

---

# LineageOT

*Release 0.2.0*

**Aden Forrow**

**Jan 12, 2022**



**CONTENTS:**

<b>1</b>	<b>Modules</b>	<b>3</b>
1.1	Core pipeline . . . . .	3
1.2	Simulations . . . . .	5
1.3	Inference . . . . .	8
1.4	Evaluation . . . . .	13
<b>2</b>	<b>LineageOT examples</b>	<b>17</b>
2.1	Minimal pipeline example . . . . .	17
2.2	LineageOT with static lineage tracing . . . . .	18
2.3	LineageOT on a convergent trajectory . . . . .	20
2.4	LineageOT on a curled trajectory . . . . .	32
<b>3</b>	<b>Core pipeline</b>	<b>45</b>
	<b>Python Module Index</b>	<b>49</b>
	<b>Index</b>	<b>51</b>



LineageOT is a package for analyzing lineage-traced single-cell sequencing time series. It extends [Waddington-OT](#) to compute temporal couplings using measurements of both gene expression and lineage trees. The LineageOT couplings can be used directly by the downstream analysis tools of the Waddington-OT package, which we do not duplicate here. For full details, see our [paper](#).

All of the functionality required for running LineageOT is in the `core` module. The remaining modules have implementation functions and code for reproducing analyses in the paper.

The source code, with installation instructions and examples, is available at <https://github.com/aforrr/LineageOT>.



## MODULES

## 1.1 Core pipeline

This module contains only the core functions required for applying LineageOT to new data. The fitted couplings produced can be used directly by the downstream analysis tools of the Waddington-OT package. See <https://broadinstitute.github.io/wot/> for more details.

```
lineageot.core.fit_lineage_coupling(adata, time_1, time_2, lineage_tree_t2, time_key='time',
                                   state_key=None, epsilon=0.05, normalize_cost=True,
                                   ot_method='sinkhorn', marginal_1=[], marginal_2=[],
                                   balance_reg=inf)
```

Fits a LineageOT coupling between the cells in adata at time\_1 and time\_2. In the process, annotates the lineage tree with observed and estimated cell states.

### Parameters

- **adata** (*AnnData*) – Annotated data matrix
- **time\_1** (*Number*) – The earlier time point in adata. All times are relative to the root of the tree.
- **time\_2** (*Number*) – The later time point in adata. All times are relative to the root of the tree.
- **lineage\_tree\_t2** (*Networkx DiGraph*) – The lineage tree fitted to cells at time\_2. Nodes should already be annotated with times. Annotations related to cell state will be added.
- **time\_key** (*str (default 'time')*) – Key in adata.obs and lineage\_tree\_t2 containing cells' time labels
- **state\_key** (*str (default None)*) – Key in adata.obsm containing cell states. If None, uses adata.X.
- **epsilon** (*float (default 0.05)*) – Entropic regularization parameter for optimal transport
- **normalize\_cost** (*bool (default True)*) – Whether to rescale the cost matrix by its median before fitting a coupling. Normalizing this way allows us to choose a reasonable default epsilon for data of any scale
- **ot\_method** (*str (default 'sinkhorn')*) – Method used for the optimal transport solver. Choose from 'sinkhorn', 'greenkhorn', 'sinkhorn\_stabilized' and 'sinkhorn\_epsilon\_scaling' for balanced transport and 'sinkhorn', 'sinkhorn\_stabilized', and 'sinkhorn\_reg\_scaling' for unbalanced transport. 'sinkhorn' is recommended unless you encounter numerical problems. See PythonOT docs for more details.

- **marginal\_1** (*Vector (default [])*) – Marginal distribution (relative growth rates) for cells at time 1. If empty, assumed uniform.
- **marginal\_2** (*Vector (default [])*) – Marginal distribution (relative growth rates) for cells at time 2. If empty, assumed uniform.
- **balance\_reg** (*Number*) – Regularization parameter for unbalanced transport. Smaller values allow more flexibility in growth rates. If infinite, marginals are treated as hard constraints.

**Returns coupling** – AnnData containing the lineage coupling. Cells from time\_1 are in coupling.obs, cells from time\_2 are in coupling.var, and the coupling matrix is coupling.X

**Return type** AnnData

```
lineageot.core.fit_tree(adata, time, barcodes_key='barcodes', clones_key='X_clone',
                        clone_times=None, method='neighbor join')
```

Fits a lineage tree to lineage barcodes of all cells in adata. To compute the lineage tree for a specific time point, filter adata before calling fit\_tree. The fitted tree is annotated with node times but not states.

#### Parameters

- **adata** (*AnnData*) – Annotated data matrix with lineage-traced cells
- **time** (*Number*) – Time of sampling of the cells of adata relative to most recent common ancestor (for dynamic lineage tracing) or labeling time (for static lineage tracing).
- **barcodes\_key** (*str, default 'barcodes'*) – Key in adata.obsm containing cell barcodes. Ignored if using clonal data. If using barcode data, each row of adata.obsm[barcodes\_key] should be a barcode where each entry corresponds to a possibly-mutated site. A positive number indicates an observed mutation, zero indicates no mutation, and -1 indicates the site was not observed.
- **clones\_key** (*str, default 'X\_clone'*) – Key in adata.obsm containing clonal data. Ignored if using barcodes directly. If using clonal data, adata.obsm[clones\_key] should be a num\_cells x num\_clones boolean matrix. Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (*Vector of length num\_clones, default None*) – Ignored unless method is 'clones'. Each entry contains the time of labeling of the corresponding column of adata.obsm[clones\_key].
- **method** (*str*) – Inference method used to fit tree. Current options are 'neighbor join' (for barcodes from dynamic lineage tracing), 'non-nested clones' (for non-nested clones from static lineage tracing), or 'clones' (for possibly-nested clones from static lineage tracing).

**Returns tree** – A fitted lineage tree. Each node is annotated with 'time\_to\_parent' and 'time' (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with 'time' equal to the child node's 'time\_to\_parent'. Observed node indices correspond to their row in adata.

**Return type** Networkx DiGraph

```
lineageot.core.read_newick(filename, leaf_labels, leaf_time=None)
```

Loads a tree saved in Newick format and adds annotations required for LineageOT.

#### Parameters

- **filename** (*str*) – The name of the file to load from.
- **leaf\_labels** (*list*) – The label of each leaf in the Newick tree, sorted to align with the gene expression AnnData object filtered to cells at the corresponding time.



- **leaf\_time** (*float (default None)*) – The time of sampling of the leaves. If unspecified, the root of the tree is assigned time 0.

**Returns tree** – The saved tree, in LineageOT’s format. Each node is annotated with ‘time\_to\_parent’ and ‘time’ (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with ‘time’ equal to the child node’s ‘time\_to\_parent’. Observed node indices correspond to their index in leaf\_labels, which should match their row in the gene expression AnnData object filtered to cells at the corresponding time.

**Return type** Networkx DiGraph

`lineageot.core.save_coupling_as_tmap(coupling, time_1, time_2, tmap_out)`

Saves a LineageOT coupling for downstream analysis with Waddington-OT. A sequence of saved couplings can be loaded in wot with `wot.tmap.TransportMapModel.from_directory(tmap_out)`

#### Parameters

- **coupling** (*AnnData*) – The coupling to save.
- **time\_1** (*Number*) – The earlier time point in adata. All times are relative to the root of the tree.
- **time\_2** (*Number*) – The later time point in adata. All times are relative to the root of the tree.
- **tmap\_out** (*str*) – The path and prefix to the save file name.

## 1.2 Simulations

This module contains functions for generating the simulated data used in the LineageOT paper. Most of this is not required for applying LineageOT to experimental data, and none of it needs to be used directly.

**class** `lineageot.simulation.Cell(x, barcode, seed=None)`

Bases: `object`

Wrapper for (rna expression, barcode) arrays

**deepcopy** ()

**reset\_seed** ()

### Storing the parameters for simulated data

Returns the center of the distribution  $p(x_0|\text{barcode})$

Single bifurcation followed by convergence of the two clusters

Converts a list of cells to two ndarrays, one for expression and one for barcodes

Returns the new barcode after mutations have occurred for some time

Returns a new cell after both barcode and  $x$  have evolved for some time

Returns a sample from Langevin dynamics following `potential_gradient`

`lineageot.simulation.flatten_list_of_lists` (*tree\_data*)  
 Converts a dataset of cells with their ancestral tree structure to a list of cells (with ancestor and time information dropped)

`lineageot.simulation.mask_barcode` (*barcode, p*)  
 Replaces a subset of the entries of barcode with -1 to simulate missing data  
 Entries are masked independently with probability p  
 Also works for an array of barcodes

`lineageot.simulation.mismatched_clusters_flow` (*x, params*)  
 Single bifurcation followed by bifurcation of each cluster

`lineageot.simulation.mutate_barcode` (*barcode, params*)  
 Randomly changes one entry of the barcode

`lineageot.simulation.partial_convergent_flow` (*x, params*)  
 Single bifurcation followed by bifurcation of each cluster, where two of the new clusters subsequently merge

`lineageot.simulation.reproducible_poisson` (*rate*)  
 Samples a single Poisson random variable, in a way that is reproducible, i.e. after  
`np.random.seed(s) a = divisible_poisson(r1) np.random.seed(s) b = divisible_poisson(r2)`  
 with  $r1 > r2$ ,  $b \sim \text{binomial}(n = a, p = r2/r1)$   
 This is the standard numpy Poisson sampling algorithm for rate  $\leq 10$ .  
 Note that this is relatively slow, running in  $O(\text{rate})$  time.

`lineageot.simulation.sample_barcode` (*params*)  
 Samples an initial barcode

`lineageot.simulation.sample_cell` (*params*)  
 Samples an initial cell

`lineageot.simulation.sample_descendants` (*initial\_cell, time, params, target\_num\_cells=None*)  
 Samples the descendants of an initial cell

`lineageot.simulation.sample_division_time` (*params*)  
 Samples the time until a cell divides

`lineageot.simulation.sample_pop` (*num\_initial\_cells, time, params*)  
 Samples a population after some intervening time  
*num\_initial\_cells*: Number of cells in the population at time 0 *time*: Time when population is measured *params*: Simulation parameters

`lineageot.simulation.sample_population_descendants` (*pop, time, params*)  
 Samples the descendants of each cell in a population *pop*: list of (expression, barcode) tuples

`lineageot.simulation.sample_x0` (*barcode, params*)  
 Samples the initial position in gene expression space

`lineageot.simulation.single_bifurcation_flow` (*x*)

`lineageot.simulation.split_targets_between_daughters` (*time\_remaining, target\_num\_cells, params*)  
 Given a target number of cells to sample, divides the samples between daughters assuming both have the expected number of descendants at the sampling time

`lineageot.simulation.subsample_list` (*sample, target\_num\_cells*)  
 Randomly samples *target\_num\_cells* from the sample

If there are fewer than `target_num_cells` in the sample, returns the whole sample

`lineageot.simulation.subsample_pop(sample, target_num_cells, params, num_cells=None)`

Randomly samples `target_num_cells` from the sample. Subsampling during the simulation by setting `params.target_num_cells` is a more efficient approximation of this.

If there are fewer than `target_num_cells` in the sample, returns the whole sample

sample should be either:

- a list of cells, if `params.keep_tree` is False
- nested lists of lists of cells encoding the tree structure, if `params.keep_tree` is True

(i.e., it should match the output of `sample_descendants` with the same `params`)

`lineageot.simulation.vector_field(x, params)`

Selects a vector field and returns its value at `x`

## 1.3 Inference

This module contains the implementation of LineageOT used by the core functions.

**class** `lineageot.inference.NeighborJoinNode(subtree, subtree_root, has_global_root)`

Bases: object

`lineageot.inference.OT_cost(coupling, cost)`

`lineageot.inference.add_conditional_means_and_variances(tree, observed_nodes)`

Adds the mean and variance of the posterior on 'x' for each of the unobserved nodes, conditional on the observed values of 'x' in `observed_nodes`, assuming that differences along edges are Gaussian with variance equal to the length of the edge.

In doing so, also adds inverse time annotations to edges.

If no nodes in tree are observed, inverse time annotations are added but conditional means and variances are not (as there is nothing to condition on).

`lineageot.inference.add_division_times_from_vertex_times(tree, current_node='root')`

Adds 'time\_to\_parent' variables to nodes, based on 'time' annotations

`lineageot.inference.add_inverse_times_to_edges(tree)`

Labels each edge of the tree with 'inverse time' equal to `1/edge['time']`

`lineageot.inference.add_leaf_barcodes(tree, barcode_array)`

Adds barcodes from `barcode_array` to the corresponding leaves of the tree

`lineageot.inference.add_leaf_times(tree, final_time)`

Adds the known final time to all leaves and 0 as the root time

`lineageot.inference.add_leaf_x(tree, x_array)`

Adds expression vectors from `x_array` to the corresponding leaves of the tree

`lineageot.inference.add_node_times_from_dict(tree, current_node, time_dict)`

Adds times from `time_dict` to `current_node` and its descendants

`lineageot.inference.add_node_times_from_division_times(tree, current_node='root', overwrite=False)`

Adds 'time' variable to all descendants of `current_node` based on the 'time\_to\_parent' variable

`lineageot.inference.add_nodes_at_time` (*tree*, *time\_to\_add*, *current\_node='root'*,  
*num\_nodes\_added=0*)

Splits every edge (u,v) where  $u[\text{'time'}] < \text{time\_to\_add} < v[\text{'time'}]$

into (u, w) and (w, v) with  $w[\text{'time'}] = \text{time\_to\_add}$

Newly added nodes {w} are labeled as tuples (time\_to\_add, i)

The input tree should be annotated with node times already

`lineageot.inference.add_samples_to_clone_tree` (*clone\_matrix*, *clone\_times*,  
*clone\_reference\_tree*, *sampling\_time*)

Adds a leaf for each row in *clone\_matrix* to *clone\_reference\_tree*. The parent is set as the clone that the cell is a member of with the latest labeling time.

*clone\_reference\_tree* is edited in place rather than returned.

#### Parameters

- **clone\_matrix** (*Boolean array with shape [num\_cells, num\_clones]*) – Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (*Vector of length num\_clones*) – Each entry has the time of labeling of the corresponding clone.
- **clone\_reference\_tree** – The tree of lineage relationships among clones.
- **sampling\_time** (*Number*) – The time of sampling of the cells. Should be greater than all clone labeling times.

`lineageot.inference.add_times` (*tree*, *mutation\_rates*, *time\_inference\_method*, *overwrite=False*)

Adds estimated division times/edge lengths to a tree

The tree should already have all node barcodes estimated

`lineageot.inference.add_times_to_edges` (*tree*)

Labels each edge of tree with 'time' taken from 'time\_to\_parent' of its endpoint

`lineageot.inference.annotate_tree` (*tree*, *mutation\_rates*, *time\_inference\_method='independent'*,  
*overwrite\_times=False*)

Adds barcodes and times to internal (ancestor) nodes so likelihoods can be computed

Barcodes are inferred by putting minimizing the number of mutation events required, assuming a model with no back mutations and a known initial barcode

`lineageot.inference.barcode_distances` (*barcode\_array*)

Computes all pairwise lineage distances between barcodes

`lineageot.inference.compute_leaf_times` (*tree*, *num\_leaves*)

Computes the list of times of the leaves by adding 'time\_to\_parent' along the path to 'root'

`lineageot.inference.compute_new_distances` (*distance\_matrix*, *nodes\_to\_join*)

`lineageot.inference.compute_q_matrix` (*distance\_matrix*)

Computes the Q-matrix for neighbor joining

`lineageot.inference.compute_tree_distances` (*tree*)

Computes the matrix of pairwise distances between leaves of the tree

`lineageot.inference.convert_newick_to_networkx` (*newick\_tree*, *leaf\_labels*,  
*leaf\_time=None*, *root\_label='root'*,  
*unlabeled\_nodes\_added=0*,  
*at\_global\_root=True*)

Converts a tree from the Newick package's format to LineageOT's annotated NetworkX DiGraph. Ignores

existing annotations, except edge lengths.

