BandHiC

Weibing Wang

Jul 18, 2025

CONTENTS

Given that most informative chromatin contacts occur within a limited genomic distance (typically within 2 Mb), **BandHiC** adopts a banded storage scheme that stores only a configurable diagonal bandwidth of the dense Hi-C contact matrices. This design can reduce memory usage by up to 99% compared to dense matrices, while still supporting fast random access and user-friendly indexing operations. In addition, BandHiC supports flexible masking mechanisms to efficiently handle missing values, outliers, and unmappable genomic regions. It also provides a suite of vectorized operations optimized with NumPy, making it both scalable and practical for ultra-high-resolution Hi-C data analysis.

CHAPTER

ONE

QUICK START

This section shows basic usage of the **BandHiC** package.

1.1 Overview

BandHiC is a Python package for efficient storage, manipulation, and analysis of Hi-C matrices using a banded matrix representation.

Given that most informative chromatin contacts occur within a limited genomic distance (typically within 2 Mb), BandHiC adopts a banded storage scheme that stores only a configurable diagonal bandwidth of the dense Hi-C contact matrices. This design can reduce memory usage by up to 99% compared to dense matrices, while still supporting fast random access and user-friendly indexing operations. In addition, BandHiC supports flexible masking mechanisms to efficiently handle missing values, outliers, and unmappable genomic regions. It also provides a suite of vectorized operations optimized with NumPy, making it both scalable and practical for ultra-high-resolution Hi-C data analysis.

1.2 Installation

1.2.1 Required Package

BandHiC can be installed on Linux-like systems and requires the following dependencies:

1. python >= 3.11
2. numpy >= 2.3
3. pandas >= 2.3
4. scipy >= 1.16
5. cooler >= 0.10
6. hic_straw >= 1.3

There are two recommended ways to install **BandHiC**:

1.2.2 Option 1: Install via pip

If you already have Python >= 3.11 installed:

```
pip install bandhic
```

1.2.3 Option 2: Install from source with conda

```
# 1. Clone the repository
git clone https://github.com/xdwwb/BandHiC-Master.git
cd BandHiC-Master

# 2. Create the environment and activate it
conda env create -f environment.yml
conda activate bandhic

# 3. Install BandHiC
pip install .
```

1.2.4 Build Troubleshooting for hic-straw

If you encounter an error like the following while installing or building `hic-straw`:

```
fatal error: curl/curl.h: No such file or directory
```

This means the C++ extension in `hic-straw` requires the **libcurl development headers**, which are not installed by default on many systems.

Solution 1: Install system dependencies (for pip installation)

You need to install the `libcurl` development package before building:

- **On Ubuntu/Debian**:

  ```
  sudo apt-get update
  sudo apt-get install libcurl4-openssl-dev
  ```

- **On Fedora/CentOS/RHEL**:

  ```
  sudo dnf install libcurl-devel
  ```

- **On macOS (with Homebrew)**:

  ```
  brew install curl
  ```

  If Homebrew's curl is not found automatically, you may need to set environment variables:

  ```
  export CPATH="$(brew --prefix curl)/include"
  export LIBRARY_PATH="$(brew --prefix curl)/lib"
  ```

Solution 2: Use Conda (recommended for convenience)

Instead of building `hic-straw` from source, you can install a prebuilt binary via Bioconda:

```
conda install -c bioconda hic-straw
```

To avoid conflicts and ensure reproducibility, we recommend installing it in a fresh Conda environment:

```
conda create -n bandhic-env python=3.11
conda activate bandhic-env
conda install -c bioconda hic-straw
```

(continues on next page)

(continued from previous page)

```
# Install BandHiC
pip install bandhic
```

1.3 What is BandHiC?

1.3.1 Data structure of BandHiC

To address the growing memory demands posed by high-resolution Hi-C data, we introduce `band_hic_matrix`, the core class implemented in the BandHiC package. For a Hi-C contact matrix $A \in \mathbb{R}^{n \times n}$ at resolution *r*, `band_hic_matrix` retains only the diagonals within a user-defined bandwidth *k*, yielding a compact representation $D \in \mathbb{R}^{n \times k}$. This format ensures that each column in *D* corresponds to a fixed diagonal of *A*, such that the mapping $A[i,j] = D[i,j-i]$ holds for $|i-j| \leq k$.

The memory efficiency achieved by this strategy is substantial. When $k \ll n$, the memory footprint of `band_hic_matrix` is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(nk)$. For example, assuming a resolution of 1 kb and a bandwidth of 2 Mb (k = 200), the representation of chromosome 1 of the human genome (~249 Mb) requires 3.7 GB of memory, less than 1% of the memory required by the dense matrix (~461.9 GB). This compression enables the use of high-resolution Hi-C data on commodity hardware without sacrificing random access efficiency.

To further enhance flexibility of usage, `band_hic_matrix` integrates an optional two-tier masking mechanism. The element-wise mask matrix $M \in \{0,1\}^{n \times k}$ allows users to selectively ignore missing or outlier contacts, enabling robust statistical estimation on unmasked subsets. Additionally, a bin-level mask $X \in \{0,1\}^n$ supports the exclusion of entire rows or columns, which is particularly useful for removing repetitive genomic regions devoid of a valid Hi-C signal. These masking features facilitate downstream tasks such as the estimation of average contact intensity at given distances, while confirming statistical validity.

Lastly, a scalar default value *d* is defined to fill in the undefined entries of *A* not covered by the band matrix *D*. This default is typically set to 0, consistent with the assumption that long-range interactions are negligibly sparse. It also ensures that reconstruction of the full matrix *A* (if required) can be achieved seamlessly by combining *D*, *M*, *X*, and *d*. Overall, `band_hic_matrix` provides an efficient, flexible data structure for scalable Hi-C data analysis.

1.3.2 Features of BandHiC

A key feature of `band_hic_matrix` is its direct coordinate mapping between the banded matrix *B* and the full dense matrix *A*. For any pair of genomic loci *(i, j)* satisfying the band constraint $|i-j| \leq k$, the interaction frequency $A[i,j]$ can be accessed in constant time via $D[i,j-i]$. This structure ensures random access in $\mathcal{O}(1)$ time, which is critical for performance-sensitive Hi-C analyses, particularly when memory constraints prohibit the use of fully dense matrices. Data access in `band_hic_matrix` is fully consistent with that of a dense matrix, as each entry is accessed via $B[i,j] = D[i,j-i] = A[i,j]$, allowing users to interact with `band_hic_matrix` objects as if they were dense matrices without needing to consider the underlying implementation.

Owing to this random-access capability, `band_hic_matrix` supports NumPy-like indexing semantics, including slicing, boolean indexing, and fancy indexing. This design allows users to easily query local chromatin contacts. For instance, a slice operation such as `B[i:j, i:j]` retrieves a banded submatrix. Combined with the `todense` operation, this enables reconstruction of the dense submatrix for downstream analysis or visualization.

In addition to flexible data access, `band_hic_matrix` also supports a wide range of numerical operations, including element-wise arithmetic operations and reduction operations. These operations are implemented using NumPy for efficiency. Standard reduction operations such as `sum`, `min`, and `max` are supported along conventional axes (rows or columns), as well as along the diagonal axis, which is particularly useful in summarizing interaction intensities by genomic distance. This feature facilitates common Hi-C analyses such as distance-decay profiling.

Taken together, `band_hic_matrix` combines the memory efficiency of a banded storage model with the expressiveness of NumPy's interface. By mimicking both NumPy's `ndarray` and `MaskedArray` behaviors, it provides an intuitive and powerful interface for users, substantially lowering the barrier to adoption and promoting integration into existing Hi-C data analysis workflows.

1.3.3 BandHiC reduces the memory consumption of TopDom

To evaluate the effectiveness of BandHiC, we benchmarked the TopDom algorithm on chromosome 1 of mouse embryonic stem cell (mESC) Micro-C data across multiple resolutions using both dense matrix and `band_hic_matrix`. TopDom was reimplemented in Python to support both NumPy's `ndarray` and BandHiC's `band_hic_matrix` class. As shown in Fig. 1B, BandHiC substantially reduces memory usage at all tested resolutions. At 1000 bp and 500 bp resolution, the dense matrices require over 72 GB of memory, exceeding the available system memory (60 GB RAM and 12 GB swap space), and causing the program to terminate due to memory allocation failure. In contrast, the banded format completes successfully, with memory usage of only 5,989 MB and 23,902 MB at 1000 bp and 500 bp resolution, respectively. The banded format introduces modest additional runtime compared to the dense matrix (Fig. 1C). This overhead stems not from masking—which was disabled in this evaluation—but from index computation performed during element access in the `band_hic_matrix` object.

These results indicate that `band_hic_matrix` enables domain-calling algorithms like TopDom to be applied to high-resolution Hi-C data on standard hardware, making large-scale analysis feasible without incurring significant computational cost. Owing to BandHiC's NumPy-like API, other Hi-C-based pattern identification methods can be reimplemented with minimal modifications to the original code, thereby significantly improving their scalability and adaptability to higher-resolution data.

CHAPTER

TWO

TUTORIALS

2.1 Prerequisites

BandHiC can serve as an alternative to the NumPy package when managing and manipulating Hi-C data, aiming to address the issue of excessive memory usage caused by storing dense matrices using NumPy's `ndarray`. At the same time, BandHiC supports masking operations similar to NumPy's `ma.MaskedArray` module, with enhancements tailored for Hi-C data.

