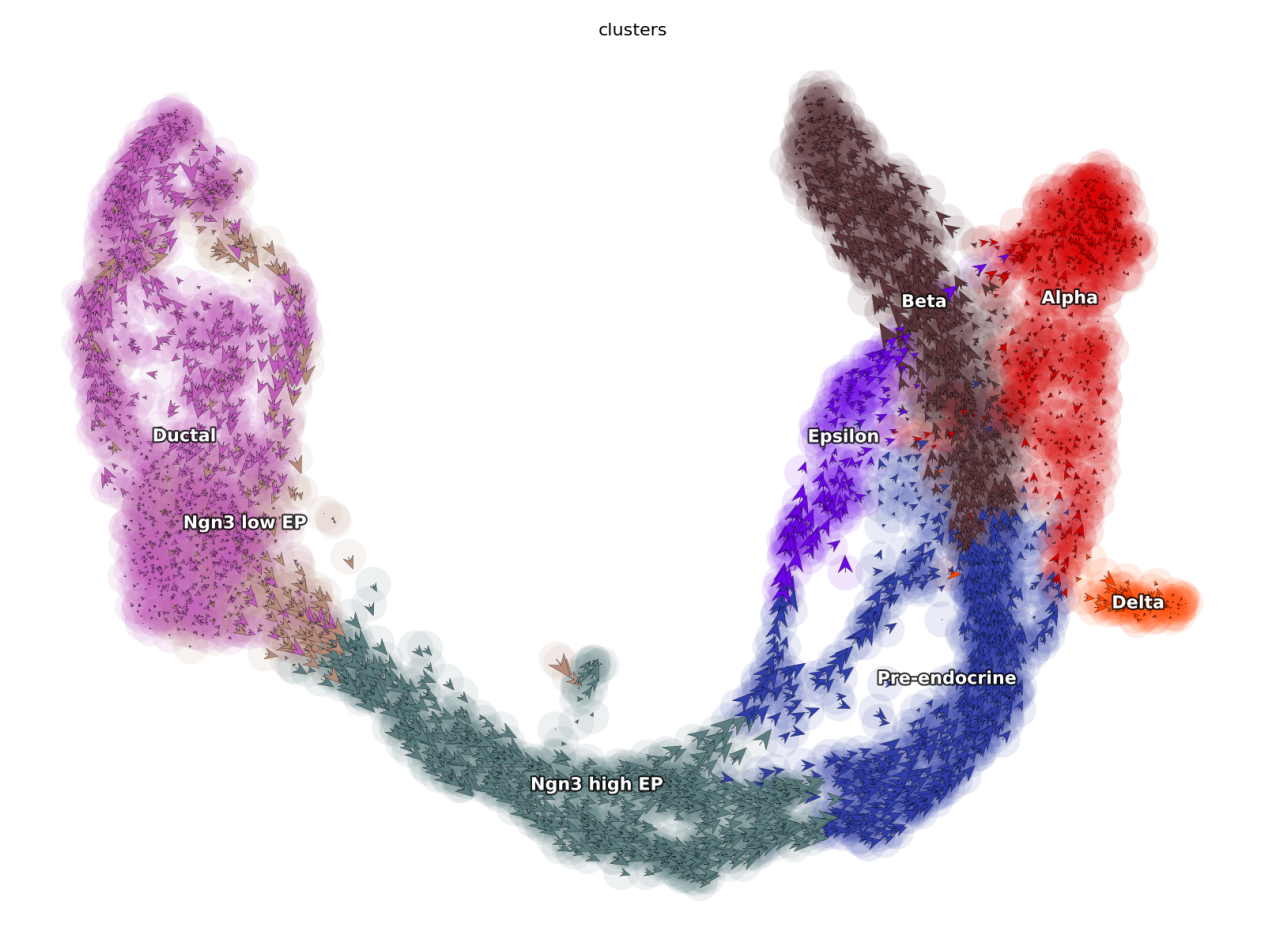
[6]:
```
dyn.pl.streamline_plot(adata, color=['clusters'], basis='umap', show_legend='on data')
```

```
<Figure size 600x400 with 0 Axes>
```



[7]:
```
dyn.pl.cell_wise_vectors(adata, color=['clusters'], basis='umap', show_legend='on data
↪', quiver_length=6, quiver_size=6, figsize=(8, 6), show_arrowed_spines=False)
```

```
<Figure size 800x600 with 0 Axes>
```

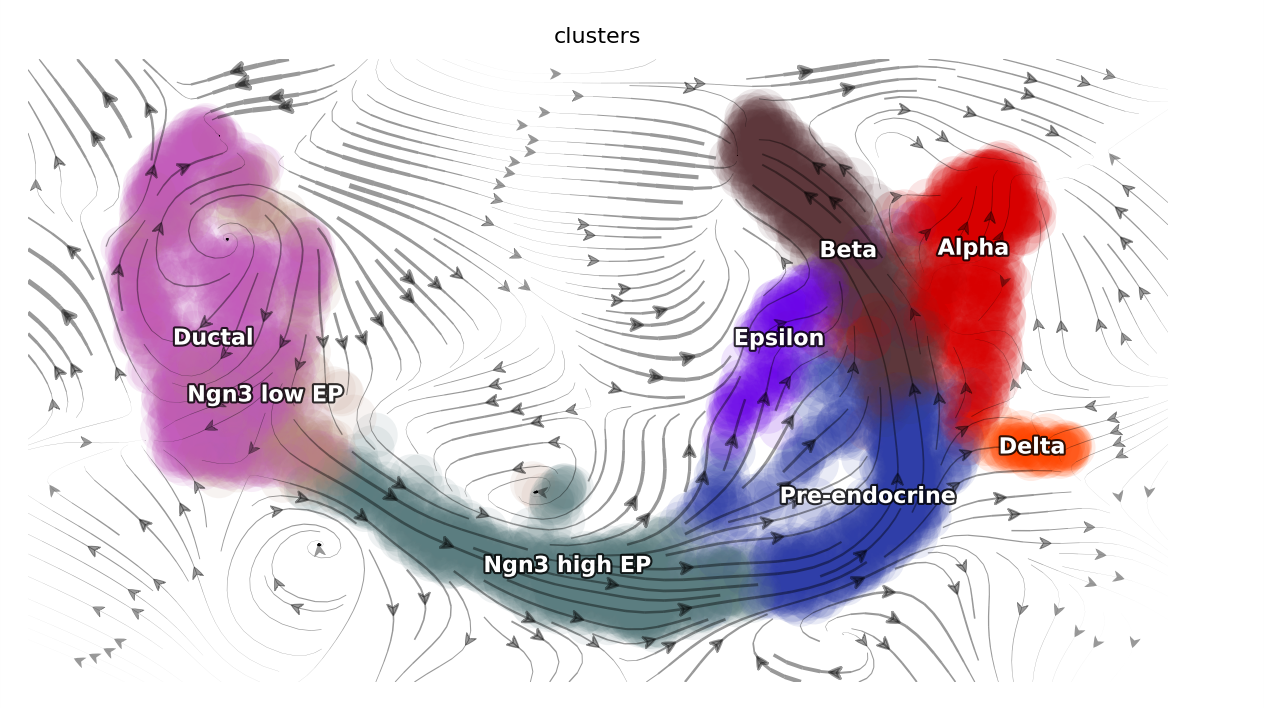


[8]:
```
# ok some exciting vector field analysis

# you can set `verbose = 1/2/3` to obtain different levels of running information of␣
↪vector field reconstruction

dyn.vf.VectorField(adata, basis='umap', pot_curl_div=True) # , M=1000, MaxIter=1000
```

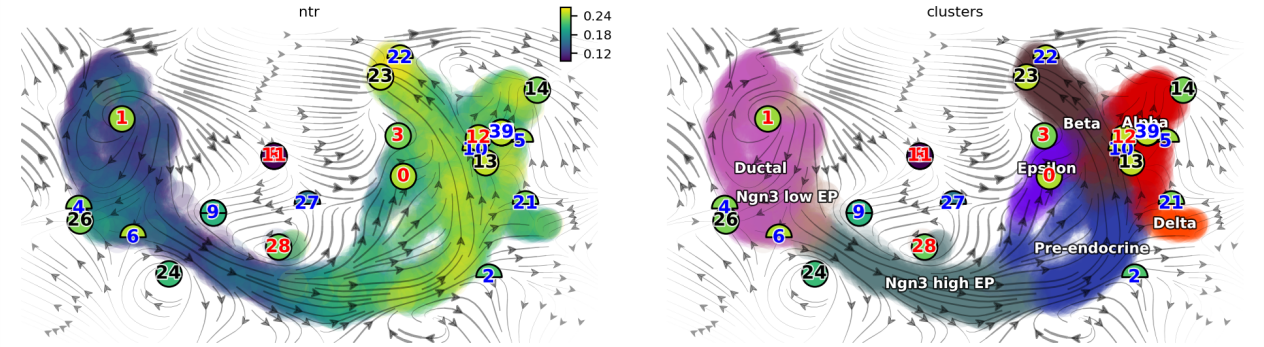
```
Constructing diffusion graph from reconstructed vector field: 3696it [00:58, 63.53it/
↪s]
Calculating 2-D curl: 100%|| 3696/3696 [00:00<00:00, 16788.15it/s]
Calculating divergence: 100%|| 3696/3696 [00:00<00:00, 13627.22it/s]
```

[9]:
```
dyn.pl.topography(adata, color=['clusters'], basis='umap', background='white',
                  streamline_color='black', show_legend='on data', terms=("streamline
↪"))
```

```
<Figure size 600x400 with 0 Axes>
```



[10]:
```
dyn.pl.topography(adata, basis='umap', background='white', color=['ntr', 'clusters'],␣
↪streamline_color='black', show_legend='on data')
```



[11]:
```
dyn.tl.cell_velocities(adata, basis='pca')
dyn.vf.VectorField(adata, basis='pca')
dyn.vf.speed(adata)
dyn.vf.divergence(adata)
dyn.vf.acceleration(adata)
dyn.vf.curl(adata)
```

```
projecting velocity vector to low dimensional embedding...:  10%|          | 378/3696␣
↪[00:00<00:00, 3769.11it/s]
```

```
Using existing pearson_transition_matrix found in .obsp.
```

