vDNAmic data pipeline

WLab at UChicago

October 2024

Contents

1 Sequence Analysis

1.1 Sequence Analysis Pipeline

The `lib` command is a critical component of the DNA microscopy data analysis pipeline. It processes raw sequencing data obtained from DNA microscopy experiments, performing tasks such as quality filtering, sequence parsing, UMI/UEI extraction, clustering, and initial data organization. The outputs generated are essential for downstream analyses, including spatial reconstruction and gene expression profiling.

1.1.1 Workflow Context

The processing steps facilitated by the `lib` command include:

- Quality control and filtering of raw reads.
- Extraction of Unique Molecular Identifiers (UMIs) and Unique Event Identifiers (UEIs).
- Clustering of UMIs and UEIs to reduce sequencing errors.
- Assembly of consensus sequences for cDNA amplicons.
- Optional alignment of sequences to a reference genome.
- Generation of subsampled datasets for rarefaction analysis.

1.1.2 Subsequent Steps

The outputs from the `lib` command serve as inputs for further analysis steps, such as the `GSE` (Geodesic Spectral Embedding) command, which performs spatial reconstruction based on the processed data.

1.1.3 Prerequisites

Software Dependencies Ensure the following software and packages are installed:

- **Python 3.8+** with packages:
  - `numpy`
  - `pandas`
  - `Biopython`
- **STAR Aligner** version 2.7.9a or later.
- **awk**, **bioawk**, and **sed** utilities.

Hardware Requirements Processing DNA microscopy data can be resource-intensive. Recommended hardware includes:

- Multi-core CPU (4 cores or more).
- At least 16 GB of RAM.
- Sufficient storage space (depending on dataset size; at least 100 GB recommended).

Environmental Setup

- Install dependencies:

```
pip install numpy pandas biopython
conda install bioconda::bioawk
```

Setting Up STAR Aligner

1. **Download STAR Aligner:** `https://github.com/alexdobin/STAR`
2. **Build Genome Indices:**

```
STAR --runMode genomeGenerate --genomeDir /path/to/genomeDir \
     --genomeFastaFiles genome.fa --sjdbGTFfile annotations.gtf \
     --runThreadN 4
```

3. **Update `lib.settings` with STAR Paths:**

```
-STARindexdir /path/to/genomeDir
-gtffile /path/to/annotations.gtf
```

Configuring Paths Ensure all file paths in `lib.settings` are absolute or correctly relative to the execution directory.

1.2 Usage

1.2.1 Command-Line Execution

To run the `lib` command:

```
python main.py lib /path/to/data//
```

Replace `/path/to/data//` with the actual path where your data and `lib.settings` file are located.

1.2.2 Sample Data

Sample data can be downloaded from `https://github.com/wlab-bio/vdnamic/tree/main/SAMPLE_DIRECTORY`. Unpack the data into a directory and use the provided `lib.settings` file.

1.2.3 Expected Output

Upon successful execution, the console will display messages indicating the progress of each processing step.

1.3 Input

1.3.1 Required Directory Structure

The input directory should contain:

- `lib.settings`
- A path to raw sequencing data files (`R1.fastq`, `R2.fastq`) from read 1 and read 2 of a paired-end sequencing run, respectively. If there are multiple runs, fastqs from read-1 should have full paths separated by commas. Read-2 fastqs should have the same, in the corresponding order.

1.3.2 Required Input Files

1. `lib.settings`: Configuration file with processing parameters.
2. `R1.fastq`: Forward/read-1 FASTQ file.
3. `R2.fastq`: Reverse/read-2 read FASTQ file.

1.3.3 FASTQ File Specifications

- Files should be in standard FASTQ format with Phred+33 encoding.
- Paired-end reads with consistent naming conventions.
- Ensure that read lengths encompass sequence features in position ranges.

1.3.4 Directory Structure Visualization

```
/path/to/data//
 lib.settings
 R1.fastq
 R2.fastq
```

1.4 `lib.settings` File Format

The `lib.settings` file contains key-value pairs defining processing parameters. Each parameter is specified on a new line.

1.4.1 Parameter Descriptions

- `-source_for`: Path to the forward read FASTQ file.
  - **Expected Value**: Absolute or relative file path.
  - **Example**:

    ```
    -source_for ./R1.fastq
    ```

- `-source_rev`: Path to the reverse read FASTQ file.
  - **Expected Value**: Absolute or relative file path.
  - **Example**:

    ```
    -source_rev ./R2.fastq
    ```

- `-seqform_rev`: Reverse read sequence format specification.
  - **Purpose**: Defines the expected sequence structure in reverse reads.
  - **Syntax**: `[TYPE]_[SEQUENCE]_[RANGE]`
  - **Example**:

    ```
    -seqform_rev U_GCTNWWNNNNWWSWNNNWSWNNNWSWWNNNNNWWNTGA_2:39
    ```

  - **Explanation**:
    * `U`: Indicates a UMI sequence.
    * `GCTNWW...NTGA`: Expected nucleotide pattern.
    * `2:39`: Position range within the read (indexing starts at 0, up through position 38).
  - **Pattern Symbols**:
    * `N`: Any nucleotide (A, C, G, T).
    * `W`: Weak nucleotides (A or T).
    * `S`: Strong nucleotides (G or C).
- `-seqform_for`: Forward read sequence format specification(s).