#### Parameters

- **newick\_tree** (*newick.Node* or [*newick.Node*]) – A tree loaded by the Newick package.
- **leaf\_labels** (*list*) – The label of each leaf in the Newick tree, sorted to align with the gene expression AnnData object filtered to cells at the corresponding time
- **leaf\_time** (*float* (default *None*)) – The time of sampling of the leaves. If unspecified, the root of the tree is assigned time 0.
- **root\_label** (*str* (default *'root'*)) – The label of the root node of the tree
- **unlabeled\_nodes\_added** (*int* (default *0*)) – The number of previously-unlabeled nodes that have already been added to the tree. Leave as 0 for any top-level use of the function.
- **at\_global\_root** (*bool* (default *True*)) – Whether the function is being called to convert a full tree or a subtree.

**Returns tree** – The saved tree, in LineageOT’s format. Each node is annotated with ‘time\_to\_parent’ and ‘time’ (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with ‘time’ equal to the child node’s ‘time\_to\_parent’. Observed node indices correspond to their index in leaf\_labels.

**Return type** Networkx DiGraph

`lineageot.inference.cvxopt_qp_from_numpy(P, q, G, h)`

Converts arguments to cvxopt matrices and runs cvxopt’s quadratic programming solver

`lineageot.inference.distances_to_joined_node(distance_matrix, nodes_to_join)`

`lineageot.inference.estimate_division_time(child, parent, mutation_rates)`

Estimates the lifetime of child, i.e. the time between when parent divided to make child and when child divided

Input arguments are nodes in a lineage tree, i.e. dicts

`lineageot.inference.extract_ancestor_data_arrays(late_tree, time, params)`

Returns arrays of the RNA expression and barcodes for ancestors of leaves of the tree

Each row of each array is a leaf node

`lineageot.inference.extract_data_arrays(tree)`

Returns arrays of the RNA expression and barcodes from leaves of the tree

Each row of each array is a cell

`lineageot.inference.find_parent_clone(clone, clone_matrix, clone_times)`

Returns the parent of a subclone, assuming this is uniquely defined as the clone from an earlier time point whose barcode was observed in a cell from the subclone.

#### Parameters

- **clone** (*int*) – Index of clone whose parent will be returned
- **clone\_matrix** (*Boolean array with shape [num\_cells, num\_clones]*) – Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (*Vector of length num\_clones*) – Each entry has the time of labeling of the corresponding clone.

**Returns parent** – Index of parent clone.

**Return type** int

`lineageot.inference.get_ancestor_data(tree, time, leaf=None)`

`lineageot.inference.get_components(graph, edge_length_key='time')`

Returns subgraph views corresponding to connected components of the graph if edges of infinite length are removed

**Parameters**

- **graph** (*NetworkX graph*) –
- **edge\_length\_key** (*default 'time'*) –

**Returns subgraphs**

**Return type** List of NetworkX subgraph views

`lineageot.inference.get_internal_nodes(tree)`

Returns a list of the non-leaf nodes of a tree

`lineageot.inference.get_leaf_descendants(tree, node)`

Returns a list of the leaf nodes of the tree that are descendants of node

`lineageot.inference.get_leaves(tree, include_root=True)`

Returns a list of the leaf nodes of a tree including the root

`lineageot.inference.get_lineage_distances_across_time(early_tree, late_tree)`

Returns the matrix of lineage distances between leaves of early\_tree and leaves in late\_tree. Assumes that early\_tree is a truncated version of late\_tree

`lineageot.inference.get_parent_clone_of_leaf(leaf, clone_matrix, clone_times)`

Returns the index of the clone that the leaf is a member of with the latest labeling time.

`lineageot.inference.get_true_coupling(early_tree, late_tree)`

Returns the coupling between leaves of early\_tree and their descendants in late\_tree. Assumes that early\_tree is a truncated version of late\_tree

The marginal over the early cells is uniform; if cells have different numbers of descendants, the marginal over late cells will not be uniform.

`lineageot.inference.join_nodes(node1, node2, new_root, distances)`

`lineageot.inference.list_tree_to_digraph(list_tree)`

Converts a tree stored as nested lists to a networkx DiGraph

Internal nodes are indexed by negative integers, leaves by nonnegative integers

`lineageot.inference.make_clone_reference_tree(clone_matrix, clone_times, root_time=-inf)`

Makes a tree with nodes for each clone.

**Parameters**

- **clone\_matrix** (*Boolean array with shape [num\_cells, num\_clones]*) – Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (*Vector of length num\_clones*) – Each entry has the time of labeling of the corresponding clone.
- **root\_time** (*Number, default -np.inf*) – The time of the most recent common ancestor of all clones. If -np.inf, clone subtrees are effectively treated independently.

**Returns** `clone_reference_tree` – A tree of clones (not sampled cells), annotated with edge and node times

**Return type** NetworkX DiGraph

`lineageot.inference.make_tree_from_clones(clone_matrix, time, clone_times, root_time=-inf)`

Adds a leaf for each row in `clone_matrix` to `clone_reference_tree`. The parent is set as the clone that the cell is a member of with the latest labeling time.

`clone_reference_tree` is edited in place rather than returned.

#### Parameters

- **clone\_matrix** (Boolean array with shape `[num_cells, num_clones]`) – Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (Vector of length `num_clones`) – Each entry has the time of labeling of the corresponding clone.
- **time** (Number) – The time of sampling of cells.
- **root\_time** (Number, default `-np.inf`) – The time of the most recent common ancestor of all clones. If `-np.inf`, clones are effectively treated as unrelated

**Returns** `fitted_tree` – A tree annotated with edge and node times

**Return type** NetworkX DiGraph

`lineageot.inference.make_tree_from_nonnested_clones(clone_matrix, time, root_time_factor=1000)`

Creates a forest of stars from clonally-labeled data. The centers of the stars are connected to a root far in the past.

#### Parameters

- **clone\_matrix** (Boolean array with shape `[num_cells, num_clones]`) – Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise. Each cell should belong to exactly one clone.
- **time** (Number) – The time of sampling of cells relative to initial clonal labelling.
- **root\_time\_factor** (Number, default `1000`) – Relative time to root of tree (i.e., most recent common ancestor of all cells). The time of the root is set to `-root_time_factor*time`. The default is large so minimal information is shared across clones.

**Returns** `fitted_tree` – A tree annotated with edge and node times

**Return type** NetworkX DiGraph

`lineageot.inference.neighbor_join(distance_matrix)`

Creates a tree by neighbor joining with the input distance matrix

Final row/column of `distance_matrix` assumed to correspond to the root (unmutated) barcode

`lineageot.inference.pick_joined_nodes(Q)`

In default neighbor joining, returns the indices of the pair of nodes with the lowest Q value

TODO: extend to allow stochastic neighbor joining

`lineageot.inference.rate_estimator(barcode_array, time)`

Estimates the mutation rate based on the number of unmutated barcodes remaining.



`lineageot.inference.recursive_add_barcodes` (*tree*, *current\_node*)  
 Fills in the barcodes for internal nodes for a tree whose leaves have barcodes  
 Minimizes the number of mutation events that occur, assuming no backmutations and a known initial barcode

`lineageot.inference.recursive_list_tree_to_digraph` (*list\_tree*, *next\_internal\_node*,  
*next\_leaf\_node*)  
 Recursive helper function for `list_tree_to_digraph`  
 Returns (*current\_tree*, *next\_internal\_node\_label*, *root\_of\_current\_tree*)

`lineageot.inference.remove_node_and_descendants` (*tree*, *node*)  
 Removes a node and all its descendants from the tree

`lineageot.inference.remove_times` (*tree*)  
 Removes time annotations from nodes and edges of a tree

`lineageot.inference.resample_cells` (*tree*, *params*, *current\_node*='root', *inplace*=False)  
 Runs a new simulation of the cell evolution on a fixed tree

`lineageot.inference.robinson_foulds` (*tree1*, *tree2*)  
 Computes the Robinson-Foulds distance between two trees

`lineageot.inference.scaled_Hamming_distance` (*barcode1*, *barcode2*)  
 Computes the distance between two barcodes, adjusted for  
 (1) the number of sites where both cells were measured and  
 (2) distance between two scars is twice the distance from  
 scarred to unscarred

`lineageot.inference.split_edge` (*tree*, *edge*, *new\_node*)

`lineageot.inference.subtree_to_ete3` (*tree*, *current\_root*)  
 Converts the subtree from *current\_root* to ete3 format

`lineageot.inference.tree_accuracy` (*tree1*, *tree2*)  
 Returns the fraction of nontrivial splits appearing in both trees

`lineageot.inference.tree_discrepancy` (*tree1*, *tree2*)  
 Computes a version of the Robinson-Foulds distance between two trees rescaled to be between 0 and 1

`lineageot.inference.tree_to_ete3` (*tree*)  
 Converts a tree to ete3 format. Useful for calculating Robinson-Foulds distance.

`lineageot.inference.truncate_tree` (*tree*, *new\_end\_time*, *params*, *inplace*=False, *current\_node*='root', *next\_leaf\_to\_add*=0)  
 Removes all nodes at times greater than *new\_end\_time* and adds new leaves at exactly *new\_end\_time*  
*params*: simulation parameters used to create tree

## 1.4 Evaluation

This module contains functions for examining couplings after they are fitted, including comparing to a known ground truth. Nothing here is required for applying LineageOT to experimental data.

`lineageot.evaluation.coupling_W2` (*coupling\_1*, *coupling\_2*, *source*, *target*, *epsilon*)  
 Returns the entropically-regularized W2 distance between two couplings

`lineageot.evaluation.coupling_to_coupling_cost_matrix` (*source*, *target*)  
 Returns the (*n\_source*\**n\_target*)\*(*n\_source*\**n\_target*) cost matrix for a W2 distance between two couplings of *source* and *target*

Source and target here are just expression samples, without barcodes

```
lineageot.evaluation.expand_coupling(c, true_coupling, distances, matched_dim=0,
                                     max_dims_used=inf, xs_used=None)
```

#### Parameters

- **c** (*ndarray, shape (nx, ny) if matched\_dim == 0, (ny, nx) if matched\_dim == 1*) – Coupling between source x and target y
- **true\_coupling** (*ndarray, shape (nx, nz) if matched\_dim == 0, (nz, nx) if matched\_dim == 1*) – Reference coupling between x and z
- **distances** (*ndarray, shape (nz, ny)*) – Pairwise distances between z and y
- **matched\_dim** (*int*) – Dimension in which c and true coupling
- **max\_dims\_used** (*int or np.inf*) – Set a finite value here to do an approximate calculation based on min(nx, max\_dims\_used) elements of x
- **xs\_used** (*list or None*) – Indices of matched\_dim to use in approximate calculation. If None and max\_dims\_used < nx, indices are randomly selected.

**Returns** **expanded\_coupling** – Optimal coupling between z and y consistent with the coupling c

**Return type** *ndarray, shape same as true\_couplings*

```
lineageot.evaluation.expand_coupling_independent(c, true_coupling)
```

```
lineageot.evaluation.l2_difference(coupling_1, coupling_2)
```

```
lineageot.evaluation.normalize_columns(coupling)
```

```
lineageot.evaluation.pairwise_squared_distances(data)
```

Returns the pairwise squared distances between rows of the data matrix

```
lineageot.evaluation.plot2D_samples_mat(xs, xt, G, thr=1e-08, alpha_scale=1, **kwargs)
```

Plot matrix M in 2D with lines using alpha values

Plot lines between source and target 2D samples with a color proportional to the value of the matrix G between samples.

Copied function from PythonOT and added alpha\_scale parameter

#### Parameters

- **xs** (*ndarray, shape (ns, 2)*) – Source samples positions
- **b** (*ndarray, shape (nt, 2)*) – Target samples positions
- **G** (*ndarray, shape (na, nb)*) – OT matrix
- **thr** (*float, optional*) – threshold above which the line is drawn
- **\*\*kwargs** (*dict*) – parameters given to the plot functions (default color is black if nothing given)

```
lineageot.evaluation.plot_metrics(couplings, cost_func, cost_func_name, epsilons,
                                  log=False, points=False, scale=1.0, label_font_size=18,
                                  tick_font_size=12)
```

Plots cost\_func evaluated as a function of epsilon

```
lineageot.evaluation.print_metrics(couplings, cost_func, cost_func_name, log=False)
```

Prints cost\_func evaluated for each coupling in the dictionary couplings

```
lineageot.evaluation.sample_coordinates_from_coupling(c, row_points, column_points,
                                                    num_samples=None, return_all=False, thr=1e-06)
```

Generates [x, y] samples from the coupling *c*.

If *return\_all* is True, returns [x,y] coordinates of every pair with coupling value >thr

```
lineageot.evaluation.sample_indices_from_coupling(c, num_samples=None, return_all=False, thr=1e-06)
```

Generates [row, column] samples from the coupling *c*

If *return\_all* is True, then returns all indices with coupling values above the threshold

```
lineageot.evaluation.sample_interpolant(coupling, row_points, column_points, t=0.5,
                                         num_samples=None, return_all=False, thr=1e-06)
```

Samples from the interpolated distribution implied by the coupling

If *return\_all* is True, returns the interpolants between every pair with coupling value >thr. This is the exact interpolant distribution if and only if all nonzero values of the coupling are identical and >thr.

```
lineageot.evaluation.scaled_l2_difference(coupling_1, coupling_2)
```

```
lineageot.evaluation.squeeze_coupling(c, row_cluster_labels=None, column_cluster_labels=None)
```

```
lineageot.evaluation.squeeze_coupling_by_late_cluster(c, index)
```

```
lineageot.evaluation.tv(coupling1, coupling2)
```



## LINEAGEOT EXAMPLES

Here is a gallery of examples of LineageOT.

### 2.1 Minimal pipeline example

```
import anndata
import lineageot
import numpy as np

rng = np.random.default_rng()
```

#### 2.1.1 Creating data

First we make a minimal fake AnnData object to run LineageOT on.

```
t1 = 5;
t2 = 10;

n_cells_1 = 5;
n_cells_2 = 10;
n_cells = n_cells_1 + n_cells_2;

n_genes = 5;

barcode_length = 10;

adata = anndata.AnnData(X = np.random.rand(n_cells, n_genes),
                        obs = {"time" : np.concatenate([t1*np.ones(n_cells_1), t2*np.
↳ ones(n_cells_2)])},
                        obsm = {"barcodes" : rng.integers(low = -1, high = 10, size =
↳ (n_cells, barcode_length))}
                        )
```

### 2.1.2 Fitting a lineage tree

Before running LineageOT, we need to build a lineage tree from the observed barcodes. This step is not optimized. We provide an implementation of a heuristic algorithm called neighbor joining. Feel free to use your own preferred tree construction algorithm. You can import a tree saved in Newick format with `lineageot.read_newick`.

