

Algorithmic Flow: MNN Detection & Alignment Scoring

Two strategies for mutual nearest neighbor detection (MNN), dataset indexing (FILL_TABLE), and alignment scoring

Original Scanorama

Set-based MNN + IntervalTree

1. Nearest neighbor search (nn_approx):

1. Build new AnnoyIndex every call

2. Query each cell; collect indices

3. Return Python set of (a, b_i) tuples

Index rebuilt on each invocation;
set-of-tuples has high Python overhead

2. Mutual nearest neighbors (mnn):

1. $\text{match1} \leftarrow \text{nn_approx}(X_1, X_2)$ (set)

2. $\text{match2} \leftarrow \text{nn_approx}(X_2, X_1)$ (set)

3. $\text{mutual} \leftarrow \text{match1} \cap \{(b, a) : (a, b) \in \text{match2}\}$

Python set intersection over $O(Nk)$ tuples

3. Dataset indexing (fill_table):

1. Build IntervalTree for dataset boundaries

2. For each (d, r) pair: query tree to find which dataset r belongs to

3. Store in per- (i, j) Python set

IntervalTree: $O(N \log N)$ build + query overhead

4. Alignment score:

$$\text{score}(i, j) = \max\left(\frac{|\text{unique_ds}|}{N_i}, \frac{|\text{unique_ref}|}{N_j}\right)$$

Coverage-only; no distance/bias quality signal

scanorama_optimized

CSR-based MNN + searchsorted

1. Nearest neighbor search (nn_approx):

1. `_annoy_cache`: reuse built index

Key = (id, data_ptr, shape, metric, n_trees)

2. Query each cell; collect indices

3. Return dense numpy array (`return_ind=True`)

Annoy index built once, cached;
dense array avoids per-tuple Python overhead

2. Mutual nearest neighbors (mnn):

1. $\text{ind12} \leftarrow \text{nn_approx}(X_1, X_2)$ (array)

2. $\text{ind21} \leftarrow \text{nn_approx}(X_2, X_1)$ (array)

3. Build CSR adjacency $\text{adj21} \in \mathbb{R}^{N_2 \times N_1}$

4. $\text{mask} \leftarrow \text{adj21}[b_{\text{flat}}, a_{\text{rep}}].A1$

Sparse matrix lookup: $O(\text{nnz})$ vs set hash

3. Dataset indexing (fill_table):

1. `np.cumsum(sizes) + searchsorted`

2. Vectorized $(d, r) \rightarrow$ dataset-id mapping

3. Per- (i, j) set (same downstream compat.)

$O(N \log B)$ searchsorted vs $O(N \log N)$ tree

4. Alignment score (enriched):

1. Coverage: same as original

2. Tightness: $\exp(-\text{med_dist}/t)$

3. Consistency: $\exp(-\text{bias_var}/u)$

4. $\text{score} = \text{cov} \times \text{tight} \times \text{consist}$

Quality-aware edge weights for MSF

Algorithmic Flow: Batch Correction Assembly (assemble)

Two strategies for merging datasets: pairwise panorama merging vs. global symmetric iterative optimization

Original Scanorama

Greedy pairwise panorama merge

1. Find alignments; sort by score (coverage)

2. For each alignment (i, j) :

- Find panorama membership of i, j
- If i not in panorama: correct $i \rightarrow \text{pan}(j)$
- If j not in panorama: correct $j \rightarrow \text{pan}(i)$
- If both in panoramas: correct $\text{pan}(i) \rightarrow \text{pan}(j)$
- Merge panoramas

Sequential, order-dependent;
each pair corrected independently

3. Correction (`transform` \rightarrow `batch_bias`):

- $\text{bias} \leftarrow X_{\text{ref}}[\text{matched}] - X_{\text{ds}}[\text{matched}]$
- $W \leftarrow \text{rbf_kernel}(X_{\text{all}}, X_{\text{matched}})$
Full $N \times M$ dense kernel matrix
- $W \leftarrow \text{normalize}(W, \mathbf{1})$
- $X_{\text{ds}} \leftarrow X_{\text{ds}} + W \cdot \text{bias}$