Users can leverage their experience with NumPy when using the BandHiC package, so it is recommended that users have some basic knowledge of NumPy. A link to NumPy is provided below: https://numpy.org

2.2 Import `bandhic` package

```
>>> import bandhic as bh
```

2.3 Initialize a `band_hic_matrix` object

Initialize from a SciPy `coo_matrix` object:

```
>>> import bandhic as bh
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> coo = coo_matrix(([1, 2, 3], ([0, 1, 2],[0, 1, 2])), shape=(3,3))
>>> mat1 = bh.band_hic_matrix(coo, diag_num=2)
```

Initialize from a tuple (data, (row, col)):

```
>>> mat2 = bh.band_hic_matrix(([4, 5, 6], ([0, 1, 2],[2, 1, 0])), diag_num=1)
```

Initialize from a full dense array, only upper-triangular part is stored, lower part is symmetrized:

```
>>> arr = np.arange(16).reshape(4,4)
>>> mat3 = bh.band_hic_matrix(arr, diag_num=3)
```

2.4 Load or save a `band_hic_matrix` object

```
>>> bh.save_npz('./sample.npz', mat)
>>> mat = bh.load_npz('./sample.npz')
```

Load from .hic file:

```
>>> mat = bh.straw_chr('sample.hic',
                  'chr1',
                  resolution=10000,
                  diag_num=200
                  )
```

Load from .mcool file:

```
>>> mat = bh.cooler_chr('sample.mcool',
                  'chr1',
                  diag_num=200
                  resolution=10000,
                  )
```

2.5 Construct a band_hic_matrix object

```
# Create a band_hic_matrix object filled with zeros.
>>> mat1 = bh.zeros((5, 5), diag_num=3, dtype=float)

# Create a band_hic_matrix object filled with ones.
>>> mat2 = bh.ones((5, 5), diag_num=3, dtype=float)

# Create a band_hic_matrix object filled as an identity matrix.
>>> mat3 = bh.eye((5, 5), diag_num=3, dtype=float)

# Create a band_hic_matrix object filled with a specified value.
>>> mat4 = bh.full((5, 5), fill_value=0.1, diag_num=3, dtype=float)

# Create a band_hic_matrix object matching another matrix, filled with zeros.
>>> mat5 = bh.zeros_like(mat1, diag_num=3, dtype=float)

# Create a band_hic_matrix object matching another matrix, filled with ones.
>>> mat6 = bh.ones_like(mat1, diag_num=3, dtype=float)

# Create a band_hic_matrix object matching another matrix, filled as an identity matrix.
>>> mat7 = bh.eye_like(mat1, diag_num=3, dtype=float)

# Create a band_hic_matrix object matching another matrix, filled with a specified value.
>>> mat8 = bh.full_like(mat1, fill_value=0.1, diag_num=3, dtype=float)
```

2.6 Indexing on band_hic_matrix

```
# First, we create a band_hic_matrix object:
>>> import numpy as np
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(16).reshape(4,4), diag_num=2)

# Single-element access (scalar)
>>> mat[1, 2]
```

(continues on next page)

(continued from previous page)

```
6

# Masked element returns masked
>>> mat2 = bh.band_hic_matrix(np.eye(4), dtype=int, diag_num=2, mask=([0],[1]))
>>> mat2[0, 1]
masked

# Square submatrix via two-slice indexing returns band_hic_matrix
>>> sub = mat[1:3, 1:3]
>>> isinstance(sub, bh.band_hic_matrix)
True

# Single-axis slice returns band_hic_matrix for square region
>>> sub2 = mat[0:2]  # equivalent to mat[0:2, 0:2]
>>> isinstance(sub2, bh.band_hic_matrix)
True

# Fancy indexing returns ndarray or MaskedArray
>>> arr = mat[[0,2,3], [1,2,0]]
>>> isinstance(arr, np.ndarray)
True

>>> mat.add_mask([0,1],[1,2])  # Add mask to some entries
>>> masked_arr = mat[[0,1], [1,2]]
>>> isinstance(masked_arr, np.ma.MaskedArray)
True

# Boolean indexing with band_hic_matrix
>>> mat3 = bh.band_hic_matrix(np.eye(4), diag_num=2, mask=([0,1],[1,2]))
>>> bool_mask = mat3 > 0  # Create a boolean mask
>>> result = mat3[bool_mask]  # Use boolean mask for indexing
>>> isinstance(result, np.ma.MaskedArray)
True
>>> result
masked_array(data=[1.0, 1.0, 1.0, 1.0],
         mask=[False, False, False, False],
   fill_value=0.0)
```

2.7 Masking

```
# Add item-wise mask:
>>> mat.add_mask([0, 1], [1, 2])

# Add row/column mask:
>>> mask = np.array([True, False, False])
>>> mat.add_mask_row_col(mask)

# Remove mask for specified indices.
>>> mat.unmask(( [0],[1] ))

# Remove all item-wise mask and row/column mask.
```

(continues on next page)

(continued from previous page)

```
>>> mat.unmask()

# Remove all item-wise mask and row/column mask.
>>> mat.clear_mask()

# Drop all item-wise mask but preserve all row/column mask.
>>> mat.drop_mask()

# Drop all row/column mask.
>>> mat.drop_mask_row_col()

# Access masked `band_hic_matrix` will obtain `np.ma.MaskedArray` object:
>>> mat.add_mask([0, 1], [1, 2])
>>> masked_arr = mat[[0,1], [1,2]]
>>> isinstance(masked_arr, np.ma.MaskedArray)
True
```

2.8 Universal functions (ufunc)

Universal functions that BandHiC supports:

Function 1	Description 1	Function 2	Description 2
absolute	Absolute value	add	Element-wise addition
arccos	Inverse cosine	arccosh	Inverse hyperbolic cosine
arcsin	Inverse sine	arcsinh	Inverse hyperbolic sine
arctan	Inverse tangent	arctan2	Arctangent of y/x with quadrant
arctanh	Inverse hyperbolic tangent	bitwise_and	Element-wise bitwise AND
bitwise_or	Element-wise bitwise OR	bitwise_xor	Element-wise bitwise XOR
cbrt	Cube root	conj	Complex conjugate
conjugate	Alias for conj	cos	Cosine function
cosh	Hyperbolic cosine	deg2rad	Degrees to radians
degrees	Radians to degrees	divide	Element-wise division
divmod	Quotient and remainder	equal	Element-wise equality test
exp	Exponential	exp2	Base-2 exponential
expm1	exp(x) - 1	fabs	Absolute value (float)
float_power	Floating-point power	floor_divide	Integer division (floor)
fmod	Modulo operation	gcd	Greatest common divisor
greater	Element-wise greater-than test	greater_equal	Greater-than or equal test
heaviside	Heaviside step function	hypot	Euclidean norm
invert	Bitwise inversion	lcm	Least common multiple
left_shift	Bitwise left shift	less	Element-wise less-than test
less_equal	Less-than or equal test	log	Natural logarithm
log1p	log(1 + x)	log2	Base-2 logarithm
log10	Base-10 logarithm	logaddexp	log(exp(x) + exp(y))
logaddexp2	Base-2 version of logaddexp	logical_and	Element-wise logical AND
logical_or	Element-wise logical OR	logical_xor	Element-wise logical XOR
maximum	Element-wise maximum	minimum	Element-wise minimum
mod	Remainder (modulo)	multiply	Element-wise multiplication
negative	Element-wise negation	not_equal	Element-wise inequality test
positive	Returns input unchanged	power	Raise to power

continues on next page

Table 1 – continued from previous page

Function 1	Description 1	Function 2	Description 2
rad2deg	Radians to degrees	radians	Degrees to radians
reciprocal	Element-wise reciprocal	remainder	Modulo remainder
right_shift	Bitwise right shift	rint	Round to nearest integer
sign	Sign of input	sin	Sine function
sinh	Hyperbolic sine	sqrt	Square root
square	Square of input	subtract	Element-wise subtraction
tan	Tangent function	tanh	Hyperbolic tangent
true_divide	Division that returns float		

BandHiC supports these universal functions, and they can be used in the following three ways:

1. As methods of the `band_hic_matrix` object:

```
# When two band_hic_matrix objects are involved, their shape and diag_num must match
>>> mat3 = mat1.add(mat2)
>>> mat4 = mat1.less(mat2)
>>> mat5 = mat1.negative()
```

2. Using mathematical operators:

```
>>> mat3 = mat1 + mat2
>>> mat4 = mat1 < mat2
>>> mat5 = - mat1
```

3. Calling NumPy's universal functions:

```
>>> mat3 = np.add(mat1, mat2)
>>> mat4 = np.less(mat1, mat2)
>>> mat5 = np.negative(mat1)
```

2.9 Other Array Functions

Function	Description
sum	Compute the sum of all elements along the specified axis
prod	Compute the product of all elements along the specified axis
min	Return the minimum value along the specified axis
max	Return the maximum value along the specified axis
mean	Compute the arithmetic mean along the specified axis
var	Compute the variance (average squared deviation)
std	Compute the standard deviation (square root of variance)
ptp	Compute the range (max - min) of values along the axis
all	Return `True` if all elements evaluate to `True`
any	Return `True` if any element evaluates to `True`
clip	Limit values to a specified min and max range

BandHiC supports these functions, and they can be used in the following two ways:

1. As methods of the `band_hic_matrix` object:

```
# Compute the sum of all elements including out-of-band values filled with `default_
↪value`.
>>> result0 = mat1.sum()

# Compute the sum of all elements along the `row` axis
>>> result1 = mat1.sum(axis=0)
>>> result1 = mat1.sum(axis='row')

# Compute the sum of all elements along the `diag` axis
>>> result2 = mat1.sum(axis='diag')
```

2. Calling NumPy's functions:

```
# Compute the sum of all elements including out-of-band values filled with `default_
↪value`.
>>> result0 = np.sum(mat1)

# Compute the sum of all elements along the `row` axis
>>> result1 = np.sum(mat1, axis=0)

# Compute the sum of all elements along the `diag` axis
>>> result2 = np.sum(mat1, axis='diag')
```

CHAPTER

THREE

API REFERENCE

This page provides a full API overview of the BandHiC module, including the core band_hic_matrix class and its utilities.