  - **Purpose**: Defines the expected sequence structures in forward reads.
  - **Multiple features**: Separate multiple features with a `|` character.
  - **Example**:

    ```
    -seqform_for U_NWNNNNWWSWNNNWSWNNNWSWWNNNNWNGCG_0:31|U_AGGNWWNNNNWWSWNNNWSWNNNWSWWNNNNNWWNAGC_39:76
    ```

- `-min_mean_qual`: Minimum average Phred quality score for reads.
  - **Expected Value**: Integer between 0 and 40.
  - **Example**:

    ```
    -min_mean_qual 30
    ```

  - **Impact**: Only accepts reads with an average quality score of 30 or higher.
- `-filter_umi0_amp_len`: Minimum length for the amplicon associated with UMI0. -filter_umi1_amp_len does the same for UMI1.
  - **Expected Value**: Positive integer.
  - **Example**:

    ```
    -filter_umi0_amp_len 25
    ```

- `-min_uei_per_umi`: Minimum number of unique UEIs required to be associated with a UMI.
  - **Expected Value**: Positive integer.
  - **Example**:

    ```
    -min_uei_per_umi 2
    ```

- `-min_reads_per_assoc`: Minimum number of reads required for a distinct UMI-UMI association to be considered valid.
  - **Expected Value**: Positive integer.
  - **Example**:

    ```
    -min_reads_per_assoc 2
    ```

- `-min_assoc_per_umi`: Minimum number of total valid associations required for a UMI to be retained.
  - **Expected Value**: Positive integer.
  - **Example**:

    ```
    -min_assoc_per_umi 1
    ```

- `-u0`, `-u1`, `-u2`: UMI and UEI specifications.
  - **Purpose**: Define how to construct UMIs and UEIs from the parsed sequences.
  - **Syntax**: `-u[0/1/2] "[FORWARD/READ1_INDEX]","[REVERSE/READ1_INDEX]",[UMI/UEI_INDEX]`
  - **Example**:

    ```
    -u0 "","",1
    ```

- **Explanation**:
  - The format -uW X,Y,Z means that this UMI (if W=0 or 1) or UEI (if W=2) exists on all pairings of FORWARD/READ1 sequence-form X (as enumerated by the -seq_form keys) and REVERSE/READ2 sequence-form Y at "U"-labeled position (enumerated from the beginning of READ1) Z.
  - `"*"` or `""`: Wildcard or any index acceptable.

1.4.2 Formatting Rules

- Each UMI/UEI must be on a separate line.
- Ensure there are no trailing spaces or hidden characters.

1.5 Processing Steps

1.5.1 Step-by-Step Breakdown

1. **Quality Filtering of Raw FASTQ Files**
   - Reads are filtered based on the minimum mean quality score specified by `-min_mean_qual`.
   - Any reads failing this threshold are discarded.
2. **Sequence Parsing According to Specified Formats**
   - Reads are parsed to extract UMIs, UEIs, and amplicon sequences using the patterns defined in `-seqform_for` and `-seqform_rev`.
3. **Extraction and Organization of UMIs, UEIs, and Amplicon Sequences**
   - UMIs and UEIs are constructed based on the `-u0`, `-u1`, and `-u2` parameters.
   - Amplicon sequences (where labeled) are extracted and trimmed according to termination sequences and length requirements.
4. **Clustering and Pairing of UMIs and UEIs**
   - UMIs and UEIs are clustered to account for sequencing errors.
   - Pairings are established based on per-UEI consensus.
   - Thresholds for clustering are set by parameters such as `-min_uei_per_umi`.
5. **Generation of Subsampled (rarefaction) Datasets**
   - Subsampling is performed to enable rarefaction analysis.
   - Subsampled datasets are stored in `sub*//` directories.
6. **Optional Alignment of Amplicon Sequences to Reference Genome**
   - If STAR aligner paths are specified, amplicon sequences are aligned to the reference genome.
   - Alignment outputs are stored in `STARalignment*/` directories.

1.5.2 Intermediate Files

At each step, intermediate files are generated to facilitate processing:

- `part_*_*.txt`: Partitioned data files.
- `filtered_src_data.txt`: Quality-filtered reads.
- `tmp_*.txt`: Temporary files used during sorting and merging.

1.6 Output

The command generates several output files and directories in the specified `PATH//`.