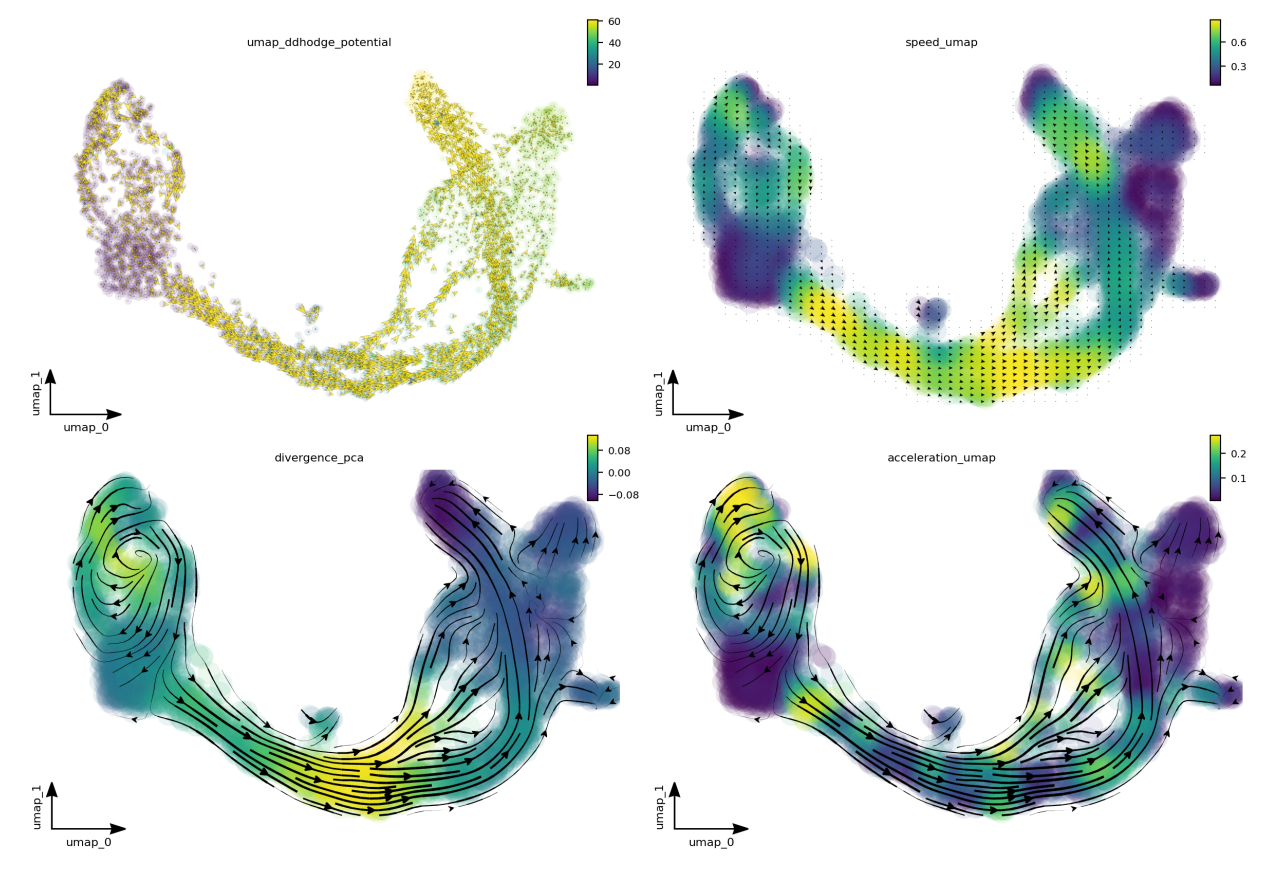
```
projecting velocity vector to low dimensional embedding...: 100%|| 3696/3696 [00:00
↪<00:00, 4015.84it/s]
Calculating divergence: 100%|| 3696/3696 [00:00<00:00, 7500.08it/s]
Calculating acceleration: 100%|| 3696/3696 [00:00<00:00, 292963.20it/s]
Calculating 2-D curl: 100%|| 3696/3696 [00:00<00:00, 17275.38it/s]
```

[12]:
```
adata
```

[12]:
```
AnnData object with n_obs × n_vars = 3696 × 27998
    obs: 'clusters_coarse', 'clusters', 'S_score', 'G2M_score', 'nGenes', 'nCounts',
↪'pMito', 'use_for_pca', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'Size_
↪Factor', 'initial_cell_size', 'unspliced_Size_Factor', 'initial_unspliced_cell_size
↪', 'ntr', 'cell_cycle_phase', 'umap_ddhodge_div', 'umap_ddhodge_potential', 'curl_
↪umap', 'divergence_umap', 'speed_umap', 'divergence_pca', 'acceleration_umap'
    var: 'highly_variable_genes', 'pass_basic_filter', 'log_m', 'score', 'log_cv',
↪'use_for_pca', 'ntr', 'beta', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b
↪', 'gamma_r2', 'gamma_logLL', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0
↪', 'S0', 'total0', 'use_for_dynamics', 'use_for_transition'
    uns: 'clusters_coarse_colors', 'clusters_colors', 'day_colors', 'neighbors', 'pca
↪', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_fit',
↪'feature_selection', 'dynamics', 'grid_velocity_umap', 'VecFld_umap', 'VecFld',
↪'grid_velocity_pca', 'VecFld_pca'
    obsm: 'X_pca', 'X_umap', 'X', 'cell_cycle_scores', 'velocity_umap', 'velocity_
↪umap_SparseVFC', 'X_umap_SparseVFC', 'velocity_pca', 'velocity_pca_SparseVFC', 'X_
↪pca_SparseVFC', 'acceleration_umap'
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced', 'M_u', 'M_uu', 'M_s',
↪'M_us', 'M_ss', 'velocity_S'
    obsp: 'distances', 'connectivities', 'moments_con', 'pearson_transition_matrix',
↪'umap_ddhodge'
```

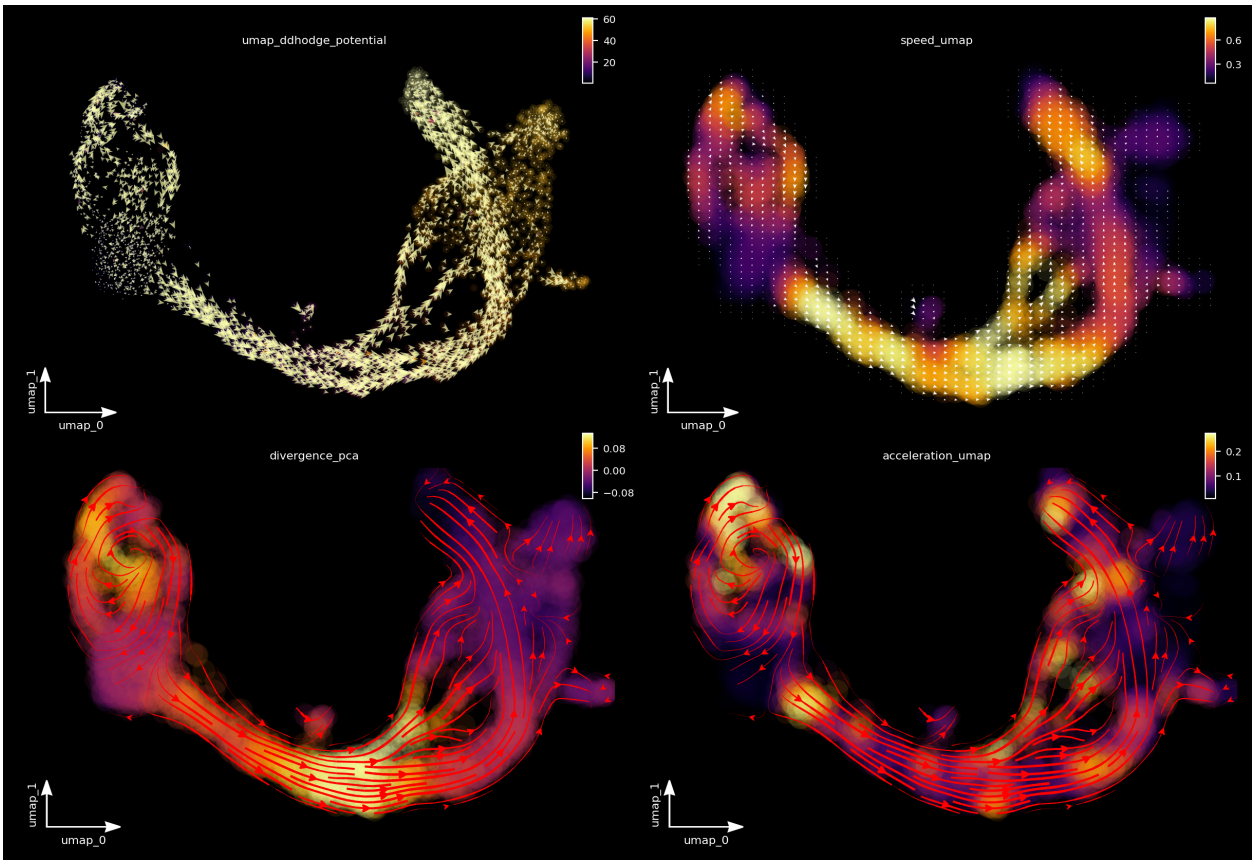
this just shows how flexible dynamo' plotting function can be.