The *bandhic* module defines data structures and utilities for efficiently storing and manipulating Hi-C contact matrices in a banded form. It enables NumPy-compatible operations, matrix masking, diagonal reduction, file I/O, and domain-level interaction analysis.

Core components: - *band_hic_matrix*: A memory-efficient matrix class supporting banded Hi-C data. - Utility functions: Loaders (*cooler_chr*) and constructors (*ones*, *zeros*, etc.).

3.1 *band_hic_matrix* Class API

`class bandhic.`**`band_hic_matrix`**(*contacts*, *diag_num=1*, *mask_row_col=None*, *mask=None*, *dtype=None*, *default_value=0*, *band_data_input=False*)

Symmetric banded matrix stored in upper-triangular format. This storage format is motivated by high-resolution Hi-C data characteristics: 1. Symmetry of contact maps. 2. Interaction frequency concentrated near the diagonal; long-range contacts are sparse (mostly zero). 3. Contact frequency decays sharply with genomic distance.

By storing only the main and a fixed number of super-diagonals as columns of a band matrix (diagonal-major storage: diagonal k stored in column k), we drastically reduce memory usage while enabling random access to Hi-C contacts. Additionally, mask and mask_row_col arrays track invalid or masked contacts to support downstream analysis.

Operations on this band_hic_matrix are as simple as on a numpy.ndarray; users can ignore these storage details.

This class stores only the main diagonal and up to (diag_num - 1) super-diagonals, exploiting symmetry by mirroring values for lower-triangular access.

Parameters

- **`contacts`** (*`coo_array | coo_matrix | tuple | ndarray`*)
- **`diag_num`** (*`int`*)
- **`mask_row_col`** (*`ndarray | None`*)
- **`mask`** (*`Tuple[ndarray, ndarray] | None`*)
- **`dtype`** (*`type | None`*)
- **`default_value`** (*`int | float`*)
- **`band_data_input`** (*`bool`*)

`shape`

Shape of the original full Hi-C contact matrix (bin_num, bin_num), regardless of internal band storage format.

Type
tuple of int

dtype

Data type of the matrix elements, compatible with numpy dtypes.

Type
data-type

diag_num

Number of diagonals stored.

Type
int

bin_num

Number of bins (rows/columns) of the Hi-C matrix.

Type
int

data

Array of shape (*bin_num*, *diag_num*) storing banded Hi-C data.

Type
ndarray

mask

Mask for individual invalid entries. Stored as a boolean ndarray of shape (*bin_num*, *diag_num*) with the same shape as *data*.

Type
ndarray of bool or None

mask_row_col

Mask for entire rows and corresponding columns, indicating invalid bins. Stored as a boolean ndarray of shape (*bin_num*,). For computational convenience, row/column masks are also applied to the *mask* array to track masked entries.

Type
ndarray of bool or None

default_value

Default value for out-of-band entries. Entries out of the banded region and not stored in the data array will be set to this value.

Type
scalar

Examples

```
>>> import bandhic as bh
>>> import numpy as np
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.shape
(4, 4)
```

`__init__`(*contacts*, *diag_num=1*, *mask_row_col=None*, *mask=None*, *dtype=None*, *default_value=0*, *band_data_input=False*)

Initialize a band_hic_matrix instance.

Parameters

- **`contacts`** (*`{coo_array, coo_matrix, tuple, ndarray}`*) – Input Hi-C data in COO format, tuple (data, (row, col)), full square array, or banded stored ndarray. For non-symmetric full arrays, only the upper-triangular part is used and the matrix is symmetrized. Full square arrays are not recommended for large matrices due to memory constraints.
- **`diag_num`** (*`int, optional`*) – Number of diagonals to store. Must be >=1 and <= matrix dimension. Default is 1.
- **`mask_row_col`** (*`ndarray of bool or indices, optional`*) – Mask for invalid rows/columns. Can be specified as: - A boolean array of shape (bin_num,) indicating which rows/columns to mask. - A list of indices to mask. Defaults to None (no masking).
- **`mask`** (*`ndarray pair of (row_indices, col_indices), optional`*) – Mask for invalid matrix entries. Can be specified as: - A tuple of two ndarray (row_indices, col_indices) listing positions to mask. Defaults to None (no masking).
- **`dtype`** (*`data-type, optional`*) – Desired numpy dtype; defaults to 'contacts' data dtype; compatible with numpy dtypes.
- **`default_value`** (*`scalar, optional`*) – Default value for unstored out-of-band entries. Default is 0.
- **`band_data_input`** (*`bool, optional`*) – If True, contacts is treated as precomputed band storage. Default is False.

Raises

`ValueError` – If contacts type is invalid, diag_num out of range, or array shape invalid.

Return type

None

Examples

Initialize from a SciPy COO matrix: >>> import bandhic as bh >>> import numpy as np >>> from scipy.sparse import coo_matrix >>> coo = coo_matrix(([1, 2, 3], ([0, 1, 2],[0, 1, 2])), shape=(3,3)) >>> mat1 = bh.band_hic_matrix(coo, diag_num=2) >>> mat1.data.shape (3, 2)

Initialize from a tuple (data, (row, col)): >>> mat2 = bh.band_hic_matrix(([4, 5, 6], ([0, 1, 2],[2, 1, 0])), diag_num=1) >>> mat2.data.shape (3, 1)

Initialize from a full dense array, only upper-triangular part is stored, lower part is symmetrized: >>> arr = np.arange(16).reshape(4,4) >>> mat3 = bh.band_hic_matrix(arr, diag_num=3) >>> mat3.data.shape (4, 3)

Initialize with row/column mask, this masks entire rows and corresponding columns: >>> mask = np.array([True, False, False, True]) >>> mat4 = bh.band_hic_matrix(arr, diag_num=2, mask_row_col=mask) >>> mat4.mask_row_col array([True, False, False, True])

mask_row_col is also supported as a list of indices: >>> mat4 = bh.band_hic_matrix(arr, diag_num=2, mask_row_col=[0, 3]) >>> mat4.mask_row_col array([True, False, False, True])

Initialize from precomputed banded storage: >>> band = mat3.data.copy() >>> mat5 = bh.band_hic_matrix(band, band_data_input=True) >>> mat5.data.shape (4, 3)

3.1.1 Masking Operations

These methods allow for the application and manipulation of masks within the matrix. Masks can be used to exclude or highlight specific parts of the matrix during computations.

band_hic_matrix.**init_mask**()

Initialize mask for invalid entries based on matrix shape.

Raises
ValueError – If mask is already initialized.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.init_mask()
```

band_hic_matrix.**add_mask**(*row_idx*, *col_idx*)

Add mask entries for specified indices.

Parameters

- **row_idx** (*array-like of int*) – Row indices to mask.
- **col_idx** (*array-like of int*) – Column indices to mask.

Raises
ValueError – If row_idx and col_idx have different shapes.

Return type
None

Examples

```
>>> import bandhic as bh
>>> import numpy as np
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.add_mask([0, 1], [1, 2])
```

band_hic_matrix.**add_mask_row_col**(*mask_row_col*)

Mask entire rows and corresponding columns.

Parameters
mask_row_col (*array-like of int or bool*) – If boolean array of shape (bin_num,), True entries indicate rows/columns to mask. If integer array or sequence, treated as indices of rows/columns to mask.

Raises
ValueError – If integer indices are out of bounds.

Return type
None

Examples

```
>>> import bandhic as bh
>>> mask = np.array([True, False, False])
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> mat.add_mask_row_col(mask)
```

`band_hic_matrix.`**`get_mask`**`()`

Get current mask array.

Returns
Current mask array or None if no mask.

Return type
ndarray or None

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mask = mat.get_mask()
```

`band_hic_matrix.`**`get_mask_row_col`**`()`

Get current row/column mask.

Returns
Current row/column mask or None if no mask.

Return type
ndarray or None

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mask_row_col = mat.get_mask_row_col()
```

`band_hic_matrix.`**`unmask`**`(`*indices=None*`)`

Remove mask entries for specified indices or clear all.

Parameters
`indices` (*`tuple of array-like or None`*) – Tuple (row_idx, col_idx) to unmask, or None to clear all.

Raises
`Warning` – If no mask exists when trying to remove.

Return type
`None`

Notes

If *indices* is None, this will clear all masks (both entry-level and row/column), equivalent to *clear_mask()*.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> mat.unmask(( [0],[0] ))
>>> mat.unmask()
```

`band_hic_matrix.`**`clear_mask`**`()`

Clear all masks (both entry-level and row/column-level).

Return type
`None`

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.clear_mask()
```

`band_hic_matrix.`**`drop_mask`**`()`

Clear the current mask by entry-level, but retain the row/column mask.

Return type
`None`

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2, mask=(np.array([[0, 1], [1,␣
↪2]])))
>>> mat.drop_mask()
```

Notes

This method is useful when you want to keep the row/column mask but clear the entry-level mask. It will not affect the row/column mask, allowing you to maintain the masking of entire rows and columns. >>> mat.mask

`band_hic_matrix.`**`drop_mask_row_col`**`()`

Clear the current row/column mask.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.drop_mask_row_col()
```

`band_hic_matrix.`**`count_masked`**`()`

Count the number of masked entries in the banded matrix. This counts the number of entries in the upper triangular part of the matrix, excluding the diagonal and valid entries. :returns: Number of valid entries in the banded matrix. :rtype: int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_masked()
0
```

`band_hic_matrix.`**`count_unmasked`**`()`

Count the number of unmasked entries in the banded matrix.