Dense $O(NM)$ RBF; no robust weighting;
no control for overcorrection

4. Expression correction:

Same `transform` on expression matrices;
dense RBF on full gene-expression space

Greedy order \rightarrow inconsistent corrections;
dense kernel \rightarrow memory-heavy;
no global optimization

scanorama_optimized

MSF + global iterative optimization

1. Maximum Spanning Forest (MSF):

- Sort edges by quality-aware score \downarrow
- Union-Find: add edge if it connects two disjoint components
- Result: tree of $D-1$ edges (no cycles)
Avoids conflicting correction paths

2. Precompute edge cache:

For each MSF edge (d, e) :

- Anchor points: $A_d = X_0[d][\text{matched}]$
- Precompute kNN: all cells \rightarrow anchors
- Gaussian weights $g = e^{-d^2/2\sigma^2}$
- CSR sparsity pattern for smoother S
Avoids recomputation across iterations

3. Global iterative optimization ($T=5$):

- $X_{\text{corr}} \leftarrow X_0 + f$ (apply current field)
- For each MSF edge (d, e) :
 - Symmetric residuals: R_d, R_e
 - Huber weights from median $\|R\|$
 - Smooth via `batch_bias` (CSR path)
 - Clip at 95th percentile
- $f_d \leftarrow (1-\eta\lambda)f_d + \eta \cdot \text{step}$
 $\eta=0.5, \lambda=0.05$ (L2 shrink)
- Laplacian smooth: $f_d \leftarrow (1-\mu)f_d + \mu \bar{f}_{\text{nbr}}$
 $\mu=0.10$; preserves local structure

4. Expression correction:

Single pass on matched anchors only;
sparse multiply (no dense `toarray`)

Detailed Pseudocode Comparison

Algorithmic differences in MNN detection (MNN) and batch correction assembly (ASSEMBLE)

1. Mutual Nearest Neighbor Detection (mnn)

Algorithm 1a: Original Scanorama — Set-based MNN with IntervalTree

Input: Datasets $X_1 \in \mathbb{R}^{N_1 \times d}$, $X_2 \in \mathbb{R}^{N_2 \times d}$, k neighbors

Output: Set of mutual nearest neighbor pairs $\{(a, b)\}$

```
1: Build AnnoyIndex on  $X_2$  (rebuild every call) ▷ No caching
2: for  $i = 1, \dots, N_1$  do
3:   Query  $k$  nearest neighbors of  $X_1[i]$  in  $X_2$ 
4:   for each neighbor  $b_j$  do
5:     match1.ADD( $(i, b_j)$ ) ▷ Python set of tuples
6:   end for
7: end for
8: Repeat: build AnnoyIndex on  $X_1$ , query  $X_2 \rightarrow$  match2
9: mutual  $\leftarrow$  match1  $\cap \{(b, a) : (a, b) \in \text{match2}\}$  ▷ Set intersection over  $O(Nk)$  tuples
10: Dataset indexing: Build IntervalTree for dataset boundaries; query per  $(d, r)$  pair
```

Algorithm 1b: scanorama_optimized — CSR-based MNN with Annoy Caching

Input: Datasets $X_1 \in \mathbb{R}^{N_1 \times d}$, $X_2 \in \mathbb{R}^{N_2 \times d}$, k neighbors

Output: Set of mutual nearest neighbor pairs $\{(a, b)\}$

```
1: Lookup AnnoyIndex in _annoy_cache; build only if miss ▷ Index reuse
2: ind12  $\leftarrow$  QUERYALL( $X_1, X_2, k$ ) ▷ Dense  $N_1 \times k$  array
3: ind21  $\leftarrow$  QUERYALL( $X_2, X_1, k$ ) ▷ Dense  $N_2 \times k$  array
4: Build CSR adjacency: adj21  $\in \{0, 1\}^{N_2 \times N_1}$  from ind21 ▷ Sparse membership
5:  $a_{\text{rep}} \leftarrow \text{np.repeat}(\text{arange}(N_1), k)$ ;  $b_{\text{flat}} \leftarrow \text{ind12.reshape}(-1)$ 
6: mask  $\leftarrow \text{adj21}[b_{\text{flat}}, a_{\text{rep}}].\text{A1.astype}(\text{bool})$  ▷ Vectorized mutual test
7: mutual  $\leftarrow \{(a_{\text{rep}}[m], b_{\text{flat}}[m]) : m \in \text{mask}\}$ 
8: Dataset indexing: np.searchsorted(cumsum(sizes), r) ▷ Replaces IntervalTree
```