[13]:
```
import matplotlib.pyplot as plt
fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True, figsize=(12,␣
↪8))
f1_axes
f1_axes[0, 0] = dyn.pl.cell_wise_vectors(adata, color='umap_ddhodge_potential',␣
↪pointsize=0.1, alpha = 0.7, ax=f1_axes[0, 0], quiver_length=6, quiver_size=6, save_
↪show_or_return='return')
f1_axes[0, 1] = dyn.pl.grid_vectors(adata, color='speed_umap', ax=f1_axes[0, 1],␣
↪quiver_length=12, quiver_size=12, save_show_or_return='return')
f1_axes[1, 0] = dyn.pl.streamline_plot(adata, color='divergence_pca', ax=f1_axes[1,␣
↪0], save_show_or_return='return')
f1_axes[1, 1] = dyn.pl.streamline_plot(adata, color='acceleration_umap', ax=f1_axes[1,
↪ 1], save_show_or_return='return')
plt.show()
```



[14]:
```
# emulate ggplot2 plotting styple with black background
dyn.configuration.set_figure_params('dynamo', background='black')
```

[15]:
```
fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True, figsize=(12,␣
↪8))
f1_axes
f1_axes[0, 0] = dyn.pl.cell_wise_vectors(adata, color='umap_ddhodge_potential',␣
↪pointsize=0.1, alpha = 0.7, ax=f1_axes[0, 0], quiver_length=6, quiver_size=6, save_
↪show_or_return='return', background='black')
f1_axes[0, 1] = dyn.pl.grid_vectors(adata, color='speed_umap', ax=f1_axes[0, 1],␣
↪quiver_length=12, quiver_size=12, save_show_or_return='return', background='black')
f1_axes[1, 0] = dyn.pl.streamline_plot(adata, color='divergence_pca', ax=f1_axes[1,␣
↪0], save_show_or_return='return', background='black')
f1_axes[1, 1] = dyn.pl.streamline_plot(adata, color='acceleration_umap', ax=f1_axes[1,
↪ 1], save_show_or_return='return', background='black')
plt.show()
```



[16]:
```
progenitor = adata.obs_names[adata.obs.clusters.isin(['Ductal'])]
len(progenitor)
```

[16]:
```
916
```

[17]:
```
import numpy as np
dyn.pd.fate(adata, basis='umap', init_cells=np.random.choice(progenitor, 100),␣
↪interpolation_num=100,  direction='forward',
   inverse_transform=False, average=False, cores=3)
```

[17]:
```
AnnData object with n_obs × n_vars = 3696 × 27998
    obs: 'clusters_coarse', 'clusters', 'S_score', 'G2M_score', 'nGenes', 'nCounts',
↪'pMito', 'use_for_pca', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'Size_
↪Factor', 'initial_cell_size', 'unspliced_Size_Factor', 'initial_unspliced_cell_size
↪', 'ntr', 'cell_cycle_phase', 'umap_ddhodge_div', 'umap_ddhodge_potential', 'curl_
↪umap', 'divergence_umap', 'speed_umap', 'divergence_pca', 'acceleration_umap'
    var: 'highly_variable_genes', 'pass_basic_filter', 'log_m', 'score', 'log_cv',
↪'use_for_pca', 'ntr', 'beta', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b
↪', 'gamma_r2', 'gamma_logLL', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0
↪', 'S0', 'total0', 'use_for_dynamics', 'use_for_transition'
    uns: 'clusters_coarse_colors', 'clusters_colors', 'day_colors', 'neighbors', 'pca
↪', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_variance_ratio_', 'pca_fit',
↪'feature_selection', 'dynamics', 'grid_velocity_umap', 'VecFld_umap', 'VecFld',
↪'grid_velocity_pca', 'VecFld_pca', 'fate_umap'
    obsm: 'X_pca', 'X_umap', 'X', 'cell_cycle_scores', 'velocity_umap', 'velocity_
↪umap_SparseVFC', 'X_umap_SparseVFC', 'velocity_pca', 'velocity_pca_SparseVFC', 'X_
↪pca_SparseVFC', 'acceleration_umap'
```

(continues on next page)

(continued from previous page)

```
    layers: 'spliced', 'unspliced', 'X_spliced', 'X_unspliced', 'M_u', 'M_uu', 'M_s',
↪'M_us', 'M_ss', 'velocity_S'
    obsp: 'distances', 'connectivities', 'moments_con', 'pearson_transition_matrix',
↪'umap_ddhodge'
```

[18]:
```
%%capture
fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='clusters', ax=ax, save_show_or_return='return')
```

[19]:
```
%%capture
instance = dyn.mv.StreamFuncAnim(adata=adata, ax=ax, color='clusters')
```

[20]:
```
import matplotlib
matplotlib.rcParams['animation.embed_limit'] = 2**128 # Ensure all frames will be␣
↪embedded.

from matplotlib import animation
import numpy as np

anim = animation.FuncAnimation(instance.fig, instance.update, init_func=instance.init_
↪background,
                               frames=np.arange(100), interval=100, blit=True)
from IPython.core.display import display, HTML
HTML(anim.to_jshtml()) # embedding to jupyter notebook.
```

[20]:
```
<IPython.core.display.HTML object>
```

[21]:
```
%%capture
fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='clusters', ax=ax, save_show_or_return='return')
dyn.mv.animate_fates(adata, color='clusters', basis='umap', n_steps=200, fig=fig,␣
↪ax=ax,
                     save_show_or_return='save', logspace=True, max_time=None, save_
↪kwargs={"filename": 'pancreas.gif'})
```

2.9 Dentate gyrus dataset

This tutorial uses raw data from scvelo package. Special thanks go to the scvelo team!

[ ]:
```
# get the latest version from pypi
# for other installations approaches, see https://dynamo-release.readthedocs.io/en/
↪latest/ten_minutes_to_dynamo.html#how-to-install
!pip install dynamo-release --upgrade --quiet
```

[1]:
```
# from IPython.core.display import display, HTML
# display(HTML("<style>.container { width:90% !important; }</style>"))
# %matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

(continues on next page)

(continued from previous page)

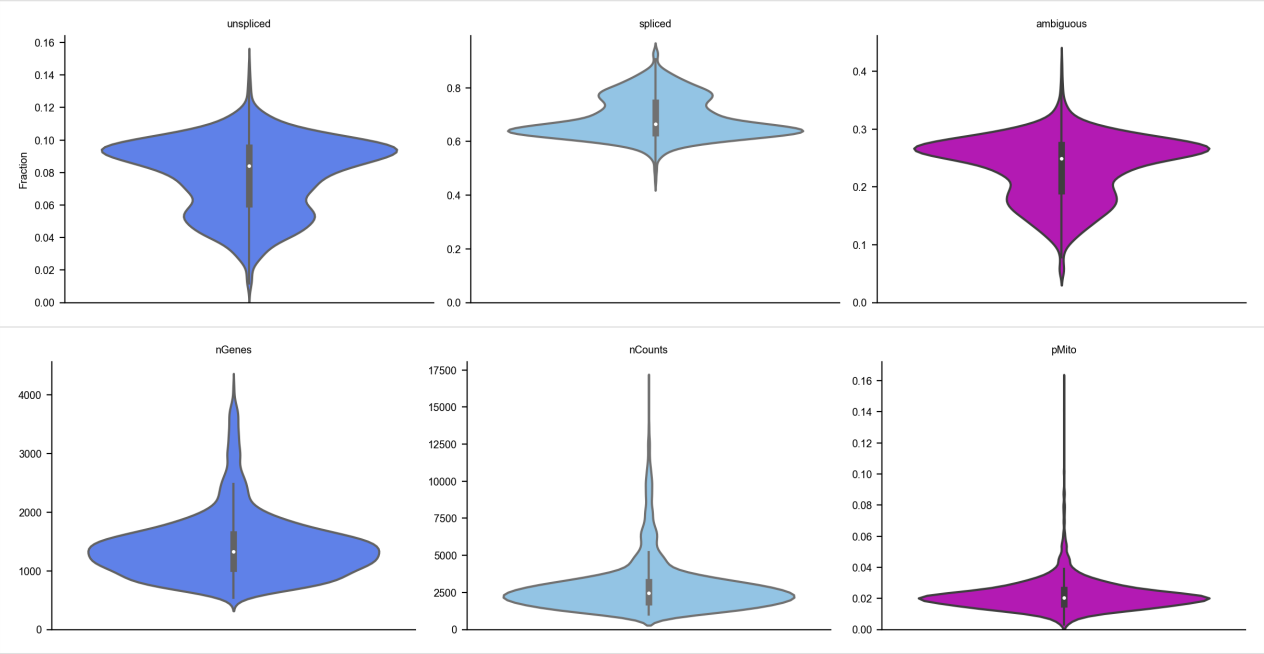
```
import dynamo as dyn

dyn.get_all_dependencies_version()
```

```
package dynamo-release umap-learn anndata cvxopt hdbscan loompy matplotlib  \
version         0.95.2      0.4.6   0.7.4  1.2.3  0.8.26  3.0.6      3.3.0

package   numba   numpy pandas pynndescent python-igraph scikit-learn  scipy  \
version  0.51.0  1.19.1  1.1.1       0.4.8         0.8.2       0.23.2  1.5.2

package seaborn setuptools statsmodels    tqdm  trimap numdifftools colorcet
version   0.9.0      49.6.0      0.11.1  4.48.2  1.0.12       0.9.39    2.0.2
```

[2]:
```
# emulate ggplot2 plotting styple with white background
dyn.configuration.set_figure_params('dynamo', background='white')
```

[3]:
```
adata = dyn.sample_data.DentateGyrus_scvelo()

adata.obsm['X_umap_ori'] = adata.obsm['X_umap'].copy()
```

[4]:
```
dyn.pl.show_fraction(adata)
dyn.pl.basic_stats(adata)
```



[5]:
```
dyn.pp.recipe_monocle(adata, n_top_genes=2000, fg_kwargs={'shared_count': 30})
```

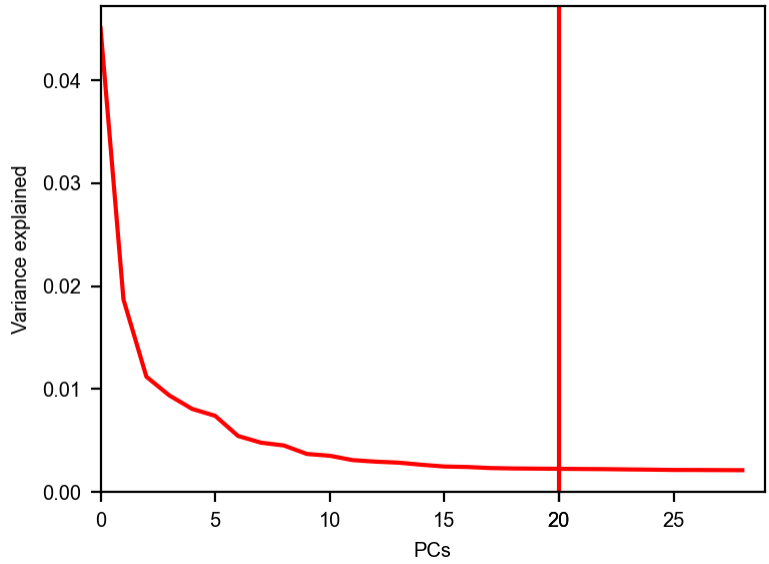
[5]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'ntr',
↪'cell_cycle_phase'
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr'
```