Returns
Number of unmasked entries in the banded matrix.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_unmasked()
16
```

band_hic_matrix.**count_in_band_masked**()

Count the number of masked entries in the in-band region.

Returns
Number of masked entries in the in-band region.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_in_band_masked()
0
```

band_hic_matrix.**count_out_band_masked**()

Count the number of masked entries in the out-of-band region.

Returns
Number of masked entries in the out-of-band region.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_out_band_masked()
0
```

band_hic_matrix.**count_in_band**()

Count the number of valid entries in the in-band region.

Returns
Number of valid entries in the in-band region.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_in_band()
10
```

`band_hic_matrix.`**`count_out_band`**`()`

Count the number of valid entries in the out-of-band region.

Returns
Number of valid entries in the out-of-band region.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2)
>>> mat.count_out_band()
6
```

3.1.2 Data Indexing and Modification

The following methods provide functionality to access, modify, and index the matrix data. These are essential for manipulating individual elements or subsets of the matrix.

`band_hic_matrix.`**`__getitem__`**`(`*index*`)`

Retrieve matrix entries or submatrix using NumPy-like indexing.

Supports: - Integer indexing: mat[i, j] returns a single value. - Slice indexing: mat[i:j, i:j] returns a band_hic_matrix for square slices. - Single-axis slice: mat[i:j] returns a band_hic_matrix same as mat[i:j, i:j]. - Fancy (array) indexing: mat[[i1, i2], [j1, j2]] returns an ndarray or MaskedArray according to *mask*. - Mixed indexing: combinations of integer, slice, and array-like indices. - Boolean indexing: 'band_hic_matrix' object with dtype *bool* can be used to index entries.

When both row and column indices specify the same slice (or a single slice is provided), a new band_hic_matrix representing that square submatrix is returned. New submatrix is the view of the original matrix, sharing the same data and mask. If the mask or data is altered in the submatrix, the original matrix will reflect those changes as well. If a single integer index is provided for both row and column, a scalar value is returned. If a mask is set, masked entries will return as *numpy.ma.masked* for scalars, or as a *numpy.ma.MaskedArray* for arrays. If a mask is not set, the scalar value is returned directly, or a numpy.ndarray for arrays. If a square slice is provided, a new band_hic_matrix is returned with the same diagonal number and shape as the original matrix. If a single slice is provided, it returns a band_hic_matrix with the same diagonal number and shape as the original matrix. If fancy indexing is used, it returns a numpy.ndarray or numpy.ma.MaskedArray depending on whether the mask is set. In all other cases, a numpy.ndarray (if no mask) or numpy.ma.MaskedArray (if mask present) is returned.

Parameters
`index` (*`int, slice,`* `band_hic_matrix`*`, array-like of int, or tuple of these`*) – Index expression for rows and columns. May be: - A pair *(row_idx, col_idx)* of ints, slices, or array-like for mixed indexing. - A single slice selecting a square region. - A *band_hic_matrix* object with dtype of *bool* for boolean indexing.

Returns

- scalar : when both row and column are integer indices.
- numpy.ndarray : for fancy or mixed indexing without mask.
- numpy.ma.MaskedArray : for fancy or mixed indexing when mask is set.
- band_hic_matrix : for square slice results.

Return type
scalar or ndarray or MaskedArray or *band_hic_matrix*

Raises

`ValueError` – If a slice step is not 1, or if indices are out of bounds.

Examples

```
>>> import numpy as np
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(16).reshape(4,4), diag_num=2)
```

# Single-element access (scalar) >>> mat[1, 2] 6

# Masked element returns masked >>> mat2 = bh.band_hic_matrix(np.eye(4), dtype=int, diag_num=2, mask=([0],[1])) >>> mat2[0, 1] masked

# Square submatrix via two-slice indexing returns band_hic_matrix >>> sub = mat[1:3, 1:3] >>> isinstance(sub, bh.band_hic_matrix) True

# Single-axis slice returns band_hic_matrix for square region >>> sub2 = mat[0:2] # equivalent to mat[0:2, 0:2] >>> isinstance(sub2, bh.band_hic_matrix) True

# Fancy indexing returns ndarray or MaskedArray >>> arr = mat[[0,2,3], [1,2,0]] >>> isinstance(arr, np.ndarray) True

```
>>> mat.add_mask([0,1],[1,2])  # Add mask to some entries
>>> masked_arr = mat[[0,1], [1,2]]
>>> isinstance(masked_arr, np.ma.MaskedArray)
True
```

# Boolean indexing with band_hic_matrix >>> mat3 = bh.band_hic_matrix(np.eye(4), diag_num=2, mask=([0,1],[1,2])) >>> bool_mask = mat3 > 0 # Create a boolean mask >>> result = mat3[bool_mask] # Use boolean mask for indexing >>> isinstance(result, np.ma.MaskedArray) True

```
>>> result
masked_array(data=[1.0, 1.0, 1.0, 1.0],
            mask=[False, False, False, False],
    fill_value=0.0)
```

`band_hic_matrix.`**`__setitem__`**(*index*, *values*)

Assign values to matrix entries using NumPy-like indexing.

Parameters

- **`index`** (*`int, tuple of (row_idx, col_idx), slice, or band_hic_matrix`*) – Index expression for rows and columns. May be: - A single integer for both row and column. - A tuple of row and column indices (can be int, slice, or array-like). - A single slice selecting a square region. - A *band_hic_matrix* object with dtype of *bool* for boolean indexing.
- **`values`** (*`scalar or array-like`*) – Values to assign. Can be a single scalar or an array-like object.

Raises

- **`ValueError`** – If index is a slice with step not equal to 1, or if indices exceed matrix dimensions.
- **`TypeError`** – If *values* is not a scalar or array-like object.
- **`Supports:`** –
- **`- Integer indexing`** – mat[i, j] = value assigns to a single element.:

- **- Slice indexing** – mat[i:j, i:j] = array or scalar assigns to a square submatrix.:
- **- Single-axis slice** – mat[i:j] = … is equivalent to mat[i:j, i:j].:
- **- Fancy (array) indexing** – mat[[i1, i2], [j1, j2]] = array or scalar for scattered assignments.:
- **- Mixed indexing** – combinations of integer, slice, and array-like indices.:
- **- Boolean indexing** – boolean mask (another band_hic_matrix with dtype=bool) selects entries to set.:

Return type
`None`

Examples

```
>>> import bandhic as bh
>>> import numpy as np
>>> mat = bh.band_hic_matrix(np.zeros((4,4)), diag_num=2, dtype=int)
```

# Single element assignment >>> mat[1, 2] = 5 >>> mat[1, 2] 5

# Slice assignment to square submatrix >>> mat[0:2, 0:2] = [[1, 2], [2, 4]] >>> mat[0:2, 0:2].todense() array([[1, 2],

[2, 4]])

# Single-axis slice assignment (equivalent square slice) >>> mat[2:4] = 0 >>> mat[2:4].todense() array([[0, 0],

[0, 0]])

# Fancy indexing for scattered assignments >>> mat[[0, 3], [1, 2]] = [7, 8] >>> mat[0, 1], mat[3, 2] (7, 8)

# Boolean mask assignment >>> mat2 = bh.band_hic_matrix(np.eye(4), diag_num=2, dtype=int) >>> bool_mask = mat2 > 0 >>> mat2[bool_mask] = 9 >>> mat2.todense() array([[9, 0, 0, 0],

[0, 9, 0, 0], [0, 0, 9, 0], [0, 0, 0, 9]])

Notes

- Assigning to masked entries updates underlying data but does not automatically unmask.
- For multidimensional assignments, scalar values broadcast to all selected positions, while array values must match the number of targeted elements.
- If a boolean mask is used, it must be a *band_hic_matrix* with dtype *bool* and the same shape as the original matrix.
- If a single slice is provided, it behaves like mat[i:j, i:j] for square submatrices.

`band_hic_matrix.`**`get_values`**`(`*row_idx*, *col_idx*`)`

Retrieve values considering mask.

Parameters

- **row_idx** (*array-like of int*) – Row indices.
- **col_idx** (*array-like of int*) – Column indices.

Returns
Retrieved values; masked entries yield masked results.

Return type
ndarray or MaskedArray

Raises
ValueError – If indices exceed matrix dimensions.

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat.get_values([0,1],[1,2])
array([1., 1.])
```

band_hic_matrix.**set_values**(*row_idx*, *col_idx*, *values*)

Set values at specified row and column indices.

Parameters

- **row_idx** (*array-like of int*) – Row indices where values will be set.
- **col_idx** (*array-like of int*) – Column indices where values will be set.
- **values** (*scalar or array-like*) – Values to assign.

Return type
None

Examples

```
>>> import bandhic as bh
>>> mat = bh.zeros((3,3), diag_num=2, dtype=int)
>>> mat.set_values([0,1], [1,2], [4,5])
>>> mat[0,1]
4
```

Notes

Writing to masked positions will update the underlying data but will not clear the mask.

band_hic_matrix.**filled**(*fill_value=None*, *copy=True*)

Fill masked entries in data with default value.

Parameters

- **fill_value** (*scalar*) – Value to assign to masked entries.
- **copy** (*bool*)

Raises
ValueError – If no mask is initialized.