2. Batch Correction Assembly (assemble)

Algorithm 2a: Original Scanorama — Greedy Pairwise Panorama Merging

Input: Datasets $\{X_d\}_{d=1}^D$, alignments sorted by coverage, MNN matches

Output: Corrected datasets

```
1: for each alignment  $(i, j)$  in descending score order do
2:   Determine panorama membership; collect MNN pairs  $(a, b)$ 
3:   bias  $\leftarrow X_{\text{ref}}[\text{ref\_ind}] - X_{\text{ds}}[\text{ds\_ind}]$  ▷ Raw correction vectors
4:    $W \leftarrow \text{RBF\_KERNEL}(X_{\text{all}}, X_{\text{matched}}, \gamma = \sigma/2)$  ▷ Dense  $N \times M$  matrix
5:    $W \leftarrow W / \sum_j W$  ▷ L1-normalize rows
6:    $X_{\text{ds}} \leftarrow X_{\text{ds}} + W \cdot \text{bias}$  ▷ Apply smoothed correction
7:   Merge panoramas
8: end for
```

Algorithm 2b: scanorama_optimized — MSF + Global Symmetric Iterative Optimization**Input:** Datasets $\{X_d\}_{d=1}^D$, MNN matches, edge scores**Output:** Corrected datasets

```

1: — Phase 1: Maximum Spanning Forest —
2: Sort edges by score (coverage  $\times$  tightness  $\times$  consistency) descending
3: Union-Find MSF: greedily add edges connecting disjoint components ▷ No redundant paths
4: — Phase 2: Precompute edge cache —
5: for each MSF edge  $(d, e)$  do
6:    $A_d \leftarrow X_0[d][\text{ds\_ind}]; \quad A_e \leftarrow X_0[e][\text{ref\_ind}]$  ▷ Anchor points
7:   Fit kNN on anchors; query all cells  $\rightarrow$  neighbor indices + distances
8:    $g \leftarrow \exp(-d^2/(2\sigma_i^2))$  where  $\sigma_i = \text{median}(d_i)$  ▷ Precomputed Gaussian
9:   Build CSR pattern:  $S \in \mathbb{R}^{N_d \times M}$  (indices, indptr cached)
10: end for
11: — Phase 3: Global iterative optimization —
12: Initialize correction fields  $f_d \leftarrow \mathbf{0}$  for each dataset  $d$ 
13: for  $t = 1, \dots, T$  do ▷  $T = 5$  outer iterations
14:    $X_{\text{corr}}^{(d)} \leftarrow X_0^{(d)} + f_d$  for all  $d$  ▷ Current corrected coordinates
15:   for each MSF edge  $(d, e)$  with weight  $w_e$  do
16:      $R_d \leftarrow X_{\text{corr}}^{(e)}[\text{ref}] - X_{\text{corr}}^{(d)}[\text{ds}]$  ▷ Symmetric residuals
17:      $R_e \leftarrow -R_d$ 
18:      $\delta \leftarrow \max(0.25, 2 \cdot \text{median}(\|R_d\|))$  ▷ Adaptive Huber threshold
19:      $w_d^{(\text{rob})} \leftarrow \text{HUBERWEIGHTS}(\|R_d\|^2, \delta)$  ▷  $w = \min(1, \delta/\|R\|)$ 
20:      $\text{upd}_d \leftarrow \text{BATCHBIAS}(X_0^{(d)}, R_d; \text{weights} = w_d^{(\text{rob})}, \text{CSR cache})$  ▷ Precomputed kNN + CSR
21:     Clip:  $\|\text{upd}_d\| \leftarrow \min(\|\text{upd}_d\|, q_{95}(\|\text{upd}_d\|))$  ▷ 95th percentile cap
22:      $\Delta f_d += w_e \cdot \text{upd}_d; \quad \Delta f_e += w_e \cdot \text{upd}_e$ 
23:   end for
24:    $\text{step}_d \leftarrow \Delta f_d / \sum w_e$  for each  $d$ 
25:    $f_d \leftarrow (1 - \eta\lambda) f_d + \eta \text{step}_d$  ▷  $\eta=0.5$  damping,  $\lambda=0.05$  L2 shrink
26:    $f_d \leftarrow (1 - \mu) f_d + \mu \text{mean}(f_d[\text{kNN}])$  ▷  $\mu=0.10$  Laplacian smooth
27: end for
28:  $X_d \leftarrow X_0^{(d)} + f_d$  for all  $d$  ▷ Apply final corrections
29: — Phase 4: Expression-space correction —
30: Single pass on matched anchors only; sparse multiply ▷ No dense toarray