The tree should be formatted as a `NetworkX DiGraph` in the same way as the output of `lineageot.fit_tree()`. Each node is annotated with `'time'` (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with `'time'` equal to the child node's `'time_to_parent'`. Observed node indices correspond to their row in `adata[adata.obs['time'] == t2]`.

```
lineage_tree_t2 = lineageot.fit_tree(adata[adata.obs['time'] == t2], t2)
```

### 2.1.3 Running LineageOT

Once we have a lineage tree annotated with time, we can compute a LineageOT coupling.

```
coupling = lineageot.fit_lineage_coupling(adata, t1, t2, lineage_tree_t2)
```

### 2.1.4 Saving

The LineageOT package does not include functionality for downstream analysis and plotting. We recommend transitioning to other packages, like [Waddington-OT](#), after computing a coupling. This saves the fitted coupling in a format Waddington-OT can import.

```
lineageot.save_coupling_as_tmap(coupling, t1, t2, './tmaps/example')
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 2.2 LineageOT with static lineage tracing

While designed for dynamic lineage tracing with continuously edited barcodes, LineageOT can be applied to any time course where a lineage tree can be created, including static barcoding data.

With some forms of static barcoding, more information is available than LineageOT uses. LineageOT does not account for the possibility that the same barcode could be observed at multiple time points. If that happens in your data, you can still use LineageOT, but should also consider other methods.

```
import anndata
import lineageot
import numpy as np

rng = np.random.default_rng()
```

### 2.2.1 Creating data

First we make a minimal fake AnnData object to run LineageOT on. Here, the lineage information is encoded in a Boolean matrix with cells as rows and clones as column, where entry  $[i, j]$  is 1 if and only if cell  $i$  belongs to clone  $j$ . This example has two initial clones labeled at time 0 and four subclones labeled at time 7.

In addition to the clone identities, LineageOT also needs a time for each clone. This is encoded in the vector `clone_times`, whose entries give the time of labeling of the clones.

```
t1 = 5;
t2 = 10;

n_cells_1 = 4;
n_cells_2 = 8;
n_cells = n_cells_1 + n_cells_2;

n_genes = 5;

# clones labeled at time 0
time_0_clones = np.concatenate([np.kron(np.identity(2), np.ones((2,1))),
                                np.kron(np.identity(2), np.ones((4,1)))])

# clones labeled at time 7
time_7_clones = np.concatenate([np.zeros((4,4)),
                                np.kron(np.identity(4), np.ones((2,1)))])
clones = np.concatenate([time_0_clones, time_7_clones], 1)

clone_times = np.array([0, 0, 7, 7, 7, 7])

adata = anndata.AnnData(X = np.random.rand(n_cells, n_genes),
                        obs = {"time" : np.concatenate([t1*np.ones(n_cells_1), t2*np.
→ones(n_cells_2)])},
                        obsm = {"X_clone" : clones}
                        )

print(clones)
```

Out:

```
[[1. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [1. 0. 0. 1. 0. 0.]
 [1. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 1. 0.]
 [0. 1. 0. 0. 1. 0.]
 [0. 1. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 1.]]
```

## 2.2.2 Fitting a lineage tree

Before running LineageOT, we need to build a lineage tree from the observed barcodes. For static lineage tracing data, we provide an algorithm to construct a tree of possibly-nested clones, assuming there are no barcode collisions across clones so the phylogeny is straightforward to reconstruct. This step is not optimized. Feel free to use your own preferred tree construction algorithm. You can import a tree saved in Newick format with `lineageot.read_newick`.

The tree should be formatted as a `NetworkX DiGraph` in the same way as the output of `lineageot.fit_tree()`. Each node is annotated with `'time'` (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with `'time'` equal to the child node's `'time_to_parent'`. Observed node indices correspond to their row in `adata[adata.obs['time'] == t2]`.

```
lineage_tree_t2 = lineageot.fit_tree(adata[adata.obs['time'] == t2], t2, clone_times_  
↪= clone_times, method = 'clones')
```

## 2.2.3 Running LineageOT

Once we have a lineage tree annotated with time, we can compute a LineageOT coupling.

```
coupling = lineageot.fit_lineage_coupling(adata, t1, t2, lineage_tree_t2)
```

## 2.2.4 Saving

The LineageOT package does not include functionality for downstream analysis and plotting. We recommend transitioning to other packages, like [Waddington-OT](#), after computing a coupling. This saves the fitted coupling in a format Waddington-OT can import.

```
lineageot.save_coupling_as_tmap(coupling, t1, t2, './tmaps/example')
```

**Total running time of the script:** ( 0 minutes 0.210 seconds)

## 2.3 LineageOT on a convergent trajectory

This shows results of applying LineageOT to a simulation of convergent trajectories, closely following `simulation_demo.ipynb` in the source code.

```
import copy  
import matplotlib.pyplot as plt  
import numpy as np  
import ot  
  
import lineageot.simulation as sim  
import lineageot.evaluation as sim_eval  
import lineageot.inference as sim_inf
```



### 2.3.1 Generating simulated data

```
flow_type = 'convergent'
np.random.seed(257)
```

Setting simulation parameters

```
if flow_type == 'bifurcation':
    timescale = 1
else:
    timescale = 100

x0_speed = 1/timescale

sim_params = sim.SimulationParameters(division_time_std = 0.01*timescale,
                                      flow_type = flow_type,
                                      x0_speed = x0_speed,
                                      mutation_rate = 1/timescale,
                                      mean_division_time = 1.1*timescale,
                                      timestep = 0.001*timescale
                                      )

mean_x0_early = 2
time_early = 4*timescale # Time when early cells are sampled
time_late = time_early + 4*timescale # Time when late cells are sampled
x0_initial = mean_x0_early - time_early*x0_speed
initial_cell = sim.Cell(np.array([x0_initial, 0, 0]), np.zeros(sim_params.barcode_
    ↪ length))
sample_times = {'early' : time_early, 'late' : time_late}

# Choosing which of the three dimensions to show in later plots
if flow_type == 'mismatched_clusters':
    dimensions_to_plot = [1,2]
else:
    dimensions_to_plot = [0,1]
```

Running the simulation

```
sample = sim.sample_descendants(initial_cell.deepcopy(), time_late, sim_params)
```

### 2.3.2 Processing simulation output

```
# Extracting trees and barcode matrices
true_trees = {'late': sim_inf.list_tree_to_digraph(sample)}
true_trees['late'].nodes['root']['cell'] = initial_cell

true_trees['early'] = sim_inf.truncate_tree(true_trees['late'], sample_times['early'],
    ↪ sim_params)

# Computing the ground-truth coupling
couplings = {'true': sim_inf.get_true_coupling(true_trees['early'], true_trees['late'
    ↪ ''])}

data_arrays = {'late' : sim_inf.extract_data_arrays(true_trees['late'])}
```

(continues on next page)

(continued from previous page)

```
rna_arrays = {'late': data_arrays['late'][0]}
barcode_arrays = {'late': data_arrays['late'][1]}

rna_arrays['early'] = sim_inf.extract_data_arrays(true_trees['early'])[0]
num_cells = {'early': rna_arrays['early'].shape[0], 'late': rna_arrays['late'].
↳ shape[0]}

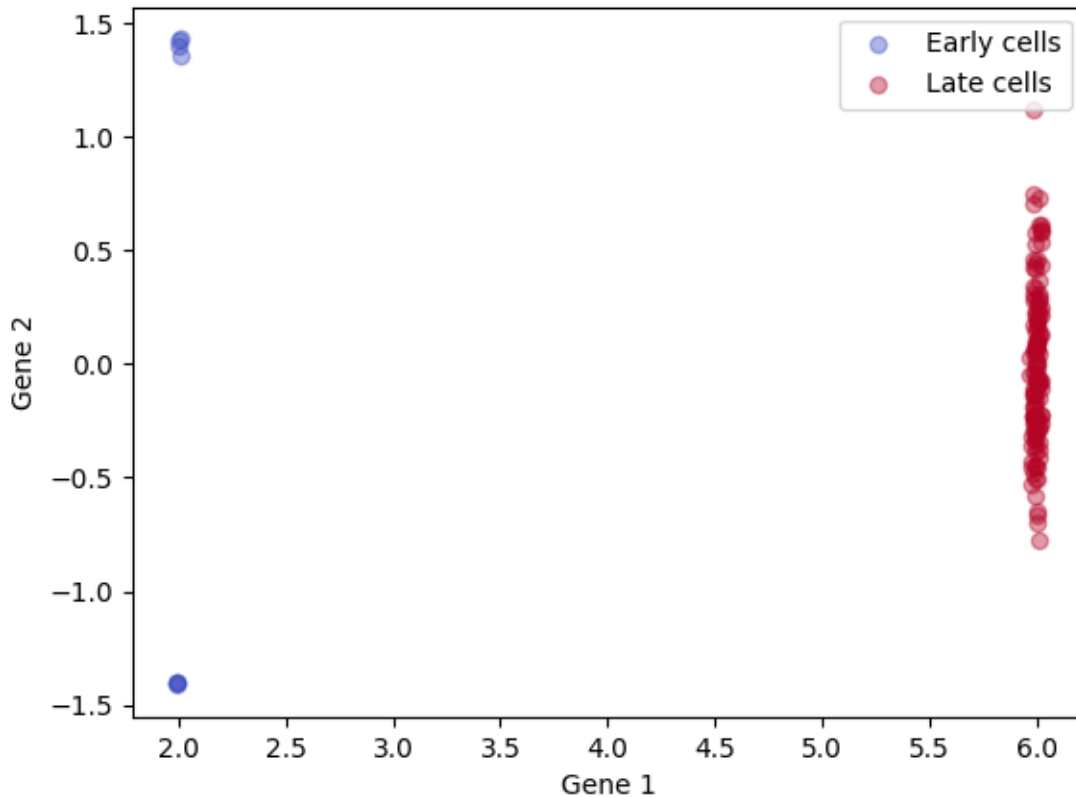
print("Times      : ", sample_times)
print("Number of cells: ", num_cells)

# Creating a copy of the true tree for use in LineageOT
true_trees['late, annotated'] = copy.deepcopy(true_trees['late'])
sim_inf.add_node_times_from_division_times(true_trees['late, annotated'])
sim_inf.add_nodes_at_time(true_trees['late, annotated'], sample_times['early']);

# Scatter plot of cell states

cmap = "coolwarm"
colors = [plt.get_cmap(cmap)(0), plt.get_cmap(cmap)(256)]
for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
↳ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
↳ label = label, color = c)

plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.legend();
```



Out:

```
Times      : {'early': 400, 'late': 800}
Number of cells: {'early': 8, 'late': 128}

<matplotlib.legend.Legend object at 0x7fe924b79150>
```

Since these are simulations, we can compute and plot inferred ancestor locations based on the true tree.

```
# Infer ancestor locations for the late cells based on the true lineage tree
observed_nodes = [n for n in sim_inf.get_leaves(true_trees['late, annotated'],
↳ include_root=False)]
sim_inf.add_conditional_means_and_variances(true_trees['late, annotated'], observed_
↳ nodes)

ancestor_info = {'true tree': sim_inf.get_ancestor_data(true_trees['late, annotated'],
↳ sample_times['early'])}

# Scatter plot of cell states, with inferred ancestor locations for the late cells

for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
↳ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
↳ label = label, color = c)

plt.scatter(ancestor_info['true tree'][0][:, dimensions_to_plot[0]],
```

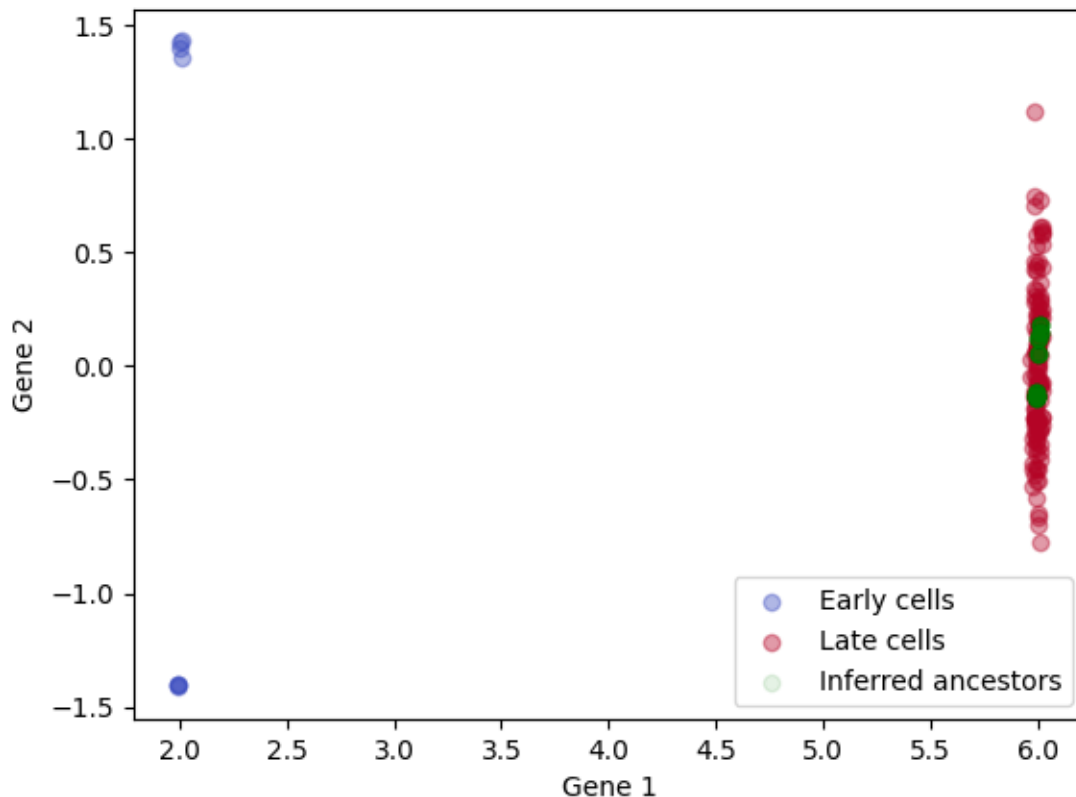
(continues on next page)

(continued from previous page)

```

        ancestor_info['true tree'][0][:,dimensions_to_plot[1]],
        alpha = 0.1,
        label = 'Inferred ancestors',
        color = 'green')
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.legend();

```



Out:

```
<matplotlib.legend.Legend object at 0x7fe924abb150>
```

To better visualize cases where there were two clusters at the early time point, we can color the late cells (and their inferred ancestors) by their cluster of origin. Cells in orange are from the late time point with ancestors on the left; cells in green are from the late time point with ancestors on the right. Though the green and orange distributions substantially overlap, the estimated ancestor distributions in red and purple are separate.