1.6.1 Output Files

1. `uxi*.txt`: Files containing UMI and UEI sequences.
   - **Format**:

Read Number	Forward Index	Reverse Index	Sequence

   - **Purpose**: Stores extracted sequences for UMIs and UEIs.
2. `amp*.txt`: Files containing amplicon sequences.
   - **Format**:

Read Number	Forward Index	Reverse Index	Sequence

   - **Purpose**: Contains amplicon sequences associated with UMIs.
3. `readcounts.txt`: Summary of read counts for different sequence combinations.
   - **Purpose**: Provides an overview of the number of reads processed at each step.
4. `umi_stats.txt` and `pairing_stats.txt`: Statistics on UMI and UMI-UEI pairings.
   - **Content**: Contains counts of UMIs with different read abundances.
5. `rejected.txt`: Information on rejected reads.
   - **Purpose**: Logs reads that failed quality filters or did not match sequence patterns.
6. `ncrit.txt`: Critical values for UMI/UEI clustering.
   - **Purpose**: Stores thresholds used in clustering algorithms.
7. **Temporary and Intermediate Files**:
   - Various `part_*_*.txt` files.
   - `filtered_src_data.txt`.

1.6.2 Output Directories

- `sub*//`: Subdirectories containing subsampled data for rarefaction analysis.
- `STARalignment*/`: Directories with STAR aligner output.
- `uei_grp*//`: Directories containing grouped UEI associations for inference.

1.6.3 Example Content of Output Files

uxi0.txt

```
000001,0,0,ATCGTAGCTA
000002,0,0,GCTAGCTAGC
...
```

amp0_seqcons_trimmed.txt

```
0,3,ATCGTAGCTAGCTAGCTA
1,2,GCTAGCTAGCTAGCTAGC
...
```

1.7 Additional Notes

1.7.1 Best Practices and Recommendations

- **Parameter Tuning**:
  - Adjust mismatch rates based on sequencing quality.
  - Set `-min_mean_qual` higher for better-quality datasets.
- **Data Preparation**:
  - Ensure input FASTQ files are not corrupted.
  - Verify that the sequence formats in `lib.settings` match your experimental design.
- **Performance Optimization**:
  - Utilize multi-threading capabilities where possible.
  - Monitor system resources during processing to prevent bottlenecks.

1.7.2 Troubleshooting and FAQ

Common Errors

- **Error: Unrecognized pipeline input**:
  - **Cause**: Incorrect command-line arguments.
  - **Solution**: Ensure the command is in the correct format: `python main.py lib /path/to/data//`.
- **Reads failing quality filtering**:
  - **Cause**: Low-quality sequencing data.
  - **Solution**: Lower the `-min_mean_qual` threshold or improve data quality.
- **Alignment step taking a long time**:
  - **Cause**: Large dataset or limited computational resources.
  - **Solution**: Increase computational resources or perform alignment on a high-performance computing cluster.

Frequently Asked Questions

- **How can I adjust parameters for different read lengths?**
  - Update the position ranges in `-seqform_for` and `-seqform_rev` to match the new read lengths.
- **What do I do if my reads do not match the expected sequence patterns?**
  - Verify the sequence formats in `lib.settings`.
  - Check for issues in library preparation or sequencing.

1.7.3 Appendices

Data Formats
Custom Symbols in Sequence Patterns

- `N`: Any nucleotide (A, C, G, T).
- `W`: Weak nucleotides (A or T).
- `S`: Strong nucleotides (G or C).
- `U`: Indicates a UMI sequence.
- `RANGE`: Position range in the read, specified as `start:end`.

Command Reference

- `python main.py lib /path/to/data//`
  - Initiates the library processing pipeline.
- `-source_for`
  - Specifies the forward read FASTQ file.
- `-source_rev`
  - Specifies the reverse read FASTQ file.
- `-seqform_for`, `-seqform_rev`
  - Define sequence formats for parsing reads.

2 GSE

2.1 Introduction to GSE

GSE (Geodesic Spectral Embedding) is a scalable algorithm designed to compute embeddings for large graphs efficiently. It employs a multi-faceted approach that combines subsampling, iterative refinement, filtering, optimization, and parallel processing to generate high-quality embeddings.

- **Subsampling**: GSE creates smaller subsets of the graph to capture global structure while reducing computational overhead.
- **Iterative Refinement**: Initial embeddings from subsamples are integrated and refined iteratively to improve the accuracy of the global embedding.
- **Filtering**: Nodes are filtered based on connectivity metrics to remove noise and outliers, enhancing the quality of the embedding.
- **Optimization**: GSE uses projected gradient descent with Hessian computations to optimize the embedding with respect to a defined objective function.
- **Parallel Processing**: Leveraging multiple CPU cores accelerates processing, making GSE suitable for large datasets.

2.2 Usage

2.2.1 Environment Setup

Before using GSE, ensure that your Python environment is correctly configured with the necessary dependencies. The key requirements include:

- **Python 3.x**: GSE requires Python 3.
- **NumPy**: For numerical computations.
- **SciPy**: For sparse matrix operations and linear algebra functions.
- **Joblib**: For parallel processing.
- **Optional - Faiss**: For efficient k-nearest neighbors search in high-dimensional spaces.

Installing Dependencies You can install the required packages using `pip`:

```
pip install numpy scipy joblib
# Optional for approximate nearest neighbors:
pip install faiss-cpu
```

2.2.2 Command-Line Interface and Parameter Usage

GSE is executed via the command line with various parameters to customize its behavior.