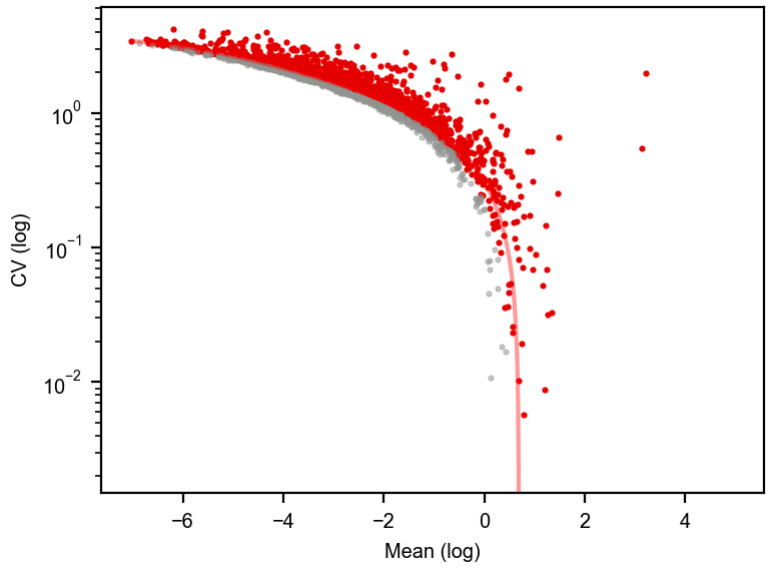
(continues on next page)

(continued from previous page)

```
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced'
```

[6]:
```
dyn.pl.variance_explained(adata)
dyn.pl.feature_genes(adata)
```





[7]:
```
dyn.tl.dynamics(adata, model='stochastic', cores=3)
# or dyn.tl.dynamics(adata, model='deterministic')
# or dyn.tl.dynamics(adata, model='stochastic', est_method='negbin')
```

[7]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'nTr',
↪'cell_cycle_phase'
```

(continues on next page)

(continued from previous page)

```
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics'
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection', 'dynamics'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced', 'M_u',
↪'M_uu', 'M_s', 'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con'
```

[8]:
```
# enforce recalculating the umap embedding. By default dynamo will avoid␣
↪recalculation if a reduced dimension space exists.
dyn.tl.reduceDimension(adata, enforce=True)
```

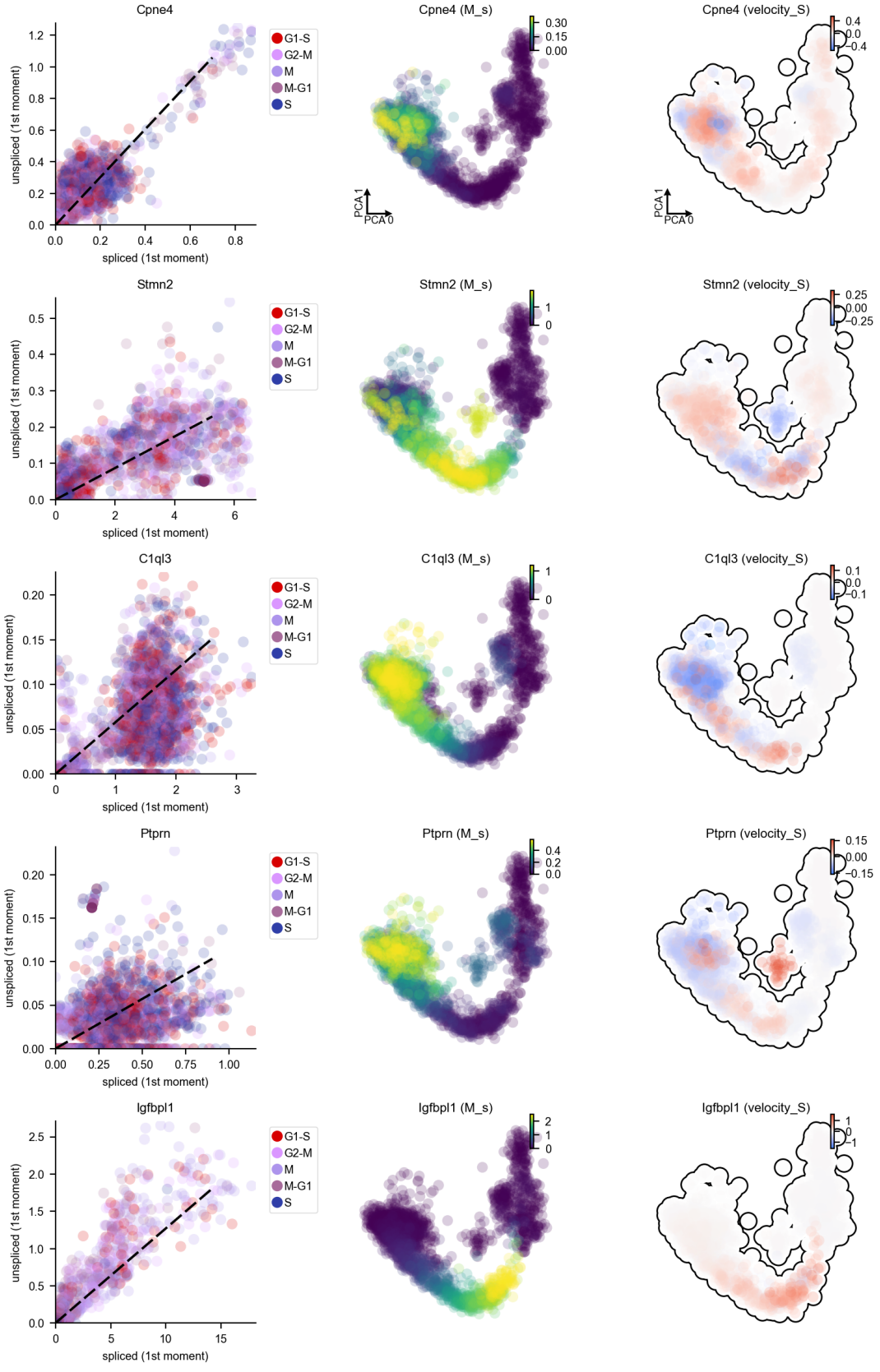
[8]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'ntr',
↪'cell_cycle_phase'
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics'
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit
↪'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced', 'M_u',
↪'M_uu', 'M_s', 'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con', 'connectivities', 'distances'
```

[9]:
```
dyn.tl.cell_velocities(adata)
```

```
calculating transition matrix via pearson kernel with sqrt transform.: 100%|| 2930/
↪2930 [00:05<00:00, 498.05it/s]
projecting velocity vector to low dimensional embedding...: 100%|| 2930/2930 [00:00
↪<00:00, 3557.37it/s]
```

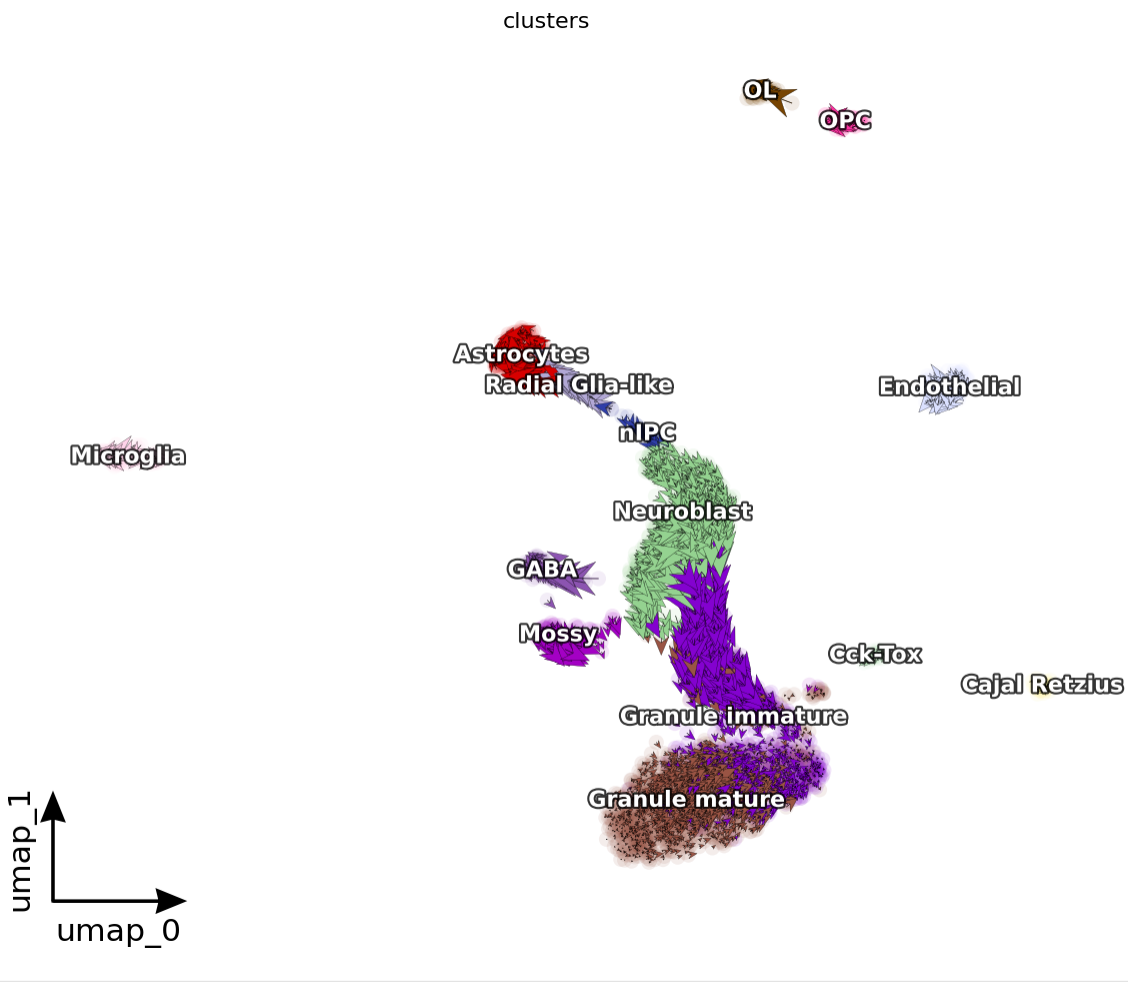
[9]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'ntr',
↪'cell_cycle_phase'
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics', 'use_for_transition'
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit
↪', 'grid_velocity_umap'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores', 'velocity_umap'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced', 'M_u',
↪'M_uu', 'M_s', 'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con', 'connectivities', 'distances', 'pearson_transition_matrix'
```

[10]:
```
DentateGyrus_genes = ["Tnc", "Gfap", "Tac2", "Pdgfra", "Igfbpl1", 'Ptprn', "Sema3c",
↪"Neurod6", "Stmn2", "Sema5a", "C1ql3", "Cpne4", "Cck"]