```

is_from_left = sim_inf.extract_ancestor_data_arrays(true_trees['late'], sample_times[
    → 'early'], sim_params)[0][:,1] < 0
for a,label in zip([rna_arrays['early'], rna_arrays['late'][is_from_left,:], rna_
    → arrays['late'][~is_from_left,:]], ['Early cells', 'Late cells from left', 'Late_
    → cells from right']):
    plt.scatter(a[:, 1], a[:, 2], alpha = 0.4)

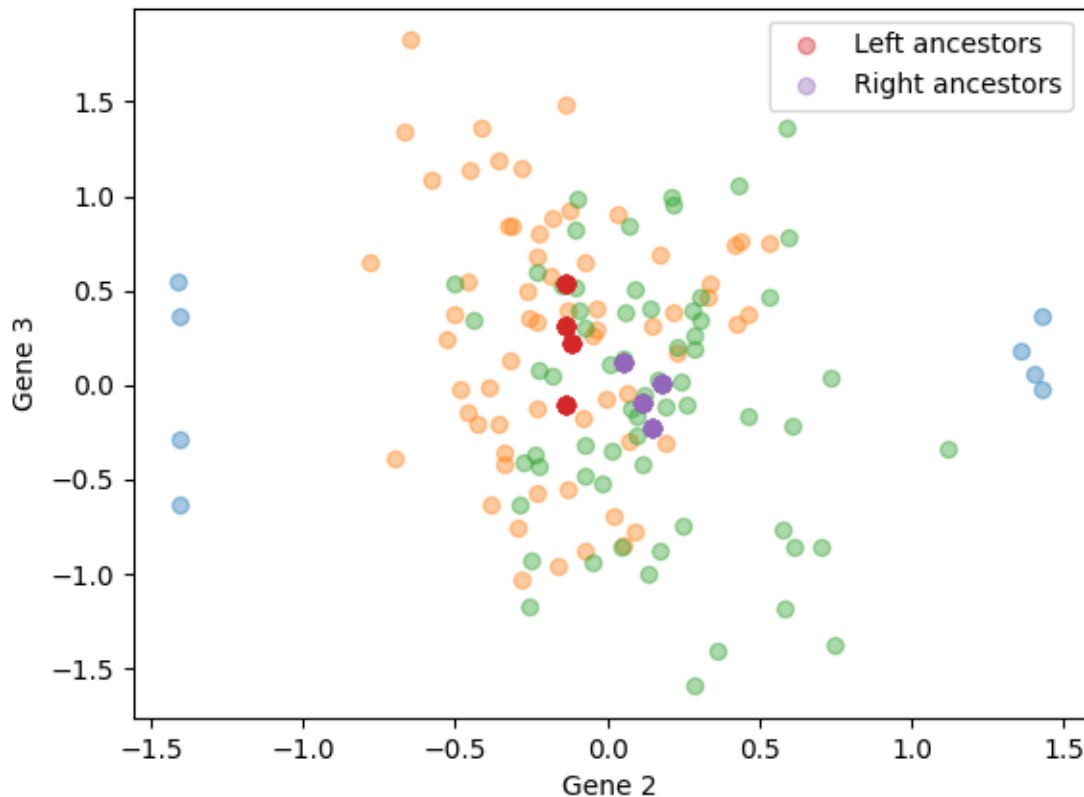
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Gene 2')
plt.ylabel('Gene 3')

for a, label in zip([ancestor_info['true tree'][0][is_from_left, :], ancestor_info[
    ↪ 'true tree'][0][~is_from_left, :]], ['Left ancestors', 'Right ancestors']):
    plt.scatter(a[:,1], a[:,2], alpha = 0.4, label = label)
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7fe924a726d0>
```

### 2.3.3 Running LineageOT

The first step is to fit a lineage tree to observed barcodes

```
# True distances
true_distances = {key:sim_inf.compute_tree_distances(true_trees[key]) for key in true_
    ↪ trees}

# Estimate mutation rate from fraction of unmutated barcodes
```

(continues on next page)

(continued from previous page)

```

rate_estimate = sim_inf.rate_estimator(barcode_arrays['late'], sample_times['late'])

# Compute Hamming distance matrices for neighbor joining
hamming_distances_with_roots = {'late':sim_inf.barcode_distances(np.
    ↳concatenate([barcode_arrays['late'],
                                                                np.
    ↳zeros([1,sim_params.barcode_length]))))}

# Compute neighbor-joining tree
fitted_tree = sim_inf.neighbor_join(hamming_distances_with_roots['late'])

```

Once the tree is computed, we need to annotate it with node times and states

```

# Annotate fitted tree with internal node times
sim_inf.add_leaf_barcodes(fitted_tree, barcode_arrays['late'])
sim_inf.add_leaf_x(fitted_tree, rna_arrays['late'])
sim_inf.add_leaf_times(fitted_tree, sample_times['late'])
sim_inf.annotate_tree(fitted_tree,
                      rate_estimate*np.ones(sim_params.barcode_length),
                      time_inference_method = 'least_squares');

# Add inferred ancestor nodes and states
sim_inf.add_node_times_from_division_times(fitted_tree)
sim_inf.add_nodes_at_time(fitted_tree, sample_times['early'])
observed_nodes = [n for n in sim_inf.get_leaves(fitted_tree, include_root = False)]
sim_inf.add_conditional_means_and_variances(fitted_tree, observed_nodes)
ancestor_info['fitted tree'] = sim_inf.get_ancestor_data(fitted_tree, sample_times[
    ↳'early'])

```

Out:

```

      pcost      dcost      gap      pres      dres
0: -4.0661e+07 -4.2066e+07  6e+06  1e-01  2e-01
1: -4.0696e+07 -4.1441e+07  8e+05  8e-03  2e-02
2: -4.0803e+07 -4.1023e+07  2e+05  2e-03  4e-03
3: -4.0851e+07 -4.0887e+07  4e+04  1e-16  1e-16
4: -4.0862e+07 -4.0866e+07  4e+03  1e-16  2e-16
5: -4.0863e+07 -4.0864e+07  3e+02  1e-16  2e-16
6: -4.0863e+07 -4.0863e+07  1e+01  1e-16  4e-16
Optimal solution found.

```

We're now ready to compute LineageOT cost matrices

```

# Compute cost matrices for each method
coupling_costs = {}
coupling_costs['lineageOT, true tree'] = ot.utils.dist(rna_arrays['early'], ancestor_
    ↳info['true tree'][0])@np.diag(ancestor_info['true tree'][1]**(-1))
coupling_costs['OT'] = ot.utils.dist(rna_arrays['early'], rna_arrays['late'])
coupling_costs['lineageOT, fitted tree'] = ot.utils.dist(rna_arrays['early'],
    ↳ancestor_info['fitted tree'][0])@np.diag(ancestor_info['fitted tree'][1]**(-1))

early_time_rna_cost = ot.utils.dist(rna_arrays['early'], sim_inf.extract_ancestor_
    ↳data_arrays(true_trees['late'], sample_times['early'], sim_params)[0])
late_time_rna_cost = ot.utils.dist(rna_arrays['late'], rna_arrays['late'])

```

Given the cost matrices, we can fit couplings with a range of entropy parameters.

```

epsilons = np.logspace(-2, 3, 15)

couplings['OT'] = ot.emd([], [], coupling_costs['OT'])
couplings['lineageOT'] = ot.emd([], [], coupling_costs['lineageOT, true tree'])
couplings['lineageOT, fitted'] = ot.emd([], [], coupling_costs['lineageOT, fitted tree
↳'])
for e in epsilons:
    if e >= 0.1:
        f = ot.sinkhorn
    else:
        # Epsilon scaling is more robust at smaller epsilon, but slower than simple_
↳sinkhorn
        f = ot.bregman.sinkhorn_epsilon_scaling
        couplings['entropic rna ' + str(e)] = f([], [], coupling_costs['OT'], e)
        couplings['lineage entropic rna ' + str(e)] = f([], [], coupling_costs['lineageOT,
↳ true tree'], e*np.mean(ancestor_info['true tree'][1]**(-1)))
        couplings['fitted lineage rna ' + str(e)] = f([], [], coupling_costs['lineageOT,
↳ fitted tree'], e*np.mean(ancestor_info['fitted tree'][1]**(-1)))

```

Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/lineageot/envs/develop/lib/python3.7/
↳site-packages/ot/bregman.py:1112: UserWarning: Sinkhorn did not converge. You might
↳want to increase the number of iterations `numItermax` or the regularization_
↳parameter `reg`.
    warnings.warn("Sinkhorn did not converge. You might want to "

```

## 2.3.4 Evaluation of couplings

First compute the independent coupling as a reference

```

couplings['independent'] = np.ones(couplings['OT'].shape)/couplings['OT'].size
ind_ancestor_error = sim_inf.OT_cost(couplings['independent'], early_time_rna_cost)
ind_descendant_error = sim_inf.OT_cost(sim_eval.expand_coupling(couplings['independent
↳'],
                                                                    couplings['true'],
                                                                    late_time_rna_cost),
                                                                    late_time_rna_cost)

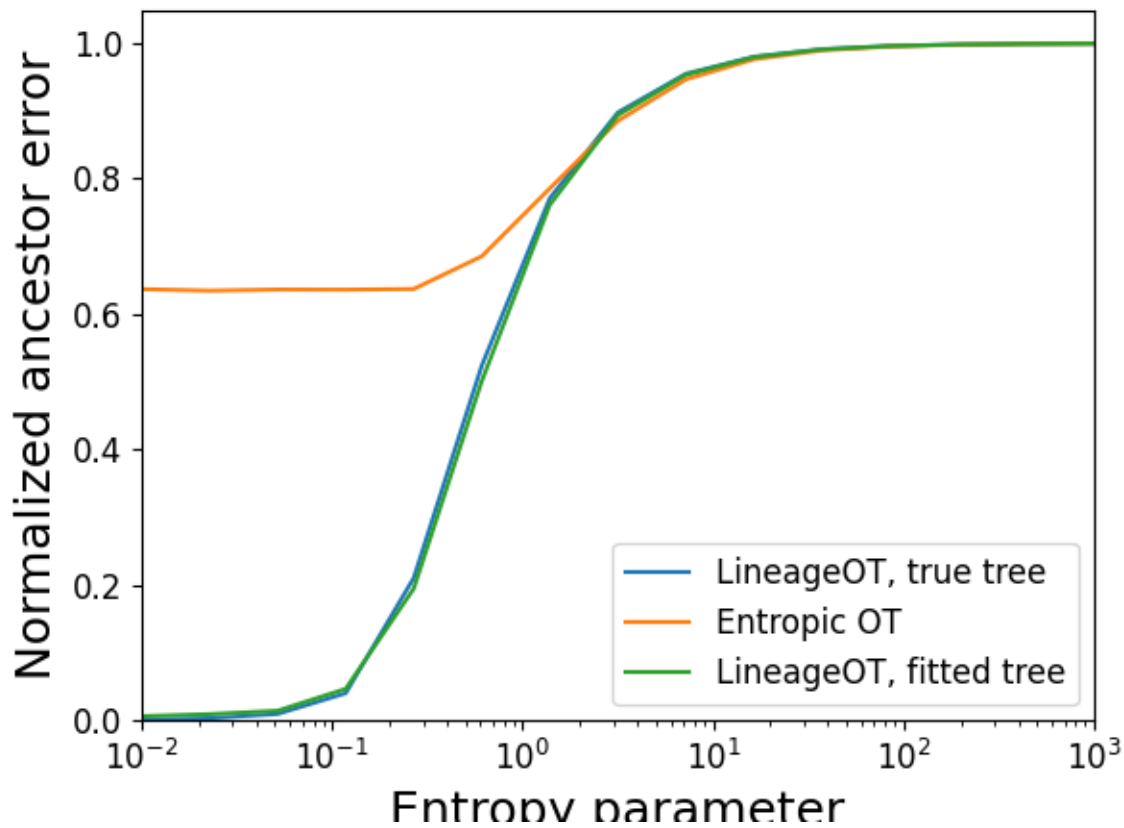
```

Plotting the accuracy of ancestor prediction

```

ancestor_errors = sim_eval.plot_metrics(couplings,
                                       lambda x: sim_inf.OT_cost(x, early_time_rna_
↳cost),
                                       'Normalized ancestor error',
                                       epsilons,
                                       scale = ind_ancestor_error,
                                       points=False)

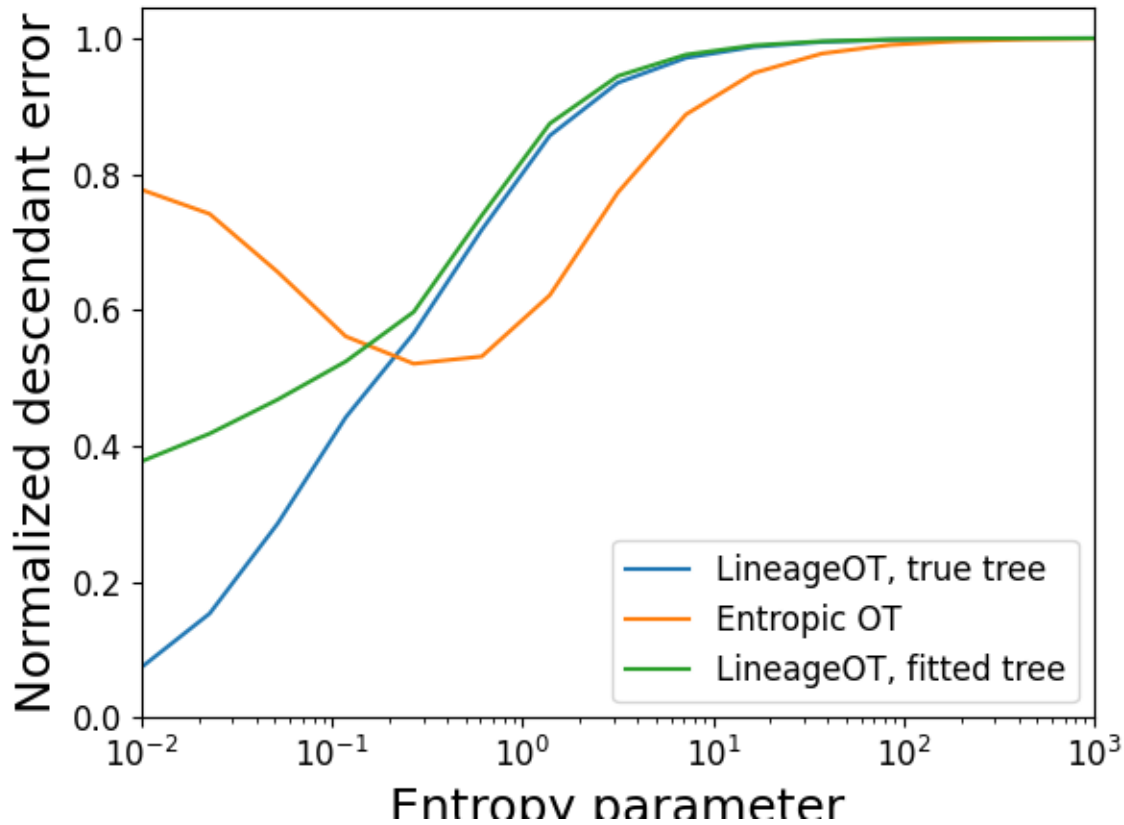
```



Plotting the accuracy of descendant prediction

```
descendant_errors = sim_eval.plot_metrics(couplings,
                                         lambda x:sim_inf.OT_cost(sim_eval.expand_
↳ coupling(x,
                                         couplings['true'],
↳ late_time_rna_cost),
                                         late_time_rna_
↳ cost),
                                         'Normalized descendant error',
                                         epsilon, scale = ind_descendant_error)
```