Parsing Mechanics The command-line parser interprets parameters and values in the following way:

- Parameters start with a hyphen (`-`) and are followed by their corresponding values.
- Single-value parameters: Specify the parameter followed by its value (e.g., `-inference_dim 3`).
- Multi-value parameters: For parameters that accept multiple values, list them after the parameter flag (e.g., `-filter_criterion 5 10`).
- Boolean parameters: The presence of a parameter flag without a value is interpreted as `True` (e.g., `-verbose`).

Command Syntax **Basic Execution**:

```
python main.py GSE -path PATH// \
-inference_dim 2
```

Advanced Execution:

```
python main.py GSE -path PATH// \
-inference_dim 2 -inference_eignum 25 -final_eignum 225 \
-sub_num 30 -sub_size 15000 -ncpus 10 -filter_criterion 5 -calc_final ../
```

2.3 Command-Line Options

GSE offers a variety of parameters to customize its execution. Below are the key command-line options, along with detailed explanations, default values, acceptable ranges, and practical examples.

2.3.1 `-inference_dim INT`

- **Description**: Defines the target dimensionality d for the embedding space.
- **Default**: `2`
- **Acceptable Range**: Any positive integer (e.g., 2, 3, 5).
- **Impact**:
  - Higher d allows the embedding to capture more complex structures.
  - Increases computational complexity proportionally to $O(Nd^2)$, where N is the number of nodes.
- **Example**:

```
-inference_dim 3
```

2.3.2 `-inference_eignum INT`

- **Description**: Specifies the number of eigenvectors k used in initial computations.
- **Default**: `25`
- **Acceptable Range**: Integer satisfying $d \leq k \leq$ `-final_eignum`.
- **Impact**:
  - Increasing k can improve the approximation of the graph's spectral properties.
  - Leads to additional computational resources and time.
- **Example**:

```
-inference_eignum 50
```

2.3.3 `-final_eignum INT`

- **Description**: Determines the total number of eigenvectors for the final embedding.
- **Default**: `225`
- **Acceptable Range**: Any integer greater than or equal to `-inference_eignum`.
- **Impact**:
  - A higher value can capture more nuanced structures in the graph.
  - Increases memory usage and computational time.
- **Example**:

```
-final_eignum 300
```

2.3.4 `-sub_num INT`

- **Description**: Number of subsamples to create.
- **Default**: `0`
- **Acceptable Range**: Non-negative integers (e.g., 0, 10, 30).
- **Impact**:
  - Increasing this value enhances robustness by incorporating diverse graph structures.
  - Increases computational time linearly.
- **Example**:

```
-sub_num 30
```

2.3.5 `-sub_size INT`

- **Description**: Size of each subsample.
- **Default**: `0`
- **Acceptable Range**: Positive integers less than or equal to the number of nodes.
- **Impact**:
  - Larger sizes capture more global structure.
  - Increase computational complexity, especially during eigenvalue decomposition (scales cubically with `sub_size`).
- **Example**:

```
-sub_size 15000
```

2.3.6 `-ncpus INT`

- **Description**: Number of CPU cores to utilize for parallel processing.
- **Default**: One less than the total number of available CPU cores.
- **Acceptable Range**: Positive integers up to the number of CPU cores.
- **Impact**:
  - Increasing this value can speed up processing.
  - Setting it too high may lead to diminishing returns or system instability.
- **Example**:

```
-ncpus 10
```

2.3.7 `-filter_criterion FLOAT`

- **Description**: Percentile threshold (0-100) to filter out nodes based on connectivity metrics.
- **Default**: `None` (no filtering applied).
- **Acceptable Range**: Floating-point numbers between 0 and 100.
- **Impact**:
  - Removes nodes below the specified percentile, reducing noise.
  - Can reduce long-range structure (advantage or disadvantage, depending on context).
- **Example**:

```
-filter_criterion 5
```

2.3.8 `-calc_final PATH`

- **Description**: Directory path to save final calculation outputs.
- **Default**: `None`
- **Impact**:
  - Specifies where to store the final embedding results and associated files.
  - Ensure sufficient disk space is available in the specified directory.
- **Example**:

```
-calc_final ../results/
```

2.3.9 `-exit_code STRING`

- **Description**: Determines the execution mode of GSE.
- **Default**: `'full'`
- **Acceptable Values**: `'full'`, `'gd'`, `'eig'`
- **Impact**:
  - `'full'`: Executes the full embedding process.
  - `'gd'`: Runs gradient descent optimization only.
  - `'eig'`: Performs eigenvalue decomposition and exits.
- **Example**:

```
-exit_code gd
```

2.3.10 Recommendations Based on Dataset Size

- **Small Datasets** (less than 10,000 nodes):
  - Use a higher `-inference_dim` and `-inference_eignum` to capture more details.
  - Subsampling may not be necessary (`-sub_num 0`).
- **Medium Datasets** (10,000 to 1 million nodes):
  - Set `-sub_num` between 10 and 30.
  - Adjust `-sub_size` to balance between capturing global structure and computational feasibility.
- **Large Datasets** (over 1 million nodes):
  - Increase `-sub_size` to capture sufficient global structure (e.g., `50000`).
  - Reduce `-sub_num` if computational resources are limited.
  - Consider increasing `-filter_criterion` to reduce dataset size.

2.4 Algorithm Workflow Explanation

2.4.1 Step-by-Step Walkthrough

GSE follows a structured approach to perform spectral embedding on large-scale graphs:

1. **Data Loading**:
   - Reads the `link_assoc.npy` file.
   - Constructs a sparse adjacency matrix A.
2. **Filtering** (Optional):
   - Calculates an importance metric for each node using an initial embedding to calculate UEI dispersion.
   - Removes nodes below the specified `-filter_criterion` percentile.
3. **Subsampling**:
   - Creates multiple subsamples of the graph using random sampling.
   - Each subsample is embedded independently.
4. **Eigenvalue Decomposition**:
   - Performs eigenvalue decomposition on each subsample to obtain local embeddings.
   - Aggregates eigenvectors from subsamples to form a global eigenbasis.
5. **Iterative Refinement**:
   - Combines embeddings from different subsamples.
   - Aligns them using techniques like random rotations.
6. **Optimization with Projected Gradient Descent**:
   - Optimizes the embedding to minimize a defined objective function.
   - Uses gradient and Hessian computations for efficient convergence.
7. **Final Embedding**:
   - Expands the initial embedding using additional eigenvectors (`-final_eignum`).
   - Performs post-processing steps such as normalization.

2.4.2 Subsampling Mechanics

Subsampling is a crucial step in GSE, especially when dealing with large graphs. The goal is to capture the global structure of the graph while maintaining computational feasibility. The subsampling process involves:

1. **Contiguous Subgraph Identification:** GSE employs functions like `get_contig()` and `make_subset()` to identify and extract contiguous subgraphs. This ensures that the sampled subsets retain meaningful structural properties of the original graph.

2. **Iterative Refinement:** The algorithm iteratively adjusts the subset size and composition based on connectivity metrics. Parameters such as `-sub_num` (number of subsamples) and `-sub_size` (size of each subsample) allow users to balance between capturing global structures and managing computational resources.

3. **Seed Selection and Random Rotations:** To enhance the diversity of subsamples, GSE selects seed points based on nearest neighbor distances and applies random rotations using functions like `random_rotation_matrix()` and `interleave_arrays()`. This promotes variability in the embedding space and helps in avoiding local minima during optimization.

4. **Aggregation of Subsample Embeddings:** After individual subsamples are embedded, their eigenvectors are aggregated using `generate_evecs_from_subsets()` to form a global eigenbasis. This aggregated basis serves as the foundation for the final embedding, ensuring consistency and accuracy across the entire graph.

By meticulously managing the subsampling process, GSE ensures that the resulting embeddings are both representative of the original graph's structure and computationally tractable.

2.4.3 Random Rotation and Data Interleaving

To refine the embedding space and ensure robustness, GSE incorporates the following techniques:

- **Random Rotation:** The function `random_rotation_matrix()` generates random rotation matrices for 2D or 3D spaces. These rotations are applied to subsample embeddings to introduce variability and prevent alignment biases. By rotating the embedding space randomly, GSE ensures that the algorithm explores diverse configurations, enhancing the quality of the final embedding.

- **Data Interleaving:** The `interleave_arrays()` function merges two arrays by alternating their elements in a block-wise manner. This interleaving is crucial when combining multiple subsample embeddings, as it maintains a balanced representation across different regions of the graph. Interleaving facilitates the alignment and integration of embeddings from various subsamples, contributing to a cohesive global embedding.

These methods collectively enhance the embedding process by promoting diversity and balance, ensuring that the final representation accurately captures the intricate structures within the graph.

2.4.4 Optimization with Projected Gradient Descent

After generating an initial embedding, GSE optimizes it to better capture the underlying graph structure. This optimization involves:

- **Gradient Calculation:** Utilizing the `GSEobj` class, GSE computes gradients with respect to the embedding coordinates. The method `calc_grad_and_hessp()` calculates both the gradient and the Hessian-vector product, essential for efficient optimization. Gradients are derived based on the likelihood of node associations, ensuring that the embedding aligns with the graph's connectivity.

- **Hessian-Vector Product:** The same method also computes the Hessian-vector product, which provides second-order information about the objective function. This information is leveraged by optimization algorithms like the Trust Region Reflective algorithm to achieve faster convergence rates.

- **Projected Conjugate Gradient (PCG) Method:** The function `proj_cg()` solves the linear system arising from the optimization problem using a modified conjugate gradient method. It incorporates L2 regularization to stabilize the solution and ensures that the embedding does not overfit to noisy data.
- **Regularization and Convergence:** To prevent overfitting and ensure numerical stability, GSE applies L2 regularization during optimization. The algorithm monitors convergence by evaluating the norm of the residuals, terminating the optimization process once the solution meets the specified tolerance criteria.

Through meticulous gradient and Hessian computations, GSE refines the embedding to accurately reflect the graph's structural properties, resulting in a high-quality low-dimensional representation.