```

3. Summary of Key Algorithmic Differences

Component	Original	scanorama_optimized
MNN detection	Python set of tuples; set intersection	CSR adjacency matrix; vectorized sparse lookup <code>adj21[b,a].A1</code>
ANN index	Rebuilt every call	Cached in <code>_annoy_cache</code> ; reused across calls
Dataset indexing	<code>IntervalTree</code> per <code>fill_table</code> call	<code>np.searchsorted</code> on prefix sums (vectorized)
Alignment score	Coverage only	Coverage \times tightness \times consistency (quality-aware)
Merge strategy	Greedy pairwise panorama merging (order-dependent)	Maximum Spanning Forest (MSF) via <code>union-find</code> ; cycle-free
Optimization	Single-pass pairwise correction	$T=5$ global symmetric iterations over all MSF edges simultaneously
Robust weighting	None (all anchors weighted equally)	Huber IRLS: adaptive δ from median $\ R\ $; downweights outliers
Correction control	None	Update clipping (95th pctile), damping $\eta=0.5$, L2 shrink $\lambda=0.05$
Structure preservation	None	Laplacian smoothing ($\mu=0.10$) on correction field via kNN graph
Correction smoothing	Dense RBF kernel ($N \times M$)	Precomputed kNN + Gaussian weights + CSR smoother
Expression correction	Full dense correction via <code>transform</code>	Single pass on matched anchors only; sparse multiply

4. Performance Comparison

Metric	Original	scanorama_optimized	Δ
Batch mixing score	0.5440	0.7542	+38.6%
Bio conservation score	0.6736	0.7313	+8.6%
Speed score	0.6579	0.7555	+14.8%
Execution time	0.520s	0.324s	$\times 1.6$
Evolution generation	0	6	—
Evolution order	0	235	—

5. Why scanorama_optimized Improved

scanorama_optimized: Comprehensive improvement via global optimization + robust weighting

- Batch mixing (+38.6%)**: The original greedy pairwise merging applies corrections sequentially, which can create inconsistent corrections when datasets are corrected toward different references. The MSF-based global optimization corrects *all* datasets simultaneously over $T=5$ iterations with symmetric residuals ($R_d = X_e - X_d$ and $R_e = X_d - X_e$), ensuring consistent convergence. The quality-aware alignment scores (coverage \times tightness \times consistency) select more reliable MNN edges as the MSF backbone.
- Bio conservation (+8.6%)**: Multiple mechanisms prevent overcorrection:
 - Huber IRLS weights** downweight outlier MNN pairs (adaptive δ from median $\|R\|$), preventing false matches from distorting corrections.
 - Update clipping** at the 95th percentile prevents extreme per-cell corrections.
 - Laplacian smoothing** ($\mu=0.10$) on the correction field via precomputed kNN graphs preserves within-dataset neighborhood structure.
 - L2 shrinkage** ($\lambda=0.05$) and **damping** ($\eta=0.5$) prevent oscillation and limit correction magnitude.
- Speed ($\times 1.6$)**: Annoy index caching eliminates redundant index construction. CSR-based MNN detection replaces Python set operations with vectorized sparse matrix lookups. `searchsorted` replaces `IntervalTree` for dataset boundary queries. Precomputed per-edge kNN, Gaussian weights, and CSR sparsity patterns avoid recomputation across the 5 iterations. Expression-space correction operates on anchors only (sparse multiply) rather than converting full matrices to dense.