Return type
band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(4), diag_num=2, mask = ([0,1],[1,2]))
>>> mat.filled()
    band_hic_matrix(shape=(4, 4), diag_num=2, dtype=float64)
```

`band_hic_matrix.`**`diag`**(*k*)

Retrieve the k-th diagonal from the matrix.

Parameters
`k` (*`int`*) – The diagonal index to retrieve.

Returns
The k-th diagonal values; masked if mask is set.

Return type
ndarray or MaskedArray

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat.diag(1)
array([1., 1.])
```

`band_hic_matrix.`**`set_diag`**(*k*, *values*)

Set values in the k-th diagonal of the matrix.

Parameters

- **`k`** (*`int`*) – The diagonal index to set.
- **`values`** (*`array-like`*) – The values to set in the diagonal.

Raises
`ValueError` – If k is out of range or values length mismatch.

Return type
`None`

Examples

```
>>> import bandhic as bh
>>> mat = bh.zeros((4,4), diag_num=3)
>>> mat.set_diag(1, [9,9,9])
>>> mat.diag(1)
array([9., 9., 9.])
```

3.1.3 Data Reduction Methods

These methods enable various data reduction operations, such as summing, averaging, or aggregating matrix values along specific axes or dimensions.

`band_hic_matrix.`**`min`**(*axis=None*)

Compute the minimum value in the matrix or along a given axis.

Parameters
`axis` (*`None, int, or {'row','col','diag'}, optional`*) – Axis along which to compute the

minimum: - None: compute over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Minimum value(s) along the specified axis.

Return type
scalar or ndarray

Raises
`ValueError` – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.min()  # global minimum
0
>>> mat.min(axis='row')
array([0, 1, 0])
```

`band_hic_matrix.`**`max`**(*axis=None*)

Compute the maximum value in the matrix or along a given axis.

Parameters
`axis` (*`None, int, or {'row','col','diag'}, optional`*) – Axis along which to compute the maximum: - None: compute over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Maximum value(s) along the specified axis.

Return type
scalar or ndarray

Raises
`ValueError` – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.max()  # global maximum
8
>>> mat.max(axis='row')
array([1, 5, 8])
```

`band_hic_matrix.`**`sum`**(*axis=None*)

Compute the sum of the values in the matrix or along a given axis.

Parameters
`axis` (*`None, int, or {'row','col','diag'}, optional`*) – Axis along which to compute the sum: - None: sum over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Sum(s) along the specified axis.

Return type
scalar or ndarray

Raises
ValueError – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.sum()  # sum of all elements
24
>>> mat.sum(axis='row')
array([ 1, 10, 13])
```

band_hic_matrix.**mean**(*axis=None*)

Compute the mean value of the matrix or along a given axis.

Parameters
axis (*None, int, or {'row','col','diag'}, optional*) – Axis along which to compute the mean: - None: mean over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Mean value(s) along the specified axis.

Return type
scalar or ndarray

Raises
ValueError – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.mean()  # mean of all elements
2.6666666666666665
>>> mat.mean(axis='diag')
array([4., 3.])
```

band_hic_matrix.**prod**(*axis=None*)

Compute the product of the values in the matrix or along a given axis.

Parameters
axis (*None, int, or {'row','col','diag'}, optional*) – Axis along which to compute the product: - None: product over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Product(s) along the specified axis.

Return type
scalar or ndarray

Raises
ValueError – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(1, 10).reshape(3,3), diag_num=2)
>>> mat.prod()  # product of all elements
0
>>> mat.prod(axis='row')
array([ 0, 60, 0])
```

band_hic_matrix.**std**(*axis=None*)

Compute the standard deviation of the values in the matrix or along a given axis.

Parameters

axis (*None, int, or {'row','col','diag'}, optional*) – Axis along which to compute the standard deviation: - None: std over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns

Standard deviation(s) along the specified axis.

Return type

scalar or ndarray

Raises

ValueError – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.std()  # std of all elements
2.748737083745107
>>> mat.std(axis='diag')
array([3.26598632, 2.        ])
```

band_hic_matrix.**var**(*axis=None*)

Compute the variance of the values in the matrix or along a given axis.

Parameters

axis (*None, int, or {'row','col','diag'}, optional*) – Axis along which to compute the variance: - None: variance over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns

Variance(s) along the specified axis.

Return type

scalar or ndarray

Raises

ValueError – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.var()  # variance of all elements
7.555555555555555
>>> mat.var(axis='row')
array([ 0.22222222,  2.88888889, 10.88888889])
```

`band_hic_matrix.`**`normalize`**(*inplace=False*)

Normalize each diagonal of the matrix to have zero mean and unit variance. This modifies the matrix in place. :raises UserWarning: If any diagonal has zero standard deviation, it will be set to zero.

Return type
`None`

Parameters
`inplace` (*`bool`*)

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat = mat.normalize()
```

`band_hic_matrix.`**`ptp`**(*axis=None*)

Compute the peak-to-peak (maximum - minimum) value of the matrix or along a given axis.

Parameters
`axis` (*`None, int, or {'row','col','diag'}, optional`*) – Axis along which to compute the peak-to-peak value: - None: ptp over all stored values (and default for missing). - 0 or 'row': per-row reduction. - 1 or 'col': per-column reduction. - 'diag': per-diagonal reduction. Default is None.

Returns
Peak-to-peak value(s) along the specified axis.

Return type
scalar or ndarray

Raises
`ValueError` – If axis is not one of the supported values.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.ptp()  # ptp of all elements
8
>>> mat.ptp(axis='row')
array([1, 4, 8])
```

`band_hic_matrix.`**`all`**(*axis=None*, *banded_only=False*)

Test whether all (or any) array elements along a given axis evaluate to True.

Parameters

- **`axis`** (*`None, int, or {'row','col','diag'}, optional`*) – Axis along which to test.

- **banded_only** (*bool, optional*) – If True, only consider stored band elements; ignore out-of-band values. Default is False.

Returns
Boolean result(s) of the test.

Return type
bool or ndarray

Raises
ValueError – If axis is not supported.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> mat.all()
False
>>> mat.any(axis='diag', banded_only=True)
array([ True, False])
```

band_hic_matrix.**any**(*axis=None*, *banded_only=False*)

Test whether all (or any) array elements along a given axis evaluate to True.

Parameters

- **axis** (*None, int, or {'row','col','diag'}, optional*) – Axis along which to test.
- **banded_only** (*bool, optional*) – If True, only consider stored band elements; ignore out-of-band values. Default is False.

Returns
Boolean result(s) of the test.

Return type
bool or ndarray

Raises
ValueError – If axis is not supported.

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> mat.all()
False
>>> mat.any(axis='diag', banded_only=True)
array([ True, False])
```

band_hic_matrix.**__contains__**(*item*)

Check whether a value exists in the band_hic_matrix.

Parameters
item (*scalar*) – Value to check for membership in the matrix (including masked values if present).

Returns
True if *item* is present in the stored matrix (ignoring masked entries), False otherwise.

Return type
bool

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> 1 in mat
True
>>> 99 in mat
False
```

3.1.4 Universal Functions

These methods allow for the application of universal functions (ufuncs) to the matrix, enabling element-wise operations similar to those in NumPy. .. automethod:: numpy.absolute .. automethod:: numpy.add .. automethod:: numpy.arccos .. automethod:: numpy.arccosh .. automethod:: numpy.arcsin .. automethod:: numpy.arcsinh .. automethod:: numpy.arctan .. automethod:: numpy.arctan2 .. automethod:: numpy.arctanh .. automethod:: numpy.bitwise_and .. automethod:: numpy.bitwise_or .. automethod:: numpy.bitwise_xor .. automethod:: numpy.cbrt .. automethod:: numpy.conj .. automethod:: numpy.conjugate .. automethod:: numpy.cos .. automethod:: numpy.cosh .. automethod:: numpy.deg2rad .. automethod:: numpy.degrees .. automethod:: numpy.divide .. automethod:: numpy.divmod .. automethod:: numpy.equal .. automethod:: numpy.exp .. automethod:: numpy.exp2 .. automethod:: numpy.expm1 .. automethod:: numpy.fabs .. automethod:: numpy.float_power .. automethod:: numpy.floor_divide .. automethod:: numpy.fmod .. automethod:: numpy.gcd .. automethod:: numpy.greater .. automethod:: numpy.greater_equal .. automethod:: numpy.heaviside .. automethod:: numpy.hypot .. automethod:: numpy.invert .. automethod:: numpy.lcm .. automethod:: numpy.left_shift .. automethod:: numpy.less .. automethod:: numpy.less_equal .. automethod:: numpy.log .. automethod:: numpy.log1p .. automethod:: numpy.log2 .. automethod:: numpy.log10 .. automethod:: numpy.logaddexp .. automethod:: numpy.logaddexp2 .. automethod:: numpy.logical_and .. automethod:: numpy.logical_or .. automethod:: numpy.logical_xor .. automethod:: numpy.maximum .. automethod:: numpy.minimum .. automethod:: numpy.mod .. automethod:: numpy.multiply .. automethod:: numpy.negative .. automethod:: numpy.not_equal .. automethod:: numpy.positive .. automethod:: numpy.power .. automethod:: numpy.rad2deg .. automethod:: numpy.radians .. automethod:: numpy.reciprocal .. automethod:: numpy.remainder .. automethod:: numpy.right_shift .. automethod:: numpy.rint .. automethod:: numpy.sign .. automethod:: numpy.sin .. automethod:: numpy.sinh .. automethod:: numpy.sqrt .. automethod:: numpy.square .. automethod:: numpy.subtract .. automethod:: numpy.tan .. automethod:: numpy.tanh .. automethod:: numpy.true_divide

3.1.5 Other Methods

This section includes additional methods that do not fall into the categories above but provide other useful operations for *bandhic.band_hic_matrix*.