dyn.pl.phase_portraits(adata, genes=DentateGyrus_genes, ncols=3, figsize=(3, 3),␣
↪basis='pca', show_quiver=False)
```


Cpne4
Cpne4 (M_s)
Cpne4 (velocity_S)
Stmn2
Stmn2 (M_s)
Stmn2 (velocity_S)
C1ql3
C1ql3 (M_s)
C1ql3 (velocity_S)
Ptprn
Ptprn (M_s)
Ptprn (velocity_S)
Igfbpl1
Igfbpl1 (M_s)
Igfbpl1 (velocity_S)
unspliced (1st moment)
spliced (1st moment)
G1-S
G2-M
M
M-G1
S
PCA 1
PCA 0

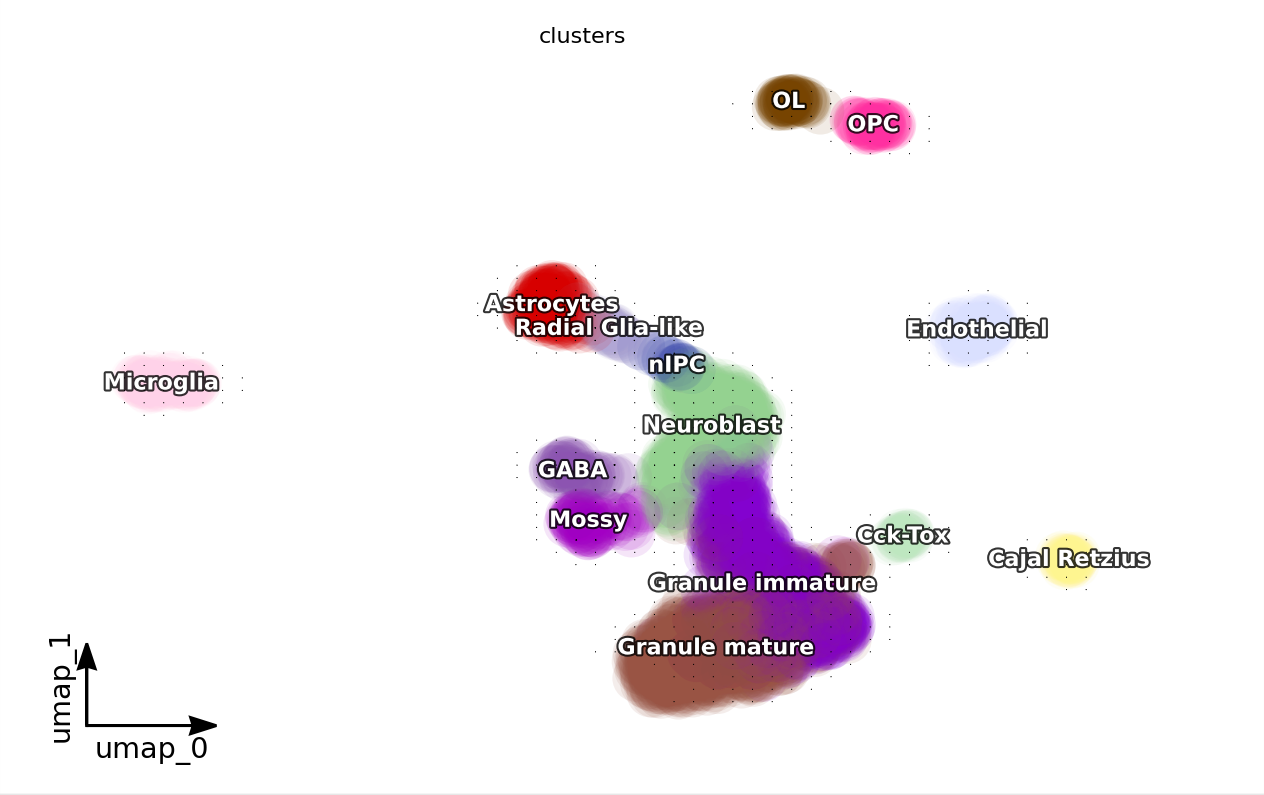
[11]:
```
dyn.pl.cell_wise_vectors(adata, color=['clusters'], quiver_size=6, quiver_length=6,␣
↪figsize=(6, 5), pointsize=0.1) # ['GRIA3', 'LINC00982', 'AFF2']
```

```
<Figure size 600x500 with 0 Axes>
```



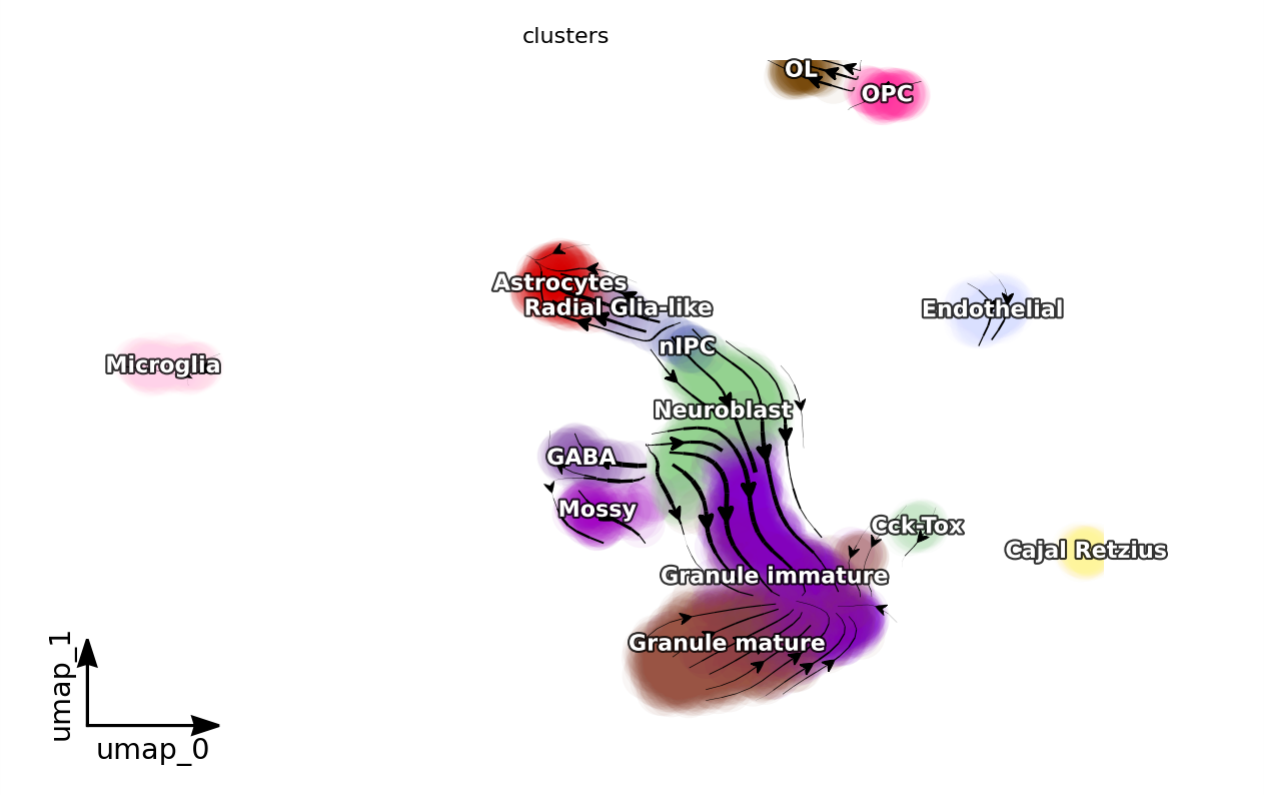
[12]:
```
dyn.pl.grid_vectors(adata, color=['clusters'], method='gaussian')
```

```
<Figure size 600x400 with 0 Axes>
```



[13]:
```
dyn.pl.streamline_plot(adata, color=['clusters'], basis='umap', density=1, background=
↪'white',s_kwargs_dict={"alpha": 0.05})
```