2.5 Output Files and Their Interpretation

GSE produces several output files representing the embedded coordinates and associated labels for each node.

2.5.1 `GSEoutput.txt`

- **Description**: A CSV file containing the embedded coordinates for each node.
- **Columns**:
  1. `node_index` (int): Original node index.
  2. `x1, x2, ..., xn` (float): Coordinates in the embedding space.
- **Usage**:
  - Can be loaded into data analysis tools for visualization or further processing.

2.5.2 `final_labels.txt`

- **Description**: A CSV file containing node attributes and labels. The content can vary widely depending on the input data and the nature of the graph being analyzed.
- **Generation**: Created when the `-calc_final` option is used, incorporating information from the embedding process and any available node metadata.
- **Typical Columns**:
  1. `point_type` (int): Indicates the category or type of node.
  2. `true_raw` (int): The original raw index of the node in the input data.
  3. Additional columns may be present, depending on the input data and analysis parameters.

2.5.3 `final_coords.txt`

- **Description**: A CSV file containing the final embedding coordinates for each node in the graph.
- **Generation**: Created when the `-calc_final` option is used, representing the output of the GSE algorithm.

2.5.4 `evecs.npy`

- **Description**: NumPy binary file containing the eigenvectors from the spectral decomposition.
- **Shape**: (N, k) where N is the number of nodes and k is `-final_eignum`.
- **Usage**:
  - Load using `numpy.load` for downstream tasks like visualization or machine learning applications.

2.5.5 `evals.txt`

- **Description**: Plain text file listing the eigenvalues corresponding to each eigenvector in `evecs.npy`.
- **Usage**:
  - Assess the contribution of each eigenvector to the embedding.

2.5.6 `orig_evecs_gapnorm.npy`

- **Description**: NumPy binary file containing the gap-normalized eigenvectors of the graph Laplacian.
- **Shape**: (N, k) where N is the number of nodes and k is the number of eigenvectors (typically equal to `-inference_eignum`).
- **Generation**: Created during the initial eigendecomposition phase, before the iterative refinement process.
- **Purpose**: Serves as an intermediate representation of the graph's spectral properties, incorporating gap normalization to enhance the separation between eigenvectors.
- **Usage**:
  - Can be used as a starting point for further embedding refinement.
  - Useful for analyzing the initial spectral properties of the graph before optimization.
  - Load using `numpy.load` for custom post-processing or analysis tasks.
- **Note**: This file is generated and used internally by GSE. It may be overwritten in subsequent runs unless explicitly preserved.

2.5.7 Performance Optimizations and Parallel Processing

To handle large-scale graphs efficiently, GSE incorporates several performance optimization strategies:

- **Parallel Processing:** GSE utilizes Python's `joblib` and `concurrent.futures` libraries to parallelize computationally intensive tasks. Parameters like `-ncpus` allow users to specify the number of CPU cores to employ, thereby accelerating processes such as subsampling, eigenvalue decomposition, and nearest neighbor searches.
- **Just-In-Time (JIT) Compilation:** Leveraging Numba's `@njit` decorator, functions like `recalculate_nn_distances(` and `get_dxpts()` are compiled to machine code at runtime. This significantly enhances execution speed, especially for operations involving large numerical computations.
- **Efficient Data Structures:** Sparse matrices are employed extensively using SciPy's `csc_matrix` and `csr_matrix` formats to optimize memory usage and computational efficiency during matrix operations and eigenvalue decompositions.
- **Batch Processing and Memory Management:** GSE processes data in manageable chunks, ensuring that memory usage remains optimal even when dealing with graphs containing millions of nodes. Techniques like subsampling and iterative refinement further contribute to efficient memory utilization.

These optimizations collectively enable GSE to scale effectively with increasing graph sizes, maintaining high performance without compromising accuracy.

2.5.8 Post-Processing Guidance

Visualization Examples You can visualize the embeddings using libraries like Matplotlib:

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

# Load embeddings
embeddings = np.loadtxt(’GSEoutput.txt’, delimiter=’,’, skiprows=1)[:, 1:]

# Plot for 2D embeddings
plt.scatter(embeddings[:, 0], embeddings[:, 1], s=1)
plt.xlabel(’Dimension 1’)
plt.ylabel(’Dimension 2’)
plt.title(’GSE Embedding’)
plt.show()
```

2.6 Error Handling and Troubleshooting Tips

2.6.1 Common Issues and Resolutions

Convergence Problems **Issue**: During eigenvalue decomposition, convergence issues may arise.
Solutions:

- Increase the `maxiter` parameter in the eigenvalue decomposition function.
- Ensure the adjacency matrix is symmetric and well-conditioned.
- Check for isolated nodes or disconnected components.

Memory Errors **Issue**: Insufficient memory when handling large graphs.
Solutions:

- Reduce `-final_eignum`.
- Increase system memory or use machines with higher RAM.
- Optimize subsampling parameters to reduce memory footprint.

Invalid Input Data **Issue**: Missing or incorrectly formatted input files.
Solutions:

- Verify that `link_assoc.npy` exists and follows the required format.
- Check for data inconsistencies, such as non-integer node indices.

2.6.2 Debugging Strategies

- Use logging options to gain insights into the algorithm’s execution.
- Monitor system resources to detect bottlenecks.
- Validate input data before running the algorithm.

2.7 Code Documentation and Comments

The GSE implementation is provided in the `optimOps.py` file. Below are key functions and their descriptions:

Parameter	Default Value	Description
-inference_dim	2	Embedding dimensionality
-inference_eignum	25	Number of eigenvectors for initial computations
-final_eignum	225	Total number of eigenvectors for final embedding
-sub_num	0	Number of subsamples to create
-sub_size	0	Size of each subsample
-ncpus	All available CPUs minus one	Number of CPU cores to use
-filter_criterion	None	Percentile threshold for node filtering
-calc_final	None	Directory for final outputs
-exit_code	'full'	Execution mode

Table 1: GSE Command-Line Parameters

2.7.1 Function run_GSE

```
def run_GSE(output_name, params):
    """
    Main function to run the Geodesic Spectral Embedding (GSE) algorithm.

    Args:
    - output_name (str): Name of the output file to save embeddings.
    - params (dict): Dictionary of parameters for GSE execution.

    Steps:
    1. Parse parameters and initialize variables.
    2. Load and preprocess graph data.
    3. Perform filtering if specified.
    4. Generate eigenvectors through subsampling or full computation.
    5. Run the embedding optimization process.
    6. Save outputs and perform final calculations if required.
    """
    # Function implementation...
```

2.7.2 Function generate_evecs_from_subsets

```
def generate_evecs_from_subsets(this_GSEobj, orig_evecs_list, ref_index_list, empirical_distribution,
    ↪ subsample_this_iteration, GSE_final_eigenbasis_size, output_preorthbasis=False, retain_all_eigs=False)
    ↪ :
    """
    Aggregates eigenvectors from subsamples to form a global eigenbasis.

    Args:
    - this_GSEobj: GSE object containing graph data.
    - orig_evecs_list: List of eigenvector arrays from subsamples.
    - ref_index_list: List of indices corresponding to subsample nodes.
    - empirical_distribution: List indicating whether to apply quantile transform.
    - subsample_this_iteration: Number of subsamples in the current iteration.
    - GSE_final_eigenbasis_size: Desired size of the final eigenbasis.
    - output_preorthbasis (bool): Whether to save the pre-orthogonalized basis.
    - retain_all_eigs (bool): Whether to retain all eigenvectors.

    Returns:
    - full_evecs: Global eigenvectors for the full graph.
    """
    # Function implementation...
```

2.7.3 Function calc_grad_and_hessp

```
def calc_grad_and_hessp(self, X, inp_vec):
    """
    Calculates the gradient and Hessian-vector product for the optimization.

    Args:
    - X (numpy.ndarray): Current embedding flattened into a vector.
    - inp_vec (numpy.ndarray): Vector for Hessian-vector product. If None, computes gradient only.

    Returns:
    - If computing gradient: Tuple (negative log-likelihood, negative gradient).
    - If computing Hessian-vector product: Negative Hessian-vector product.
    """
    # Function implementation...
```

2.7.4 Comprehensive Function and Class Documentation

To facilitate a deeper understanding of GSE's internal workings, the following functions and classes are documented:

`partition_graph_csc_matrix()` Partitions the graph represented by a CSC (Compressed Sparse Column) matrix into specified subsets using METIS's k-way partitioning. This function is pivotal for dividing the graph into manageable subgraphs for parallel processing.

`make_subset()` Creates a subset of the graph based on specified parameters such as coverage and minimum association thresholds. It ensures that each subset maintains a contiguous structure, which is essential for accurate embedding.

`generate_evecs_from_subsets()` Aggregates eigenvectors derived from individual subsamples to form a global eigenbasis. This aggregation is critical for maintaining consistency across different subsamples and ensuring the integrity of the final embedding.

`run_GSE()` The primary function that orchestrates the entire GSE process. It handles parameter parsing, data loading, subsampling, eigenvalue decomposition, optimization, and result generation. This function serves as the entry point for executing the GSE algorithm.

`filter_data()` Applies filtering criteria to the graph data, removing nodes and links that do not meet specified percentile thresholds. This preprocessing step reduces noise and focuses the embedding on the most significant structural components of the graph.

`GSEobj` Class Encapsulates all relevant data and methods for performing GSE. Key methods include:

- `load_data()`: Loads and preprocesses the graph data from input files.
- `eigen_decomp()`: Performs eigenvalue decomposition on the graph's adjacency matrix to derive initial embeddings.
- `calc_grad_and_hessp()`: Computes gradients and Hessian-vector products necessary for optimization.
- `calc_grad()` and `calc_hessp()`: Wrapper methods for gradient and Hessian computations.

`proj_cg()` Implements the Projected Conjugate Gradient method for solving linear systems with L2 regularization. This function is integral to the optimization phase, ensuring that embeddings converge efficiently and accurately.

`parallel_knn()` and Related Functions Handles parallel nearest neighbor searches using libraries like Annoy, FAISS, and scikit-learn. These functions enable efficient computation of nearest neighbors, which are essential for understanding local graph structures during embedding.

`filter_extremes_pairwise()` Filters out extreme values in the data based on specified percentile thresholds for each feature pair. This function enhances the robustness of the embedding by mitigating the impact of outliers.

`generate_final_eigenbasis()` Generates the final eigenbasis by performing additional eigenvalue decompositions on refined adjacency matrices. This step ensures that the final embedding captures the most significant structural features of the graph.

`linear_interp()` Performs linear interpolation on embedding coordinates, facilitating the expansion of the initial embedding space to incorporate additional eigenvectors. This function is crucial for refining and scaling the embedding.

`spec_GSEobj()` Executes the spectral GSEobj likelihood maximization, integrating embedding adjustments and optimization to finalize the embedding coordinates.

`get_contig()` **and** `min_contig_edges()` Identify and manage contiguous subgraphs within the larger graph, ensuring that each subsample maintains structural coherence.

`GSEobj` Class Methods

- `load_data()`:
  - **Purpose**: Loads and preprocesses graph data from input files. It handles the reindexing of node identifiers and constructs the necessary data structures for subsequent embedding steps.
  - **Process**: Reads the `link_assoc_reindexed.npy` file, computes link counts for each node, and filters out nodes with no connections.
- `eigen_decomp()`:
  - **Purpose**: Performs eigenvalue decomposition on the graph's adjacency matrix to derive initial embeddings.
  - **Parameters**:
    - `orth` (bool): If `True`, orthogonalizes the resulting eigenvectors.
    - `krylov_approx` (str): Path to precomputed Krylov subspace vectors for approximation.
    - `print_evecs` (bool): If `True`, saves the computed eigenvectors to disk.
  - **Process**: Depending on the parameters, it either loads precomputed eigenvectors, performs a full eigenvalue decomposition, or uses a Krylov subspace method for approximation. It handles exceptions like convergence failures and ensures that trivial eigenvectors are excluded from the final set.
- `calc_grad_and_hessp()`:
  - **Purpose**: Calculates both the gradient and the Hessian-vector product for the optimization process.
  - **Parameters**:
    - `X` (numpy array): Current embedding vector.
    - `inp_vec` (numpy array): Input vector for Hessian-vector product computation.
  - **Process**: Depending on whether a Hessian-vector product is requested, it computes the necessary gradients and updates based on the embedding's likelihood and regularization terms.
- `calc_grad()` and `calc_hessp()`:
  - **Purpose**: Wrapper methods that call `calc_grad_and_hessp()` with appropriate arguments to compute gradients or Hessian-vector products independently.

2.7.5 License

This software is distributed under the MIT License. See `LICENSE` for details.

3 Glossary and Parameter Definitions

3.1 Glossary

Embedding Dimensionality (`-inference_dim`) The number of dimensions in which the graph is embedded. Higher dimensions can capture more complex structures but may increase computational complexity.

Eigenvectors (`-inference_eignum`, `-final_eignum`) Principal components derived from the graph's adjacency matrix, used to define the embedding space. Initial eigenvectors capture fundamental structures, while final eigenvectors refine the embedding.

Subsampling (`-sub_num`, `-sub_size`) The process of selecting subsets of the graph for separate embedding computations. `-sub_num` specifies the number of subsamples, and `-sub_size` defines the size of each subsample.

Filtering Criterion (`-filter_criterion`) A percentile threshold used to remove nodes with low connectivity metrics, reducing noise in the embedding process.

Gradient Descent Optimization (`-exit_code gd`) An optimization mode that focuses solely on refining the embedding through gradient descent, without performing a full embedding process.

Eigenvalue Decomposition (`-exit_code eig`) An execution mode that performs eigenvalue decomposition and exits, allowing users to inspect eigenvectors and eigenvalues independently.

CPU Core Allocation (`-ncpus`) The number of CPU cores allocated for parallel processing tasks within GSE. Optimizing this parameter can significantly impact performance.

3.2 Parameter Definitions

- `-inference_dim FLOAT`: Sets the embedding dimensionality. Default is 2.
- `-inference_eignum FLOAT`: Number of eigenvectors for initial computations. Default is 25.
- `-final_eignum FLOAT`: Total number of eigenvectors for the final embedding. Default is 225.
- `-sub_num FLOAT`: Number of subsamples to create. Default is 0 (no subsampling).
- `-sub_size FLOAT`: Size of each subsample. Default is 0.
- `-ncpus FLOAT`: Number of CPU cores to use. Default is all available CPUs minus one.
- `-filter_criterion FLOAT`: Percentile threshold for node filtering. Default is `None`.
- `-calc_final PATH`: Directory path to save final calculation outputs. Default is `None`.
- `-exit_code STRING`: Determines the execution mode of GSE. Acceptable values are `'full'`, `'gd'`, `'eig'`. Default is `'full'`.