`band_hic_matrix.`**`clip`**(*min_val*, *max_val*)

Clip data values to given range.

Parameters

- **`min_val`** (*`scalar`*)
- **`max_val`** (*`scalar`*)

Return type
`None`

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat = mat.clip(0, 10)
```

`band_hic_matrix.`**`todense`**`()`

Convert the band matrix to a dense format.

Returns
The dense (square) matrix. Masked if mask is set.

Return type
ndarray or MaskedArray

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> dense = mat.todense()
>>> dense.shape
(3, 3)
```

`band_hic_matrix.`**`tocoo`**`(`*drop_zeros=True*`)`

Convert the matrix to COO format.

Parameters
drop_zeros (*bool, optional*) – If True, zero entries will be dropped from the COO format. Default is True.

Returns
The matrix in scipy COO sparse format.

Return type
coo_array

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> coo = mat.tocoo()
>>> coo.shape
(3, 3)
```

`band_hic_matrix.`**`tocsr`**`()`

Convert the matrix to CSR format.

Returns
The matrix in scipy CSR sparse format.

Return type
csr_array

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> csr = mat.tocsr()
>>> csr.shape
(3, 3)
```

band_hic_matrix.**copy**()

Deep copy the object.

Returns
A deep copy.

Return type
band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat2 = mat.copy()
```

band_hic_matrix.**memory_usage**()

Compute memory usage of *band_hic_matirx* object.

Returns
Size in bytes.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat.memory_usage()
772
```

band_hic_matrix.**astype**(*dtype*, *copy=False*)

Cast data to new dtype.

Parameters

- **type** (*data-type*) – Target dtype.
- **copy** (*bool*) – If True, the operation is performed in place. Default is False.
- **dtype** (*type*)

Return type
band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat = mat.astype(np.float32)
```

`band_hic_matrix.__repr__()`

Return a string representation of the band_hic_matrix object.

Returns

A string representation of the object.

Return type

str

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> repr(mat)
    "band_hic_matrix(shape=(3, 3), diag_num=2, dtype=<class 'numpy.float64'>)"
```

`band_hic_matrix.__str__()`

Return a string representation of the band_hic_matrix object.

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> print(mat)
    band_hic_matrix(shape=(3, 3), diag_num=2, dtype=<class 'numpy.float64'>)
```

`band_hic_matrix.__len__()`

Return the number of rows in the band_hic_matrix object.

Returns

Number of rows.

Return type

int

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> len(mat)
3
```

`band_hic_matrix.__iter__()`

Iterate over diagonals of the matrix.

Yields

ndarray – Values of each diagonal.

Return type

`Iterator[ndarray]`

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
```

(continues on next page)

(continued from previous page)

```
>>> for band in mat:
...     print(band)
[1. 1. 1.]
[1. 1.]
```

`band_hic_matrix.`**`__hash__`**`()`

Return a hash value for the band_hic_matrix object.

Returns
Hash value.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> hashed=hash(mat)
```

`band_hic_matrix.`**`__bool__`**`()`

Truth value of the band_hic_matrix, following NumPy semantics.

Returns
If the matrix has exactly one element (shape (1,1)), returns its truth value; otherwise raises a ValueError.

Return type
bool

Raises
`ValueError` – If the matrix contains more than one element.

`band_hic_matrix.`**`__array__`**`(`*copy=False*`)`

Return the data as a NumPy array.

Parameters
copy (*`bool, optional`*) – If True, returns a copy. Default is False.

Returns
Underlying band data array.

Return type
ndarray

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> arr = np.array(mat)
```

`band_hic_matrix.`**`__array_priority__`**`()`

Return the priority for array operations.

Returns
Priority value.

Return type
int

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(3), diag_num=2)
>>> mat.__array_priority__()
100
```

`band_hic_matrix.`**`iterwindows`**`(`*width*, *step=1*`)`

Iterate over the diagonals of the matrix with a specified window size.

Parameters

- **`width`** (*`int`*) – The size of the window to iterate over.
- **`step`** (*`int, optional`*) – Step size between windows. Default is 1.

Yields
band_hic_matrix – The values in the current window.

Return type
`Iterator[`*`band_hic_matrix`*`]`

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> for win in mat.iterwindows(2):
...     print(win)
band_hic_matrix(shape=(2, 2), diag_num=2, dtype=<class 'numpy.float64'>)
band_hic_matrix(shape=(2, 2), diag_num=2, dtype=<class 'numpy.float64'>)
```

`band_hic_matrix.`**`iterrows`**`()`

Iterate over the rows of the band_hic_matrix object.

Yields
ndarray – The values in the current row.

Return type
`Iterator[ndarray]`

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> for row in mat.iterrows():
...     print(row)
[1. 1. 1.]
[1. 1. 1.]
[1. 1. 1.]
```

`band_hic_matrix.`**`itercols`**`()`

Iterate over the columns of the band_hic_matrix object.

Yields
ndarray – The values in the current column.

Return type
Iterator[ndarray]

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> for col in mat.itercols():
...     print(col)
[1. 1. 1.]
[1. 1. 1.]
[1. 1. 1.]
```

band_hic_matrix.**dump**(*filename*)

Save the band_hic_matrix object to a file.

Parameters
filename (*str*) – The name of the file to save the object to.

Return type
None

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((3,3), diag_num=2)
>>> mat.dump('myfile.npz')
```

band_hic_matrix.**extract_row**(*idx*, *extract_out_of_band=True*)

Extract stored, unmasked band values for a given row or column.

Parameters

- **idx** (*int*) – Row (or column, due to symmetry) index for which to extract band values.
- **extract_out_of_band** (*bool, optional*) – If True, include out-of-band entries filled with *default_value* and masked if appropriate. If False (default), return only the stored band values.

Returns
If *extract_out_of_band=False*, a 1D array of length up to *diag_num* containing band values. If *extract_out_of_band=True*, a 1D array of length *bin_num* with all row/column values.

Return type
ndarray or MaskedArray

Raises
ValueError – If *idx* is outside the range [0, bin_num-1].

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.arange(9).reshape(3,3), diag_num=2)
>>> mat.extract_row(0, extract_out_of_band=False)
array([0, 1])
>>> mat.extract_row(0)
array([0, 1, 0])
```

3.2 Create Functions

bandhic.ones(shape[, diag_num, dtype])	Create a band_hic_matrix object filled with ones.
bandhic.zeros(shape[, diag_num, dtype])	Create a band_hic_matrix object filled with zeros.
bandhic.eye(shape[, diag_num, dtype])	Create a band_hic_matrix object filled as an identity matrix.
bandhic.full(shape, fill_value[, diag_num, ...])	Create a band_hic_matrix object filled with a specified value.
bandhic.ones_like(other[, dtype])	Create a band_hic_matrix object matching another matrix, filled with ones.
bandhic.zeros_like(other[, dtype])	Create a band_hic_matrix object matching another matrix, filled with zeros.
bandhic.eye_like(other[, dtype])	Create a band_hic_matrix object matching another matrix, filled as an identity matrix.
bandhic.full_like(other, fill_value[, dtype])	Create a band_hic_matrix object matching another matrix, filled with a specified value.

3.2.1 bandhic.ones

bandhic.**ones**(*shape*, *diag_num=1*, *dtype=<class 'numpy.float64'>*)

Create a band_hic_matrix object filled with ones.

Parameters

- **shape** (*tuple of int*) – Matrix shape as (bins, bins).
- **diag_num** (*int, optional*) – Number of diagonals to consider. Default is 1.
- **dtype** (*data-type, optional*) – The data type of the matrix. Default is np.float64.

Returns

A band_hic_matrix object with all entries filled with ones.

Return type

band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.ones((5, 5), diag_num=3)
>>> print(mat)
band_hic_matrix(shape=(5, 5), diag_num=3, dtype=<class 'numpy.float64'>)
```

3.2.2 bandhic.zeros

bandhic.**zeros**(*shape*, *diag_num=1*, *dtype=<class 'numpy.float64'>*)

Create a band_hic_matrix object filled with zeros.

Parameters

- **shape** (*tuple of int*) – Matrix shape as (bins, bins).
- **diag_num** (*int, optional*) – Number of diagonals to consider. Default is 1.
- **dtype** (*data-type, optional*) – The data type of the matrix. Default is np.float64.

Returns
A band_hic_matrix object with all entries filled with zeros.

Return type
band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.zeros((5, 5), diag_num=3)
>>> print(mat)
band_hic_matrix(shape=(5, 5), diag_num=3, dtype=<class 'numpy.float64'>)
```

3.2.3 bandhic.eye

`bandhic.eye`(*shape*, *diag_num=1*, *dtype=<class 'numpy.float64'>*)

Create a band_hic_matrix object filled as an identity matrix.

Parameters

- **shape** (*tuple of int*) – Matrix shape as (bins, bins).
- **diag_num** (*int, optional*) – Number of diagonals to consider. Default is 1.
- **dtype** (*data-type, optional*) – The data type of the matrix. Default is np.float64.

Returns
A band_hic_matrix object with ones on the main diagonal and zeros elsewhere.

Return type
band_hic_matrix

Examples

```
>>> mat = eye((5, 5), diag_num=3)
>>> print(mat)
band_hic_matrix(shape=(5, 5), diag_num=3, dtype=<class 'numpy.float64'>)
```

3.2.4 bandhic.full

`bandhic.full`(*shape*, *fill_value*, *diag_num=1*, *dtype=<class 'numpy.float64'>*)

Create a band_hic_matrix object filled with a specified value.

Parameters

- **shape** (*tuple of int*) – Matrix shape as (bins, bins).
- **fill_value** (*scalar*) – Value to fill the matrix with.
- **diag_num** (*int, optional*) – Number of diagonals to consider. Default is 1.
- **dtype** (*data-type, optional*) – The data type of the matrix. Default is np.float64.

Returns
A band_hic_matrix object with all entries filled with *fill_value*.

Return type
band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.full((5, 5), fill_value=7, diag_num=3)
>>> print(mat)
band_hic_matrix(shape=(5, 5), diag_num=3, dtype=<class 'numpy.float64'>)
```

3.2.5 bandhic.ones_like

`bandhic.`**`ones_like`**(*other*, *dtype=None*)

Create a band_hic_matrix object matching another matrix, filled with ones.