### 2.3.5 Coupling visualizations

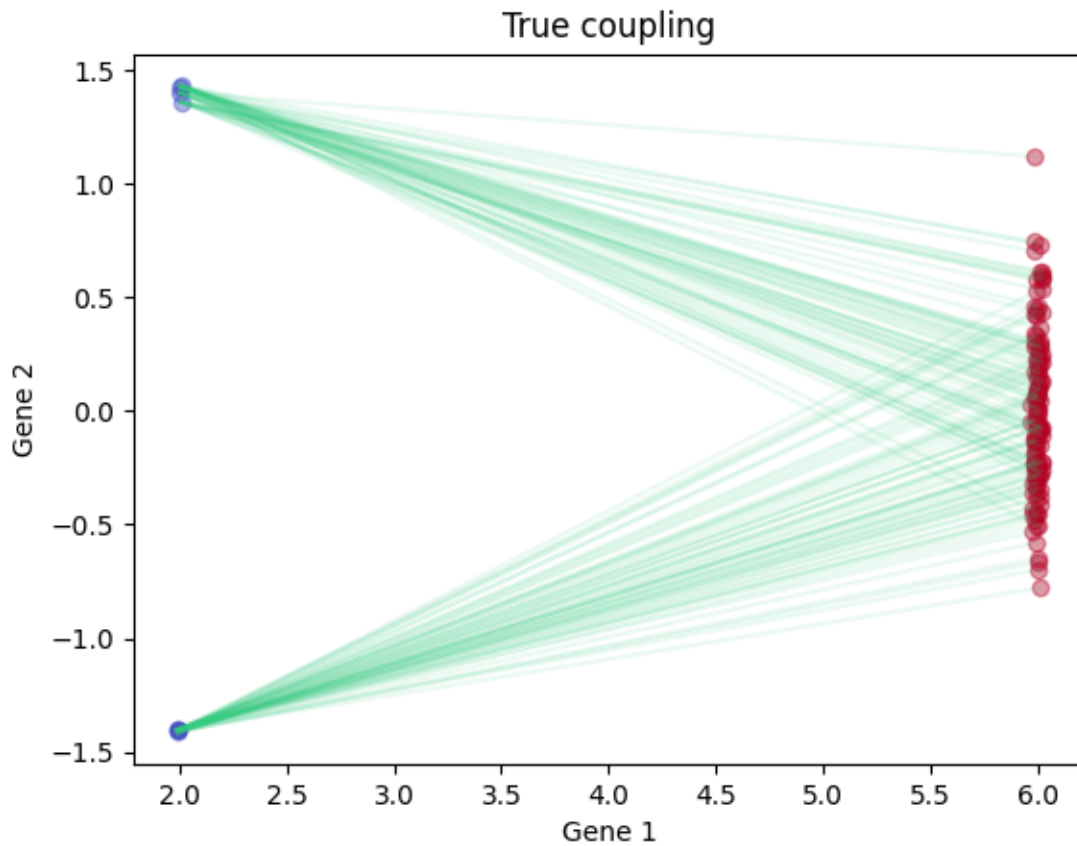
Visualizing the ground-truth coupling, zero-entropy LineageOT coupling, and zero-entropy optimal transport coupling.

Ground truth:

```
sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
→to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
→plot[1]]],
                           couplings['true'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)

plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('True coupling')

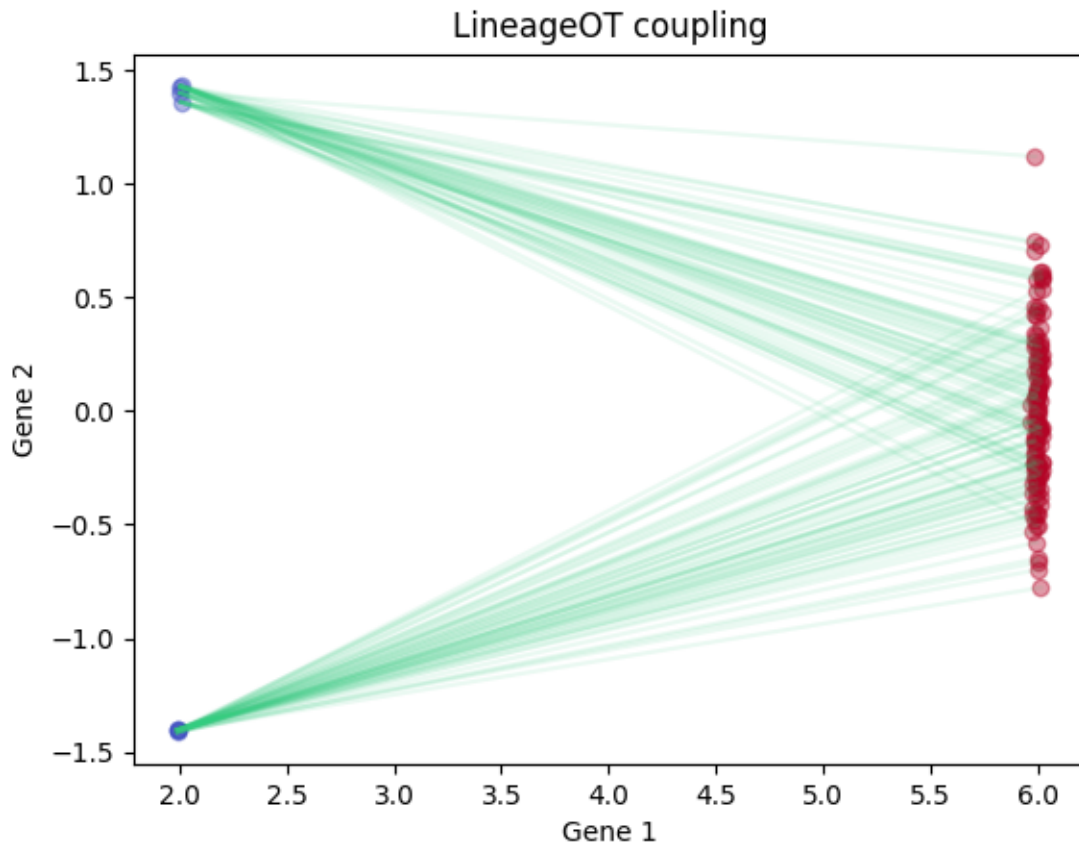
for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
→ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
    → label = label, color = c)
```



LineageOT:

```
sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
→to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
→plot[1]]],
                           couplings['lineageOT'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('LineageOT coupling')

for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
→'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
→ label = label, color = c)
```



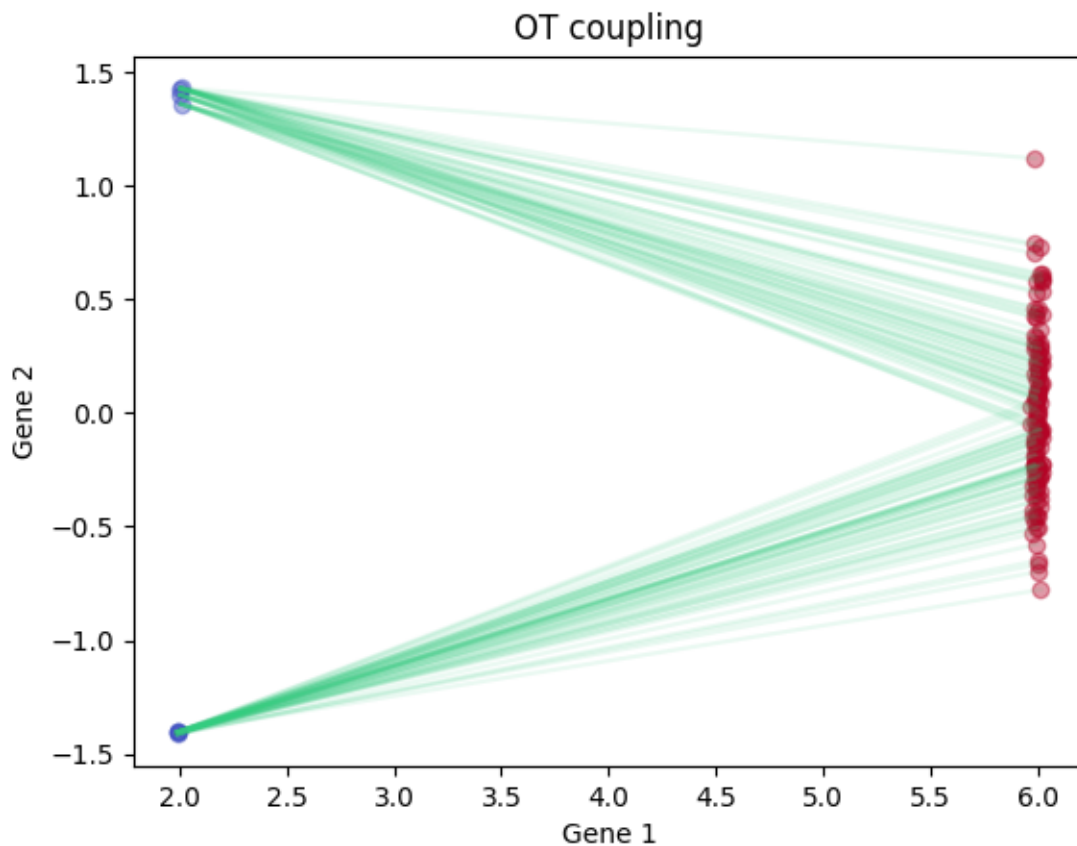
Optimal transport

```

sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
↪to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
↪plot[1]]],
                           couplings['OT'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('OT coupling')

for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
↪ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
↪ label = label, color = c)

```



Total running time of the script: ( 0 minutes 6.889 seconds)

## 2.4 LineageOT on a curled trajectory

This shows results of applying LineageOT to a simulation where descendant cells are not all closest to their ancestors, closely following `simulation_demo.ipynb` in the source code.

```
import copy
import matplotlib.pyplot as plt
import numpy as np
import ot

import lineageot.simulation as sim
import lineageot.evaluation as sim_eval
import lineageot.inference as sim_inf
```

### 2.4.1 Generating simulated data

```
flow_type = 'mismatched_clusters'
np.random.seed(257)
```

Setting simulation parameters

```
if flow_type == 'bifurcation':
    timescale = 1
else:
    timescale = 100

x0_speed = 1/timescale

sim_params = sim.SimulationParameters(division_time_std = 0.01*timescale,
                                       flow_type = flow_type,
                                       x0_speed = x0_speed,
                                       mutation_rate = 1/timescale,
                                       mean_division_time = 1.1*timescale,
                                       timestep = 0.001*timescale
                                       )

mean_x0_early = 2
time_early = 4*timescale # Time when early cells are sampled
time_late = time_early + 4*timescale # Time when late cells are sampled
x0_initial = mean_x0_early - time_early*x0_speed
initial_cell = sim.Cell(np.array([x0_initial, 0, 0]), np.zeros(sim_params.barcode_
    ↪ length))
sample_times = {'early' : time_early, 'late' : time_late}

# Choosing which of the three dimensions to show in later plots
if flow_type == 'mismatched_clusters':
    dimensions_to_plot = [1,2]
else:
    dimensions_to_plot = [0,1]
```

Running the simulation

```
sample = sim.sample_descendants(initial_cell.deepcopy(), time_late, sim_params)
```

### 2.4.2 Processing simulation output

```
# Extracting trees and barcode matrices
true_trees = {'late': sim_inf.list_tree_to_digraph(sample)}
true_trees['late'].nodes['root']['cell'] = initial_cell

true_trees['early'] = sim_inf.truncate_tree(true_trees['late'], sample_times['early'],
    ↪ sim_params)

# Computing the ground-truth coupling
couplings = {'true': sim_inf.get_true_coupling(true_trees['early'], true_trees['late'
    ↪ ''])}

data_arrays = {'late' : sim_inf.extract_data_arrays(true_trees['late'])}
```

(continues on next page)

(continued from previous page)

```
rna_arrays = {'late': data_arrays['late'][0]}
barcode_arrays = {'late': data_arrays['late'][1]}

rna_arrays['early'] = sim_inf.extract_data_arrays(true_trees['early'])[0]
num_cells = {'early': rna_arrays['early'].shape[0], 'late': rna_arrays['late'].
↳ shape[0]}

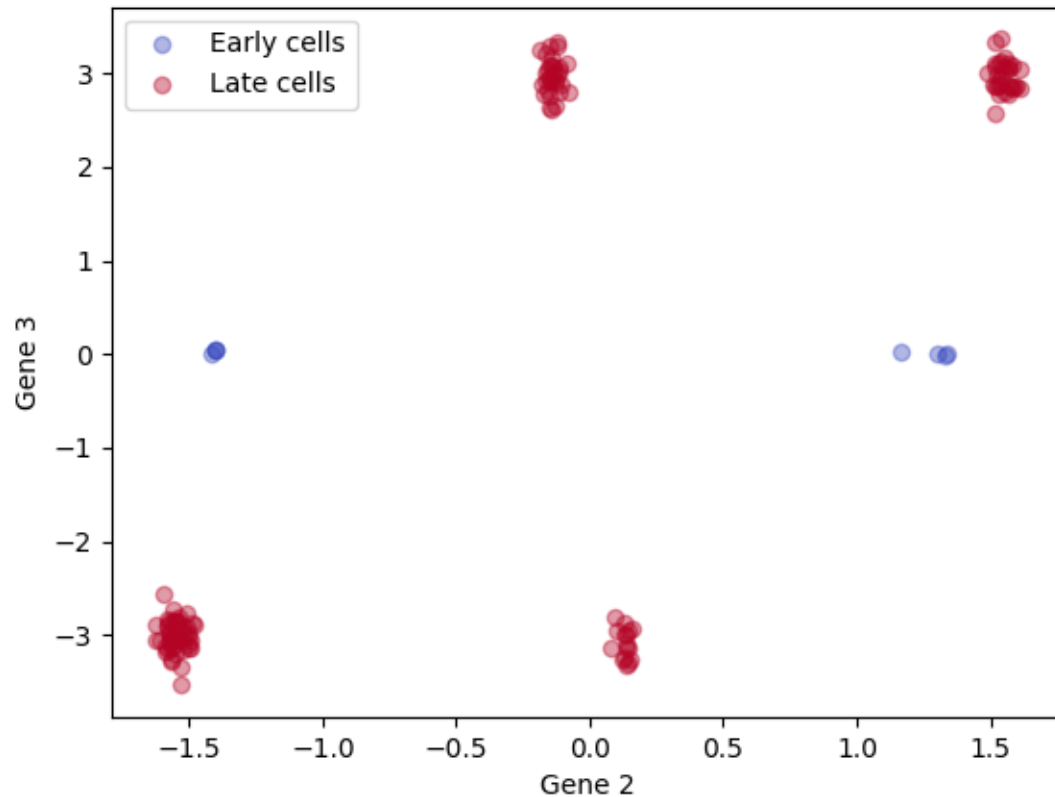
print("Times      : ", sample_times)
print("Number of cells: ", num_cells)

# Creating a copy of the true tree for use in LineageOT
true_trees['late, annotated'] = copy.deepcopy(true_trees['late'])
sim_inf.add_node_times_from_division_times(true_trees['late, annotated'])
sim_inf.add_nodes_at_time(true_trees['late, annotated'], sample_times['early']);

# Scatter plot of cell states

cmap = "coolwarm"
colors = [plt.get_cmap(cmap)(0), plt.get_cmap(cmap)(256)]
for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
↳ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
↳ label = label, color = c)

plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.legend();
```



Out:

```
Times      : {'early': 400, 'late': 800}
Number of cells: {'early': 8, 'late': 128}

<matplotlib.legend.Legend object at 0x7fe924a3dc10>
```

Since these are simulations, we can compute and plot inferred ancestor locations based on the true tree.

```
# Infer ancestor locations for the late cells based on the true lineage tree
observed_nodes = [n for n in sim_inf.get_leaves(true_trees['late, annotated'],
↳ include_root=False)]
sim_inf.add_conditional_means_and_variances(true_trees['late, annotated'], observed_
↳ nodes)

ancestor_info = {'true tree': sim_inf.get_ancestor_data(true_trees['late, annotated'],
↳ sample_times['early'])}

# Scatter plot of cell states, with inferred ancestor locations for the late cells

for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
↳ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
↳ label = label, color = c)

plt.scatter(ancestor_info['true tree'][0][:, dimensions_to_plot[0]],
```

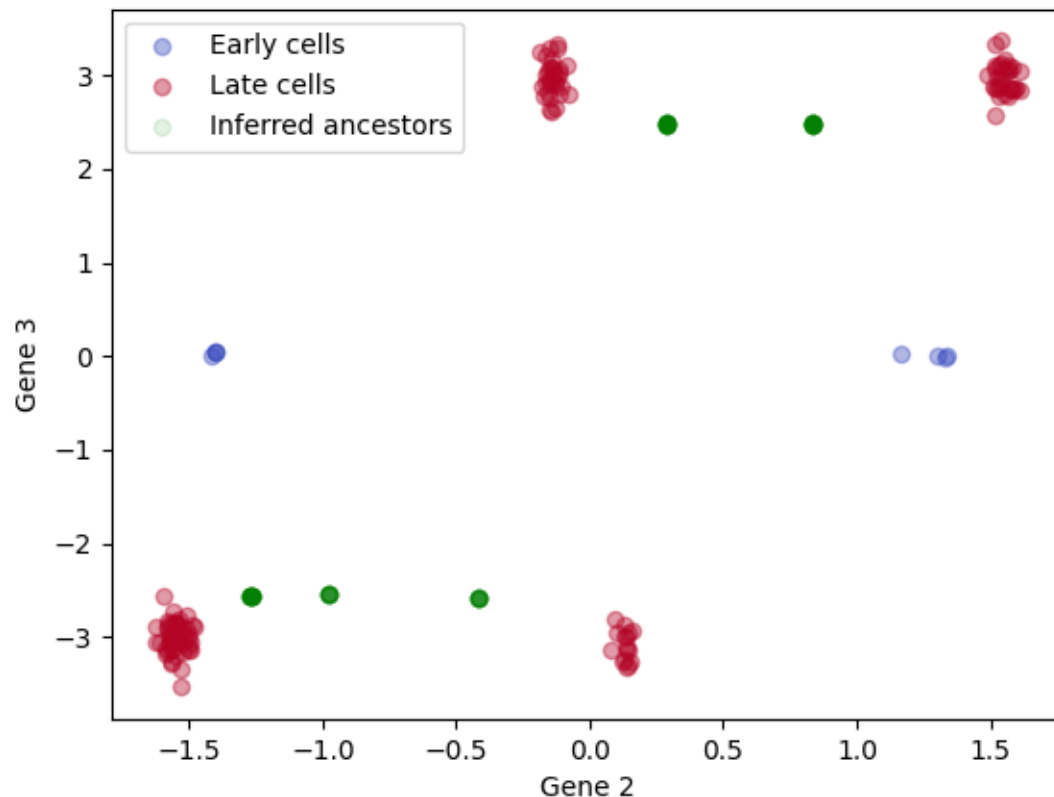
(continues on next page)

(continued from previous page)

```

        ancestor_info['true tree'][0][:, dimensions_to_plot[1]],
        alpha = 0.1,
        label = 'Inferred ancestors',
        color = 'green')
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.legend();

```



Out:

```
<matplotlib.legend.Legend object at 0x7fe91c813150>
```

To better visualize cases where there were two clusters at the early time point, we can color the late cells (and their inferred ancestors) by their cluster of origin. Cells in orange are from the late time point with ancestors on the left; cells in green are from the late time point with ancestors on the right. The estimated ancestor distributions in red and purple are closer to the true ancestors than the observations in orange and green.

```

is_from_left = sim_inf.extract_ancestor_data_arrays(true_trees['late'], sample_times[
    → 'early'], sim_params)[0][:, 1] < 0
for a, label in zip([rna_arrays['early'], rna_arrays['late'][is_from_left, :], rna_
    → arrays['late'][~is_from_left, :]], ['Early cells', 'Late cells from left', 'Late_
    → cells from right']):
    plt.scatter(a[:, 1], a[:, 2], alpha = 0.4)

```

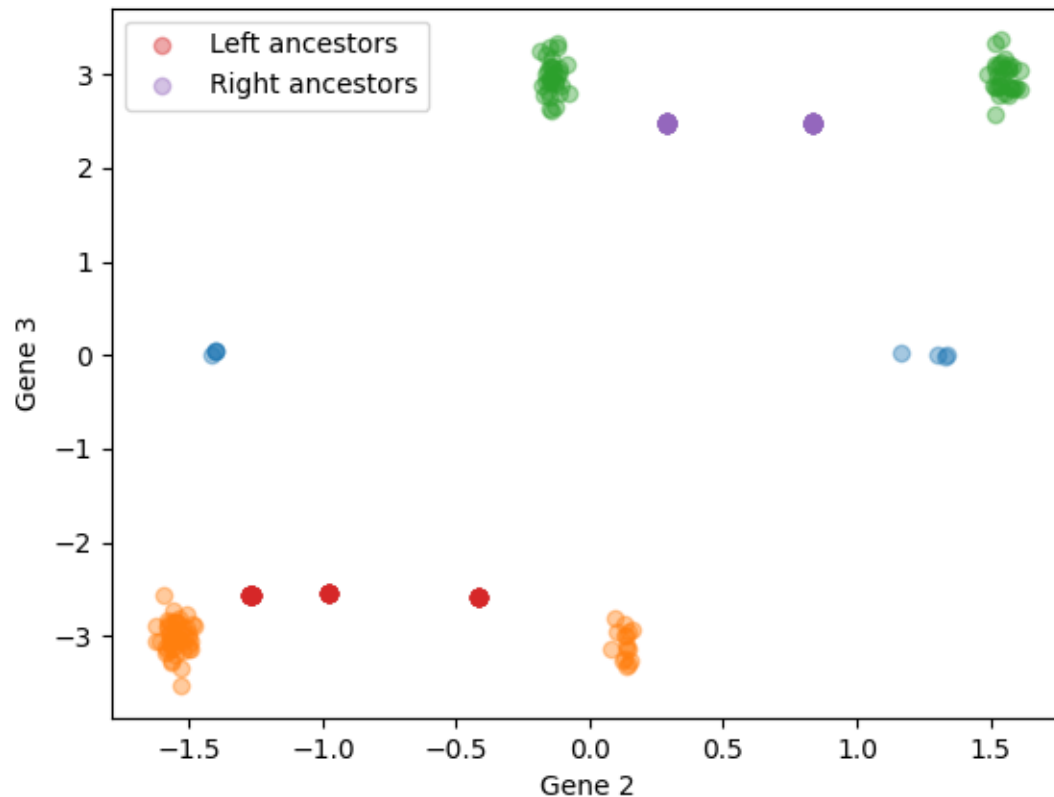
(continues on next page)



(continued from previous page)

```
plt.xlabel('Gene 2')
plt.ylabel('Gene 3')

for a, label in zip([ancestor_info['true tree'][0][is_from_left, :], ancestor_info[
    ↪ 'true tree'][0][~is_from_left, :]], ['Left ancestors', 'Right ancestors']):
    plt.scatter(a[:,1], a[:,2], alpha = 0.4, label = label)
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7fe924c6ced0>
```

### 2.4.3 Running LineageOT

The first step is to fit a lineage tree to observed barcodes

```
# True distances
true_distances = {key:sim_inf.compute_tree_distances(true_trees[key]) for key in true_
    ↪ trees}

# Estimate mutation rate from fraction of unmutated barcodes
```

(continues on next page)

(continued from previous page)

```

rate_estimate = sim_inf.rate_estimator(barcode_arrays['late'], sample_times['late'])

# Compute Hamming distance matrices for neighbor joining
hamming_distances_with_roots = {'late':sim_inf.barcode_distances(np.
    ↳concatenate([barcode_arrays['late'],
                                                                np.
    ↳zeros([1,sim_params.barcode_length]))))}

# Compute neighbor-joining tree
fitted_tree = sim_inf.neighbor_join(hamming_distances_with_roots['late'])

```

Once the tree is computed, we need to annotate it with node times and states

```

# Annotate fitted tree with internal node times
sim_inf.add_leaf_barcodes(fitted_tree, barcode_arrays['late'])
sim_inf.add_leaf_x(fitted_tree, rna_arrays['late'])
sim_inf.add_leaf_times(fitted_tree, sample_times['late'])
sim_inf.annotate_tree(fitted_tree,
                      rate_estimate*np.ones(sim_params.barcode_length),
                      time_inference_method = 'least_squares');

# Add inferred ancestor nodes and states
sim_inf.add_node_times_from_division_times(fitted_tree)
sim_inf.add_nodes_at_time(fitted_tree, sample_times['early'])
observed_nodes = [n for n in sim_inf.get_leaves(fitted_tree, include_root = False)]
sim_inf.add_conditional_means_and_variances(fitted_tree, observed_nodes)
ancestor_info['fitted tree'] = sim_inf.get_ancestor_data(fitted_tree, sample_times[
    ↳'early'])

```

Out:

```

      pcost      dcost      gap      pres      dres
0: -4.0661e+07 -4.2066e+07  6e+06  1e-01  2e-01
1: -4.0696e+07 -4.1441e+07  8e+05  8e-03  2e-02
2: -4.0803e+07 -4.1023e+07  2e+05  2e-03  4e-03
3: -4.0851e+07 -4.0887e+07  4e+04  1e-16  1e-16
4: -4.0862e+07 -4.0866e+07  4e+03  1e-16  2e-16
5: -4.0863e+07 -4.0864e+07  3e+02  1e-16  2e-16
6: -4.0863e+07 -4.0863e+07  1e+01  1e-16  4e-16
Optimal solution found.

```

We're now ready to compute LineageOT cost matrices

```

# Compute cost matrices for each method
coupling_costs = {}
coupling_costs['lineageOT, true tree'] = ot.utils.dist(rna_arrays['early'], ancestor_
    ↳info['true tree'][0])@np.diag(ancestor_info['true tree'][1]**(-1))
coupling_costs['OT'] = ot.utils.dist(rna_arrays['early'], rna_arrays['late'])
coupling_costs['lineageOT, fitted tree'] = ot.utils.dist(rna_arrays['early'],
    ↳ancestor_info['fitted tree'][0])@np.diag(ancestor_info['fitted tree'][1]**(-1))

early_time_rna_cost = ot.utils.dist(rna_arrays['early'], sim_inf.extract_ancestor_
    ↳data_arrays(true_trees['late'], sample_times['early'], sim_params)[0])
late_time_rna_cost = ot.utils.dist(rna_arrays['late'], rna_arrays['late'])

```

Given the cost matrices, we can fit couplings with a range of entropy parameters.

```

epsilons = np.logspace(-2, 3, 15)

couplings['OT'] = ot.emd([], [], coupling_costs['OT'])
couplings['lineageOT'] = ot.emd([], [], coupling_costs['lineageOT, true tree'])
couplings['lineageOT, fitted'] = ot.emd([], [], coupling_costs['lineageOT, fitted tree
↪'])
for e in epsilons:
    if e >= 0.1:
        f = ot.sinkhorn
    else:
        # Epsilon scaling is more robust at smaller epsilon, but slower than simple_
↪sinkhorn
        f = ot.bregman.sinkhorn_epsilon_scaling
        couplings['entropic rna ' + str(e)] = f([], [], coupling_costs['OT'], e)
        couplings['lineage entropic rna ' + str(e)] = f([], [], coupling_costs['lineageOT,
↪ true tree'], e*np.mean(ancestor_info['true tree'][1]**(-1)))
        couplings['fitted lineage rna ' + str(e)] = f([], [], coupling_costs['lineageOT,
↪ fitted tree'], e*np.mean(ancestor_info['fitted tree'][1]**(-1)))

```

Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/lineageot/envs/develop/lib/python3.7/
↪site-packages/ot/bregman.py:1112: UserWarning: Sinkhorn did not converge. You might
↪want to increase the number of iterations `numItermax` or the regularization_
↪parameter `reg`.
    warnings.warn("Sinkhorn did not converge. You might want to "
/home/docs/checkouts/readthedocs.org/user_builds/lineageot/envs/develop/lib/python3.7/
↪site-packages/ot/bregman.py:517: UserWarning: Sinkhorn did not converge. You might
↪want to increase the number of iterations `numItermax` or the regularization_
↪parameter `reg`.
    warnings.warn("Sinkhorn did not converge. You might want to "

```

## 2.4.4 Evaluation of couplings

First compute the independent coupling as a reference

```

couplings['independent'] = np.ones(couplings['OT'].shape)/couplings['OT'].size
ind_ancestor_error = sim_inf.OT_cost(couplings['independent'], early_time_rna_cost)
ind_descendant_error = sim_inf.OT_cost(sim_eval.expand_coupling(couplings['independent
↪'],
                                                                    couplings['true'],
                                                                    late_time_rna_cost),
                                                                    late_time_rna_cost)