```
<Figure size 600x400 with 0 Axes>
```



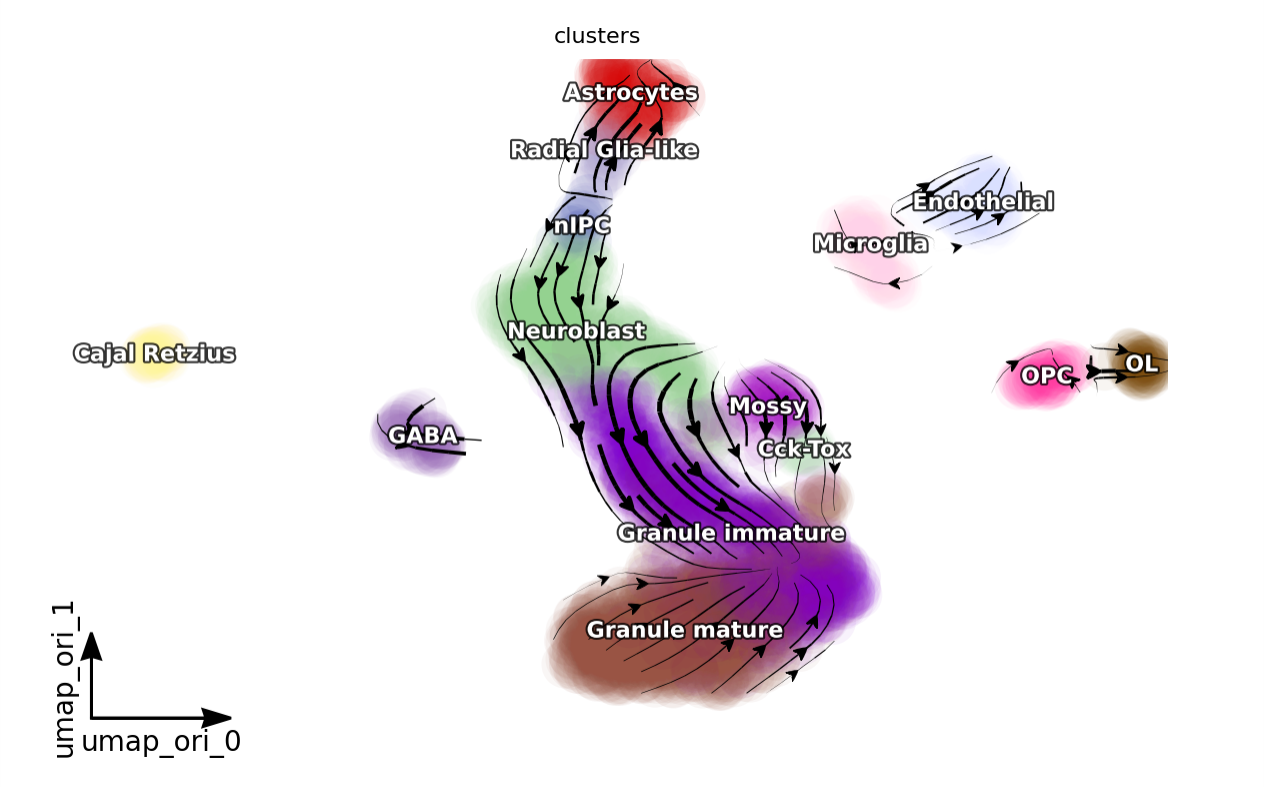
[14]:
```
dyn.pl.streamline_plot(adata, color=['clusters'], basis='umap_ori', density=1,␣
↪background='white',s_kwargs_dict={"alpha": 0.05})
```

```
projecting velocity vector to low dimensional embedding...:  27%|       | 784/2930␣
↪[00:00<00:00, 3801.14it/s]
```

```
Using existing pearson_transition_matrix found in .obsp.
```

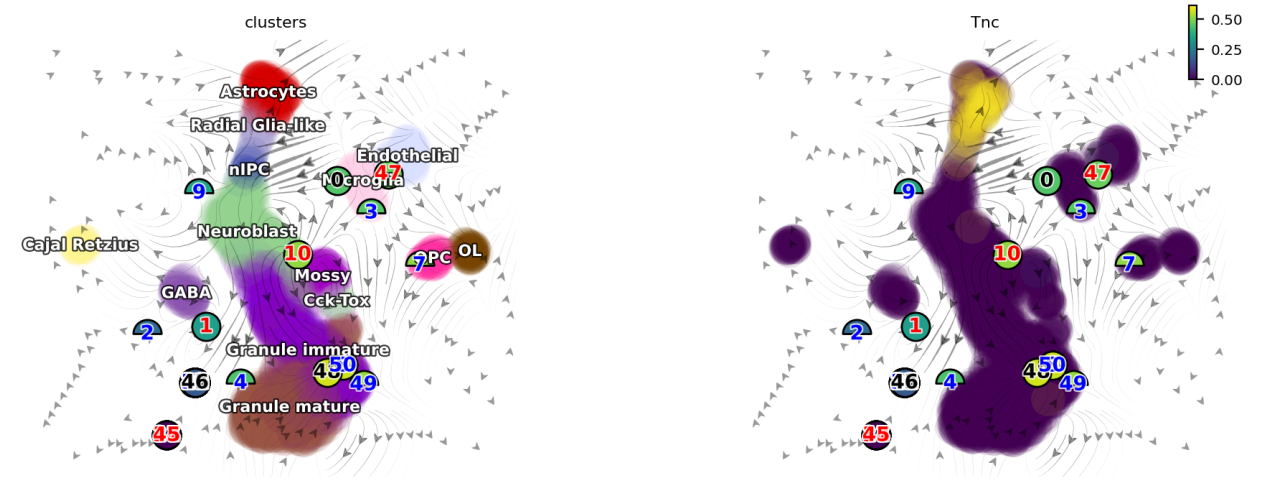
```
projecting velocity vector to low dimensional embedding...: 100%|| 2930/2930 [00:00
↪<00:00, 4048.10it/s]
```

```
<Figure size 600x400 with 0 Axes>
```



[15]:
```
# you can set `verbose = 1/2/3` to obtain different levels of running information of␣
↪vector field reconstruction
dyn.vf.VectorField(adata, basis='umap_ori', dims=[0, 1])
```

[16]:
```
dyn.pl.topography(adata, color=['clusters', 'Tnc'], basis='umap_ori', ncols=2)
```



[17]:
```
dyn.vf.VectorField(adata, basis='umap_ori', dims=[0, 1], pot_curl_div=True)
```

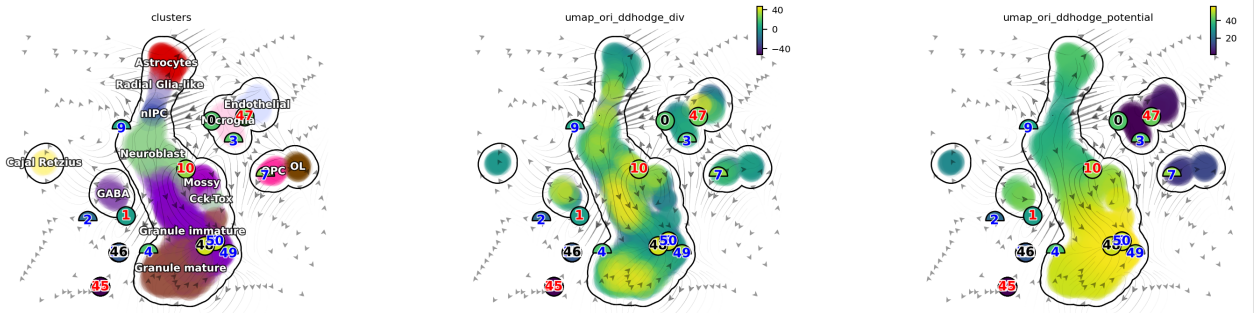
[18]:
```
dyn.ext.ddhodge(adata, basis='umap_ori')
```

```
Constructing diffusion graph from reconstructed vector field: 2930it [01:57, 24.83it/
↪s]
```

[19]:
```
adata
```

[19]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'ntr',
↪'cell_cycle_phase', 'umap_ori_ddhodge_div', 'umap_ori_ddhodge_potential'
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics', 'use_for_transition'
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit
↪', 'grid_velocity_umap', 'grid_velocity_umap_ori', 'VecFld_umap_ori', 'VecFld'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores', 'velocity_umap',
↪'velocity_umap_ori', 'velocity_umap_ori_SparseVFC', 'X_umap_ori_SparseVFC'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced', 'M_u',
↪'M_uu', 'M_s', 'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con', 'connectivities', 'distances', 'pearson_transition_matrix',
↪'umap_ori_ddhodge'
```

[20]:
```
dyn.pl.topography(adata, color=['clusters', 'umap_ori_ddhodge_div', 'umap_ori_ddhodge_
↪potential'],
                  basis='umap_ori', ncols=3, frontier=True)
```



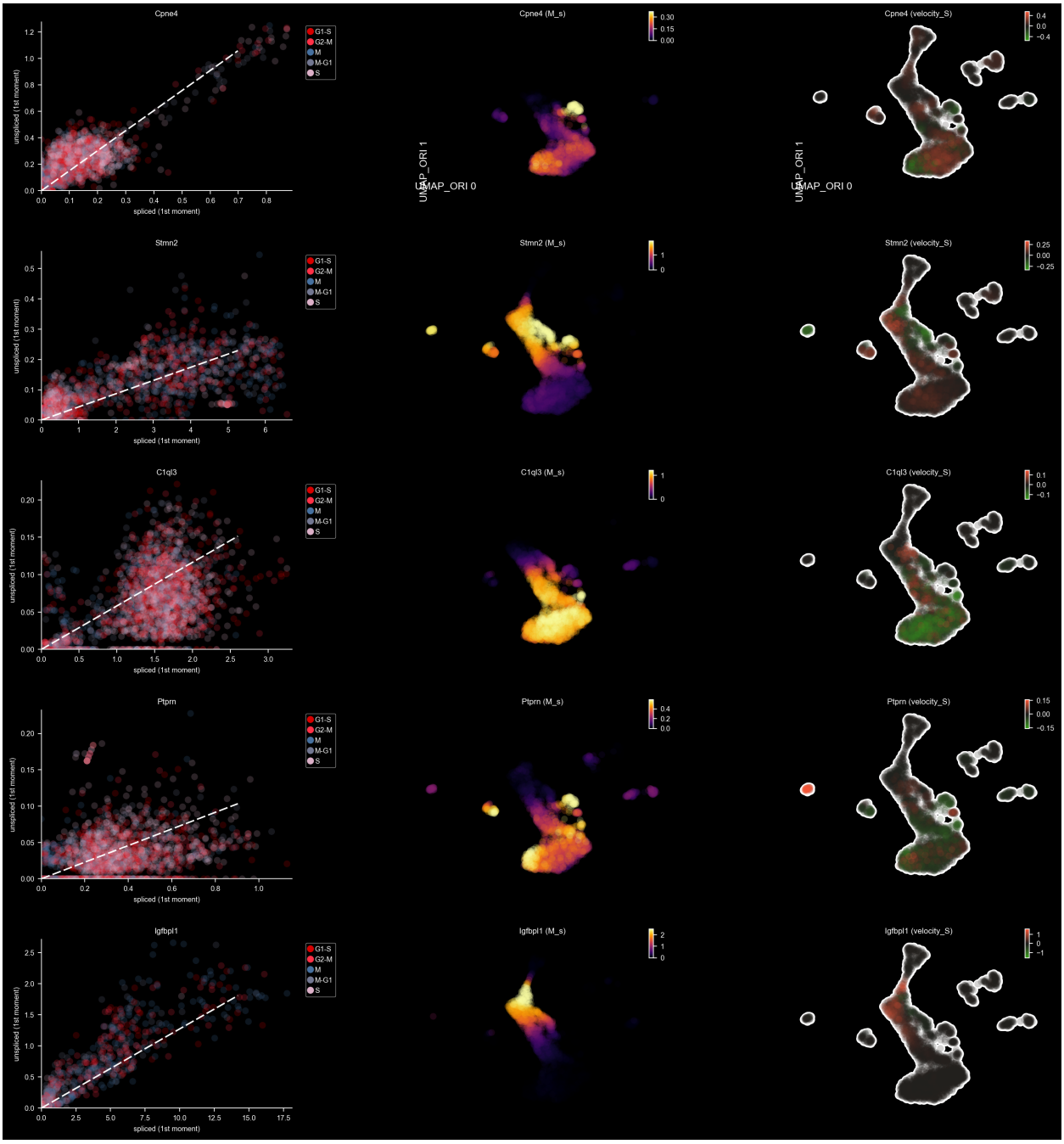
[21]:
```
# emulate ggplot2 plotting styple with black background
dyn.configuration.set_figure_params('dynamo', background='black')
```

[22]:
```
dyn.pl.phase_portraits(adata, genes=DentateGyrus_genes, ncols=3, figsize=(6, 4),␣
↪basis='umap_ori', show_quiver=False)

dyn.pl.cell_wise_vectors(adata, color=['clusters'], basis='umap_ori', pointsize=0.1,␣
↪quiver_size=4, quiver_length=4, background='black') # ['GRIA3', 'LINC00982', 'AFF2']

dyn.pl.grid_vectors(adata, color=['clusters'], basis='umap_ori', method='gaussian',␣
↪background='black')

dyn.pl.streamline_plot(adata, color=['clusters'], basis='umap_ori', density=2,␣
↪background='black')
```

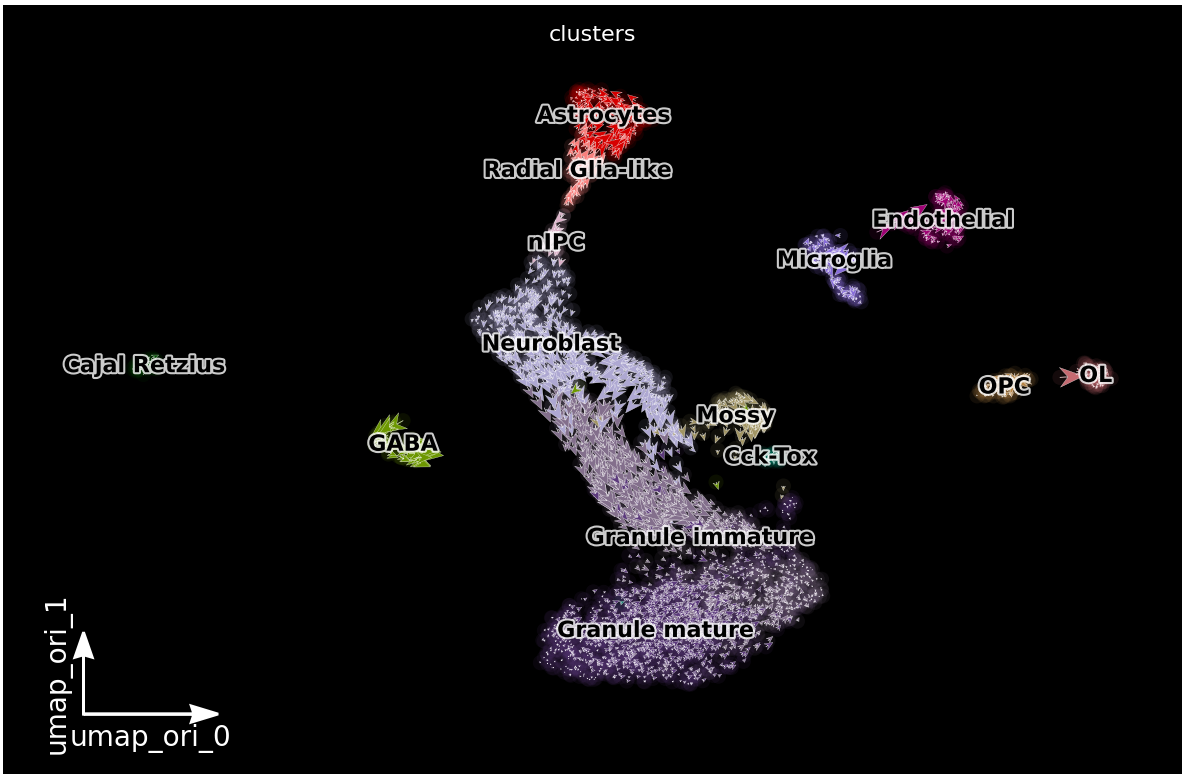
(continues on next page)

(continued from previous page)

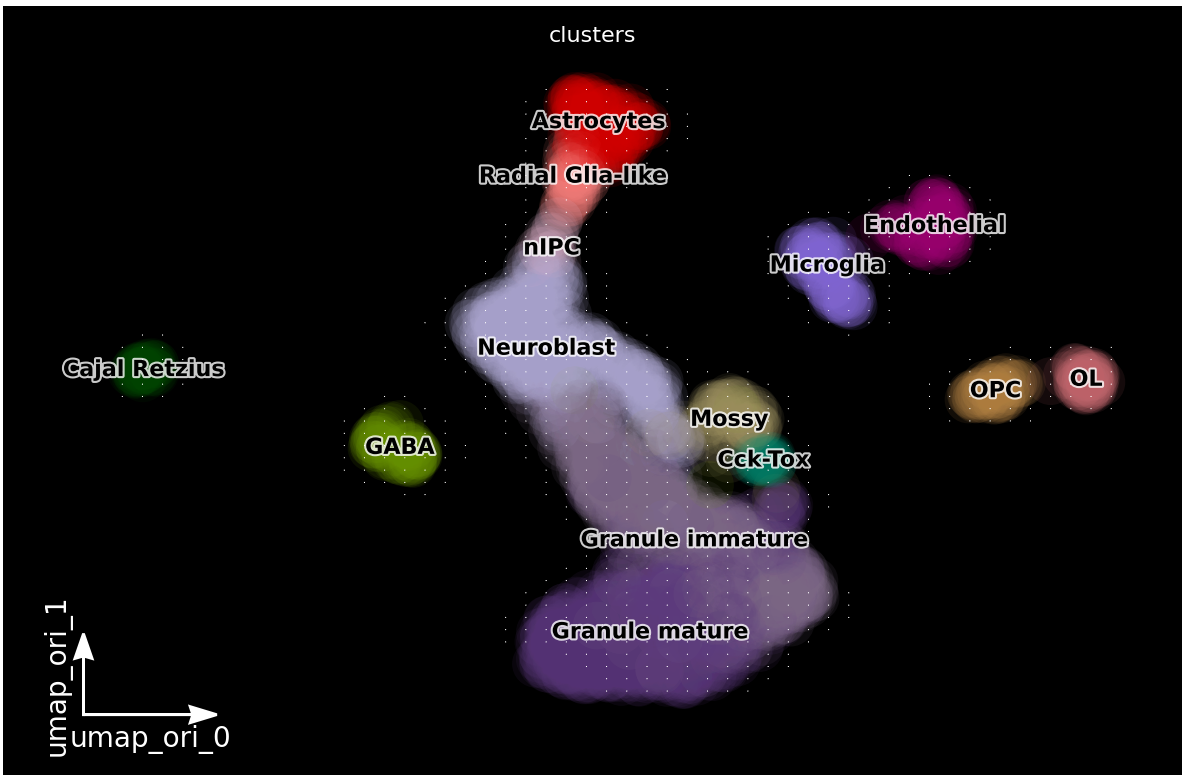
```
dyn.pl.topography(adata, color=['clusters', 'Tnc'], basis='umap_ori', ncols=2,␣
↪background='black')
```



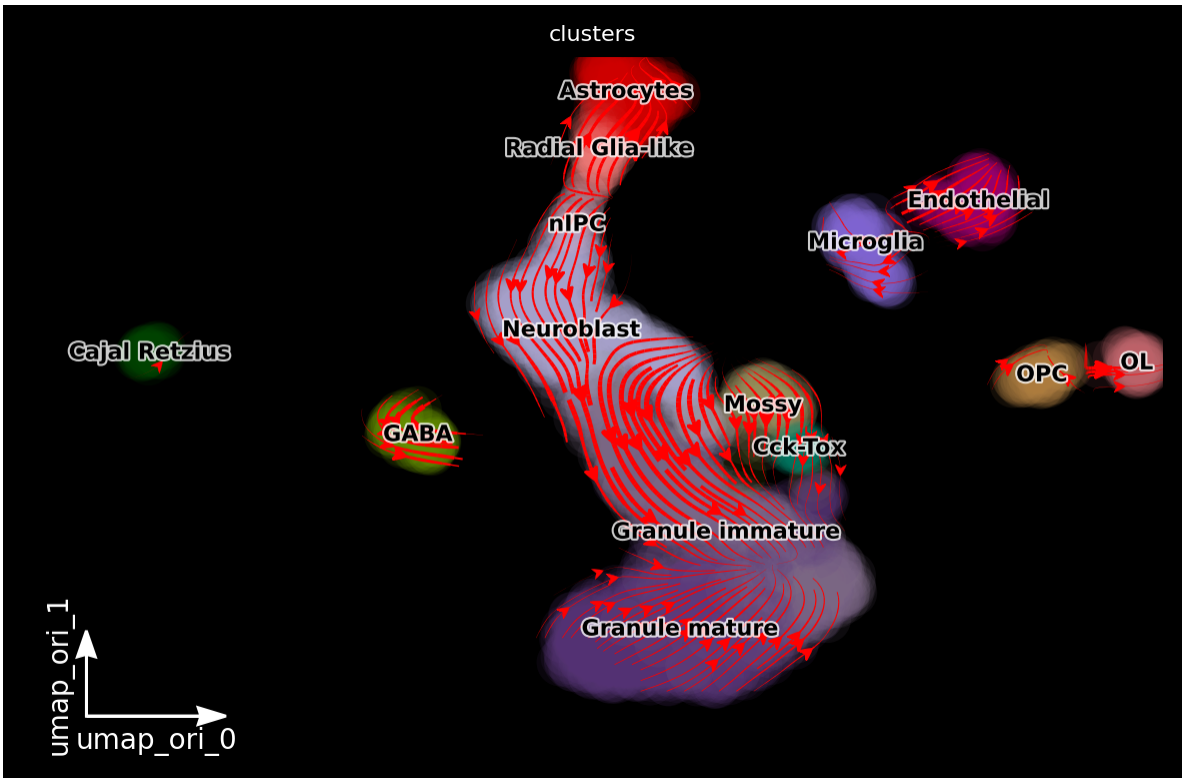

```
<Figure size 600x400 with 0 Axes>
```

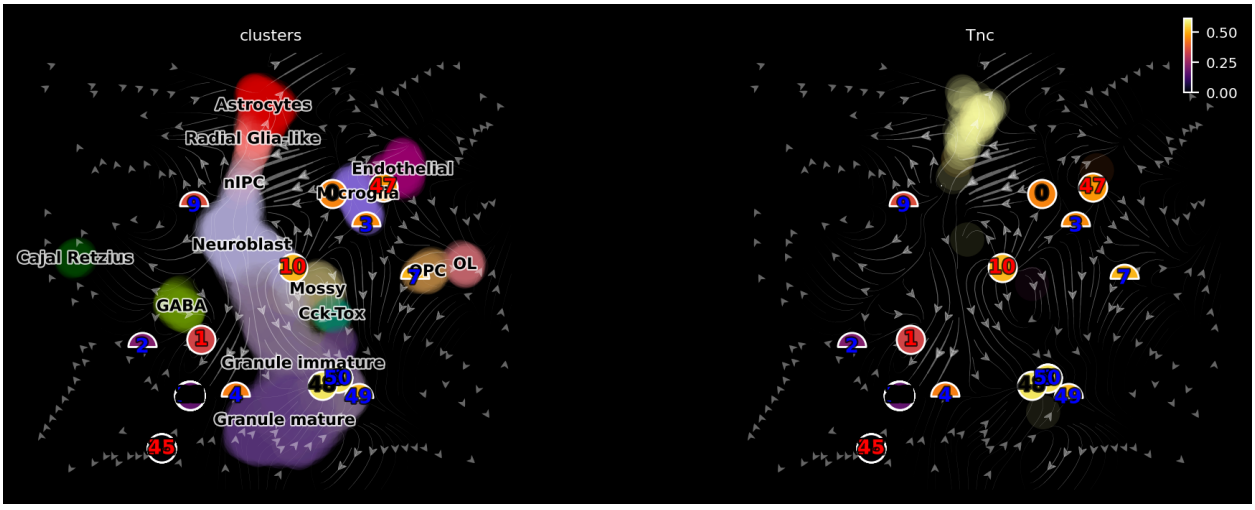



```
<Figure size 600x400 with 0 Axes>
```




```
<Figure size 600x400 with 0 Axes>
```





[23]:
```
progenitor = adata.obs_names[adata.obs.clusters.isin(['nIPC', 'Neuroblast'])]
len(progenitor)
```

[23]:
```
436
```

[24]:
```
dyn.pd.fate(adata, basis='umap_ori', init_cells=progenitor, interpolation_num=100, ␣
↪direction='forward',
   inverse_transform=False, average=False, cores=3)
```

[24]:
```
AnnData object with n_obs × n_vars = 2930 × 13913
    obs: 'clusters', 'age(days)', 'clusters_enlarged', 'nGenes', 'nCounts', 'pMito',
↪'use_for_pca', 'unspliced_Size_Factor', 'initial_unspliced_cell_size', 'Size_Factor
↪', 'initial_cell_size', 'spliced_Size_Factor', 'initial_spliced_cell_size', 'ntr',
↪'cell_cycle_phase', 'umap_ori_ddhodge_div', 'umap_ori_ddhodge_potential'
```

(continues on next page)

(continued from previous page)

```
    var: 'pass_basic_filter', 'log_m', 'log_cv', 'score', 'use_for_pca', 'ntr', 'beta
↪', 'gamma', 'half_life', 'alpha_b', 'alpha_r2', 'gamma_b', 'gamma_r2', 'gamma_logLL
↪', 'delta_b', 'delta_r2', 'uu0', 'ul0', 'su0', 'sl0', 'U0', 'S0', 'total0', 'use_
↪for_dynamics', 'use_for_transition'
    uns: 'clusters_colors', 'velocyto_SVR', 'pp_norm_method', 'PCs', 'explained_
↪variance_ratio_', 'pca_fit', 'feature_selection', 'dynamics', 'neighbors', 'umap_fit
↪', 'grid_velocity_umap', 'grid_velocity_umap_ori', 'VecFld_umap_ori', 'VecFld',
↪'fate_umap_ori'
    obsm: 'X_umap', 'X_umap_ori', 'X_pca', 'X', 'cell_cycle_scores', 'velocity_umap',
↪'velocity_umap_ori', 'velocity_umap_ori_SparseVFC', 'X_umap_ori_SparseVFC'
    layers: 'ambiguous', 'spliced', 'unspliced', 'X_unspliced', 'X_spliced', 'M_u',
↪'M_uu', 'M_s', 'M_us', 'M_ss', 'velocity_S'
    obsp: 'moments_con', 'connectivities', 'distances', 'pearson_transition_matrix',
↪'umap_ori_ddhodge'
```

[25]:
```
%%capture
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='clusters', ax=ax, save_show_or_return='return',␣
↪basis='umap_ori')
```

[26]:
```
%%capture
instance = dyn.mv.StreamFuncAnim(adata=adata, ax=ax, basis='umap_ori', color='clusters
↪')
```

[27]:
```
from matplotlib import animation
import numpy as np

anim = animation.FuncAnimation(instance.fig, instance.update, init_func=instance.init_
↪background,
                               frames=np.arange(100), interval=100, blit=True)
from IPython.core.display import display, HTML
HTML(anim.to_jshtml()) # embedding to jupyter notebook.
```

[27]:
```
<IPython.core.display.HTML object>
```

[28]:
```
%%capture
fig, ax = plt.subplots()
ax = dyn.pl.topography(adata, color='clusters', basis='umap_ori', ax=ax, save_show_or_
↪return='return')
dyn.mv.animate_fates(adata, color='clusters', basis='umap_ori', n_steps=100, fig=fig,␣
↪ax=ax,
                     save_show_or_return='save', logspace=True, max_time=None, save_
↪kwargs={"filename": 'dentategyrus.gif'})
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Qiu19] Qi Qiu, Peng Hu, *et al.* (2019), *Massively parallel, time-resolved single-cell RNA sequencing with scNT-Seq*, Biorxiv.

[Qi19] Xiaojie Qiu *et al.* (2019), *Mapping vector field of single cells*, Biorxiv.

[Qiu18] Xiaojie Qiu *et al.* (2019), *Inferring Causal Gene Regulatory Networks from Coupled Single-Cell Expression Dynamics Using Scribes*, Cell systems.

[Qiu17] Xiaojie Qiu *et al.* (2017), *Reversed graph embedding resolves complex single-cell trajectories*, Nature methods.

[Trapnell14] Cole Trapnell *et al.* (2014), *The dynamics and regulators of cell fate decisions are revealed by pseudotemporal ordering of single cells*, Nature Biotechnology.

[Bergen19] Volker Bergen *et al.* (2020), *Generalizing RNA velocity to transient cell states through dynamical modeling*, Nature biotechnology.

[Melsted19] Páll Melsted *et al.* (2019), *Modular and efficient pre-processing of single-cell RNA-seq*, Biorxiv.

[Gorin19] Gennady Gorin *et al.* (2019), *RNA velocity and protein acceleration from single-cell multiomics experiments*, Genome biology.

[Manno18] La Manno *et al.* (2018), *RNA velocity of single cells*, Nature.

[Wolf18] Wolf *et al.* (2018), *Scanpy: large-scale single-cell gene expression data analysis*, Genome Biology.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

E

F

G

M

R

S

T

V