Parameters

`other` (`band_hic_matrix`) – Reference matrix.

Returns

A band_hic_matrix object matching *other*, filled with ones.

Return type

band_hic_matrix

Examples

```
>>> mat_ref = zeros((4, 4), diag_num=2)
>>> mat = ones_like(mat_ref)
>>> isinstance(mat, band_hic_matrix)
True
```

3.2.6 bandhic.zeros_like

`bandhic.`**`zeros_like`**(*other*, *dtype=None*)

Create a band_hic_matrix object matching another matrix, filled with zeros.

Parameters

`other` (`band_hic_matrix`) – Reference matrix.

Returns

A band_hic_matrix object matching *other*, filled with zeros.

Return type

band_hic_matrix

Examples

```
>>> mat_ref = zeros((4, 4), diag_num=2)
>>> mat = zeros_like(mat_ref)
>>> isinstance(mat, band_hic_matrix)
True
```

3.2.7 bandhic.eye_like

`bandhic.`**`eye_like`**(*other*, *dtype=None*)

Create a band_hic_matrix object matching another matrix, filled as an identity matrix.

Parameters

`other` (`band_hic_matrix`) – Reference matrix.

Returns
A band_hic_matrix object matching *other*, filled as an identity matrix.

Return type
band_hic_matrix

Examples

```
>>> mat_ref = zeros((4, 4), diag_num=2)
>>> mat = eye_like(mat_ref)
>>> isinstance(mat, band_hic_matrix)
True
```

3.2.8 bandhic.full_like

`bandhic.`**`full_like`**(*other*, *fill_value*, *dtype=None*)

Create a band_hic_matrix object matching another matrix, filled with a specified value.

Parameters

- **`other`** (`band_hic_matrix`) – Reference matrix.
- **`fill_value`** (*`scalar`*) – Value to fill the matrix.

Returns
A band_hic_matrix object matching *other*, filled with *fill_value*.

Return type
band_hic_matrix

Examples

```
>>> mat_ref = zeros((4, 4), diag_num=2)
>>> mat = full_like(mat_ref, fill_value=9)
>>> isinstance(mat, band_hic_matrix)
True
```

3.3 Input/Output Functions

`bandhic.save_npz`(file_name, mat)	Save a band_hic_matrix to a .npz file.
`bandhic.load_npz`(file_name)	Load a band_hic_matrix from a .npz file.
`bandhic.straw_chr`(hic_file, chrom, ...[, ...])	Read Hi-C data from a .hic file and return a band_hic_matrix.
`bandhic.straw_all_chrs`(hic_file, resolution, ...)	Read Hi-C data from a .hic file for all chromosomes and return a dictionary of band_hic_matrix objects.
`bandhic.cooler_chr`(file_path, chrom, diag_num)	Read Hi-C data from a .cool or .mcool file and return a band_hic_matrix.
`bandhic.cooler_all_chrs`(file_path, diag_num)	Read Hi-C data from a .cool or .mcool file for all chromosomes and return a dictionary of band_hic_matrix objects.
`bandhic.cooler_chr_all_cells`(file_path, ...)	Read Hi-C data from a .scool file for a specific chromosome and return a dictionary of band_hic_matrix objects for all cells.

continues on next page

Table 2 – continued from previous page

bandhic.cooler_all_cells_all_chrs(file_path, ...)	Read Hi-C data from a .scool file for all cells and return a dictionary of dictionaries of band_hic_matrix objects.

3.3.1 bandhic.save_npz

bandhic.**save_npz**(*file_name*, *mat*)

Save a band_hic_matrix to a .npz file.

Parameters

- **file_name** (*str*) – Path to save the .npz file.
- **mat** (band_hic_matrix) – The band_hic_matrix object to save.

Return type

None

Examples

```
>>> import bandhic as bh
>>> mat = bh.band_hic_matrix(np.eye(5), diag_num=3)
>>> save_npz('./test/sample.npz', mat)
```

3.3.2 bandhic.load_npz

bandhic.**load_npz**(*file_name*)

Load a band_hic_matrix from a .npz file.

Parameters

file_name (*str*) – Path to the .npz file.

Returns

A band_hic_matrix object loaded from the file.

Return type

band_hic_matrix

Examples

```
>>> import bandhic as bh
>>> mat = bh.load_npz('./test/sample.npz')
>>> isinstance(mat, band_hic_matrix)
True
```

3.3.3 bandhic.straw_chr

bandhic.**straw_chr**(*hic_file*, *chrom*, *resolution*, *diag_num*, *data_type='observed'*, *normalization='NONE'*, *unit='BP'*)

Read Hi-C data from a .hic file and return a band_hic_matrix.

Parameters

- **hic_file** (*str*) – Path to the .hic file. This file should be in the Hi-C format compatible with hicstraw. Local or remote paths are supported.

- **chrom** (*str*) – Chromosome name (e.g., 'chr1', 'chrX'). Short names like '1', 'X' are also accepted.
- **resolution** (*int*) – Resolution of the Hi-C data. Such as 10000 for 10kb resolution.
- **diag_num** (*int*) – Number of diagonals to consider.
- **data_type** (*str, optional*) – Type of data to read from the Hi-C file. Default is 'observed'. Other options include 'expected', 'balanced', etc. See *hicstra`w* documentation for more details.
- **normalization** (*str, optional*) – Normalization method to apply. Default is 'NONE'. Other options include 'VC', 'VC_SQRT', 'KR', 'SCALE', etc. See *hicstraw* documentation for more details.
- **unit** (*str, optional*) – Unit of measurement for the Hi-C data. Default is 'BP' (base pairs). Other options include 'FRAG' (fragments), etc.

Return type

band_hic_matrix

See also

None

URL

https://github.com/aidenlab/straw/tree/master/pybind11_python

Returns

A band_hic_matrix object containing the Hi-C data.

Return type

band_hic_matrix

Raises

ValueError – If the file cannot be parsed or parameters are invalid.

Parameters

- **hic_file** (*str*)
- **chrom** (*str*)
- **resolution** (*int*)
- **diag_num** (*int*)
- **data_type** (*str*)
- **normalization** (*str*)
- **unit** (*str*)

Examples

```
>>> import bandhic as bh
>>> mat = bh.straw_chr('/Users/wwb/Documents/workspace/BandHiC-Master/data/
↪GSE130275_mESC_WT_combined_1.3B_microc.hic', 'chr1', resolution=10000, diag_
↪num=200)
>>> isinstance(mat, band_hic_matrix)
True
```

3.3.4 bandhic.straw_all_chrs

bandhic.**straw_all_chrs**(*hic_file*, *resolution*, *diag_num*, *data_type='observed'*, *normalization='NONE'*, *unit='BP'*)

Read Hi-C data from a .hic file for all chromosomes and return a dictionary of band_hic_matrix objects.

Parameters

- **hic_file** (*str*) – Path to the .hic file. This file should be in the Hi-C format compatible with hicstraw. Local or remote paths are supported.
- **resolution** (*int*) – Resolution of the Hi-C data. Such as 10000 for 10kb resolution.
- **diag_num** (*int*) – Number of diagonals to consider.
- **data_type** (*str, optional*) – Type of data to read from the Hi-C file. Default is 'observed'. Other options include 'expected', 'balanced', etc. See *hicstraw* documentation for more details.
- **normalization** (*str, optional*) – Normalization method to apply. Default is 'NONE'. Other options include 'VC', 'VC_SQRT', 'KR', 'SCALE', etc. See *hicstraw* documentation for more details.
- **unit** (*str, optional*) – Unit of measurement for the Hi-C data. Default is 'BP' (base pairs). Other options include 'FRAG' (fragments), etc.

Returns

A dictionary mapping chromosome names to band_hic_matrix objects containing the Hi-C data.

Return type

Dict[str, *band_hic_matrix*]

Raises

ValueError – If the file cannot be parsed or parameters are invalid.

Examples

```
>>> import bandhic as bh
>>> mats = bh.straw_all_chrs('/Users/wwb/Documents/workspace/BandHiC-Master/data/
↪GSE130275_mESC_WT_combined_1.3B_microc.hic', resolution=10000, diag_num=200)
>>> isinstance(mats['chr1'], band_hic_matrix)
True
```

3.3.5 bandhic.cooler_chr

bandhic.**cooler_chr**(*file_path*, *chrom*, *diag_num*, *cell_id=None*, *resolution=None*, *balance=True*)

Read Hi-C data from a .cool or .mcool file and return a band_hic_matrix.

Parameters

- **file_path** (*str*) – Path to the .cool, .mcool or .scool file.
- **chrom** (*str*) – Chromosome name.
- **diag_num** (*int*) – Number of diagonals to consider.
- **cell_id** (*str, optional*) – Cell ID for .scool files.
- **resolution** (*int, optional*) – Resolution of the Hi-C data.
- **balance** (*bool, optional*) – If True, use balanced data. Default is False. This parameter is specific to cooler files.

Returns
A band_hic_matrix object containing the Hi-C data.

Return type
band_hic_matrix

Raises
ValueError – If the cooler file is invalid or parameters are incorrect.