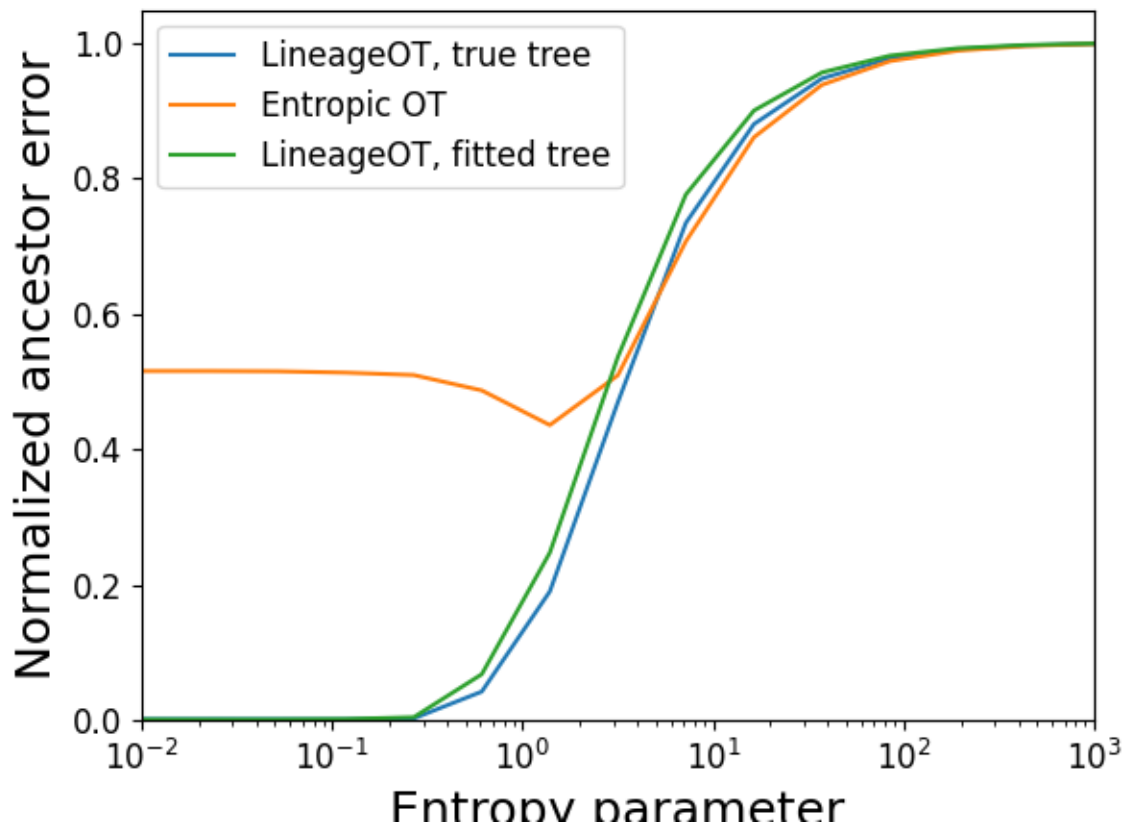
```

Plotting the accuracy of ancestor prediction

```

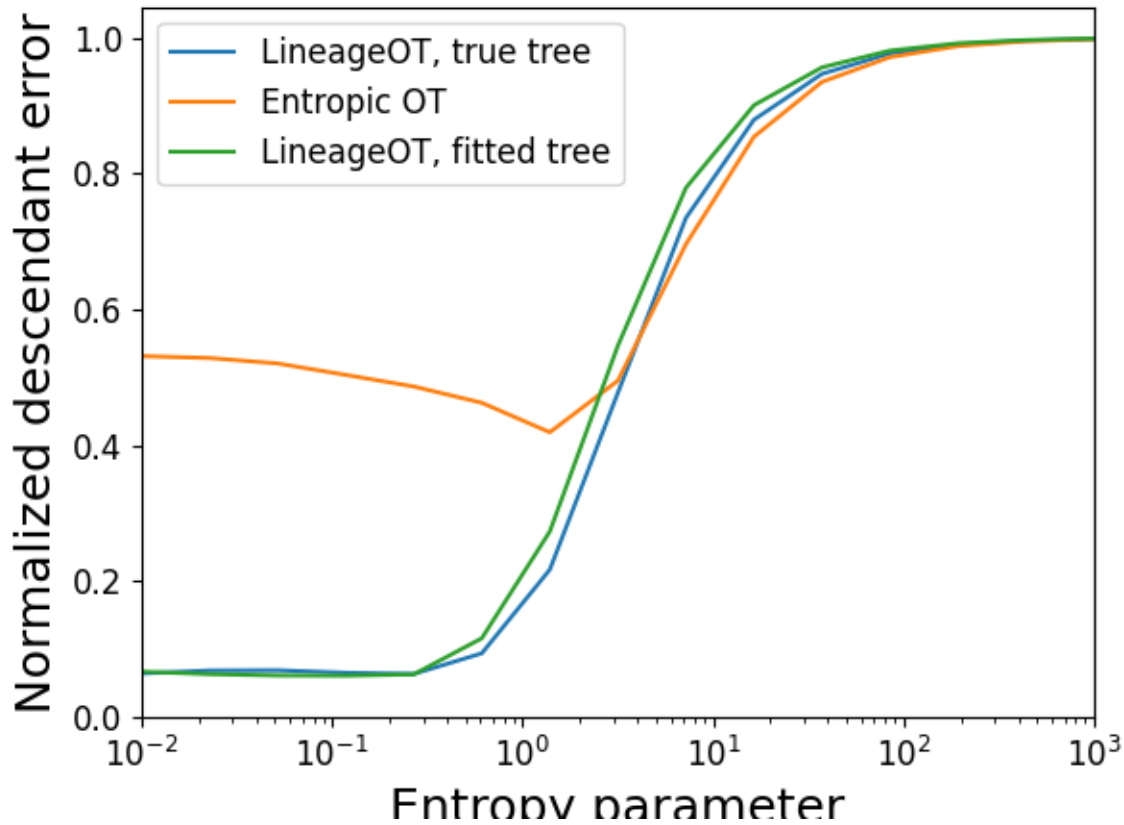
ancestor_errors = sim_eval.plot_metrics(couplings,
                                        lambda x: sim_inf.OT_cost(x, early_time_rna_
↪cost),
                                        'Normalized ancestor error',
                                        epsilons,
                                        scale = ind_ancestor_error,
                                        points=False)

```



Plotting the accuracy of descendant prediction

```
descendant_errors = sim_eval.plot_metrics(couplings,
                                         lambda x:sim_inf.OT_cost(sim_eval.expand_
↳ coupling(x,
↳ couplings['true'],
↳ late_time_rna_cost),
late_time_rna_
↳ cost),
'Normalized descendant error',
epsilons, scale = ind_descendant_error)
```



### 2.4.5 Coupling visualizations

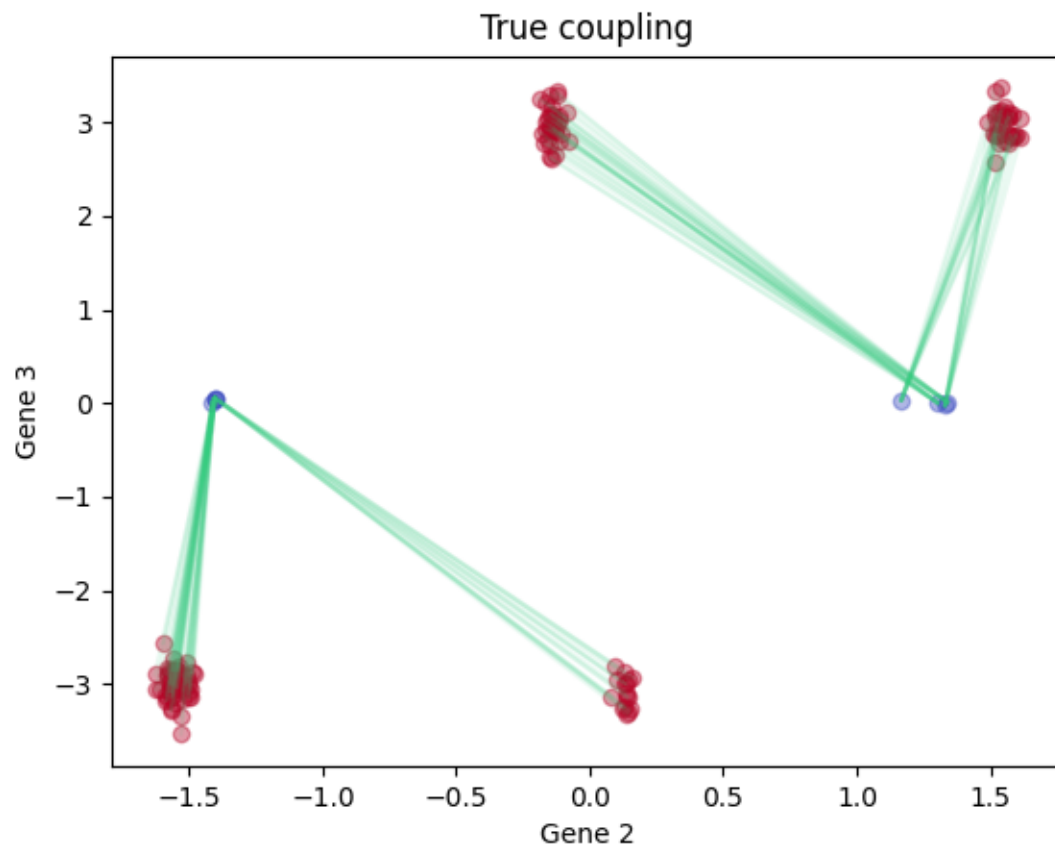
Visualizing the ground-truth coupling, zero-entropy LineageOT coupling, and zero-entropy optimal transport coupling.

Ground truth:

```
sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
→to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
→plot[1]]],
                           couplings['true'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)

plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('True coupling')

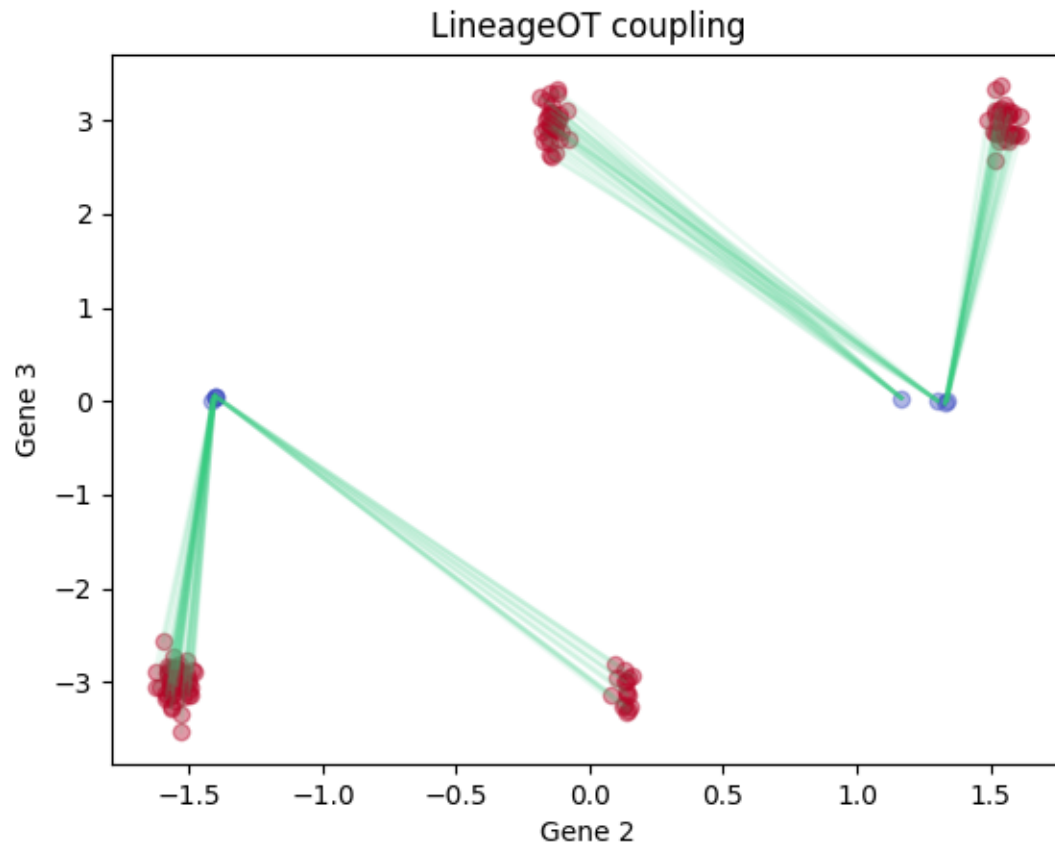
for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
→'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
→ label = label, color = c)
```



LineageOT:

```
sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
→to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
→plot[1]]],
                           couplings['lineageOT'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('LineageOT coupling')

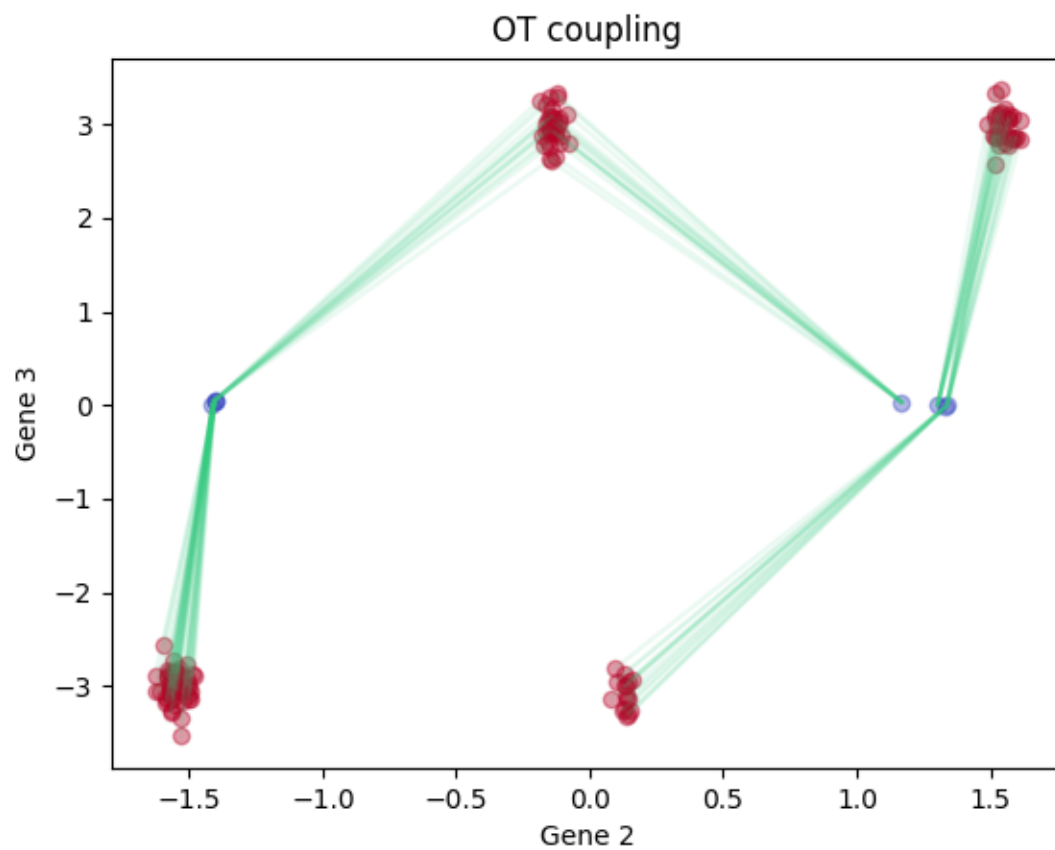
for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
→'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
→ label = label, color = c)
```



#### Optimal transport

```
sim_eval.plot2D_samples_mat(rna_arrays['early'][:, [dimensions_to_plot[0], dimensions_
→to_plot[1]]],
                           rna_arrays['late'][:, [dimensions_to_plot[0], dimensions_to_
→plot[1]]],
                           couplings['OT'],
                           c=[0.2, 0.8, 0.5],
                           alpha_scale = 0.1)
plt.xlabel('Gene ' + str(dimensions_to_plot[0] + 1))
plt.ylabel('Gene ' + str(dimensions_to_plot[1] + 1))
plt.title('OT coupling')

for a, label, c in zip([rna_arrays['early'], rna_arrays['late']], ['Early cells',
→ 'Late cells'], colors):
    plt.scatter(a[:, dimensions_to_plot[0]], a[:, dimensions_to_plot[1]], alpha = 0.4,
→ label = label, color = c)
```



**Total running time of the script:** ( 0 minutes 9.610 seconds)



## CORE PIPELINE

```
lineageot.core.fit_lineage_coupling(adata, time_1, time_2, lineage_tree_t2, time_key='time',  
                                   state_key=None, epsilon=0.05, normalize_cost=True,  
                                   ot_method='sinkhorn', marginal_1=[], marginal_2=[],  
                                   balance_reg=inf)
```

Fits a LineageOT coupling between the cells in *adata* at *time\_1* and *time\_2*. In the process, annotates the lineage tree with observed and estimated cell states.

### Parameters

- **adata** (*AnnData*) – Annotated data matrix
- **time\_1** (*Number*) – The earlier time point in *adata*. All times are relative to the root of the tree.
- **time\_2** (*Number*) – The later time point in *adata*. All times are relative to the root of the tree.
- **lineage\_tree\_t2** (*Networkx DiGraph*) – The lineage tree fitted to cells at *time\_2*. Nodes should already be annotated with times. Annotations related to cell state will be added.
- **time\_key** (*str* (default 'time')) – Key in *adata.obs* and *lineage\_tree\_t2* containing cells' time labels
- **state\_key** (*str* (default None)) – Key in *adata.obsm* containing cell states. If None, uses *adata.X*.
- **epsilon** (*float* (default 0.05)) – Entropic regularization parameter for optimal transport
- **normalize\_cost** (*bool* (default True)) – Whether to rescale the cost matrix by its median before fitting a coupling. Normalizing this way allows us to choose a reasonable default epsilon for data of any scale
- **ot\_method** (*str* (default 'sinkhorn')) – Method used for the optimal transport solver. Choose from 'sinkhorn', 'greenkhorn', 'sinkhorn\_stabilized' and 'sinkhorn\_epsilon\_scaling' for balanced transport and 'sinkhorn', 'sinkhorn\_stabilized', and 'sinkhorn\_reg\_scaling' for unbalanced transport. 'sinkhorn' is recommended unless you encounter numerical problems. See PythonOT docs for more details.
- **marginal\_1** (*Vector* (default [])) – Marginal distribution (relative growth rates) for cells at time 1. If empty, assumed uniform.
- **marginal\_2** (*Vector* (default [])) – Marginal distribution (relative growth rates) for cells at time 2. If empty, assumed uniform.

- **balance\_reg** (*Number*) – Regularization parameter for unbalanced transport. Smaller values allow more flexibility in growth rates. If infinite, marginals are treated as hard constraints.

**Returns coupling** – AnnData containing the lineage coupling. Cells from time\_1 are in coupling.obs, cells from time\_2 are in coupling.var, and the coupling matrix is coupling.X

**Return type** AnnData

```
lineageot.core.fit_tree(adata, time, barcodes_key='barcodes', clones_key='X_clone',
                        clone_times=None, method='neighbor join')
```

Fits a lineage tree to lineage barcodes of all cells in adata. To compute the lineage tree for a specific time point, filter adata before calling fit\_tree. The fitted tree is annotated with node times but not states.

#### Parameters

- **adata** (*AnnData*) – Annotated data matrix with lineage-traced cells
- **time** (*Number*) – Time of sampling of the cells of adata relative to most recent common ancestor (for dynamic lineage tracing) or labeling time (for static lineage tracing).
- **barcodes\_key** (*str*, *default* 'barcodes') – Key in adata.obsm containing cell barcodes. Ignored if using clonal data. If using barcode data, each row of adata.obsm[barcodes\_key] should be a barcode where each entry corresponds to a possibly-mutated site. A positive number indicates an observed mutation, zero indicates no mutation, and -1 indicates the site was not observed.
- **clones\_key** (*str*, *default* 'X\_clone') – Key in adata.obsm containing clonal data. Ignored if using barcodes directly. If using clonal data, adata.obsm[clones\_key] should be a num\_cells x num\_clones boolean matrix. Each entry is 1 if the corresponding cell belongs to the corresponding clone and zero otherwise.
- **clone\_times** (*Vector of length num\_clones*, *default* None) – Ignored unless method is 'clones'. Each entry contains the time of labeling of the corresponding column of adata.obsm[clones\_key].
- **method** (*str*) – Inference method used to fit tree. Current options are 'neighbor join' (for barcodes from dynamic lineage tracing), 'non-nested clones' (for non-nested clones from static lineage tracing), or 'clones' (for possibly-nested clones from static lineage tracing).

**Returns tree** – A fitted lineage tree. Each node is annotated with 'time\_to\_parent' and 'time' (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and are annotated with 'time' equal to the child node's 'time\_to\_parent'. Observed node indices correspond to their row in adata.

**Return type** Networkx DiGraph

```
lineageot.core.read_newick(filename, leaf_labels, leaf_time=None)
```

Loads a tree saved in Newick format and adds annotations required for LineageOT.

#### Parameters

- **filename** (*str*) – The name of the file to load from.
- **leaf\_labels** (*list*) – The label of each leaf in the Newick tree, sorted to align with the gene expression AnnData object filtered to cells at the corresponding time.
- **leaf\_time** (*float* (*default* None)) – The time of sampling of the leaves. If unspecified, the root of the tree is assigned time 0.

**Returns tree** – The saved tree, in LineageOT's format. Each node is annotated with 'time\_to\_parent' and 'time' (which indicates either the time of sampling (for observed cells) or the time of division (for unobserved ancestors)). Edges are directed from parent to child and

are annotated with 'time' equal to the child node's 'time\_to\_parent'. Observed node indices correspond to their index in leaf\_labels, which should match their row in the gene expression AnnData object filtered to cells at the corresponding time.

**Return type** Networkx DiGraph

`lineageot.core.save_coupling_as_tmap(coupling, time_1, time_2, tmap_out)`

Saves a LineageOT coupling for downstream analysis with Waddington-OT. A sequence of saved couplings can be loaded in wot with `wot.tmap.TransportMapModel.from_directory(tmap_out)`

**Parameters**

- **coupling** (*AnnData*) – The coupling to save.
- **time\_1** (*Number*) – The earlier time point in adata. All times are relative to the root of the tree.
- **time\_2** (*Number*) – The later time point in adata. All times are relative to the root of the tree.
- **tmap\_out** (*str*) – The path and prefix to the save file name.



## PYTHON MODULE INDEX

### I

`lineageot.core`, 3  
`lineageot.evaluation`, 13  
`lineageot.inference`, 8  
`lineageot.simulation`, 5



## A

[add\\_conditional\\_means\\_and\\_variances\(\)](#) (in module *lineageot.inference*), 8  
[add\\_division\\_times\\_from\\_vertex\\_times\(\)](#) (in module *lineageot.inference*), 8  
[add\\_inverse\\_times\\_to\\_edges\(\)](#) (in module *lineageot.inference*), 8  
[add\\_leaf\\_barcodes\(\)](#) (in module *lineageot.inference*), 8  
[add\\_leaf\\_times\(\)](#) (in module *lineageot.inference*), 8  
[add\\_leaf\\_x\(\)](#) (in module *lineageot.inference*), 8  
[add\\_node\\_times\\_from\\_dict\(\)](#) (in module *lineageot.inference*), 8  
[add\\_node\\_times\\_from\\_division\\_times\(\)](#) (in module *lineageot.inference*), 8  
[add\\_nodes\\_at\\_time\(\)](#) (in module *lineageot.inference*), 8  
[add\\_samples\\_to\\_clone\\_tree\(\)](#) (in module *lineageot.inference*), 9  
[add\\_times\(\)](#) (in module *lineageot.inference*), 9  
[add\\_times\\_to\\_edges\(\)](#) (in module *lineageot.inference*), 9  
[annotate\\_tree\(\)](#) (in module *lineageot.inference*), 9

## B

[barcode\\_distances\(\)](#) (in module *lineageot.inference*), 9

## C

[Cell](#) (class in *lineageot.simulation*), 5  
[center\(\)](#) (in module *lineageot.simulation*), 6  
[compute\\_leaf\\_times\(\)](#) (in module *lineageot.inference*), 9  
[compute\\_new\\_distances\(\)](#) (in module *lineageot.inference*), 9  
[compute\\_q\\_matrix\(\)](#) (in module *lineageot.inference*), 9  
[compute\\_tree\\_distances\(\)](#) (in module *lineageot.inference*), 9  
[convergent\\_flow\(\)](#) (in module *lineageot.simulation*), 6

[convert\\_data\\_to\\_arrays\(\)](#) (in module *lineageot.simulation*), 6  
[convert\\_newick\\_to\\_networkx\(\)](#) (in module *lineageot.inference*), 9  
[coupling\\_to\\_coupling\\_cost\\_matrix\(\)](#) (in module *lineageot.evaluation*), 13  
[coupling\\_W2\(\)](#) (in module *lineageot.evaluation*), 13  
[cvxopt\\_qp\\_from\\_numpy\(\)](#) (in module *lineageot.inference*), 10

## D

[deepcopy\(\)](#) (*lineageot.simulation.Cell* method), 5  
[distances\\_to\\_joined\\_node\(\)](#) (in module *lineageot.inference*), 10

## E

[estimate\\_division\\_time\(\)](#) (in module *lineageot.inference*), 10  
[evolve\\_b\(\)](#) (in module *lineageot.simulation*), 6  
[evolve\\_cell\(\)](#) (in module *lineageot.simulation*), 6  
[evolve\\_x\(\)](#) (in module *lineageot.simulation*), 6  
[expand\\_coupling\(\)](#) (in module *lineageot.evaluation*), 14  
[expand\\_coupling\\_independent\(\)](#) (in module *lineageot.evaluation*), 14  
[extract\\_ancestor\\_data\\_arrays\(\)](#) (in module *lineageot.inference*), 10  
[extract\\_data\\_arrays\(\)](#) (in module *lineageot.inference*), 10

## F

[find\\_parent\\_clone\(\)](#) (in module *lineageot.inference*), 10  
[fit\\_lineage\\_coupling\(\)](#) (in module *lineageot.core*), 3  
[fit\\_tree\(\)](#) (in module *lineageot.core*), 4  
[flatten\\_list\\_of\\_lists\(\)](#) (in module *lineageot.simulation*), 7

## G

[get\\_ancestor\\_data\(\)](#) (in module *lineageot.inference*), 11

`get_components()` (in module *lineageot.inference*), 11  
`get_internal_nodes()` (in module *lineageot.inference*), 11  
`get_leaf_descendants()` (in module *lineageot.inference*), 11  
`get_leaves()` (in module *lineageot.inference*), 11  
`get_lineage_distances_across_time()` (in module *lineageot.inference*), 11  
`get_parent_clone_of_leaf()` (in module *lineageot.inference*), 11  
`get_true_coupling()` (in module *lineageot.inference*), 11

## J

`join_nodes()` (in module *lineageot.inference*), 11

## L

`l2_difference()` (in module *lineageot.evaluation*), 14  
`lineageot.core` module, 3  
`lineageot.evaluation` module, 13  
`lineageot.inference` module, 8  
`lineageot.simulation` module, 5  
`list_tree_to_digraph()` (in module *lineageot.inference*), 11

## M

`make_clone_reference_tree()` (in module *lineageot.inference*), 11  
`make_tree_from_clones()` (in module *lineageot.inference*), 12  
`make_tree_from_nonnested_clones()` (in module *lineageot.inference*), 12  
`mask_barcode()` (in module *lineageot.simulation*), 7  
`mismatched_clusters_flow()` (in module *lineageot.simulation*), 7  
`module`  
     `lineageot.core`, 3  
     `lineageot.evaluation`, 13  
     `lineageot.inference`, 8  
     `lineageot.simulation`, 5  
`mutate_barcode()` (in module *lineageot.simulation*), 7

## N

`neighbor_join()` (in module *lineageot.inference*), 12  
`NeighborJoinNode` (class in *lineageot.inference*), 8

`normalize_columns()` (in module *lineageot.evaluation*), 14

## O

`OT_cost()` (in module *lineageot.inference*), 8

## P

`pairwise_squared_distances()` (in module *lineageot.evaluation*), 14  
`partial_convergent_flow()` (in module *lineageot.simulation*), 7  
`pick_joined_nodes()` (in module *lineageot.inference*), 12  
`plot2D_samples_mat()` (in module *lineageot.evaluation*), 14  
`plot_metrics()` (in module *lineageot.evaluation*), 14  
`print_metrics()` (in module *lineageot.evaluation*), 14

## R

`rate_estimator()` (in module *lineageot.inference*), 12  
`read_newick()` (in module *lineageot.core*), 4  
`recursive_add_barcodes()` (in module *lineageot.inference*), 12  
`recursive_list_tree_to_digraph()` (in module *lineageot.inference*), 13  
`remove_node_and_descendants()` (in module *lineageot.inference*), 13  
`remove_times()` (in module *lineageot.inference*), 13  
`reproducible_poisson()` (in module *lineageot.simulation*), 7  
`resample_cells()` (in module *lineageot.inference*), 13  
`reset_seed()` (*lineageot.simulation.Cell* method), 5  
`robinson_foulds()` (in module *lineageot.inference*), 13

## S

`sample_barcode()` (in module *lineageot.simulation*), 7  
`sample_cell()` (in module *lineageot.simulation*), 7  
`sample_coordinates_from_coupling()` (in module *lineageot.evaluation*), 14  
`sample_descendants()` (in module *lineageot.simulation*), 7  
`sample_division_time()` (in module *lineageot.simulation*), 7  
`sample_indices_from_coupling()` (in module *lineageot.evaluation*), 15  
`sample_interpolant()` (in module *lineageot.evaluation*), 15  
`sample_pop()` (in module *lineageot.simulation*), 7



[sample\\_population\\_descendants\(\)](#) (in module *lineageot.simulation*), 7  
[sample\\_x0\(\)](#) (in module *lineageot.simulation*), 7  
[save\\_coupling\\_as\\_tmap\(\)](#) (in module *lineageot.core*), 5  
[scaled\\_Hamming\\_distance\(\)](#) (in module *lineageot.inference*), 13  
[scaled\\_l2\\_difference\(\)](#) (in module *lineageot.evaluation*), 15  
[SimulationParameters](#) (class in *lineageot.simulation*), 5  
[single\\_bifurcation\\_flow\(\)](#) (in module *lineageot.simulation*), 7  
[split\\_edge\(\)](#) (in module *lineageot.inference*), 13  
[split\\_targets\\_between\\_daughters\(\)](#) (in module *lineageot.simulation*), 7  
[squeeze\\_coupling\(\)](#) (in module *lineageot.evaluation*), 15  
[squeeze\\_coupling\\_by\\_late\\_cluster\(\)](#) (in module *lineageot.evaluation*), 15  
[subsample\\_list\(\)](#) (in module *lineageot.simulation*), 7  
[subsample\\_pop\(\)](#) (in module *lineageot.simulation*), 8  
[subtree\\_to\\_ete3\(\)](#) (in module *lineageot.inference*), 13

## T

[tree\\_accuracy\(\)](#) (in module *lineageot.inference*), 13  
[tree\\_discrepancy\(\)](#) (in module *lineageot.inference*), 13  
[tree\\_to\\_ete3\(\)](#) (in module *lineageot.inference*), 13  
[truncate\\_tree\(\)](#) (in module *lineageot.inference*), 13  
[tv\(\)](#) (in module *lineageot.evaluation*), 15

## V

[vector\\_field\(\)](#) (in module *lineageot.simulation*), 8