Examples

```
>>> import bandhic as bh
>>> mat = bh.cooler_chr('/Users/wwb/Documents/workspace/BandHiC-Master/data/yeast.
↪10kb.cool', 'chrI', resolution=10000, diag_num=10)
>>> isinstance(mat, band_hic_matrix)
True
```

See also

None

URL
https://cooler.readthedocs.io/en/latest/index.html

3.3.6 bandhic.cooler_all_chrs

bandhic.**cooler_all_chrs**(*file_path*, *diag_num*, *resolution=None*, *cell_id=None*, *balance=True*)

Read Hi-C data from a .cool or .mcool file for all chromosomes and return a dictionary of band_hic_matrix objects.

Parameters

- **file_path** (*str*) – Path to the .cool, .mcool or .scool file.
- **diag_num** (*int*) – Number of diagonals to consider.
- **resolution** (*int, optional*) – Resolution of the Hi-C data.
- **cell_id** (*str, optional*) – Cell ID for .scool files.
- **balance** (*bool, optional*) – If True, use balanced data. Default is False. This parameter is specific to cooler files.

Returns
A dictionary mapping chromosome names to band_hic_matrix objects containing the Hi-C data.

Return type
Dict[str, *band_hic_matrix*]

Raises
ValueError – If the cooler file is invalid or parameters are incorrect.

Examples

```
>>> import bandhic as bh
>>> mats = bh.cooler_all_chrs('/Users/wwb/Documents/workspace/BandHiC-Master/data/
↪yeast.10kb.cool', diag_num=10, resolution=10000)
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(mats['chrI'], band_hic_matrix)
True
```

3.3.7 bandhic.cooler_chr_all_cells

bandhic.**cooler_chr_all_cells**(*file_path*, *chrom*, *diag_num*, *balance=True*)

Read Hi-C data from a .scool file for a specific chromosome and return a dictionary of band_hic_matrix objects for all cells.

Parameters

- **file_path** (*str*) – Path to the .scool file.
- **chrom** (*str*) – Chromosome name.
- **diag_num** (*int*) – Number of diagonals to consider.
- **balance** (*bool, optional*) – If True, use balanced data. Default is False. This parameter is specific to cooler files.

Returns

A dictionary mapping cell IDs to band_hic_matrix objects for the specified chromosome.

Return type

Dict[str, *band_hic_matrix*]

Raises

ValueError – If the scool file is invalid or parameters are incorrect.

Examples

```
>>> import bandhic as bh
>>> mats = bh.cooler_chr_all_cells('/Users/wwb/Documents/workspace/BandHiC-Master/
↪data/yeast.10kb.scool', 'chrI', diag_num=10, resolution=10000)
>>> isinstance(mats['cell1'], band_hic_matrix)
True
```

3.3.8 bandhic.cooler_all_cells_all_chrs

bandhic.**cooler_all_cells_all_chrs**(*file_path*, *diag_num*, *resolution=None*)

Read Hi-C data from a .scool file for all cells and return a dictionary of dictionaries of band_hic_matrix objects.

Parameters

- **file_path** (*str*) – Path to the .scool file.
- **diag_num** (*int*) – Number of diagonals to consider.
- **resolution** (*int, optional*) – Resolution of the Hi-C data.

Returns

A dictionary mapping cell IDs to dictionaries mapping chromosome names to band_hic_matrix objects.

Return type

Dict[str, Dict[str, *band_hic_matrix*]]

Raises

ValueError – If the scool file is invalid or parameters are incorrect.

Examples

```
>>> import bandhic as bh
>>> mats = bh.cooler_all_cells('/Users/wwb/Documents/workspace/BandHiC-Master/data/
↪yeast.10kb.scool', diag_num=10, resolution=10000)
>>> isinstance(mats['cell1']['chrI'], band_hic_matrix)
True
```

3.4 Other Functions

bandhic.matrix_equal(a, b)	Check if two band_hic_matrix objects are equal.
bandhic.assert_band_matrix_equal(a, b)	Assert that two band_hic_matrix objects are equal.
bandhic.compute_bin_bias(hic_coo[, verbose])	Compute bias values for Hi-C contact matrices using the Knight-Ruiz normalization algorithm.
bandhic.call_tad(hic_matrix, resolution, ...)	Detect TADs from a Hi-C matrix using the TopDom algorithm.
bandhic.topdom(hic_matrix, bins, window_size)	Detect TADs using TopDom algorithm.

3.4.1 bandhic.matrix_equal

bandhic.**matrix_equal**(*a*, *b*)

Check if two band_hic_matrix objects are equal.

Parameters

- **a** (band_hic_matrix) – First band_hic_matrix object.
- **b** (band_hic_matrix) – Second band_hic_matrix object.

Returns

True if the matrices are equal, False otherwise.

Return type

bool

3.4.2 bandhic.assert_band_matrix_equal

bandhic.**assert_band_matrix_equal**(*a*, *b*)

Assert that two band_hic_matrix objects are equal.

Parameters

- **a** (band_hic_matrix) – First band_hic_matrix object.
- **b** (band_hic_matrix) – Second band_hic_matrix object.

Raises

AssertionError – If the two matrices are not equal.

Return type

bool

3.4.3 bandhic.compute_bin_bias

`bandhic.`**`compute_bin_bias`**(*hic_coo*, *verbose=False*)

Compute bias values for Hi-C contact matrices using the Knight-Ruiz normalization algorithm.

Parameters

- **`hic_coo`** (*`scipy.sparse.coo_array`*) – A sparse COO matrix representing Hi-C contact data.
- **`verbose`** (*`bool, optional`*) – If True, print detailed information during processing. Default is False.

Returns

- **bias** (*numpy.ndarray*) – A 1D array containing the bias values for each bin in the Hi-C matrix.
- **is_valid** (*bool*) – A boolean indicating whether the bias vector is valid (mean and median within typical range).

Notes

This function removes a specified percentage of the most sparse bins from the Hi-C matrix before computing the bias values. The Knight-Ruiz normalization algorithm is applied to the modified matrix to compute the bias. The function iteratively removes bins with low interaction counts until a valid bias vector is obtained. The bias vector is expected to have a mean and median close to 1, indicating balanced interaction frequencies across bins.

3.4.4 bandhic.call_tad

`bandhic.`**`call_tad`**(*hic_matrix*, *resolution*, *chrom_short*, *window_size_bp=200000*, *min_TAD_size=None*, *stat_filter=True*, *verbose=False*)

Detect TADs from a Hi-C matrix using the TopDom algorithm. Different from the *TopDom* function, this function accepts a Hi-C matrix in either *band_hic_matrix* format or as a dense numpy array. This function is designed to be used with the BandHiC package and provides a convenient interface for TAD detection. :type hic_matrix: `Union[`*`band_hic_matrix`*`, ndarray]` :param hic_matrix: The Hi-C matrix, either as a band_hic_matrix object or a dense numpy array. :type hic_matrix: band_hic_matrix or np.ndarray :type resolution: `int` :param resolution: The resolution of the Hi-C matrix. :type resolution: int :type chrom_short: `str` :param chrom_short: The chromosome name without ‘chr’ prefix. :type chrom_short: str :type window_size_bp: `int` :param window_size_bp: The size of the window to consider for detecting TADs, in base pairs. Default is 200000. :type window_size_bp: int, optional :type min_TAD_size: `int` :param min_TAD_size: The minimum size of a TAD to be considered valid, in base pairs. Default is None. :type min_TAD_size: int, optional :type stat_filter: `bool` :param stat_filter: Whether to apply statistical filtering to remove false positives. Default is True. :type stat_filter: bool, optional :type verbose: `bool` :param verbose: Whether to print detailed information during processing. Default is False. :type verbose: bool, optional

Returns

- **domains** (*pd.DataFrame*) – DataFrame containing detected TADs with columns ‘chr’, ‘from.id’, ‘from.coord’, ‘to.id’, ‘to.coord’, ‘tag’.
- **bins** (*pd.DataFrame*) – DataFrame containing bin information columns ‘id’, ‘chr’, ‘from.coord’, ‘to.coord’, ‘local.ext’, ‘mean.cf’, ‘pvalue’.

Parameters

- **`hic_matrix`** (`band_hic_matrix` *`| ndarray`*)
- **`resolution`** (*`int`*)
- **`chrom_short`** (*`str`*)

- **window_size_bp** (*int*)
- **min_TAD_size** (*int*)
- **stat_filter** (*bool*)
- **verbose** (*bool*)

3.4.5 bandhic.topdom

bandhic.**topdom**(*hic_matrix*, *bins*, *window_size*, *stat_filter=True*, *verbose=False*)

Detect TADs using TopDom algorithm. :type hic_matrix: :param hic_matrix: The Hi-C matrix, either as a band_hic_matrix object or a dense numpy array. :type hic_matrix: band_hic_matrix or np.ndarray :type bins: :param bins: DataFrame containing bin information with columns 'chr', 'from.coord', 'to.coord'. :type bins: pd.DataFrame :type window_size: :param window_size: Size of the window to consider for detecting TADs. :type window_size: int :type stat_filter: :param stat_filter: Whether to apply statistical filtering to remove false positives. Default is True. :type stat_filter: bool, optional :type verbose: :param verbose: Whether to print detailed information during processing. Default is False. :type verbose: bool, optional

Returns

- **domains** (*pd.DataFrame*) – DataFrame containing detected TADs with columns 'chr', 'from.id', 'from.coord', 'to.id', 'to.coord', 'tag'.
- **bins** (*pd.DataFrame*) – Updated DataFrame containing bin information with additional columns 'local.ext', 'mean.cf', 'pvalue'.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX