
hic3defdr documentation

Release 0.2.1

Thomas Gilgenast

May 03, 2020

Contents

1	hic3defdr	3
2	Conceptual documentation	35
3	Demos	37
4	Changelog	41
5	hic3defdr	45
6	Indices and tables	93
	Python Module Index	95
	Index	97

A genome-scale differential loop finder.

Contents:

A genome-scale differential loop finder.

For the latest source, discussion, etc, please visit the Bitbucket repository at <https://bitbucket.org/creminslab/hic3defdr>

1.1 Installation

We require Python 2.7.11+ or 3.6+ and the dependencies listed in `setup.py`.

A typical quick install process should be:

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ pip install hic3defdr
```

A typical dev-mode install process should be:

```
$ git clone https://bitbucket.org/creminslab/hic3defdr.git
$ cd hic3defdr
$ virtualenv venv
$ source venv/bin/activate
(venv)$ pip install -e .
```

If installation succeeded then `hic3defdr.HiC3DeFDR` should be importable from an interactive shell started in some other directory:

```
(venv)$ cd <some other directory>
(venv)$ python
>>> from hic3defdr import HiC3DeFDR
```

1.1.1 Optional dependencies

Evaluating simulations requires scikit-learn:

```
(venv)$ pip install scikit-learn
```

To display progress bars during selected steps of the analysis, install `tqdm`:

```
(venv)$ pip install tqdm
```

1.2 Basic walkthrough

Before we start, we'll seed numpy's random number generator for reproducibility:

```
>>> import numpy as np
>>> np.random.seed(42)
```

To analyze the ES_1, ES_3, NPC_1, and NPC_2 reps of the dataset dataset from [Bonev et al. 2017](#) with default parameters, we would first describe the dataset in terms of replicate names, chromosome names, and a design matrix. We will just analyze chromosomes 18 and 19 here for illustrative purposes.

```
>>> import pandas as pd
>>>
>>> repnames = ['ES_1', 'ES_3', 'NPC_1', 'NPC_2']
>>> #chroms = ['chr%i' % i for i in range(1, 20)] + ['chrX']
>>> chroms = ['chr18', 'chr19']
>>> design = pd.DataFrame({'ES': [1, 1, 0, 0], 'NPC': [0, 0, 1, 1]},
...                       dtype=bool, index=repnames)
```

If you're following along, you can download the data like this:

```
$ python -m hic3defdr.util.demo_data
```

or from inside an interactive shell:

```
>>> from hic3defdr.util.demo_data import ensure_demo_data
>>> ensure_demo_data()
```

The data will be downloaded to a folder called `hic3defdr-demo-data` under your home directory.

The required input files consist of:

- upper triangular, raw contact matrices in `scipy.sparse NPZ` format,
- bias vectors in plain-text `np.savetxt()` format, and
- loop cluster files in sparse JSON format (see below for more details), specifying the locations of loops present in each condition

TODO: explain how to import data from common Hi-C analysis tools into this format

We would next describe the location of the input data files and use those to construct a `HiC3DeFDR` object:


```
>>> import os.path
>>> from hic3defdr import HiC3DeFDR
>>>
>>> base_path = os.path.expanduser('~/.hic3defdr-demo-data/')
>>> h = HiC3DeFDR(
...     raw_npz_patterns=[base_path + '<rep>/<chrom>_raw.npz'.replace('<rep>',
↳ repname) for repname in repnames],
...     bias_patterns=[base_path + '<rep>/<chrom>_kr.bias'.replace('<rep>', repname)
↳ for repname in repnames],
...     chroms=chroms,
...     design=design,
...     outdir='output',
...     loop_patterns={c: base_path + 'clusters/%s<chrom>_clusters.json' % c for c
↳ in ['ES', 'NPC']}},
...     res=10000
... )
creating directory output
```

This object saves itself to disk, so it can be re-loaded at any time:

```
>>> h = HiC3DeFDR.load('output')
```

To run the analysis for all chromosomes through q-values, run:

```
>>> h.run_to_qvalues()
```

To threshold, cluster, and classify the significantly differential loops, and collect all the results into a single TSV output file, run:

```
>>> h.collect()
```

The output file will be written to `output/results_0.01_3.tsv`, where “output” refers to the `outdir` we passed when constructing the `HiC3DeFDR` object, “0.01” refers to the default FDR of 1%, and “3” refers to the default cluster size threshold of 3.

```
>>> import pandas as pd
>>> pd.read_csv('output/results_0.05_3.tsv', sep='\t', index_col=0).head()
                                us_chrom  ...  classification
loop_id
chr18:3480000-3500000_chr18:4680000-4710000  chr18  ...      constitutive
chr18:3490000-3510000_chr18:3790000-3810000  chr18  ...              ES
chr18:3490000-3510000_chr18:3970000-3990000  chr18  ...      constitutive
chr18:3490000-3510000_chr18:4170000-4200000  chr18  ...      constitutive
chr18:3490000-3520000_chr18:4120000-4150000  chr18  ...      constitutive

[5 rows x 9 columns]
```

See the section “TSV output format” below for more details about the output format.

1.3 Step-by-step walkthrough

Calling `h.run_to_qvalues()` runs the four main steps of the analysis in sequence. These four steps are described in further detail below. Any kwargs passed to `h.run_to_qvalues()` will be passed along to the appropriate step; see `help(HiC3DeFDR.run_to_qvalues)` for details.

1.3.1 Step 1: Preparing input data

The function call `h.prepare_data()` prepares the input raw contact matrices and bias vectors specified by `h.raw_npz_patterns` and `h.bias_patterns` for all chromosomes specified in `h.chroms`, performs library size normalization, and determines what points should be considered for dispersion estimation. This creates intermediate files in the output directory `h.outdir` that represent the raw and scaled values, as well as the estimated size factors, and a boolean vector `disp_idx` indicating which points will be used for dispersion estimation. If `loop_patterns` was passed to the constructor, an additional boolean vector `loop_idx` is created to mark which points that pass `disp_idx` lie within looping interactions specified by `h.loop_patterns`. The raw and scaled data are stored in a rectangular matrix format where each row is a pixel of the contact matrix and each column is a replicate. If the size factors are estimated as a function of distance, the estimated size factors are also stored in this format. Two separate vectors called `row` and `col` are used to store the row and column index of the pixel represented by each row of the rectangular matrices. Together, the `row` and `col` vectors plus any of the rectangular matrices represent a generalization of a COO format sparse matrix to multiple replicates (in the standard COO format the `row` and `col` vectors are complemented by a single `value` vector).

The size factors can be estimated with a variety of methods defined in the `hic3defdr.scaling` module. The method to use is specified by the `norm` kwarg passed to `h.prepare_data()`. Some of these methods estimate size factors as a function of interaction distance, instead of simply estimating one size factor for each replicate as is common in RNA-seq differential expression analysis. When these methods are used, the number of bins to use when binning distances for distance-based estimation of the size factors can be specified with the `n_bins` kwarg. The defaults for this function use the conditional median of ratios approach (`hic3defdr.scaling.conditional_mor`) and an automatically-selected number of bins.

1.3.2 Step 2: Estimating dispersions

The function call `h.estimate_disp()` estimates the dispersion parameters at each distance scale in the data and fits a lowess curve through the graph of distance versus dispersion to obtain final smoothed dispersion estimates for each pixel.

The `estimator` kwarg on `h.estimate_disp()` specifies which dispersion estimation method to use, out of a selection of options defined in the `hic3defdr.dispersion` module. The default is to use quantile-adjusted conditional maximum likelihood (qCML).

1.3.3 Step 3: Likelihood ratio test

The function call `h.lrt()` performs a likelihood ratio test (LRT) for each pixel. This LRT compares a null model of no differential expression (fitting one true mean parameter shared by all replicates irrespective of condition) to an alternative model in which the two conditions have different true mean parameters.

1.3.4 Step 4: False discovery rate (FDR) control

The function call `h.bh()` performs Benjamini-Hochberg FDR correction on the p-values called via the LRT in the previous step, considering only a subset of pixels that are involved in looping interactions (as specified by `h.loop_patterns`; if `loop_patterns` was not passed to the constructor then all pixels are included in the BH-FDR correction). This results in final q-values for each loop pixel.

1.3.5 Thresholding, clustering, and classification

We can threshold, cluster, classify, and collect the significantly differential loops:

```
>>> h.collect(fdr=0.05, cluster_size=3)
```

We can also sweep across FDR and/or cluster size thresholds:

```
>>> h.collect(fdr=[0.01, 0.05], cluster_size=[3, 4])
```

The final output file for each combination of thresholding parameters will be written to `<h.outdir>/results_<fdr>_<cluster_size>.tsv`.

This example walkthrough should take less than 5 minutes for the two chromosomes included in the demo data. Run time for a whole-genome analysis will depend on parallelization as discussed in the next section.

1.4 Parallelization

The functions `run_to_qvalues()`, `prepare_data()`, `lrt()`, `threshold()`, `classify()`, and `simulate()` operate in a per-chromosome manner. By default, each chromosome in the dataset will be processed in series. If multiple cores and sufficient memory are available, you can use the `n_threads` kwarg on these functions to use multiple subprocesses to process multiple chromosomes in parallel. Pass either a desired number of subprocesses to use, or pass `n_threads=-1` to use all available cores. The output logged to `stderr` will be truncated to reduce clutter from multiple subprocesses printing to `stderr` at the same time. This truncation is controlled by the `verbose` kwarg which is available on some of these parallelized functions.

The function `estimate_disp()` also accepts an `n_threads` kwarg, using it to parallelize itself across distance scales.

The function `run_to_qvalues()` passes the `n_threads` kwarg through to all the steps it calls.

1.5 Intermediates and final output files

All intermediates used in the computation will be saved to the disk inside the `outdir` folder as `<intermediate>_<chrom>.npz` (most intermediates), `<intermediate>_<chrom>.json` (thresholded or classified clusters).

Two intermediates are not generated per chromosome. These are `disp_fn_<c>.pickle` (dispersion function estimated across all chromosomes for condition `<c>`) and `results_<f>_<s>.tsv` (final combined results from all chromosomes).

Step	Intermediate	Shape	Description
<code>prepare_data()</code> row		<code>(n_pixels,)</code>	Top-level row index
<code>prepare_data()</code> col		<code>(n_pixels,)</code>	Top-level column index
<code>prepare_data()</code> bias		<code>(n_bins, n_reps)</code>	Bias vectors
<code>prepare_data()</code> raw		<code>(n_pixels, n_reps)</code>	Raw count values
<code>prepare_data()</code> size_factors		<code>(n_reps,)</code> or <code>(n_pixels, n_reps)</code>	Size factors
<code>prepare_data()</code> scaled		<code>(n_pixels, n_reps)</code>	Normalized count values
<code>prepare_data()</code> disp_idx		<code>(n_pixels,)</code>	Marks pixels for which dispersion is fitted
<code>prepare_data()</code> loop_idx		<code>(disp_idx.sum(),)</code>	Marks pixels which lie in loops
<code>estimate_disp()</code> cov_per_bin		<code>(n_bins, n_conds)</code>	Average mean count or distance in each bin
<code>estimate_disp()</code> disp_per_bin		<code>(n_bins, n_conds)</code>	Pooled dispersion estimates in each bin
<code>estimate_disp()</code> disp_fn_<c>		pickled function	Fitted dispersion function
<code>estimate_disp()</code> disp		<code>(disp_idx.sum(), n_conds)</code>	Smoothed dispersion estimates
<code>lrt()</code>	<code>mu_hat_null</code>	<code>(disp_idx.sum(),)</code>	Null model mean parameters
<code>lrt()</code>	<code>mu_hat_alt</code>	<code>(disp_idx.sum(), n_conds)</code>	Alternative model mean parameters
<code>lrt()</code>	<code>llr</code>	<code>(disp_idx.sum(),)</code>	Log-likelihood ratio
<code>lrt()</code>	<code>pvalues</code>	<code>(disp_idx.sum(),)</code>	LRT-based p-value
<code>bh()</code>	<code>qvalues</code>	<code>(loop_idx.sum(),)</code>	BH-corrected q-values
<code>threshold()</code>	<code>sig_<f>_<s></code>	JSON/TSV	Significantly differential clusters
<code>threshold()</code>	<code>insig_<f>_<s></code>	JSON/TSV	Constitutive clusters
<code>classify()</code>	<code><c>_<f>_<s></code>	JSON/TSV	Classified differential clusters
<code>collect()</code>	<code>results_<f>_<s></code>	TSV	Final results table

The table uses these abbreviations to refer to variable parts of certain intermediate names:

- `<f>`: FDR threshold
- `<s>`: cluster size threshold
- `<c>`: condition/class label

1.6 Sparse JSON cluster format

This is the format used both for supplying pre-identified, per-condition loop clusters as input to HiC3DeFDR as well as the format in which differential and constitutive interaction clusters are reported as output.

The format describes the clusters on each chromosome in a separate JSON file. This JSON file contains a single JSON object, which is a list of list of list of integers. The inner lists are all of length 2 and represent specific pixels of the heatmap for that chromosome in terms of their row and column coordinates in zero-indexed bin units. The outer middle lists can be of any length and represent groups of pixels that belong to the same cluster. The length of the outer list corresponds to the number of clusters on that chromosome.

These clusters can be loaded into and written from the corresponding plain Python objects using `hic3defdr.clusters.load_clusters()` and `hic3defdr.clusters.save_clusters()`, respectively. The plain Python objects can be converted to and from scipy sparse matrix objects using `hic3defdr.clusters.clusters_to_coo()` and `hic3defdr.clusters.convert_cluster_array_to_sparse()`, respectively.

1.7 TSV output format

The final TSV output file `results_<f>_<s>.tsv` has as its first column a `loop_id`, a string of the form `<us_chrom>:<us_start>-<us_end>_<ds_chrom>:<ds_start>-<ds_end>` specifying a rectangle that surrounds the cluster of pixels that make up the loop. The next six columns are the six individual pieces of this `loop_id`. This is followed by:

- `cluster_size`: the number of pixels in the cluster
- `cluster`: a list of the exact (row, col) indices of the pixels in the cluster
- `classification`: this will be “constitutive” if the loop is not differential, or the name of the condition the loop is specific to (i.e., strongest in) if it is differential

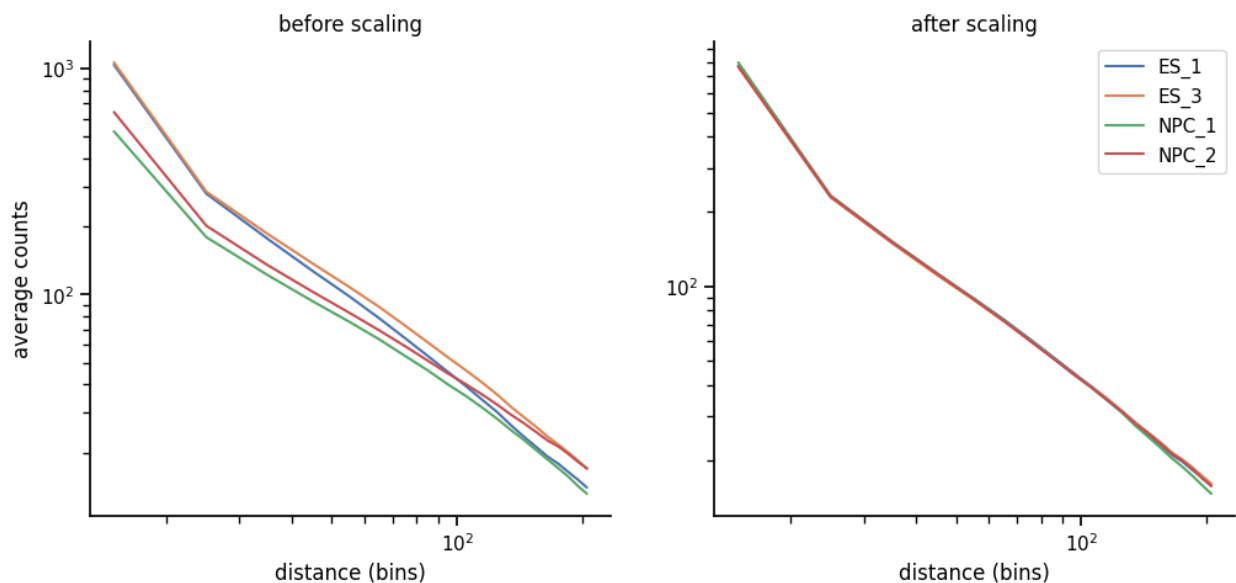
1.8 Visualizations

The `HiC3DeFDR` object can be used to draw visualizations of the analysis.

The visualization functions are wrapped with the `@plotter decorator` and therefore all support the convenience kwargs provided by that decorator (such as `outfile`).

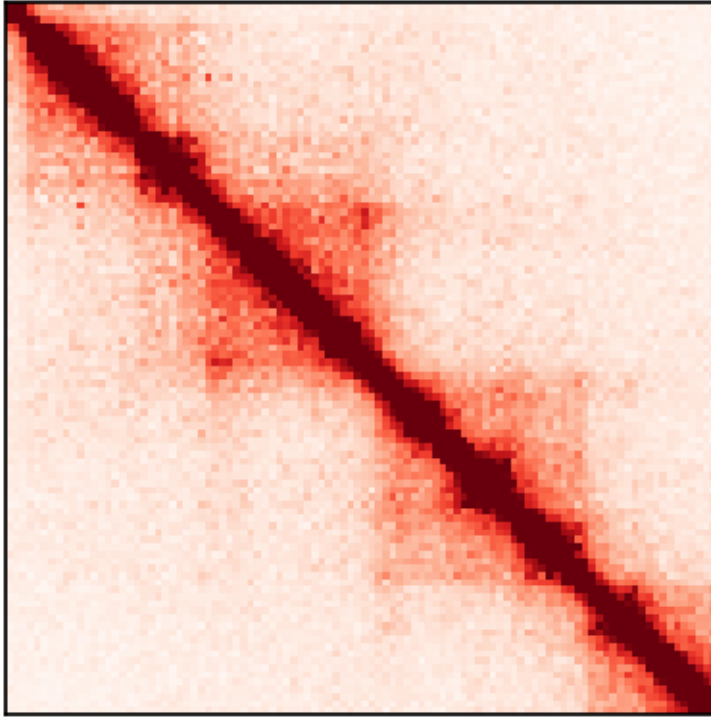
1.8.1 Distance dependence curves before and after scaling

```
>>> _ = h.plot_dd_curves('chr18', outfile='images/dd.png')
```



1.8.2 Simple heatmap plotting

```
>>> _ = h.plot_heatmap('chr18', slice(1000, 1100), slice(1000, 1100), rep='ES_1',
...                   outfile='images/heatmap.png')
```

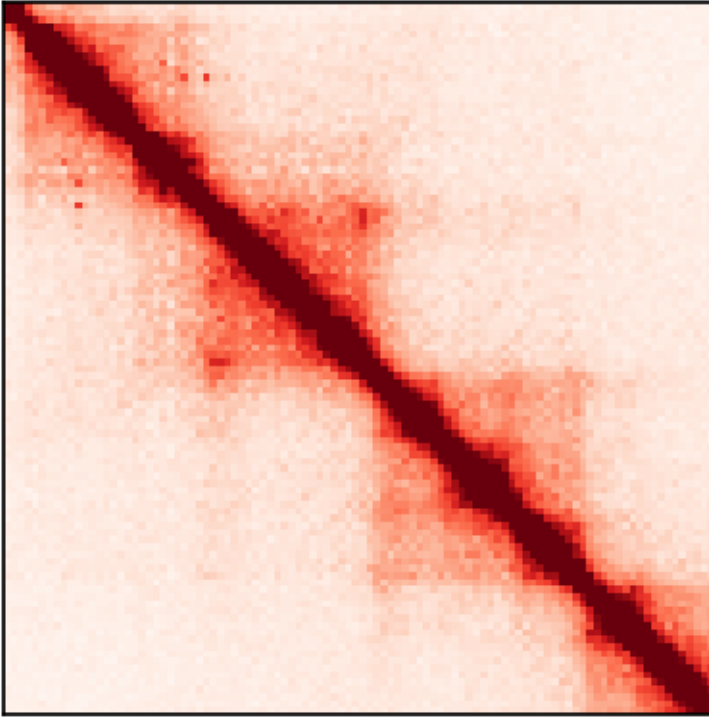


By default, this plots data at the “scaled” stage (normalized for bias and sequencing depth differences), but you can plot any stage of the data by passing a `stage` kwarg.

1.8.3 Condition-average heatmap plotting

To plot a within-condition average heatmap, pass a `stage` name with a ‘_mean’ suffix appended and `cond` to specify the condition to average within:

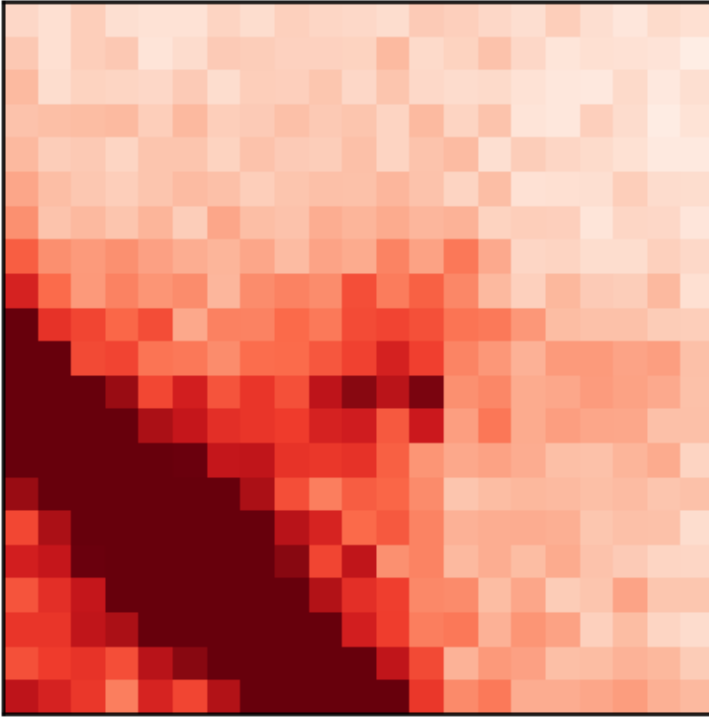
```
>>> _ = h.plot_heatmap('chr18', slice(1000, 1100), slice(1000, 1100),
...                   stage='scaled_mean', cond='ES',
...                   outfile='images/heatmap_mean.png')
```



1.8.4 Loop zoomin heatmap plotting

We can combine `h.plot_heatmap()` with the `load_clusters()` and `cluster_to_slices()` utility functions in `hic3defdr.util.clusters` to plot zoomins around specific loops:

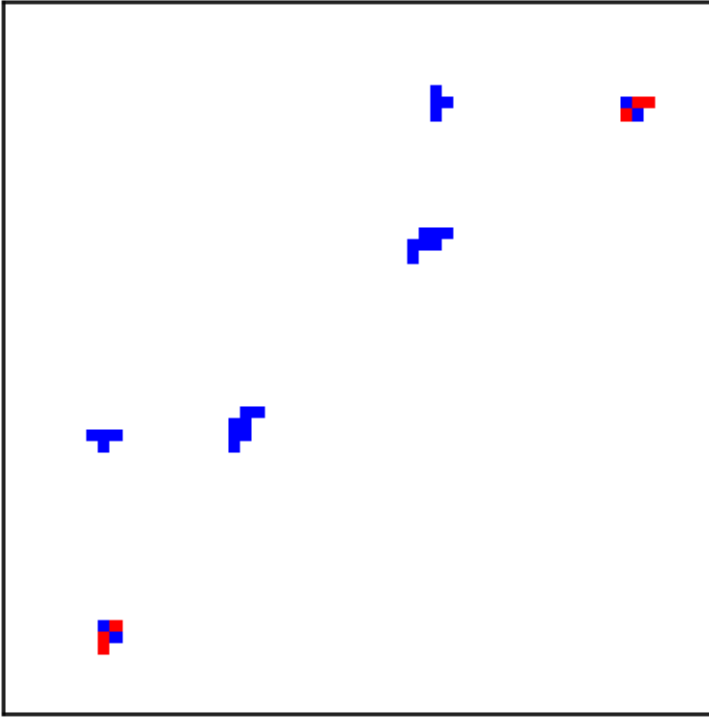
```
>>> from hic3defdr.util.clusters import load_clusters, cluster_to_slices
>>> chrom = 'chr18'
>>> clusters = load_clusters(h.loop_patterns['ES'].replace('<chrom>', chrom))
>>> slices = cluster_to_slices(clusters[23])
>>> _ = h.plot_heatmap(chrom, *slices, rep='ES_1', outfile='images/zoomin.png')
```



1.8.5 Per-pixel significance plotting

We can pass `stage='qvalues'` to `h.plot_heatmap()` to draw heatmaps showing the significance of each pixel:

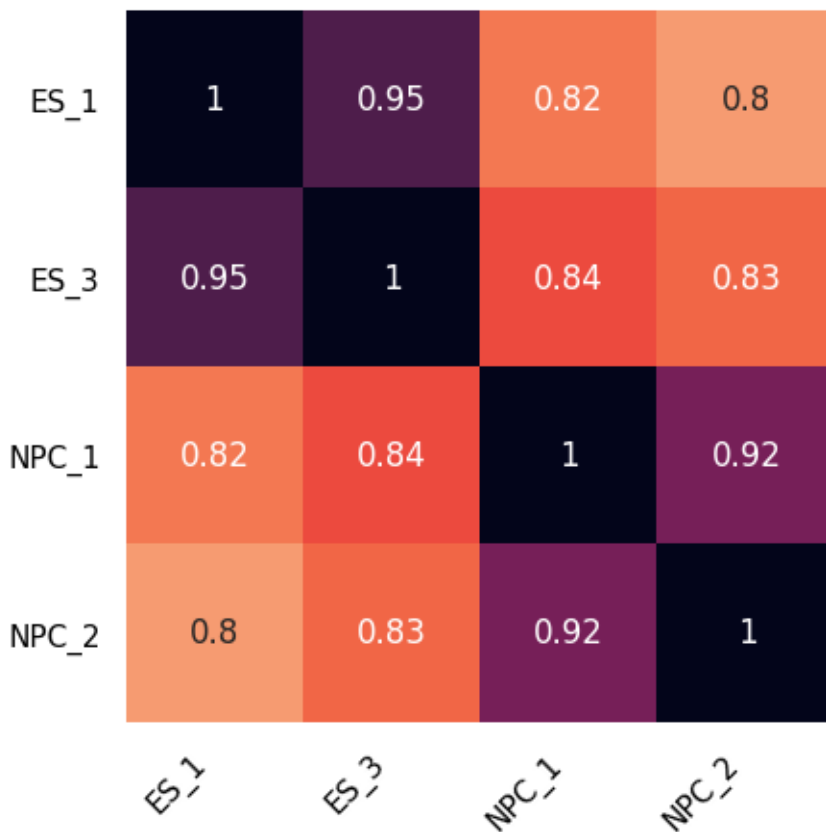
```
>>> _ = h.plot_heatmap('chr18', slice(1310, 1370), slice(1310, 1370),  
...                   stage='qvalues', cmap='bwr_r', vmin=0.099, vmax=0.101,  
...                   outfile='images/heatmap_sig.png')
```

By passing `cmap='bwr_r'` we ensure that significant, low q-value pixels will be red while insignificant, high q-value pixels will be blue. By passing `vmin=0.099`, `vmax=0.101`, we ensure that the colorbar is focused on a narrow range around an FDR threshold of 10%, allowing us to more easily tell the difference between significant and insignificant pixels.

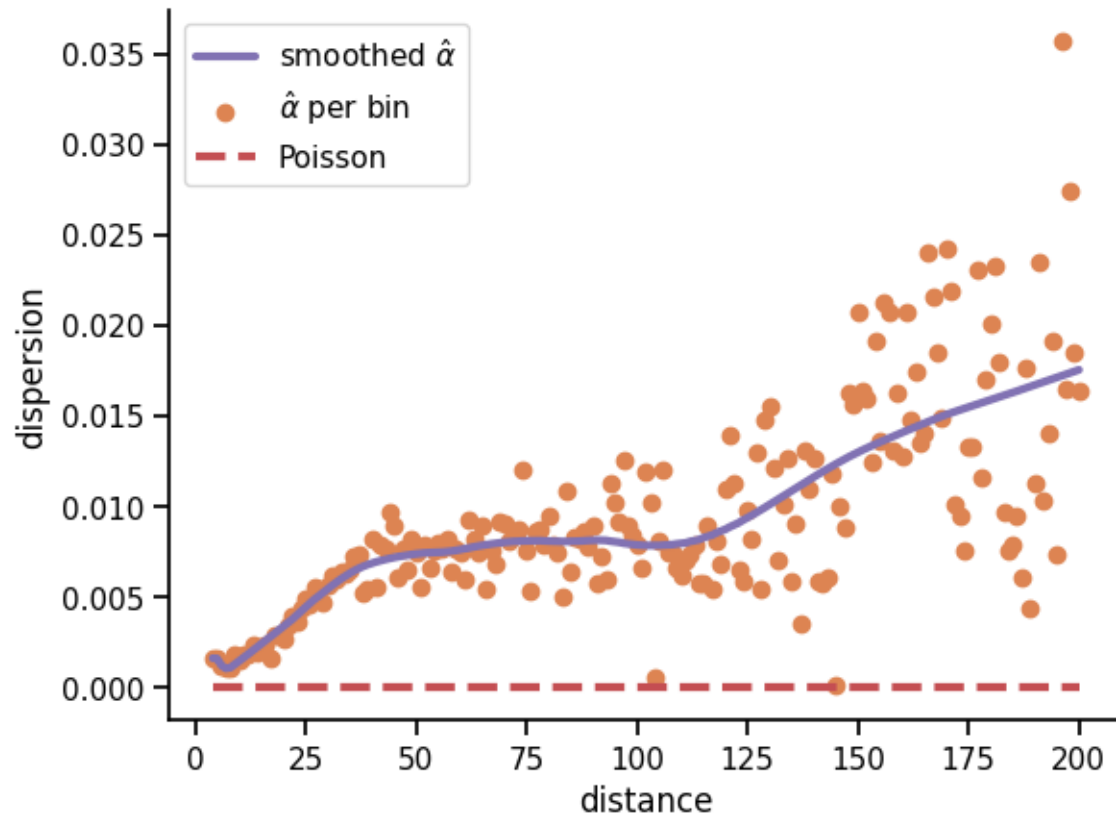
1.8.6 Correlation matrices

```
>>> _ = h.plot_correlation_matrix(outfile='images/correlation.png')
```



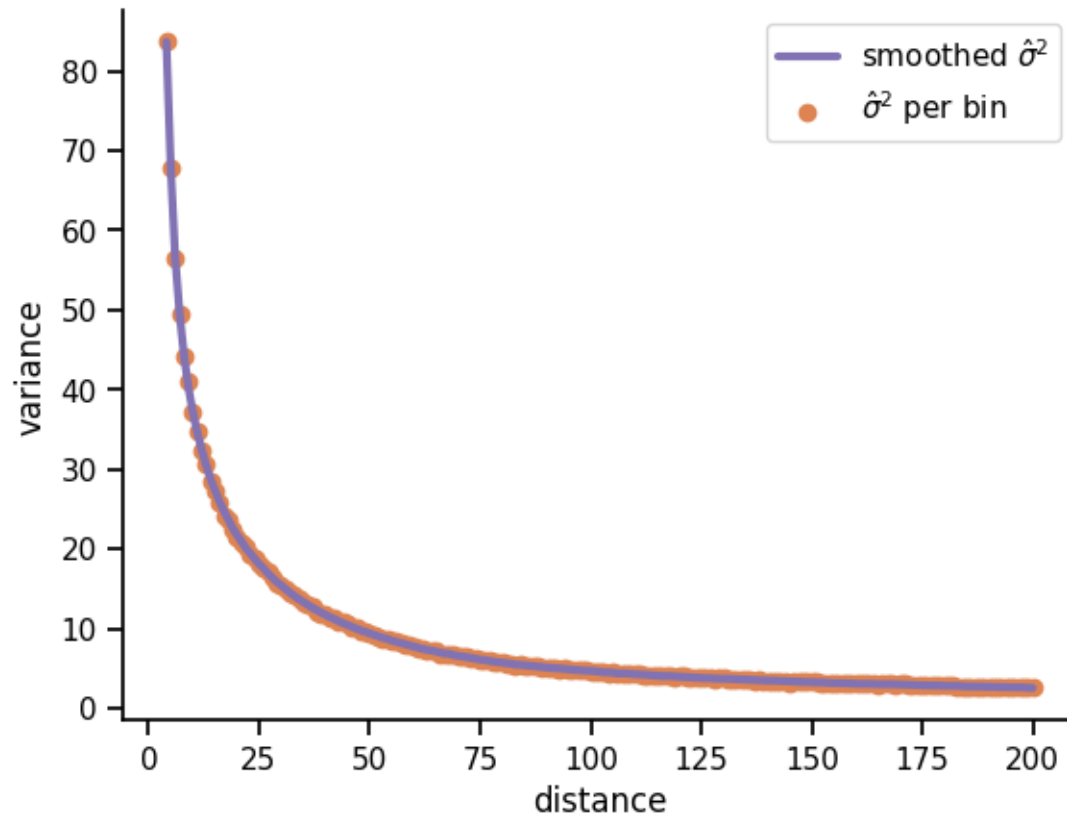
1.8.7 Dispersion fitting

```
>>> _ = h.plot_dispersion_fit('ES', outfile='images/ddr.png')
```



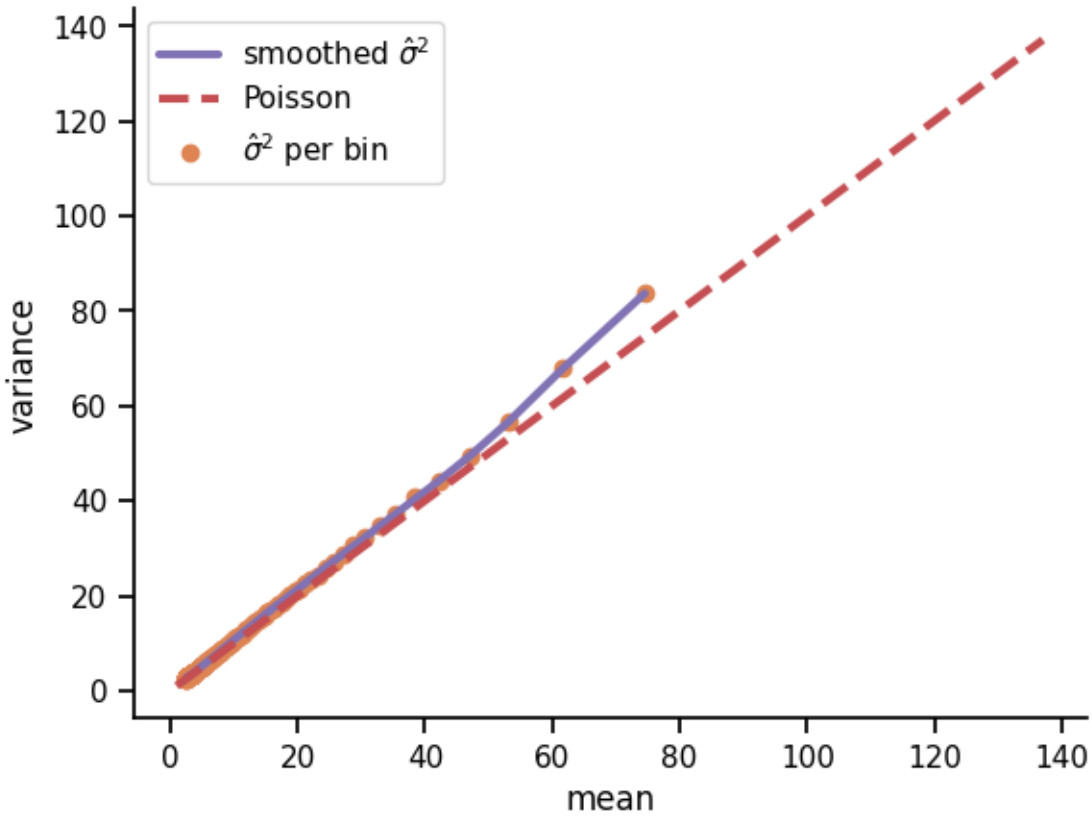
It's possible to use the one-dimensional distance dependence curve to convert distances to means. Doing so allows plotting the y-axis in terms of variance. You can do this by passing `yaxis='var'`:

```
>>> _ = h.plot_dispersion_fit('ES', yaxis='var', outfile='images/dvr.png')
```



Using the same trick, you can plot the x-axis in terms of mean by passing `xaxis='mean'`:

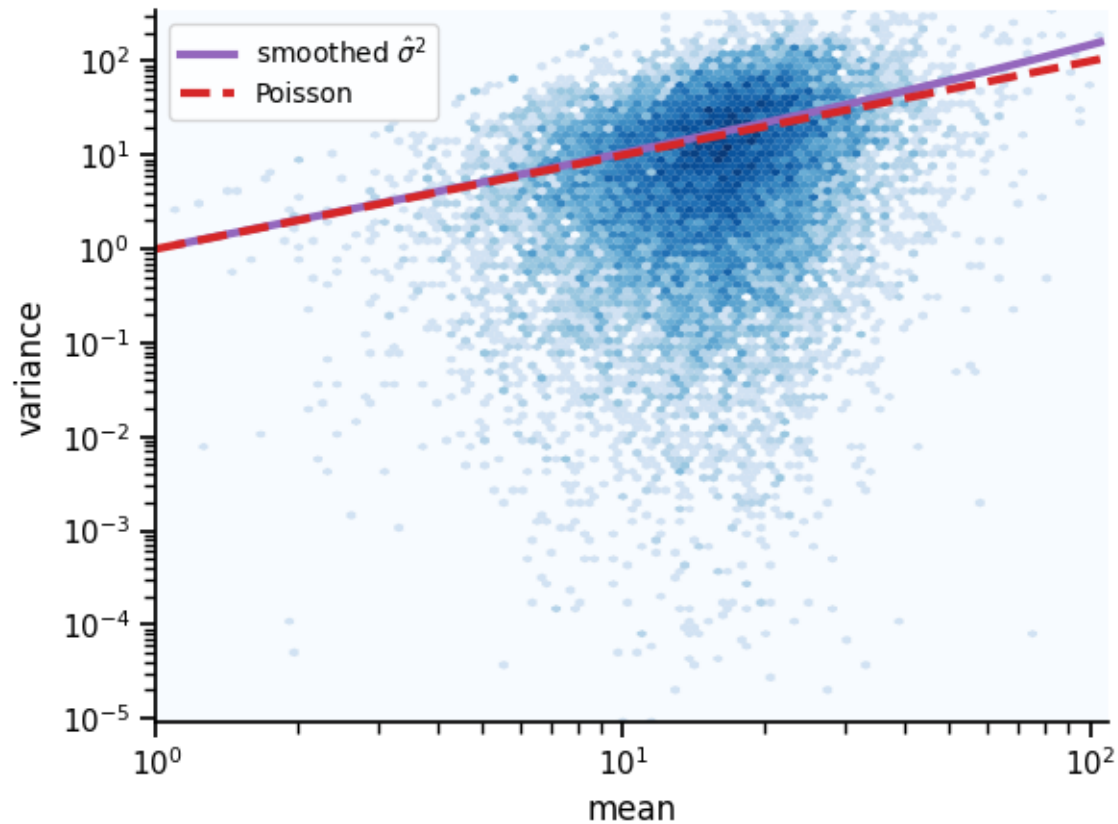
```
>>> _ = h.plot_dispersion_fit('ES', xaxis='mean', yaxis='var',  
...                           outfile='images/mvr.png')
```



At low mean and high distance, the distance dependence curve flattens out and the data become more noisy, making this conversion unreliable.

It's also possible to show the dispersion fitted at just one distance scale, overlaying the sample mean and sample variance across replicates for each pixel as a blue hexbin plot:

```
>>> _ = h.plot_dispersion_fit('ES', distance=25, hexbin=True, xaxis='mean',
...                          yaxis='var', logx=True, logy=True,
...                          outfile='images/mvr_25.png')
```

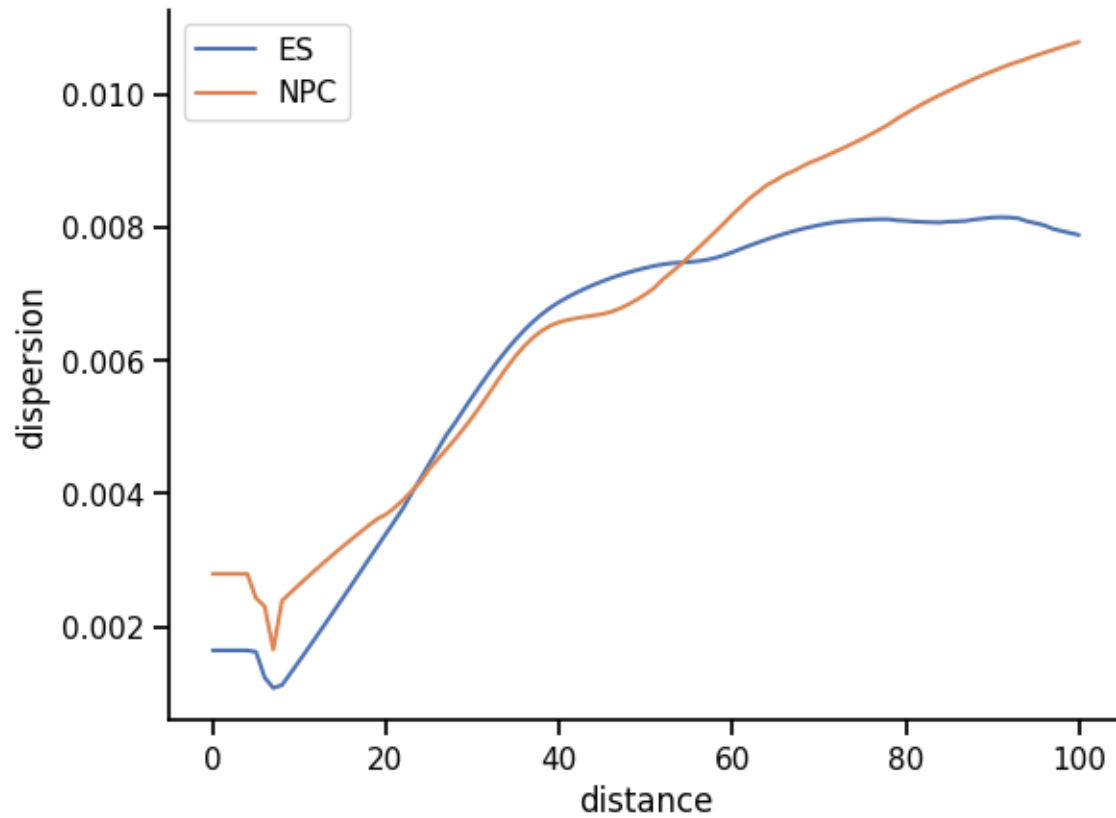


If dispersion was fitted against distance rather than mean, pass `xaxis='dist'` to plot dispersion/variance versus distance.

1.8.8 Comparing dispersion fits

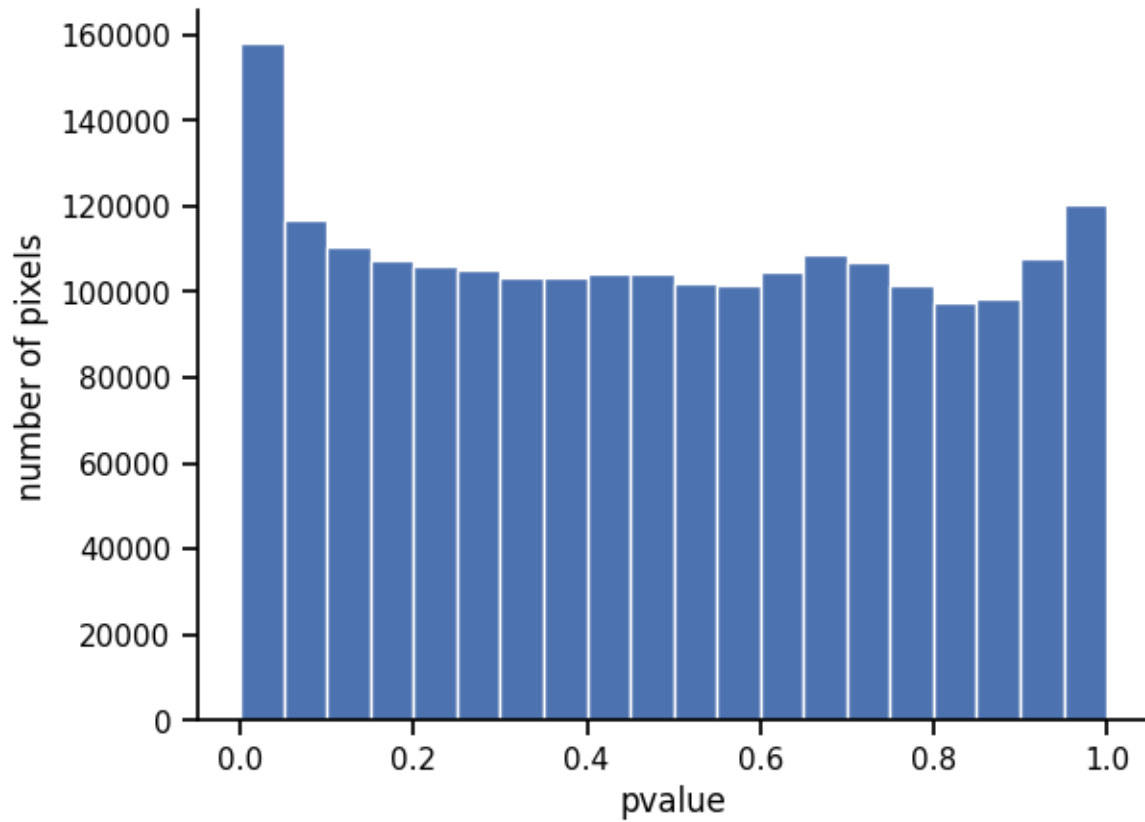
It's possible to compare different dispersion fits using the function `compare_disp_fits()` as shown here:

```
>>> from hic3defdr import compare_disp_fits
>>>
>>> _ = compare_disp_fits(
...     [h.load_disp_fn(cond) for cond in h.design.columns],
...     h.design.columns,
...     max_dist=100,
...     outfile='images/disp_comparison.png'
... )
```



1.8.9 P-value distribution

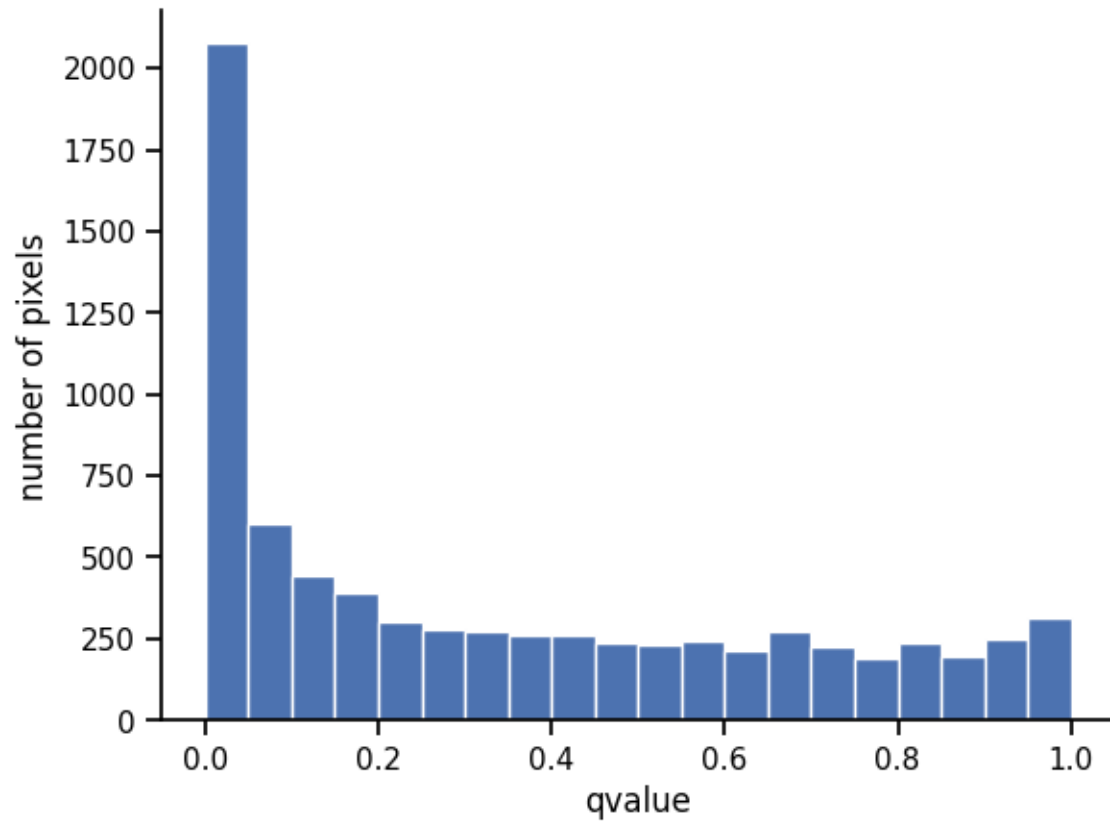
```
>>> _ = h.plot_pvalue_distribution(outfile='images/pvalue_dist.png')
```



By default, this plots the p-value distribution over all pixels for which dispersion was estimated. To plot the p-value distribution only over points in loops, pass `idx='loop'`.

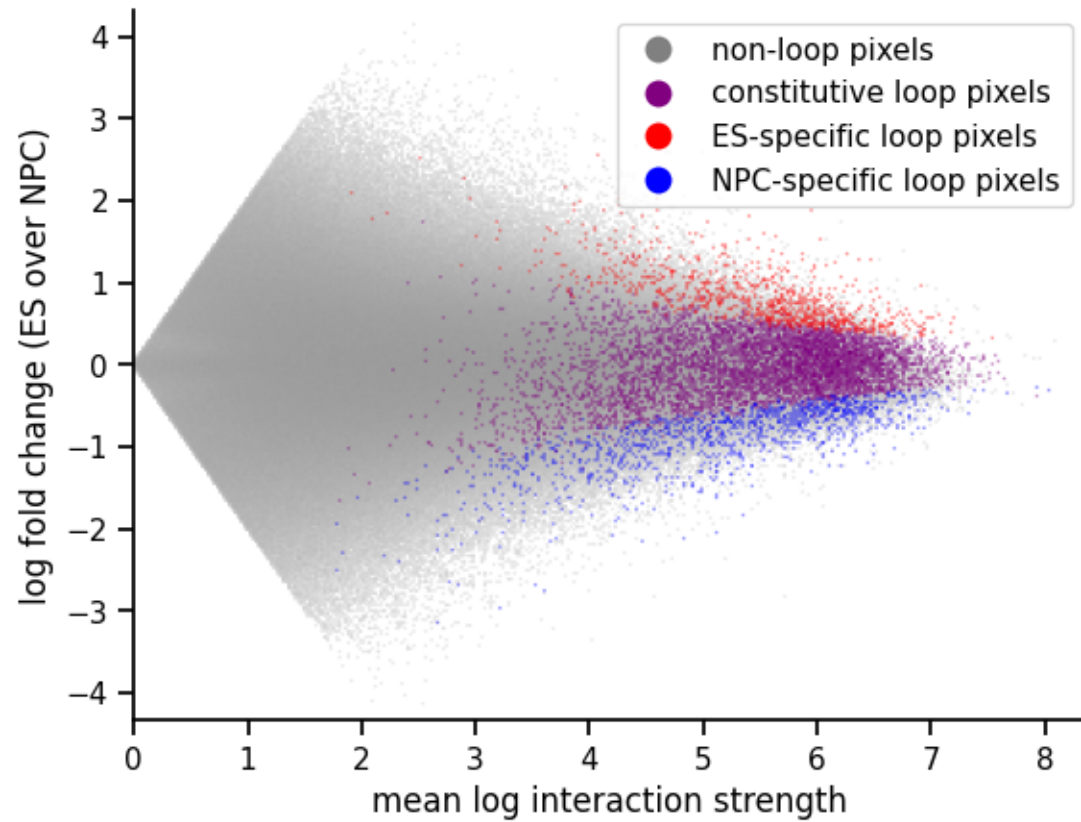
1.8.10 Q-value distribution

```
>>> _ = h.plot_qvalue_distribution(outfile='images/qvalue_dist.png')
```

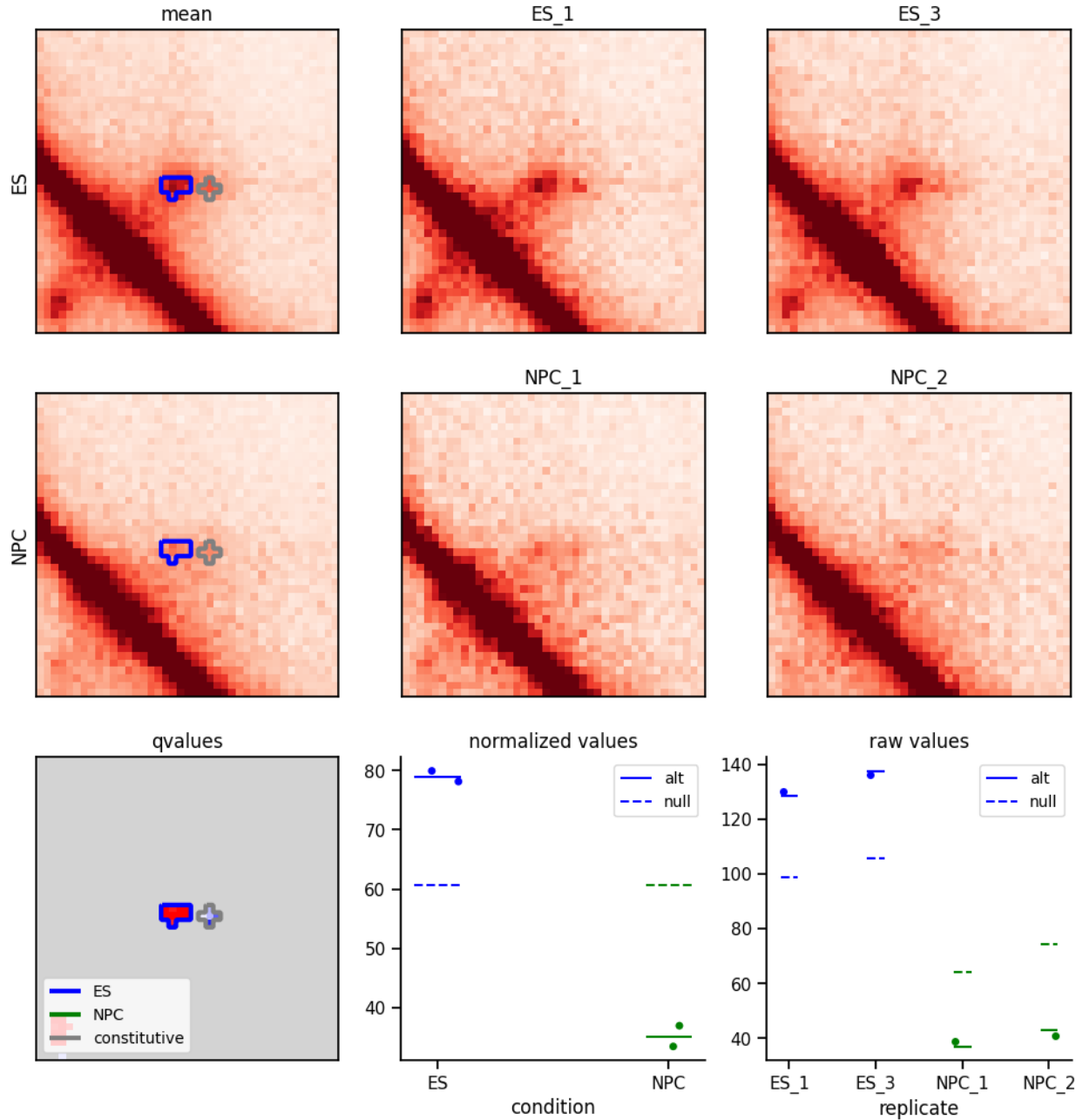
1.8.11 MA plot

```
>>> _ = h.plot_ma(outfile='images/ma.png')
```



1.8.12 Pixel detail grid

```
>>> _ = h.plot_grid('chr18', 2218, 2236, 20, outfile='images/grid.png')
```



The upper right heatmaps show the balanced and scaled values in each replicate, with each condition on its own row.

The upper left heatmaps show the alternative model mean parameter estimates for each condition. Significantly differential clusters are purple while constitutive ones are gray.

The lower left heatmap shows the q-values. Significantly differential clusters are orange while constitutive ones are gray.

The stripplots in the lower left show details information about the specific pixel in the center of the heatmaps (in this example (2218, 2236)). The dots show the values at that pixel for each replicate in normalized and raw space, respectively. The solid and dashed lines represent the mean parameters under the alt and null models, respectively.

Green points in the heatmaps represent points that have been filtered out. For the per-replicate heatmaps in the upper right of the grid, the only filters applied are the zero filter, bias filter, and distance filter. For the alt model mean

heatmaps in the upper left, this additionally includes the dispersion filter. For the q-value heatmap in the lower left, it additionally includes the loop filter if loop locations were supplied.

1.8.13 Interactive thresholding

In a Jupyter notebook environment with `ipywidgets` installed, you can play with thresholds on a live-updating plot by running:

```
%matplotlib notebook

from ipywidgets import interact
from hic3defdr import HiC3DeFDR

h = HiC3DeFDR.load('output')
_, _, outline_clusters = h.plot_grid('chr18', 2218, 2236, 50)
_ = interact(outline_clusters, fdr=[0.01, 0.05, 0.1, 0.2],
             cluster_size=[3, 4])
```

1.9 Simulation

After the `estimate_disp()` step has been run, a `HiC3DeFDR` object can be used to generate simulations of differential looping.

1.9.1 Generating simulations

To create an ES-based simulation over all chromosomes listed in `h.chroms`, we run

```
>>> from hic3defdr import HiC3DeFDR
>>>
>>> h = HiC3DeFDR.load('output')
>>> h.simulate('ES')
```

If we passed `trend='dist'` to `h.estimate_disp()`, we need to pass it to `h.simulate()` as well to ensure that the simulation function knows to treat the previously-fitted dispersion function as a function of distance.

This takes the mean of the real scaled data across the ES replicates and perturbs the loops specified in `h.loop_patterns['ES']` up or down at random to generate two new conditions called “A” and “B”. The scaled mean matrices for these conditions are then biased and scaled by the bias vectors and size factors taken from the real experimental replicates, and the ES dispersion function fitted to the real ES data is applied to the biased and scaled means to obtain dispersion values. These means and dispersions are used to draw an NB random variable for each pixel of each simulated replicate. The number of replicates in each of the simulated conditions “A” and “B” will match the design of the real analysis.

The simulated raw contact matrices will be written to disk in CSR format as `<cond><rep>_<chrom>_raw.npz` where `<cond>` is “A” or “B” and `<rep>` is the rep number within the condition. The design matrix will also be written to disk as `design.csv`.

The true labels used to perturb the loops will also be written to disk as `labels_<chrom>.txt`. This file contains as many lines as there are clusters in `h.loop_patterns['ES']`, with the `i`th line providing the label for the `i`th cluster. This file can be loaded with `np.loadtxt(..., dtype='|S7')`.

1.9.2 Evaluating simulations

After generating simulated data, HiC3DeFDR can be run on the simulated data. Then, the true labels can be used to evaluate the performance of HiC3DeFDR on the simulated data.

Evaluation of simulated data requires scikit-learn. To install this package, run

```
(venv)$ pip install scikit-learn
```

In order to run HiC3DeFDR on the simulated data, we first need to balance the simulated raw contact matrices to obtain bias vectors for each simulated replicate and chromosome. We will assume are saved next to the raw contact matrices and named `<rep>_<chrom>_kr.bias`. One example of how this can be done is shown in the following script:

```
>>> import sys
>>>
>>> import numpy as np
>>> import scipy.sparse as sparse
>>>
>>> from hic3defdr.util.filtering import filter_sparse_rows_count
>>> from hic3defdr.util.balancing import kr_balance
>>> from hic3defdr.util.printing import eprint
>>>
>>>
>>> infile_pattern = 'sim/<rep>_<chrom>_raw.npz'
>>> repnames = ['A1', 'A2', 'B1', 'B2']
>>> chroms = ['chr18', 'chr19']
>>>
>>> for repname in repnames:
...     for chrom in chroms:
...         eprint('balancing rep %s chrom %s' % (repname, chrom))
...         infile = infile_pattern.replace('<rep>', repname)\
...             .replace('<chrom>', chrom)
...         outfile = infile.replace('_raw.npz', '_kr.bias')
...         _, bias, _ = kr_balance(
...             filter_sparse_rows_count(sparse.load_npz(infile)), fl=0)
...         np.savetxt(outfile, bias)
```

Next, we create a new HiC3DeFDR object to analyze the simulated data and run the analysis through to q-values:

```
>>> import os.path
>>> from hic3defdr import HiC3DeFDR
>>>
>>> repnames = ['A1', 'A2', 'B1', 'B2']
>>> chroms = ['chr18', 'chr19']
>>> sim_path = 'sim/'
>>> base_path = os.path.expanduser('~/.hic3defdr-demo-data/')
>>> h_sim = HiC3DeFDR(
...     raw_npz_patterns=[sim_path + '<rep>_<chrom>_raw.npz'.replace('<rep>',
↪ repname) for repname in repnames],
...     bias_patterns=[sim_path + '<rep>_<chrom>_kr.bias'.replace('<rep>', repname)
↪ for repname in repnames],
...     chroms=chroms,
...     design=sim_path + 'design.csv',
...     outdir='output-sim',
...     loop_patterns={'ES': base_path + 'clusters/ES_<chrom>_clusters.json'}
... )
```

(continues on next page)

(continued from previous page)

```
creating directory output-sim
>>> h_sim.run_to_qvalues()
```

Next, we can evaluate the simulation against the clusters in `h_sim.loop_patterns['ES']` with true labels from `sim/labels_<chrom>.txt`:

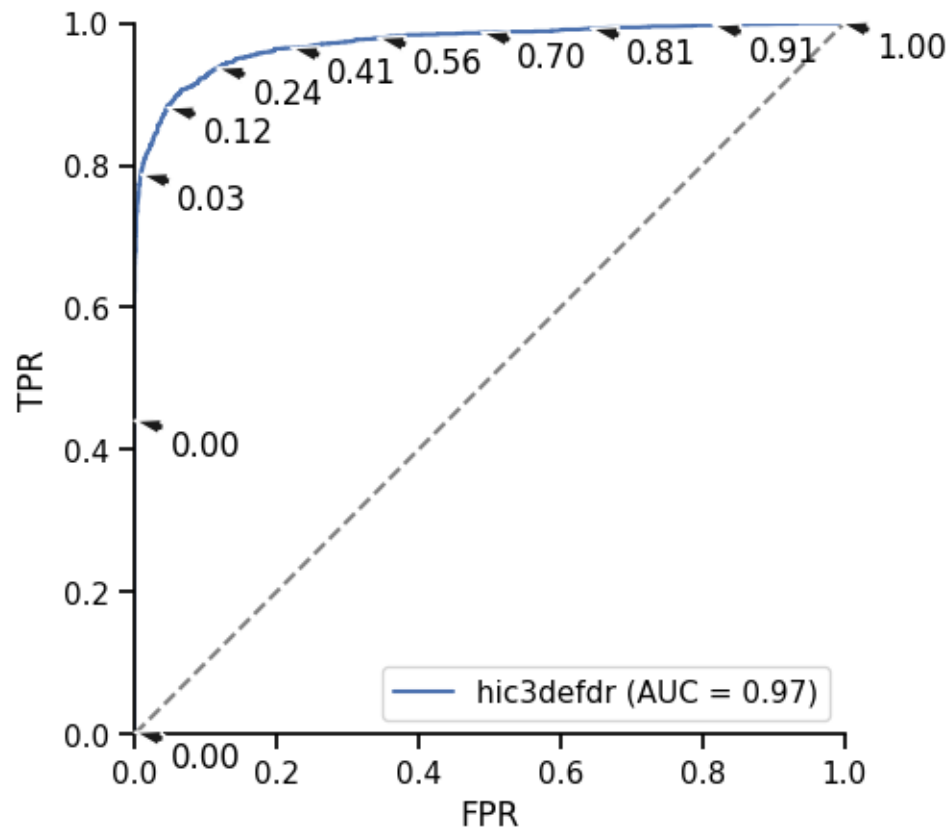
```
>>> h_sim.evaluate('ES', 'sim/labels_<chrom>.txt')
```

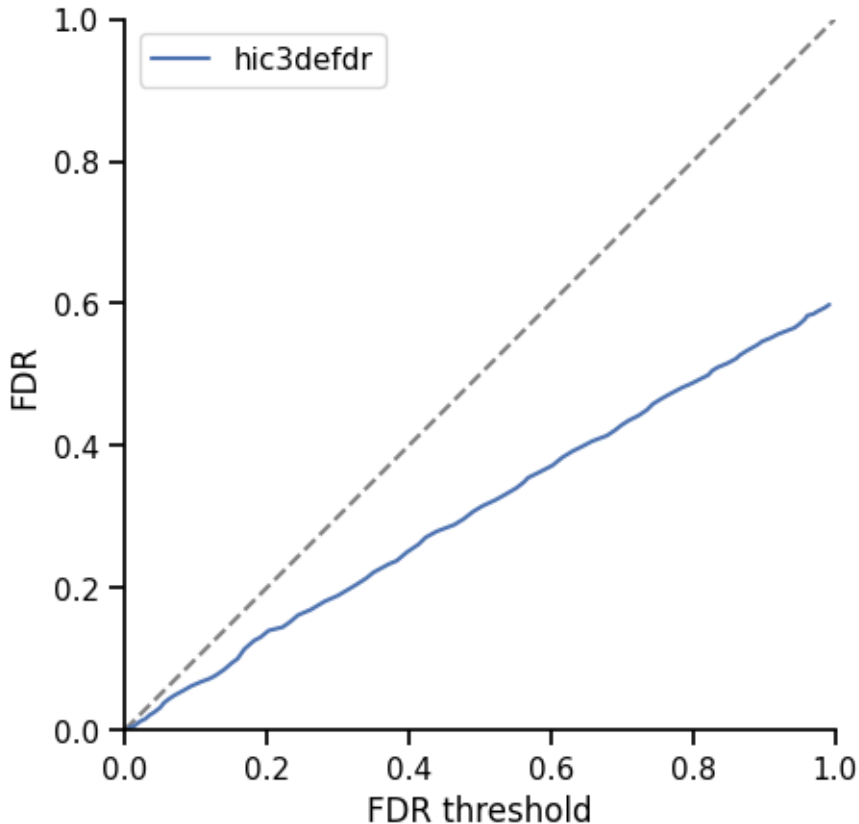
This writes a file in `h_sim`'s output directory called `eval.npz`. This file can be loaded with `np.load()` and has four keys whose values are all one dimensional vectors:

- 'thresh': the thresholds (in $1 - q$ value space) which make up the convex edge of the ROC curve; all other vectors are parallel to this one
- 'fdr': the observed false discovery rate at each threshold
- 'tpr': the observed true positive rate at each threshold
- 'fpr': the observed false positive rate at each threshold

`eval.npz` files (possibly across many runs) can be visualized as ROC curves and FDR control curves by running:

```
>>> import numpy as np
>>> from hic3defdr import plot_roc, plot_fdr
>>>
>>> _ = plot_roc([np.load('output-sim/eval.npz')], ['hic3defdr'], outfile='images/roc.
↳png')
>>> _ = plot_fdr([np.load('output-sim/eval.npz')], ['hic3defdr'], outfile='images/fdr.
↳png')
```





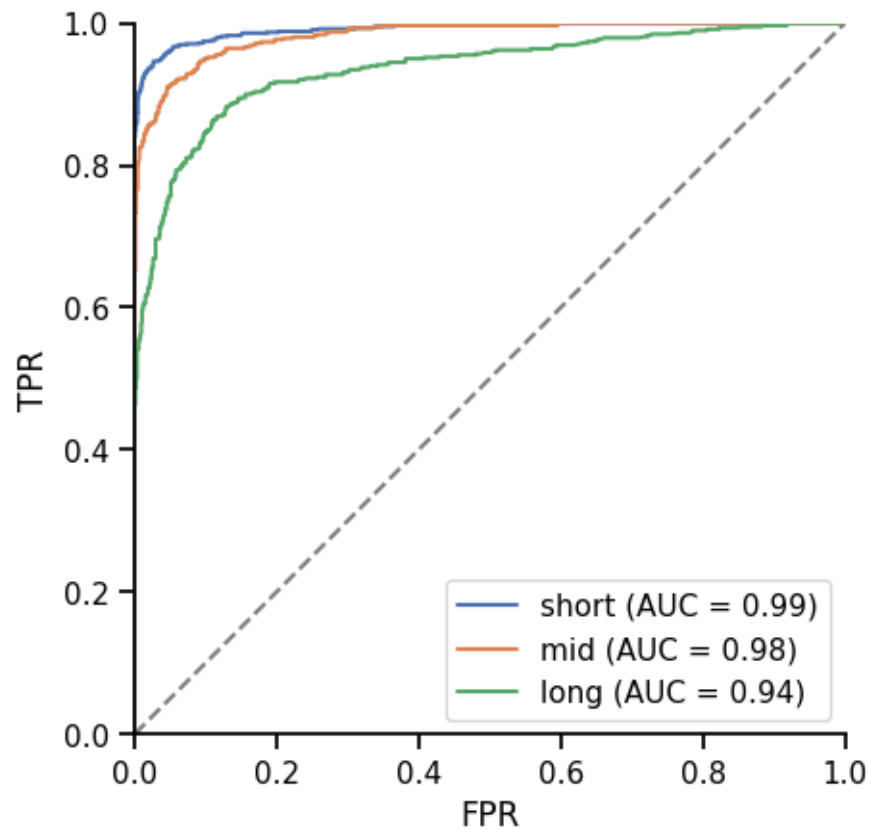
Multiple `eval.npz` files can be compared in the same plot by simply adding elements to the lists in these function calls.

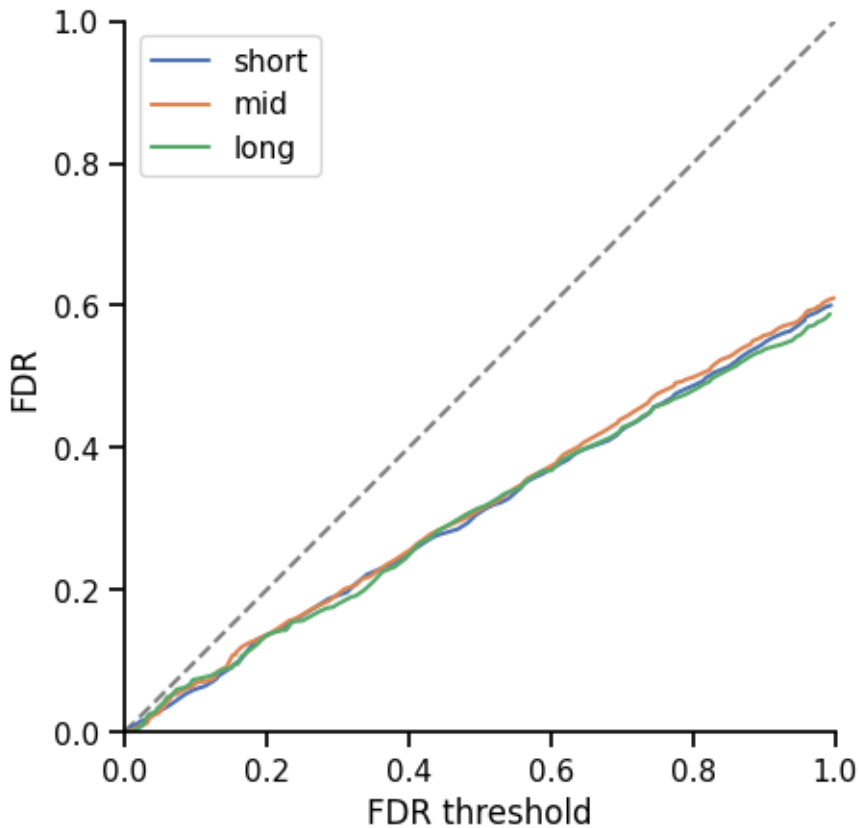
The ROC plot shows FPR versus TPR, with the gray diagonal line representing the performance of random guessing. The AUROC for each curve is shown in the legend. If only one curve is plotted, selected thresholds (in units of FDR threshold) are annotated with black arrows.

The FDR control plot shows the observed FDR as a function of the FDR threshold. Points below the gray diagonal line represent points at which FDR is successfully controlled.

As an added bonus, it's also possible to evaluate the performance on specific subsets of distance scales by using the `min_dist` and `max_dist` kwargs on `HiC3DeFDR.evaluate()` as illustrated below:

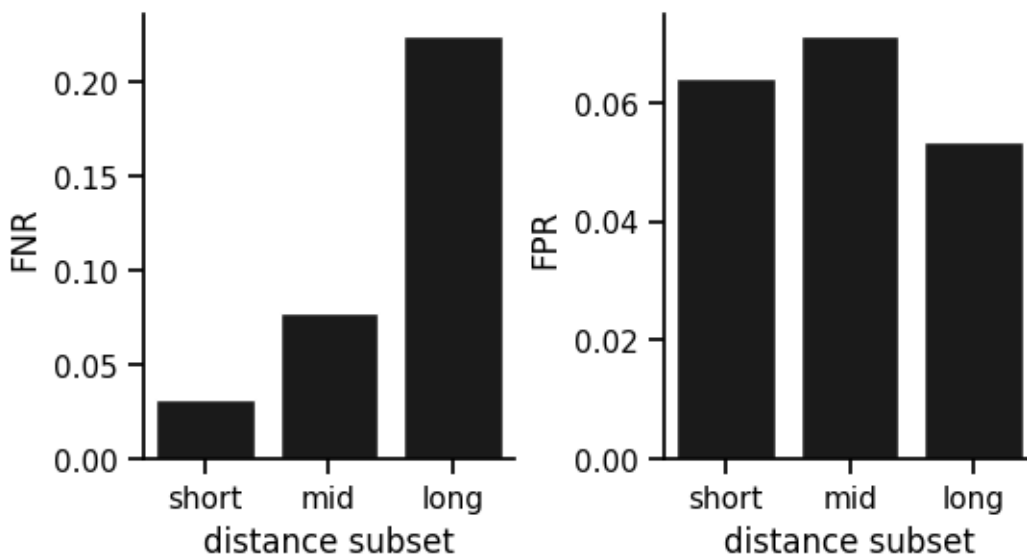
```
>>> dist_bins = [
...     ('short', (None, 15)), # distances in bin units
...     ('mid', (16, 30)),
...     ('long', (31, None))
... ]
>>> for _, (min_dist, max_dist) in dist_bins:
...     h_sim.evaluate('ES', 'sim/labels_<chrom>.txt', min_dist=min_dist,
...                   max_dist=max_dist)
>>> _ = plot_roc([np.load('output-sim/eval_{}_{}.npz'.format(min_dist, max_dist))
...               for _, (min_dist, max_dist) in dist_bins],
...              [label for label, _ in dist_bins],
...              outfile='images/roc_by_dist.png')
>>> _ = plot_fdr([np.load('output-sim/eval_{}_{}.npz'.format(min_dist, max_dist))
...               for _, (min_dist, max_dist) in dist_bins],
...              [label for label, _ in dist_bins],
...              outfile='images/fdr_by_dist.png')
```



It's also possible to compare the FPR and FNR at the different subsets:

```
>>> from hic3defdr import plot_fn_vs_fp
>>> _ = plot_fn_vs_fp([np.load('output-sim/eval_%s_%s.npz' % (min_dist, max_dist))
...                   for _, (min_dist, max_dist) in dist_bins],
...                  [label for label, _ in dist_bins], xlabel='distance subset',
...                  outfile='images/fn_vs_fp.png')
```

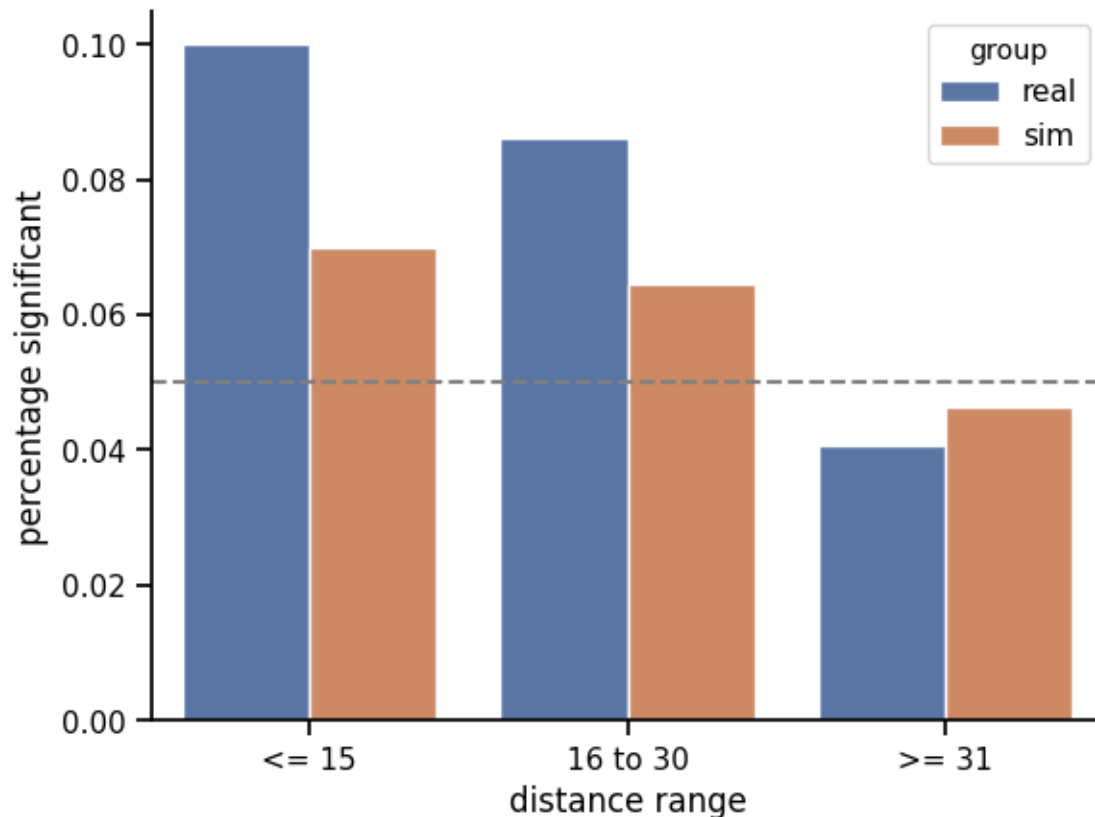


1.10 Other visualizations

1.10.1 Distance bias

We can visualize and compare the distance bias of different simulations (the degree to which low p-values are enriched or depleted in different distance scales) using the `plot_distance_bias()` function as shown below:

```
>>> from hic3defdr import plot_distance_bias
>>>
>>> dist_bins = [
...     ('short', (None, 15)), # distances in bin units
...     ('mid', (16, 30)),
...     ('long', (31, None))
... ]
>>> _ = plot_distance_bias([h, h_sim], [b for _, b in dist_bins], labels=['real', 'sim
↳'], outfile='images/distance_bias.png')
```



1.11 Package structure

The `hic3defdr` package is laid out in three parts:

1. `hic3defdr.util`: library of re-usable functions for performing computations related to differential loop calling
2. `hic3defdr.plotting`: library of re-usable functions to plotting visualizations related to differential loop calling

3. `hic3defdr.analysis`: a module that defines the `HiC3DeFDR` class, which provides an implementation of stitching together all the computational steps and visualizations in an easy-to-use way

The `HiC3DeFDR` class includes many methods, so to keep things organized these methods are defined in four separate mixin classes which are combined to form the full `HiC3DeFDR` class in `hic3defdr/analysis/constructor.py`:

- `hic3defdr.analysis.core.CoreHiC3DeFDR`
- `hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR`
- `hic3defdr.analysis.simulation.SimulatingHiC3DeFDR`
- `hic3defdr.analysis.plotting.PlottingHiC3DeFDR`

We recommend that most users simply import the `HiC3DeFDR` class and interact with this package through that interface, but the functions defined in `hic3defdr.util` and `hic3defdr.plotting` may also be useful to some users and are designed to be somewhat re-usable.

The complete layout of the package is summarized below:


```

hic3defdr/
├── _version.py           # package root
├── analysis/            # version configuration
│   ├── alternatives.py   # HiC3DeFDR class and alternatives
│   ├── analysis.py       # defines alternative analysis models
│   ├── constructor.py    # HiC3DeFDR's pipeline methods
│   ├── core.py           # HiC3DeFDR's class definition and constructor
│   ├── plotting.py       # HiC3DeFDR's core save/load methods
│   ├── simulation.py     # HiC3DeFDR's plotting methods
│   └── simulation.py     # HiC3DeFDR's simulation/evaluation methods
├── plotting/            # library of plotting functions
│   ├── dispersion.py     # dispersion/variance fit visualizations
│   ├── distance_bias.py  # distance bias comparison barplots
│   ├── distance_dependence.py # distance dependence curve comparison
│   ├── fdrr.py           # FDR control curve plotting
│   ├── fn_vs_fp.py       # FN vs FP tradeoff barplots
│   ├── grid.py           # "pixel detail grid" combination visualization
│   ├── heatmap.py        # simple contact matrix heatmap plotting
│   ├── histograms.py     # p-value/q-value histograms
│   ├── ma.py             # MA plots
│   └── roc.py            # ROC curve plots
└── util/                # library of utility functions
    ├── binning.py        # creating groups of points
    ├── balancing.py       # KR matrix balancing
    ├── banded_matrix.py   # BandedMatrix class (used for filtering)
    ├── classification.py  # classifying differential loop pixels
    ├── cluster_table.py   # creating tables summarizing cluster info
    ├── clusters.py        # interacting with called loop clusters
    ├── demo_data.py       # utilities for downloading the demo dataset
    ├── dispersion.py      # estimating dispersions in NB data
    ├── evaluation.py      # evaluating results of simulations
    ├── filtering.py       # filtering (applied before balancing)
    ├── lowess.py          # lowess fitting
    ├── lrt.py            # NB likelihood ratio testing
    ├── matrices.py        # interacting with sparse matrices
    ├── parallelization.py # parallelizing work across cores
    ├── printing.py        # printing updates and info to the console
    ├── progress.py        # showing progress bars
    ├── scaled_nb.py       # dealing with scaling factors in NB
    └── scaling.py         # scaling reps to account for sequencing depth

```

(continues on next page)

(continued from previous page)

 simulation.py	# simulating pseudoreplicates
thresholding.py	# thresholding differential pixels/clusters

1.12 Additional options

Additional options are exposed as kwargs on the functions in this library. Use `help(<function>)` to get detailed information about the options available for any function and what these options may be used for.

1.13 Testing

We run our tests with `tox`. To execute tests, install `tox` (`pip install tox`) and then run `tox` to run all tests or `tox -e <testenv>` to run a specific test environment. See `tox.ini` for the full specification of all test environments.

Conceptual documentation

This section explores some conceptual ideas in the `hic3defdr` library in deeper detail.

Contents:

2.1 `hic3defdr` data layout

In brief, the `hic3defdr` data layout is like a COO-format sparse matrix where the data vector is actually a rectangle, storing parallel data vectors for multiple replicates in the same data structure. This allows `hic3defdr` to combine the advantages of sparse matrix formats (like COO) together with the applications of analyzing data across replicates (like differential loop calling).

Like COO, the `hic3defdr` data layout keeps track of a `row` and `col` vector for each chromosome. These vectors are stored on disk as `<outdir>/row_<chrom>.npz` and `<outdir>/col_<chrom>.npz`, respectively.

In contrast to COO, where the data vector (parallel to `row` and `col`) is just a vector, in the `hic3defdr` data layout the data can be a rectangular matrix whose rows correspond to pixels (same length as `row` and `col`) and whose columns correspond to replicates or conditions. Each stage of the `hic3defdr` pipeline writes its output (in the form of this rectangle) to disk as `<outdir>/<stage>_<chrom>.npz`. The `hic3defdr` data layout is designed so that multiple “stages” of data processing can re-use the same `row` and `col` vectors, making it easy to trace pixel values across stages as well as across replicates.

One important complication is that certain steps of data processing may filter out pixels from the pipeline. This means that the number of pixels (number of rows in the rectangular data matrix) may be smaller for the output of later pipeline steps. Since these matrices have fewer rows, they don’t align with the `row` and `col` vectors, which are always the same length. To address this problem, boolean index vectors stored on disk as e.g. `<outdir>/disp_idx_<chrom>.npz` are aligned with `row` and `col` and are `True` at all pixels that are kept during filtering. This means that `row[disp_idx]` is aligned with rectangular matrices after the `disp_idx` filtering step, such as `<outdir>/disp_<chrom>.npz`. Finally, these indices can be chained, so that `row[disp_idx][loop_idx]` is aligned with rectangular matrices after the `loop_idx` filtering step, such as `<outdir>/qvalues_<chrom>.npz`.

A complete table of all the outputs, their expected shapes, and what boolean indices are needed to align them to `row` and `col` is provided in the README section “Intermediates and final output files”.

2.2 Data filtering

Points are filtered out of Hi-C datasets by HiC3DeFDR in three stages:

2.2.1 1. Initial data import

This filtering step is performed during the `prepare_data()` step.

Our motivation is to include as many points as possible. We refuse to filter out points that have zero in one replicate (leads to underestimation of variance/dispersion).

We

- exclude points beyond `HiC3DeFDR.dist_thresh_max`
- exclude points that have zero in all reps (“pixel union strategy” implemented in `hic3defdr.util.matrices.sparse_union()`)
- exclude points in rows that failed balancing (decided by `HiC3DeFDR.bias_thresh`)

Points present in the data files `row_<chrom>.npz`, `col_<chrom>.npz`, `raw_<chrom>.npz`, and `scaled_<chrom>.npz` reflect points that survive this filtering step.

2.2.2 2. `disp_idx`

This filter is computed during the `prepare_data()` step, but is not used until the `estimate_disp()` step.

Our motivation is to not try to fit dispersion to points where think dispersion estimation will be very hard. This include points very close to the diagonal where Hi-C gets crazy and points with very low coverage (low mean across reps).

We

- exclude points within `HiC3DeFDR.dist_thresh_min`
- exclude points whose mean across reps is below `HiC3DeFDR.mean_thresh`

`disp_idx_<chrom>.npz` is a boolean vector aligned to `<row/col>_<chrom>.npz` that is True for all points that survive this filtering step.

2.2.3 3. `loop_idx`

This filter is computed during the `prepare_data()` step, but is not used until the `bh()` step.

This filter is only computed and used if `HiC3DeFDR.loop_patterns` is not None.

Our motivation is to reduce the number of hypotheses that we test when performing multiple testing correction via BH-FDR. Since we are only interested in finding differential loops, we can choose to only test the hypotheses that correspond to positions where there are loops.

We

- exclude points that are not in a loop as defined by `HiC3DeFDR.loop_patterns`

`loop_idx_<chrom>.npz` is a boolean vector aligned to the positions where `disp_idx_<chrom>.npz` is True that is True for all points that survive this filtering step.

This section includes some potentially useful demos.

Contents:

3.1 Distance-conditional median of ratios demo

In an interactive shell, import `hic3defdr.util.scaling.conditional_mor()`:

```
>>> import numpy as np
>>> from hic3defdr.util.scaling import conditional_mor
```

Create a test dataset with 4 replicates (columns) and 5 pixels (rows):

```
>>> data = np.arange(20, dtype=float).reshape((5, 4))
>>> data
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

Specify a distance for each pixel:

```
>>> dist = np.array([1, 1, 1, 2, 2])
```

Normalize the data:

```
>>> conditional_mor(data, dist)
array([[0.79394639, 0.93946738, 1.08498836, 1.23050934],
       [0.79394639, 0.93946738, 1.08498836, 1.23050934],
       [0.79394639, 0.93946738, 1.08498836, 1.23050934],
       [0.90390183, 0.96968472, 1.0354676 , 1.10125049],
       [0.90390183, 0.96968472, 1.0354676 , 1.10125049]])
```

3.2 Using `sparse_union()` to load a rectangular data matrix

3.2.1 Motivation

The hic3defdr ecosystem revolves around a *specific data layout*. hic3defdr automatically imports your input npz's into this layout when running the pipeline. But what if you want to convert your npz's into hic3defdr's data layout without running the pipeline?

The key ingredient that will allow us to do this is the function `hic3defdr.util.matrices.sparse_union()`, which will be demonstrated below.

3.2.2 Walkthrough

In an interactive shell, import `hic3defdr.util.matrices.sparse_union()`:

```
>>> import numpy as np
>>> import scipy.sparse as sparse
>>> from hic3defdr.util.matrices import sparse_union
```

The kwargs on this function mostly serve to help filter out pixels to reduce the total number of pixels. The most important kwarg for this is `dist_thresh`, which simply drops all pixels with interaction distances longer than `dist_thresh`. This threshold is on by default and is highly recommended.

The remaining kwargs work together to throw out pixels that have low mean values across replicates after normalization (by `bias` and `size_factors` if these are passed). This filtering is off by default (`mean_thresh=0.0`) and is not recommended. This means that you don't need to worry about passing `bias` or `size_factors` to this function even if you have these values available.

Create some test npz's from dense matrices with zeros in different positions:

```
>>> rep1 = np.array([[0., 0., 3., 1.],
...                  [0., 6., 5., 0.],
...                  [0., 0., 0., 2.],
...                  [0., 0., 0., 7.]])
>>> rep2 = np.array([[0., 1., 3., 2.],
...                  [0., 0., 0., 0.],
...                  [0., 0., 4., 2.],
...                  [0., 0., 0., 3.]])
>>> sparse.save_npz('rep1.npz', sparse.csr_matrix(rep1))
>>> sparse.save_npz('rep2.npz', sparse.csr_matrix(rep2))
```

Use `sparse_union` to identify the union pixel set:

```
>>> rep_npzs = ['rep1.npz', 'rep2.npz']
>>> row, col = sparse_union(rep_npzs, dist_thresh=2)
>>> list(zip(row, col))
[(0, 1), (0, 2), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3)]
```

Notice that pixels (0, 0) and (1, 3) are not in the union pixel set. This is because these pixels are zero in both replicates.

Also notice that pixel (0, 3) is also not in the union pixel set. This is because its distance ($3 - 0 = 3$) is greater than the `dist_thresh` we passed to `sparse_union()`.

Finally, we can construct the data matrix:

```
>>> data = np.zeros((len(row), len(rep_npzs)))
>>> for i in range(len(rep_npzs)):
...     data[:, i] = sparse.load_npz(rep_npzs[i]).tocsr()[row, col]
>>> data
array([[0., 1.],
       [3., 3.],
       [6., 0.],
       [5., 0.],
       [0., 4.],
       [2., 2.],
       [7., 3.]])
```

If we want to know the interaction distance for each pixel (each row of data), we can calculate:

```
>>> dist = col - row
>>> dist
array([1, 2, 0, 1, 0, 1, 0], dtype=int32)
```

To clean up, we will delete the npz's we created.:

```
>>> import os
>>> for f in rep_npzs:
...     os.remove(f)
```

3.3 Making APA plots

We'll start off with some imports:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy.sparse as sparse
>>> from straw import straw
>>> from hic3defdr.util.clusters import hiccup_to_clusters
>>> from hic3defdr.util.apa import make_apu_stack
```

To keep things reasonably fast, we will run our analysis at 25 kb resolution:

```
>>> res = 25000
```

We will download loop calls from the Rao et al. 2014 GEO submission and convert it to our cluster format.

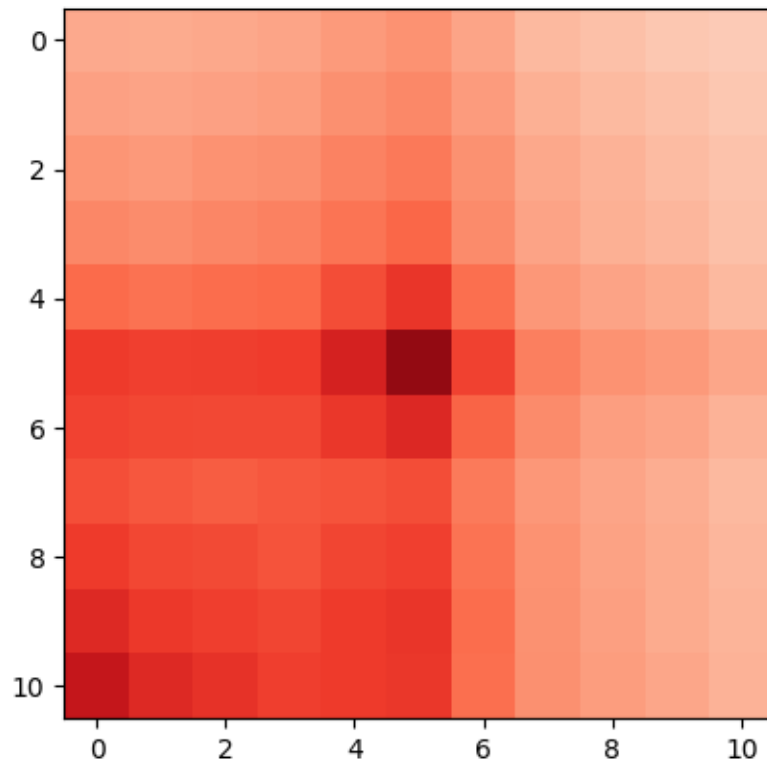
```
>>> from six.moves.urllib.request import urlretrieve
>>> _, _ = urlretrieve('https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE63525&
↳format=file&file=GSE63525%5FGM12878%5Fprimary%2Breplicate%5FHiCCUPS%5Flooplist%2Etxt
↳%2Egz', 'loops.gz')
>>> clusters = hiccup_to_clusters('loops.gz', res)
```

We will get the 25 kb resolution, KR-balanced cis matrix for chr21 from the same dataset and load it as a CSR matrix:

```
>>> hic_file = 'https://hicfiles.s3.amazonaws.com/hiseq/gm12878/in-situ/combined.hic'
>>> row, col, data = map(np.array, straw('KR', hic_file, '21', '21', 'BP', res))
HiC version: 7
>>> csr = sparse.coo_matrix((data, (row // res, col // res))).tocsr()
```

Finally, we will plot the APA plot of the called clusters on chr21:

```
>>> _ = plt.imshow(  
...     np.nanmean(make_apa_stack(csr, clusters['chr21'], 11), axis=0),  
...     cmap='Reds',  
...     vmin=0,  
...     vmax=600  
... )  
>>> plt.savefig('images/apa.png')
```



To clean up, we can delete the loops we downloaded:

```
>>> import os  
>>> os.remove('loops.gz')
```

CHAPTER 4

Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project attempts to adhere to [Semantic Versioning](#).

4.1 0.2.1 - 2020-05-03

4.1.1 Added

- A license (MIT).
- Utility functions for APA plots in `hic3defdr.util.apa` with a corresponding demo in `docs/apa.rst`.
- Utility function for importing loop calls from HiCCUPS in `hic3defdr.util.clusters.hiccups_to_clusters()`.

4.1.2 Updates/maintenance

- Demos now run as tests within the readme tox environments.
- Release pipeline no longer creates any artifacts, see comments on [this commit](#).

4.2 0.2.0 - 2020-03-13

This version adds Python 3 support!

4.2.1 Added

- A new utility module to load demo data (`hic3defdr.util.demo_data`), see [#39](#).

- New utility modules to handle balancing of simulated datasets (`hic3defdr.util.balancing`, `hic3defdr.util.banded_matrix`, and `hic3defdr.util.filtering`), see #40.

4.2.2 Changed

- numpy arrays containing strings now use a “U”-based dtype.

4.2.3 Updates/maintenance

- Bumped minimum lib5c dependency to 0.6.0.
- Version information is now provided by `setuptools-scm` instead of `versioneer`.
- Added Sphinx documentation and readthedocs configuration to build it.
- Linting, testing, running the README, and building docs are now all handled using `tox`.
- Reworked Docker image build, see [creminslab/lib5c#57](#).

4.3 0.1.1 - 2020-02-27

Streamlined data loading, see #19.

4.3.1 Added

- A new convenience function `HiC3DeFDR.get_matrix()` which selects a dense matrix slice for any stage of the data in one line.
- A new convenience kwarg `coo` on `HiC3DeFDR.load_data()` which (when `True`) causes the function to return a COO-format `row, col, data` tuple that’s guaranteed to be aligned for any stage of the data.
- New convenience kwargs `rep` and `cond` on `HiC3DeFDR.load_data()` which allow selecting the right column of rectangular data stages by name automatically (i.e., without having to manually compute `rep_idx`).
- The `stage` kwarg of `HiC3DeFDR.plot_heatmap()` can now be any stage, including ‘qvalues’. This allows easy plotting of heatmaps showing the significance of each pixel. An example showing how to do this has been added to the README.

4.3.2 Changed

- The signature of `HiC3DeFDR.plot_heatmap()` has been reworked so that `rep` is now a kwarg instead of the first positional arg.
- The signature of `hic3defdr.plotting.heatmap.plot_heatmap()` has been reworked so that it accepts a dense matrix `matrix` as the first positional arg instead of `row, col, data, row_slice, col_slice`. Clients are expected to use `hic3defdr.util.matrix.select_matrix()` to get the dense matrix before calling `plot_heatmap()`.
- `HiC3DeFDR.load_data('loop_idx', ...)` now returns a vector of `True` if `loop_patterns` was not passed to the `HiC3DeFDR` constructor.

4.4 0.1.0 - 2020-02-12

Adds a first draft of TSV output tables.

4.4.1 Added

- This changelog!
- The pipeline now generates a final table of loop calls as requested by [#23](#).
 - Added a new pipeline step `Hic3DeFDR.collect()` - we now recommend running this instead of `Hic3DeFDR.classify()`.
 - Added a new utility module `hic3defdr.util.cluster_table` to support creating tabular files summarizing cluster information
 - Added a few new functions in `hic3defdr.util.clusters` as well as some better test coverage in this module.
 - The README has been updated to cover this new step and output format.

4.4.2 Changed

- Renamed `hic3defdr.util.logging` to `hic3defdr.util.printing` to avoid a rare name clash bug related to [this issue](#), see [#28](#)

4.4.3 Fixed

- [#26](#)
- [#27](#)

4.5 0.0.9 - 2019-11-26

First official release, corresponds to what was used in the second submission of the related manuscript.

4.6 Diffs

- [0.2.1](#)
- [0.2.0](#)
- [0.1.1](#)
- [0.1.0](#)
- [0.0.9](#)

5.1 hic3defdr package

5.1.1 Subpackages

hic3defdr.analysis package

Submodules

hic3defdr.analysis.alternatives module

Experimental module exposing variants of the HiC3DeFDR model for benchmarking purposes.

```
class hic3defdr.analysis.alternatives.Global3DeFDR(raw_npz_patterns, bias_patterns,  
                                                    chroms,          design,          out-  
                                                    dir,             dist_thresh_min=4,  
                                                    dist_thresh_max=200,  
                                                    bias_thresh=0.1,  
                                                    mean_thresh=1.0,  
                                                    loop_patterns=None, res=None)
```

Bases: `hic3defdr.analysis.constructor.HiC3DeFDR`

```
estimate_disp(estimator='qcml', frac=None, auto_frac_factor=15.0, weighted_lowess=True,  
               n_threads=-1)  
Estimates dispersion parameters.
```

Parameters

- **estimator** (`'cml'`, `'qcml'`, `'mme'`, or a function) – Pass `'cml'`, `'qcml'`, `'mme'` to use conditional maximum likelihood (CML), quantile-adjusted CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.

- **frac** (*float, optional*) – The lowess smoothing fraction to use when fitting the distance vs dispersion trend. Pass None to choose a value automatically.
- **auto_frac_factor** (*float*) – When **frac** is None, this factor scales the automatically determined fraction parameter.
- **weighted_lowess** (*bool*) – Whether or not to use a weighted lowess fit when fitting the smoothed dispersion curve.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple distance scales in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the distance scales serially.

```
class hic3defdr.analysis.alternatives.Poisson3DeFDR(raw_npz_patterns,
                                                    bias_patterns, chroms, design,
                                                    outdir, dist_thresh_min=4,
                                                    dist_thresh_max=200,
                                                    bias_thresh=0.1,
                                                    mean_thresh=1.0,
                                                    loop_patterns=None,
                                                    res=None)
```

Bases: `hic3defdr.analysis.constructor.HiC3DeFDR`

```
estimate_disp(estimator='qcml', frac=None, auto_frac_factor=15.0, weighted_lowess=True,
              n_threads=-1)
    Estimates dispersion parameters.
```

Parameters

- **estimator** (*'cml', 'qcml', 'mme', or a function*) – Pass 'cml', 'qcml', 'mme' to use conditional maximum likelihood (CML), quantile-adjusted CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.
- **frac** (*float, optional*) – The lowess smoothing fraction to use when fitting the distance vs dispersion trend. Pass None to choose a value automatically.
- **auto_frac_factor** (*float*) – When **frac** is None, this factor scales the automatically determined fraction parameter.
- **weighted_lowess** (*bool*) – Whether or not to use a weighted lowess fit when fitting the smoothed dispersion curve.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple distance scales in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the distance scales serially.

```
lrt(chrom=None, refit_mu=True, n_threads=-1, verbose=True)
    Runs the likelihood ratio test to test for differential interactions.
```

Parameters

- **chrom** (*str*) – The name of the chromosome to run the LRT for. Pass None to run for all chromosomes in series.
- **refit_mu** (*bool*) – Pass True to refit the mean parameters in the NB models being compared in the LRT. Pass False to use the means across replicates directly, which is simpler and slightly faster but technically violates the assumptions of the LRT.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there

are CPUs. Pass 0 to process the chromosomes serially.

- **verbose** (*bool*) – Pass False to silence reporting of progress to stderr.

```
class hic3defdr.analysis.alternatives.Unsmoothed3DeFDR(raw_npz_patterns,
                                                    bias_patterns,      chroms,
                                                    design,          outdir,
                                                    dist_thresh_min=4,
                                                    dist_thresh_max=200,
                                                    bias_thresh=0.1,
                                                    mean_thresh=1.0,
                                                    loop_patterns=None,
                                                    res=None)
```

Bases: `hic3defdr.analysis.constructor.HiC3DeFDR`

```
estimate_disp(estimator='qcml', frac=None, auto_frac_factor=15.0, weighted_lowess=True,
               n_threads=-1)
Estimates dispersion parameters.
```

Parameters

- **estimator** (*'cml', 'qcml', 'mme', or a function*) – Pass 'cml', 'qcml', 'mme' to use conditional maximum likelihood (CML), quantile-adjusted CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.
- **frac** (*float, optional*) – The lowess smoothing fraction to use when fitting the distance vs dispersion trend. Pass None to choose a value automatically.
- **auto_frac_factor** (*float*) – When *frac* is None, this factor scales the automatically determined fraction parameter.
- **weighted_lowess** (*bool*) – Whether or not to use a weighted lowess fit when fitting the smoothed dispersion curve.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple distance scales in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the distance scales serially.

```
hic3defdr.analysis.alternatives.poisson_fit_mu_hat(raw,f)
```

```
hic3defdr.analysis.alternatives.poisson_logpmf(x,mu)
```

```
hic3defdr.analysis.alternatives.poisson_lrt(raw,f,design,refit_mu=True)
```

hic3defdr.analysis.analysis module

```
class hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR
```

Bases: object

Mixin class containing analysis functions for HiC3DeFDR.

```
bh()
```

Applies BH-FDR control to p-values across all chromosomes to obtain q-values.

Should only be run after all chromosomes have been processed through p-values.

```
classify(chrom=None, fdr=0.05, cluster_size=3, n_threads=-1)
```

Classifies significantly differential pixels according to which condition they are strongest in.

Parameters

- **chrom** (*str*) – The chromosome to classify significantly differential pixels on. Pass None to run for all chromosomes in series.
- **fdr** (*float or list of float*) – The FDR threshold used to identify clusters of significantly differential pixels via `self.threshold()`. Pass a list to do a sweep in series.
- **cluster_size** (*int or list of int*) – The cluster size threshold used to identify clusters of significantly differential pixels via `threshold()`. Pass a list to do a sweep in series.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.

collect (*fdr=0.05, cluster_size=3, n_threads=-1*)

Collects information on thresholded and classified differential interactions into a single TSV output file.

Parameters

- **fdr** (*float or list of float*) – The FDR threshold used to identify clusters of significantly differential pixels via `self.threshold()`. Pass a list to do a sweep in series.
- **cluster_size** (*int or list of int*) – The cluster size threshold used to identify clusters of significantly differential pixels via `threshold()`. Pass a list to do a sweep in series.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.

estimate_disp (*estimator='qcml', frac=None, auto_frac_factor=15.0, weighted_lowess=True, n_threads=-1*)

Estimates dispersion parameters.

Parameters

- **estimator** (*'cml', 'qcml', 'mme', or a function*) – Pass 'cml', 'qcml', 'mme' to use conditional maximum likelihood (CML), quantile-adjusted CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.
- **frac** (*float, optional*) – The lowess smoothing fraction to use when fitting the distance vs dispersion trend. Pass None to choose a value automatically.
- **auto_frac_factor** (*float*) – When `frac` is None, this factor scales the automatically determined fraction parameter.
- **weighted_lowess** (*bool*) – Whether or not to use a weighted lowess fit when fitting the smoothed dispersion curve.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple distance scales in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the distance scales serially.

lrt (*chrom=None, refit_mu=True, n_threads=-1, verbose=True*)

Runs the likelihood ratio test to test for differential interactions.

Parameters

- **chrom** (*str*) – The name of the chromosome to run the LRT for. Pass None to run for all chromosomes in series.
- **refit_mu** (*bool*) – Pass True to refit the mean parameters in the NB models being compared in the LRT. Pass False to use the means across replicates directly, which is simpler and slightly faster but technically violates the assumptions of the LRT.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.
- **verbose** (*bool*) – Pass False to silence reporting of progress to stderr.

prepare_data (*chrom=None, norm='conditional_mor', n_bins=-1, n_threads=-1, verbose=True*)

Prepares raw and normalized data for analysis.

Parameters

- **chrom** (*str*) – The name of the chromosome to prepare raw data for. Pass None to run for all chromosomes in series.
- **norm** (*str*) – The method to use to account for differences in sequencing depth. Valid options are:
 - **simple_scaling**: scale each replicate to equal total depth
 - **median_of_ratios**: use median of ratios normalization, ignoring pixels at which any replicate has a zero
 - **conditional_scaling**: apply simple scaling independently at each distance scale
 - **conditional_mor**: apply median of ratios independently at each distance scale
- **n_bins** (*int, optional*) – Number of distance bins to use during scaling normalization if **norm** is one of the conditional options. Pass 0 or None to match pixels by exact distance. Pass -1 to use a reasonable default: 1/5 of `self.dist_thesh_max`.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.
- **verbose** (*bool*) – Pass False to silence reporting of progress to stderr.

run_to_qvalues (*norm='conditional_mor', n_bins_norm=-1, estimator='qcml', frac=None, auto_frac_factor=15.0, weighted_lowess=True, refit_mu=True, n_threads=-1, verbose=True*)

Shortcut method to run the analysis to q-values.

Parameters

- **norm** (*str*) – The method to use to account for differences in sequencing depth. Valid options are:
 - **simple_scaling**: scale each replicate to equal total depth
 - **median_of_ratios**: use median of ratios normalization, ignoring pixels at which any replicate has a zero
 - **conditional_scaling**: apply simple scaling independently at each distance scale
 - **conditional_mor**: apply median of ratios independently at each distance scale
- **n_bins_norm** (*int, optional*) – Number of distance bins to use during scaling normalization if **norm** is one of the conditional options. Pass 0 or None to match pixels by exact distance. Pass -1 to use a reasonable default: 1/5 of `self.dist_thesh_max`.

- **estimator** (*'cml', 'qcml', 'mme', or a function*) – Pass 'cml', 'qcml', 'mme' to use conditional maximum likelihood (CML), qnorm-CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.
- **frac** (*float, optional*) – The lowess smoothing fraction to use when fitting the distance vs dispersion trend. Pass None to choose a value automatically.
- **auto_frac_factor** (*float*) – When *frac* is None, this factor scales the automatically determined fraction parameter.
- **weighted_lowess** (*bool*) – Whether or not to use a weighted lowess fit when fitting the smoothed dispersion curve.
- **refit_mu** (*bool*) – Pass True to refit the mean parameters in the NB models being compared in the LRT. Pass False to use the means across replicates directly, which is simpler and slightly faster but technically violates the assumptions of the LRT.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.
- **verbose** (*bool*) – Pass False to silence reporting of progress to stderr.

threshold (*chrom=None, fdr=0.05, cluster_size=3, n_threads=-1*)

Thresholds and clusters significantly differential pixels.

Should only be run after q-values have been obtained.

Parameters

- **chrom** (*str*) – The name of the chromosome to threshold. Pass None to threshold all chromosomes in series.
- **fdr** (*float or list of float*) – The FDR to threshold on. Pass a list to do a sweep in series.
- **cluster_size** (*int or list of int*) – Clusters smaller than this size will be filtered out. Pass a list to do a sweep in series.,
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass -1 to use as many threads as there are CPUs. Pass 0 to process the chromosomes serially.

hic3defdr.analysis.constructor module

```
class hic3defdr.analysis.constructor.HiC3DeFDR(raw_npz_patterns, bias_patterns,  
                                              chroms, design, out-  
                                              dir, dist_thresh_min=4,  
                                              dist_thresh_max=200, bias_thresh=0.1,  
                                              mean_thresh=1.0, loop_patterns=None,  
                                              res=None)
```

Bases: `hic3defdr.analysis.core.CoreHiC3DeFDR`, `hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR`, `hic3defdr.analysis.simulation.SimulatingHiC3DeFDR`, `hic3defdr.analysis.plotting.PlottingHiC3DeFDR`

Main object for hic3defdr analysis.

raw_npz_patterns

File path patterns to `scipy.sparse` formatted NPZ files containing raw contact matrices for each replicate, in order. Each file path pattern should contain at least one '<chrom>' which will be replaced with the chromosome name when loading data for specific chromosomes.

Type list of str

bias_patterns

File path patterns to `np.savetxt()` formatted files containing bias vector information for each replicate, in order. Each file path pattern should contain at least one '<chrom>' which will be replaced with the chromosome name when loading data for specific chromosomes.

Type list of str

chroms

List of chromosome names as strings. These names will be substituted in for '<chroms>' in the `raw_npz_patterns` and `bias_patterns`.

Type list of str

design

Pass a DataFrame with boolean dtype whose rows correspond to replicates and whose columns correspond to conditions. Replicate and condition names will be inferred from the row and column labels, respectively. If you pass a string, the DataFrame will be loaded via `pd.read_csv(design, index_col=0)`.

Type `pd.DataFrame` or str

outdir

Specify a directory to store the results of the analysis. Two different HiC3DeFDR analyses cannot co-exist in the same directory. The directory will be created if it does not exist.

Type str

dist_thresh_min, dist_thresh_max

The minimum and maximum interaction distance (in bin units) to include in the analysis.

Type int

bias_thresh

Bins with a bias factor below this threshold or above its reciprocal in any replicate will be filtered out of the analysis.

Type float

mean_thresh

Pixels with mean value below this threshold will be filtered out at the dispersion fitting stage.

Type float

loop_patterns

Keys should be condition names as strings, values should be file path patterns to sparse JSON formatted cluster files representing called loops in that condition. Each file path pattern should contain at least one '<chrom>' which will be replaced with the chromosome name when loading data for specific chromosomes.

Type dict of str, optional

res

The bin resolution, in base pair units, of the input contact matrix data. Used only when printing TSV output. Pass None to skip printing TSV output during the `threshold()` and `classify()` steps.

Type int, optional

hic3defdr.analysis.core module

class `hic3defdr.analysis.core.CoreHiC3DeFDR`

Bases: `object`

Mixin class providing core saving and loading functionality for HiC3DeFDR.

get_matrix (*name, chrom, row_slice, col_slice, rep=None, cond=None*)

Loads data with *name* as a dense matrix specified by *chrom*, *row_slice*, *col_slice*.

Parameters

- **name** (*str*) – The name (stage) of the data to load. Add a special suffix ‘_mean’ to average per-rep stages within conditions.
- **chrom** (*str*) – The chromosome to select data from.
- **col_slice** (*row_slice*,) – Row and column slice to use, respectively.
- **cond** (*rep*,) – Pass the rep name or condition name if the data specified by *name* has multiple columns.

Returns The selected dense matrix.

Return type `np.ndarray`

classmethod `load` (*outdir*)

Loads a HiC3DeFDR analysis object from disk.

It is safe to have multiple instances of the same analysis open at once.

Parameters **outdir** (*str*) – Folder path to where the HiC3DeFDR was saved.

Returns The loaded object.

Return type *HiC3DeFDR*

load_bias (*chrom*)

Loads the bias matrix for one chromosome.

The rows of the bias matrix correspond to bin indices along the chromosome. The columns correspond to the replicates.

The bias factors for bins that fail *bias_thresh* are set to zero. This is designed so that all pixels in these bins get dropped during union pixel set computation.

Parameters **chrom** (*str*) – The name of the chromosome to load the bias matrix for.

Returns The bias matrix.

Return type `np.ndarray`

load_data (*name, chrom=None, idx=None, rep=None, cond=None, coo=False*)

Loads arbitrary data for one chromosome or all chromosomes.

Parameters

- **name** (*str*) – The name of the data to load.
- **chrom** (*str, optional*) – The name of the chromosome to load data for. Pass `None` if this data is not organized by chromosome. Pass ‘all’ to load data for all chromosomes.
- **idx** (*np.ndarray or tuple of np.ndarray, optional*) – Pass a boolean array to load only a subset of the data. Pass a tuple of exactly two boolean arrays to chain the indices.

- **rep** (*str*, *optional*) – If the data is a (pixels, reps) rectangular array, pass the name of a rep to get a (pixels,) vector of data for just that rep.
- **cond** (*str*, *optional*) – If the data is a (pixels, conds) rectangular array, pass the name of a condition to get a (pixels,) vector of data for just that condition.
- **coo** (*bool*) – Pass True to return a COO-format (row, col, data) tuple of np.ndarray. chrom cannot be 'all', and idx must be None (this function will handle filtering automatically).

Returns data or (concatenated_data, offsets) or (row, col, data) – The loaded data for one chromosome, or a tuple of the concatenated data and an array of offsets per chromosome. offsets satisfies the following properties: offsets[0] == 0, offsets[-1] == concatenated_data.shape[0], and concatenated_data[offsets[i]:offsets[i+1], :] contains the data for the i`th chromosome. If called with ``coo=True, this function returns a COO-format (row, col, data) tuple of np.ndarray.

Return type np.ndarray

load_disp_fn (*cond*)

Loads the fitted dispersion function for a specific condition from disk.

Parameters **cond** (*str*) – The condition to load the dispersion function for.

Returns Vectorized. Takes in the value of the covariate the dispersion was fitted against and returns the appropriate dispersion.

Return type function

picklefile

save_data (*data*, *name*, *chrom=None*)

Saves arbitrary data for one chromosome to disk.

Parameters

- **data** (*np.ndarray*) – The data to save.
- **name** (*str*) – The name of the data to save.
- **chrom** (*str or np.ndarray, optional*) – The name of the chromosome to save data for, or None if this data is not organized by chromosome. Pass an np.ndarray of offsets to save data for all chromosomes.

save_disp_fn (*cond*, *disp_fn*)

Saves the fitted dispersion function for a specific condition and chromosome to disk.

Parameters

- **cond** (*str*) – The condition to save the dispersion function for.
- **disp_fn** (*function*) – The dispersion function to save.

hic3defdr.analysis.plotting module

class hic3defdr.analysis.plotting.**PlottingHiC3DeFDR**

Bases: object

Mixin class containing plotting functions for HiC3DeFDR.

plot_correlation_matrix (*stage='scaled', idx='loop', correlation='spearman', colorscale=(0.75, 1.0), **kwargs*)

Plots a matrix of pairwise correlations among all replicates.

Parameters

- **stage** (*{'raw', 'scaled'}*) – Specify the stage of the data to compute correlations between.
- **idx** (*{'disp', 'loop'}*) – Pass 'disp' to compute correlations for all points for which dispersion was estimated. Pass 'loop' to compute correlations for all points which are in loops (available only if `loop_patterns` was passed to the constructor).
- **correlation** (*{'spearman', 'pearson'}*) – Which correlation coefficient to compute.
- **colorscale** (*tuple of float*) – The min and max values of the correlation to use to color the matrix.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_dd_curves (*chrom, log=True, **kwargs*)

Plots the distance dependence curve before and after size factor adjustment.

Parameters

- **chrom** (*str*) – The name of the chromosome to plot the curve for.
- **log** (*bool*) – Whether or not to log the axes of the plot.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_ddr (*cond, dist_max=None, scatter_size=36, **kwargs*)

Fast alternative to `plot_dispersion_fit()` that only supports plotting distance versus dispersion, with no `hexbin` or `scatter_points` support.

Parameters

- **cond** (*str*) – The name of the chromosome and condition, respectively, to plot the fit for.
- **dist_max** (*int*) – If `xaxis` is 'dist', the maximum distance to include on the plot in bin units. Pass `None` to use `self.dist_thresh_max`.
- **scatter_size** (*int*) – The marker size when plotting scatterplots.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_dispersion_fit (*cond, xaxis='dist', yaxis='disp', dist_max=None, scatter_fit=-1, scatter_size=36, distance=None, hexbin=False, logx=False, logy=False, **kwargs*)

Plots a hexbin plot of pixel-wise distance vs either dispersion or variance, overlaying the estimated and fitted dispersions.

Parameters

- **cond** (*str*) – The name of the chromosome and condition, respectively, to plot the fit for.
- **xaxis** ('mean' or 'dist') – What to plot on the x-axis.
- **yaxis** ('disp' or 'var') – What to plot on the y-axis.
- **dist_max** (*int*) – If **xaxis** is 'dist', the maximum distance to include on the plot in bin units. Pass None to use `self.dist_thresh_max`.
- **scatter_fit** (*int*) – Pass a nonzero integer to draw the fitted dispersions passed in `disp` as a scatterplot of `scatter_fit` selected points. Pass -1 to plot the fitted dispersions passed in `disp` as a curve. Pass 0 to omit plotting the dispersion estimates altogether.
- **scatter_size** (*int*) – The marker size when plotting scatterplots.
- **distance** (*int*, *optional*) – Pick a specific distance in bin units to plot only interactions at that distance.
- **hexbin** (*bool*) – Pass False to skip plotting the hexbin plot, leaving only the estimated variances or dispersions.
- **logy** (*logx*,) – Whether or not to log the x- or y-axis, respectively.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_grid (*chrom*, *i*, *j*, *w*, *vmax*=100, *fdr*=0.05, *cluster_size*=3, *fdr_vmid*=0.05, *color_cycle*=('blue', 'green', 'purple', 'yellow', 'cyan', 'red'), *despine*=False, ***kwargs*)

Plots a combination visualization grid focusing on a specific pixel on a specific chromosome, combining heatmaps, cluster outlines, and stripplots.

Parameters

- **chrom** (*str*) – The name of the chromosome to slice matrices from.
- **j** (*i*,) – The row and column index of the pixel to focus on.
- **w** (*int*) – The size of the heatmap will be $2*w + 1$ bins in each dimension.
- **vmax** (*float*) – The maximum of the colorscale to use when plotting normalized heatmaps.
- **fdr** (*float*) – The FDR threshold to use when outlining clusters.
- **cluster_size** (*int*) – The cluster size threshold to use when outlining clusters.
- **fdr_vmid** (*float*) – The FDR value at the middle of the colorscale used for plotting the q-value heatmap.
- **color_cycle** (*list of matplotlib colors*) – The color cycle to use over conditions.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The first pyplot axis returned is injected by `@plotter`. The grid of pyplot axes is the second return value from the call to `plt.subplots()` that is used to create the grid. The function takes two args, an FDR and a cluster size, and redraws the cluster outlines using the new parameters.

Return type pyplot axis, grid of pyplot axes, function

plot_heatmap (*chrom*, *row_slice*, *col_slice*, *stage*='scaled', *rep*=None, *cond*=None, *cmap*='Reds', *vmin*=0, *vmax*=100, ***kwargs*)

Plots a simple heatmap of a slice of the contact matrix.

Parameters

- **chrom** (*str*) – The chromosome to plot.
- **col_slice** (*row_slice*,) – The row and column slice, respectively, to plot.
- **stage** (*str*) – The stage of the data to plot.
- **cond** (*rep*,) – Pass the rep name or condition name if the data specified by *stage* has multiple columns.
- **cmap** (*matplotlib colormap or dict*) – The colormap to use for the heatmap.
- **vmax** (*vmin*,) – The *vmin* and *vmax* to use for the heatmap colorscale.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_ma (*fdr*=0.05, *conds*=None, *include_non_loops*=True, *s*=-1, *nonloop_s*=None, *density_dpi*=72, *vmax*=None, *nonloop_vmax*=None, *ax*=None, *legend*=True, ***kwargs*)

Plots an MA plot for a given chromosome.

Parameters

- **fdr** (*float*) – The threshold to use for labeling significantly differential loop pixels.
- **conds** (*tuple of str, optional*) – Pass a tuple of two condition names to compare those two conditions. Pass None to compare the first two conditions.
- **include_non_loops** (*bool*) – Whether or not to include non-looping pixels in the MA plot.
- **s** (*float*) – The marker size to use for the scatterplot, or -1 to use a scatter density plot.
- **nonloop_s** (*float, optional*) – Pass a separate marker size to use specifically for the non-loop pixels if *include_non_loops* is True. Useful for drawing just the non-loop pixels as a density by passing *s*=1, *nonloop_s*=-1. Pass None to use *s* as the size for both loop and non-loop pixels.
- **density_dpi** (*int*) – If *s* or *nonloop_s* are -1 this specifies the DPI to use for the density grid.
- **nonloop_vmax** (*vmax*,) – The *vmax* to use for *ax.scatter_density()* if *s* or *nonloop_s* is -1, respectively. Pass None to choose values automatically.
- **ax** (*pyplot axis*) – The axis to plot to. Must have been created with *projection*='scatter_density'. Pass None to create a new axis.
- **legend** (*bool*) – Pass True to add a legend. Note that passing *legend*='outside' is not supported.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_pvalue_distribution (*idx*='disp', ***kwargs*)

Plots the p-value distribution across all chromosomes.

Parameters

- **idx** (`{'disp', 'loop'}`) – Pass ‘disp’ to plot p-values for all points for which dispersion was estimated. Pass ‘loop’ to plot p-values for all points which are in loops (available only if `loop_patterns` was passed to the constructor).
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

plot_qvalue_distribution (***kwargs*)

Plots the q-value distribution across all chromosomes.

Parameters **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.analysis.simulation module

class `hic3defdr.analysis.simulation.SimulatingHiC3DeFDR`

Bases: `object`

Mixin class containing plotting functions for HiC3DeFDR.

evaluate (*cluster_pattern, label_pattern, min_dist=None, max_dist=None, rerun_bh=False, outfile=None*)

Evaluates the results of this analysis, comparing it to true labels.

Parameters

- **cluster_pattern** (*str*) – File path pattern to sparse JSON formatted cluster files representing loop cluster locations. Should contain at least one ‘<chrom>’ which will be replaced with the chromosome name when loading data for specific chromosomes. Pass a condition name to use `self.loop_patterns[cluster_pattern]` instead.
- **label_pattern** (*str*) – File path pattern to true label files for each chromosome. Should contain at least one ‘<chrom>’ which will be replaced with the chromosome name when loading data for specific chromosomes. Files should be loadable with `np.loadtxt(..., dtype='U7')` to yield a vector of true labels parallel to the clusters pointed to by `cluster_pattern`.
- **max_dist** (*min_dist,*) – Specify minimum and maximum distances to evaluate performance within, respectively. Pass `None` to leave one or both ends unbounded.
- **rerun_bh** (*bool*) – If `min_dist` and/or `max_dist` are used to constrain the distances, pass `True` to re-run BH-FDR on the subset of p-values at the selected distances. Pass `False` to use the original dataset-wide q-values. Does nothing if `min_dist` and `max_dist` are both `None`.
- **outfile** (*str, optional*) – Name of a file to save the evaluation results to inside this object’s `outdir`. Default is ‘eval.npz’ if `min_dist` and `max_dist` are both `None`, otherwise it is ‘eval_<min_dist>_<max_dist>.npz’.

simulate (*cond, chrom=None, beta=0.5, p_diff=0.4, skip_bias=False, loop_pattern=None, outdir='sim', n_threads=-1, verbose=True*)

Simulates raw contact matrices based on previously fitted scaled means and dispersions in a specific condition.

Can only be run after `estimate_dispersions()` has been run.

Parameters

- **cond** (*str*) – Name of the condition to base the simulation on.
- **chrom** (*str, optional*) – Name of the chromosome to simulate. Pass `None` to simulate all chromosomes in series.
- **beta** (*float*) – The effect size of the loop perturbations to use when simulating. Perturbed loops will be strengthened or weakened by this fraction of their original strength.
- **p_diff** (*float or list of float*) – Pass a single float to specify the probability that a loop will be perturbed across the simulated conditions. Pass four floats to specify the probabilities of all four specific perturbations: up in A, down in A, up in B, down in B. The remaining loops will be constitutive.
- **skip_bias** (*bool*) – Pass `True` to set all bias factors and size factors to 1, effectively simulating “unbiased” raw data.
- **loop_pattern** (*str, optional*) – File path pattern to sparse JSON formatted cluster files representing loop cluster locations for the simulation. Should contain at least one ‘<chrom>’ which will be replaced with the chromosome name when loading data for specific chromosomes. Pass `None` to use `self.loop_patterns[cond]`.
- **outdir** (*str*) – Path to a directory to store the simulated data to.
- **n_threads** (*int*) – The number of threads (technically GIL-avoiding child processes) to use to process multiple chromosomes in parallel. Pass `-1` to use as many threads as there are CPUs. Pass `0` to process the chromosomes serially.
- **verbose** (*bool*) – Pass `False` to silence reporting of progress to `stderr`.

Module contents

hic3defdr.plotting package

Submodules

hic3defdr.plotting.dispersion module

`hic3defdr.plotting.dispersion.compare_disp_fits` (*fit_fns, labels, max_dist=200, colors=None, linestyle=None, legend=True, **kwargs*)

Compares multiple dispersion fit functions.

Parameters

- **fit_fns** (*list of functions*) – The fit functions to compare. Each function should be vectorized and take one argument (the distance) and return one value (the fitted dispersion at that distance).
- **labels** (*list of str*) – The labels to use for each fit function being compared.
- **max_dist** (*int*) – The maximum distance out to which the plot should be drawn.
- **colors** (*list of valid matplotlib colors, optional*) – Pass a list of colors to color each fit function with. Pass `None` to color them automatically.

- **linestyles** (*list of valid matplotlib linestyles, optional*) – Pass a list of linestyles to plot each fit function with. Pass None to use the default linestyle.
- **legend** (*bool*) – Pass True to include a legend on the plot.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

```
hic3defdr.plotting.dispersion.plot_ddd (dist_per_bin, disp_per_bin, disp_fn, scatter_size=36, legend=True, **kwargs)
```

Simplified plotter to visualized distance-dispersion relationships.

Plots per-distance dispersion estimates and the fitted dispersion curve.

Parameters

- **disp_per_bin** (*dist_per_bin,*) – The distances and estimated dispersions for each distance, respectively.
- **disp_fn** (*function*) – The smoothed dispersion function. Returns the smoothed dispersion as a function of distance.
- **scatter_size** (*int*) – The marker size when plotting scatter plots.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

```
hic3defdr.plotting.dispersion.plot_mvr (pixel_mean, pixel_var=None, pixel_disp=None, pixel_dist=None, pixel_var_fit=None, pixel_disp_fit=None, mean_per_bin=None, dist_per_bin=None, var_per_bin=None, disp_per_bin=None, fit_align_dist=False, xaxis='mean', yaxis='var', dist_max=200, mean_min=5.0, scatter_fit=-1, scatter_size=36, hexbin=True, logx=True, logy=True, xlim=None, ylim=None, legend=True, **kwargs)
```

Plots pixel-wise, bin-wise, and estimated dispersions in terms of either dispersion or variance versus either pixel-wise mean or distance.

Parameters

- **pixel_mean** (*np.ndarray*) – The pixel-wise mean.
- **pixel_disp** (*pixel_var,*) – The pixel-wise variance and dispersion. Pass only one of these.
- **pixel_dist** (*np.ndarray, optional*) – The pixel-wise distance. Not needed for simple MVR plotting.
- **pixel_disp_fit** (*pixel_var_fit,*) – The smoothed pixel-wise variance or dispersion estimates. Pass only one of these.
- **dist_per_bin** (*mean_per_bin,*) – The mean or distance of each bin used for estimating the dispersions but before any smoothing was performed. Pass only one of these.
- **disp_per_bin** (*var_per_bin,*) – The estimated variance or dispersion of each bin before any smoothing was performed. Pass only one of these.

- **fit_align_dist** (*bool*) – Pass True if the var/disp was fitted as a function of distance, in which case the fitted vars/disps will be aligned to distance rather than in a pixel-wise fashion. Use this to fix “jagged” fit lines caused by setting `xaxis` to a different value than the one the fitting was actually performed against.
- **xaxis** (*'mean' or 'dist'*) – What to plot on the x-axis.
- **yaxis** (*'var' or 'disp'*) – What to plot on the y-axis.
- **dist_max** (*int*) – If `xaxis` is “dist”, the maximum distance to plot in bin units.
- **mean_min** (*float*) – If `xaxis` is “mean”, the minimum mean to plot.
- **scatter_fit** (*int*) – Pass a nonzero integer to draw the fitted vars/disps as a scatter plot of `scatter_fit` selected points. Pass -1 to plot the fitted vars/disps as a curve. Pass 0 to omit plotting the var/disp estimates altogether.
- **scatter_size** (*int*) – The marker size when plotting scatter plots.
- **hexbin** (*bool*) – Pass False to skip plotting the hexbins, leaving only the estimated variances or dispersions.
- **logy** (*logx,*) – Whether or not to log the x- or y-axis, respectively.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.distance_bias module

```
hic3defdr.plotting.distance_bias.plot_distance_bias(ob, bins, bin_labels=None,  
                                                    idx='disp', threshold=0.05,  
                                                    colors=None, labels=None,  
                                                    xlabel='distance range', leg-  
                                                    end_label='group', **kwargs)
```

Plots a bar plot illustrating the degree to which p-values are biased among different distance scales.

This method visualizes distance bias by computing a specified percentile of all p-values called in an analysis, and then computing the proportion of pixels with p-values below this percentile in each distance bin.

Parameters

- **ob** (*hic3defdr.HiC3DeFDR object or list of hic3defdr.HiC3DeFDR objects*) – The analysis or analyses to inspect for distance bias.
- **bins** (*list of tuple of int*) – Each tuple represents a distance bin as a (min, max) pair, where min and max are distances in bin units and the ranges are inclusive. If either min or max is None, the distance bin will be considered unbounded on that end.
- **bin_labels** (*list of str*) – Pass a list of labels to describe the distance bins.
- **idx** (*{'disp', 'loop'}*) – Pass ‘disp’ to use p-values for all points for which dispersion was estimated. Pass ‘loop’ to only use p-values for points which are in loops.
- **threshold** (*float*) – The percentile to use for the comparison.
- **colors** (*str or list of str, optional*) – If `ob` is a single object, pass a single color to color the bars in the barplot. If `ob` is a list of objects, pass a list of colors. Pass None to use automatic colors.

- **labels** (*list of str, optional*) – If *ob* is a list of objects, you must pass a list of strings to label the objects. Otherwise, this kwarg does nothing.
- **xlabel** (*str*) – The label for the x-axis.
- **legend_label** (*str*) – If *ob* is a list of objects, the label to use for the legend title. Otherwise, this kwarg does nothing.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.distance_dependence module

`hic3defdr.plotting.distance_dependence.plot_dd_curves` (*row, col, before, after, rep-names=None, log=True, **kwargs*)

Plots a comparison of distance dependence curves before and after scaling.

Parameters

- **col** (*row,*) – Row and column indices identifying the location of pixels in *before* and *after*.
- **after** (*before,*) – Counts per pixel before and after scaling, respectively.
- **repnames** (*list of str, optional*) – Pass the replicate names in the same order as the columns of *before* and *after*.
- **log** (*bool*) – Pass True to log both axes of the distance dependence plot.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.fdr module

`hic3defdr.plotting.fdr.plot_fdr` (*eval_results, labels, colors=None, p_alt=None, **kwargs*)

Plots an FDR control curve from a list of `eval.npz`-style results.

Parameters

- **eval_results** (*list of dict-like*) – The dicts should have keys ‘thresh’ and ‘fdr’ whose values are parallel vectors describing the thresholds and FDRs to use for the FDR control curve. Each dict in the list represents a different FDR control curve which will be overlayed in the plot.
- **labels** (*list of str*) – List of labels parallel to *eval_results* providing names for each curve.
- **colors** (*list of valid matplotlib colors, optional*) – Colors for each FDR curve. Pass None to automatically assign colors.
- **p_alt** (*float, optional*) – Pass the true proportion of alternative (non-null) points to draw a dashed line representing the optimal BH-FDR control line and shade the zone of successful FDR control. Pass None to draw a dashed line indicating the boundary of successful FDR control.

- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.fn_vs_fp module

`hic3defdr.plotting.fn_vs_fp.plot_fn_vs_fp(eval_results, labels, threshold=0.15, colors=None, xlabel='label', **kwargs)`

Plots two bar plots, one showing FNR and the other showing FPR at a fixed threshold.

Parameters

- **eval_results** (*list of dict-like or list of list of dict-like*) – The dicts should have keys ‘thresh’, ‘fpr’, and ‘tpr’ whose values are parallel vectors describing the thresholds, FPRs, and TPRs to use for the bar plots. Each dict in the list represents a different bar which will be added to each bar plot. Pass a nested list of dicts to draw grouped bar plots, denoting the outer list by color and the inner list by x-axis position.
- **labels** (*list of str or (list of str, list of str)*) – List of labels parallel to `eval_results` providing names for each bar. If `eval_results` is a nested list, pass a tuple of two lists. The first list should provide the labels for the outer list grouping (`len(labels[0]) == len(eval_results)`) and the second should provide the labels for the inner list grouping (`len(labels[1]) == len(eval_results[i])` for any `i`).
- **threshold** (*float*) – The fixed threshold at which the FNR and FPR will be plotted. In practice, this function will use the closest threshold found in each dict in `eval_results`.
- **colors** (*matplotlib color or list of colors, optional*) – Specify the color to use for the bars in the bar plot. If `eval_results` is a nested list, pass a list of colors to color core the outer list grouping (`len(colors) == len(eval_results)`). Pass `None` to use automatic colors.
- **xlabel** (*str*) – The label to use for the x-axis.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The first pyplot axis returned is injected by `@plotter`. The array of pyplot axes is the second return value from the call to `plt.subplots()` that is used to create the pair of barplots.

Return type pyplot axis, array of pyplot axes

hic3defdr.plotting.grid module

`hic3defdr.plotting.grid.plot_grid(i, j, w, row, col, raw, scaled, mu_hat_alt, mu_hat_null, qvalues, disp_idx, loop_idx, design, fdr, cluster_size, vmax=100, fdr_ymin=0.05, color_cycle=('blue', 'green', 'purple', 'yellow', 'cyan', 'red'), despine=False, **kwargs)`

Plots a combination visualization grid focusing on a specific pixel on a specific chromosome, combining heatmaps, cluster outlines, and stripplots.

Parameters

- **j** (*i*,) – The row and column index of the pixel to focus on.
- **w** (*int*) – The size of the heatmap will be $2 * w + 1$ bins in each dimension.

- **col** (*row*,) – The row and column indices corresponding to the rows of the *raw* and *scaled* matrices.
- **scaled** (*raw*,) – The raw and scaled data for each pixel (rows) and each replicate (columns).
- **mu_hat_null** (*mu_hat_alt*,) – The estimated mean parameter under the alternate and null model, respectively. First dimension is pixels for which dispersion was estimated, whose row and column coordinates in the complete square matrix are given by *row[disp_idx]* and *col[disp_idx]*, respectively. Columns of *mu_hat_alt* correspond to conditions, while *mu_hat_null* has no second dimension.
- **qvalues** (*np.ndarray*) – Vector of q-values called per pixel whose dispersion was estimated and which lies in a loop. The row and column coordinates in the complete square matrix are given by *row[disp_idx][loop_idx]* and *col[disp_idx][loop_idx]*, respectively.
- **disp_idx** (*np.ndarray*) – Boolean matrix indicating which pixels in *zip(row, col)* had their dispersion estimated.
- **loop_idx** (*np.ndarray*) – Boolean matrix indicating which pixels in *zip(row[disp_idx], col[disp_idx])* lie within loops.
- **design** (*pd.DataFrame*) – Pass a DataFrame with boolean dtype whose rows correspond to replicates and whose columns correspond to conditions. Replicate and condition names will be inferred from the row and column labels, respectively.
- **fdr** (*float*) – The FDR threshold to use when outlining clusters.
- **cluster_size** (*int*) – The cluster size threshold to use when outlining clusters.
- **vmax** (*float*) – The maximum of the colorscale to use when plotting normalized heatmaps.
- **fdr_vmid** (*float*) – The FDR value at the middle of the colorscale used for plotting the q-value heatmap.
- **color_cycle** (*list of matplotlib colors*) – The color cycle to use over conditions.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The first pyplot axis returned is injected by `@plotter`. The grid of pyplot axes is the second return value from the call to `plt.subplots()` that is used to create the grid. The function takes two args, an FDR and a cluster size, and redraws the cluster outlines using the new parameters.

Return type pyplot axis, grid of pyplot axes, function

hic3defdr.plotting.heatmap module

`hic3defdr.plotting.heatmap.plot_heatmap` (*matrix*, *cmap='Reds'*, *vmin=0*, *vmax=100*, *despine=False*, ***kwargs*)

Plots a simple heatmap of a dense matrix.

Parameters

- **matrix** (*np.ndarray*) – The dense matrix to visualize.
- **cmap** (*matplotlib colormap*) – The colormap to use for the heatmap.
- **vmax** (*vmin*,) – The vmin and vmax to use for the heatmap colorscale.

- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.histograms module

hic3defdr.plotting.histograms.**plot_pvalue_histogram**(*data*, *xlabel*='pvalue',
***kwargs*)

Plots a p-value or q-value distribution.

Parameters

- **data** (*np.ndarray*) – The p-values or q-values to plot.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

hic3defdr.plotting.ma module

hic3defdr.plotting.ma.**plot_ma**(*data*, *sig_idx*, *loop_idx*=None, *names*=None, *s*=-1,
nonloop_s=None, *density_dpi*=72, *vmax*=None, *non-*
loop_vmax=None, *ax*=None, *legend*=True, ***kwargs*)

Plots an MA plot.

Parameters

- **data** (*np.ndarray*) – Unlogged values to use for computing M and A for each pixel (rows) in each condition (columns).
- **sig_idx** (*np.ndarray*) – Boolean vector indicating which pixels among those that are in *loop_idx* are significantly differential.
- **loop_idx** (*np.ndarray*, *optional*) – Boolean vector indicating which pixels in *data* are in loops. Pass None if all pixels in *data* are in loops.
- **names** (*tuple of str*, *optional*) – The names of the two conditions being compared.
- **s** (*float*) – The marker size to pass to *ax.scatter()*. Pass -1 to use *ax.scatter_density()* instead, avoiding plotting each point separately. See Notes below for more details and caveats.
- **nonloop_s** (*float*, *optional*) – Pass a separate marker size to use specifically for the non-loop pixels if *loop_idx* is not None. Useful for drawing just the non-loop pixels as a density by passing *s*=1, *nonloop_s*=-1. Pass None to use *s* as the size for both loop and non-loop pixels.
- **density_dpi** (*int*) – If *s* is -1 this specifies the DPI to use for the density grid.
- **nonloop_vmax** (*vmax*,) – The *vmax* to use for *ax.scatter_density()* if *s* or *nonloop_s* is -1, respectively. Pass None to choose values automatically.
- **ax** (*pyplot axis*) – The axis to plot to. Must have been created with *projection*='scatter_density'. Pass None to create a new axis.
- **legend** (*bool*) – Pass True to add a legend. Note that passing *legend*='outside' is not supported.

- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

Notes

It is recommended to use `ax.scatter_density()` from the `mpl_scatter_density` module to plot the data by passing `s=-1`. This avoids the massive slowdown experienced when plotting one separate dot for each pixel in the genome in the case where `loop_idx=None`. In order to support `ax.scatter_density()`, a new pyplot figure and axis must be created with `projection='scatter_density'`. Therefore, even though this function is decorated with `@plotter`, it has a non-standard behavior for interpreting the `ax` kwarg: if `ax=None` is passed, it will create a new figure and axis rather than re-using the current axis.

hic3defdr.plotting.roc module

`hic3defdr.plotting.roc.plot_roc` (*eval_results, labels, colors=None, **kwargs*)

Plots an ROC curve from a list of `eval.npz`-style results.

Parameters

- **eval_results** (*list of dict-like*) – The dicts should have keys ‘thresh’, ‘fpr’, and ‘tpr’ whose values are parallel vectors describing the thresholds, FPR, and TPR to use for the ROC curve. Each dict in the list represents a different ROC curve which will be overlaid in the plot.
- **labels** (*list of str*) – List of labels parallel to `eval_results` providing names for each curve.
- **colors** (*list of valid matplotlib colors, optional*) – Colors for each ROC curve. Pass `None` to automatically assign colors.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

Module contents

hic3defdr.util package

Submodules

hic3defdr.util.apa module

`hic3defdr.util.apa.make_apu_stack` (*matrix, clusters, width, min_dist=None*)

Creates a stack of square matrix slices centered on each cluster in a list.

Parameters

- **matrix** (*scipy.spmatrix*) – The matrix to take slices of.
- **clusters** (*list of clusters*) – The clusters around which to take the slices. Each cluster should be a list of [x, y] pairs representing the points in that cluster.

- **width** (*int*) – The width of the slice to take in bin units. Should be odd.
- **min_dist** (*int*, *optional*) – Clusters with interaction distances shorter than this (in bin units) will be given an slice of all nan's. Pass None to use a sane default.

Returns This array will have three dimensions. The first axis represents the clusters, the next two represent the square slice for each cluster. The array may contain nan's.

Return type np.ndarray

hic3defdr.util.balancing module

hic3defdr.util.balancing.**kr_balance** (*array*, *tol=1e-06*, *x0=None*, *delta=0.1*, *ddelta=3*, *fl=1*,
max_iter=3000)

Balances a matrix via Knight-Ruiz matrix balancing, using sparse matrix operations.

Parameters

- **array** (*np.ndarray* or *scipy.sparse.spmatrix*) – The matrix to balance. Must be 2 dimensional and square. Will be symmetrized using its upper triangular entries.
- **tol** (*float*) – The error tolerance.
- **x0** (*np.ndarray*, *optional*) – The initial guess for the bias vector. Pass None to use a vector of all 1's.
- **ddelta** (*delta*,) – How close/far the balancing vectors can get to/from the positive cone, in terms of a relative measure on the size of the elements.
- **fl** (*int*) – Determines whether or not internal convergence statistics will be printed to standard output.
- **max_iter** (*int*) – The maximum number of iterations to perform.

Returns The CSR matrix is the balanced matrix. It will be upper triangular if *array* was upper triangular, otherwise it will be symmetric. The first np.ndarray is the bias vector. The second np.ndarray is a list of residuals after each iteration.

Return type sparse.csr_matrix, np.ndarray, np.ndarray

Notes

The core logic of this implementation is transcribed (by Dan Gillis) from the original Knight and Ruiz IMA J. Numer. Anal. 2013 paper and differs only slightly from the implementation in Juicer/Juicebox (see comments).

It then uses the “sum factor” approach from the Juicer/Juicebox code to scale the bias vectors up to match the scale of the original matrix (see comments).

Overall, this function is not nan-safe, but you may pass matrices that contain empty rows (the matrix will be shrunk before balancing, but all outputs will match the shape of the original matrix).

This function does not perform any of the logic used in Juicer/Juicebox to filter out sparse rows if the balancing fails to converge.

If the *max_iter* limit is reached, this function will return the current best balanced matrix and bias vector - no exception will be thrown. Callers can compare the length of the list of residuals to *max_iter* - if they are equal, the algorithm has not actually converged, and the caller may choose to perform further filtering on the matrix and try again. The caller is also free to decide if the matrix is “balanced enough” using any other criteria (e.g., variance of nonzero row sums).

hic3defdr.util.banded_matrix module

```
class hic3defdr.util.banded_matrix.BandedMatrix(data, shape=None, copy=False,
max_range=300, upper=None)
```

Bases: `scipy.sparse.dia.dia_matrix`

```
classmethod align(*matrices)
```

```
classmethod apply(f, *args, **kwargs)
```

Applies a function that takes in raw banded matrices (rectangular dense arrays) and returns raw banded matrices to one or more BandedMatrices.

Parameters

- **f** (*function*) – The function to apply. At least its first positional arg (or as many as all of them) should be an `np.ndarray` corresponding to the data array of a BandedMatrix. It should return one `np.ndarray` or a tuple of `np.ndarray` corresponding to the data array(s) of the resulting BandedMatrix(es).
- ***args** (*positional arguments*) – Passed through to `f()`. Instances of BandedMatrix will be passed as just their data arrays. If multiple BandedMatrices are passed, no attempt will be made to align them - a `ValueError` will be raised if they are not aligned.
- ****kwargs** (*keyword arguments*) – Passed through to `f()`. No conversion from BandedMatrix will be attempted.

Returns If `f()` returns a single `np.ndarray`, `BandedMatrix.apply(f, arg1, ...)` will return one BandedMatrix, whose shape and offsets will be taken from `arg1` (which must always be an BandedMatrix) and whose data will be taken from `f(arg1.data, ...)`. If `f()` returns a tuple of `np.ndarray`, they will each be repackaged into an BandedMatrix in this fashion, and the tuple of new BandedMatrix will be returned.

Return type *BandedMatrix* or tuple of BandedMatrix

```
copy()
```

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

```
data_indices(key)
```

```
deconvolute(bias, invert=False)
```

Applies a bias vector to both the rows and the columns of the BandedMatrix.

Parameters

- **bias** (`np.ndarray`) – Bias vector to apply. Should match size of BandedMatrix.
- **invert** (`bool`) – Pass `True` to invert the bias vector (divide the BandedMatrix by the the outer product of the bias vector with itself). Pass `False` to multiply the BandedMatrix by the outer product of the bias vector with itself.

Returns The result of the deconvolution.

Return type *BandedMatrix*

Examples

```
>>> import numpy as np
>>> from hic3defdr.util.banded_matrix import BandedMatrix
>>> a = np.arange(16).reshape(4, 4).astype(float)
```

(continues on next page)

(continued from previous page)

```

>>> a += a.T
>>> bias = np.sqrt(np.sum(a, axis=0))
>>> b = ((a / bias).T / bias).T
>>> bm = BandedMatrix.from_ndarray(b, max_range=3)
>>> bm = bm.deconvolute(bias)
>>> np.allclose(bm.toarray(), a)
True

```

flatten()

classmethod from_dia_matrix (*dia_matrix*, *copy=True*)

classmethod from_ndarray (*ndarray*, *max_range=300*, *upper=None*)

classmethod from_sparse (*spmatrix*, *max_range=300*, *upper=None*)

Constructs a BandedMatrix from a `scipy.sparse.spmatrix` instance.

Parameters

- **spmatrix** (*scipy.sparse.spmatrix*) – The sparse matrix to convert to BandedMatrix.
- **max_range** (*int*) – Offdiagonals beyond the *max_range*'th offdiagonal will be discarded.
- **upper** (*bool*, *optional*) – Pass True to keep only the upper triangular entries. Pass False to keep all the entries. Pass None to guess what to do based on whether or not the input matrix is upper triangular.

Returns The new BandedMatrix.

Return type *BandedMatrix*

Notes

Mostly stolen from `sparse.coo_matrix.todia()`, plus nan padding from `BandedMatrix.from_ndarray()` and nan triangles from `BandedMatrix.roll_matrix()`.

classmethod is_bandedmatrix (*x*)

is_upper ()

classmethod load (*fname*, ***kwargs*)

log ()

static make_mask (*matrix*, *min_range=None*, *max_range=None*, *upper=False*, *nan=False*)

General-purpose function for creating masks for contact matrices.

Parameters

- **matrix** (*np.ndarray*) – Square contact matrix to make a mask for.
- **max_range** (*min_range*,) – Pass ints to specify a minimum and maximum allowed interaction range, in bin units.
- **upper** (*bool*) – Pass True to restrict the mask to the upper triangular entries.
- **nan** (*bool*) – Pass True to restrict the mask to non-NaN points.

Returns The mask. Same shape as *matrix*, with bool dtype.

Return type `np.ndarray`

make_upper (*pad=True*)

classmethod `max(*matrices)`

Returns the element-wise maximum of a list of BandedMatrices.

Parameters `*matrices` (*sequence of BandedMatrix*) – The matrices to compute the maximum of. Must already be aligned.

Returns The maximum.

Return type *BandedMatrix*

Notes

If the matrices aren't aligned, consider:

```
BandedMatrix.max(BandedMatrix.align(bm_a, bm_b, bm_c, ...))
```

max_range()

ravel()

reshape (*self, shape, order='C', copy=False*)

Gives a new shape to a sparse matrix without changing its data.

Parameters

- **shape** (*length-2 tuple of ints*) – The new shape should be compatible with the original shape.
- **order** (*{'C', 'F'}, optional*) – Read the elements using this index order. 'C' means to read and write the elements using C-like index order; e.g. read entire first row, then second row, etc. 'F' means to read and write the elements using Fortran-like index order; e.g. read entire first column, then second column, etc.
- **copy** (*bool, optional*) – Indicates whether or not attributes of self should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **reshaped_matrix** – A sparse matrix with the given *shape*, not necessarily of the same format as the current object.

Return type sparse matrix

See also:

numpy.matrix.reshape() NumPy's implementation of 'reshape' for matrices

static roll_matrix (*matrix, max_range=300*)

save (*fname*)

symmetrize ()

where (*a, b*)

`hic3defdr.util.banded_matrix.roll_footprint` (*footprint*)

"Rolls" each row to the right by its row index, then reverses the row order and transposes the result. This is equivalent to the operation `f(x) = sparse.dia_matrix(x).data` where `x` is a dense matrix, as long as `sparse.dia_matrix(x).offsets` is a sequential matrix (this is guaranteed by the BandedMatrix subclass).

The result of this is that the rolled footprint is suitable for use in convolution against `sparse.dia_matrix.data`, for example as `convolve(sparse.dia_matrix(x).data,`

`roll_footprint (footprint)` where `x` is a dense matrix and `footprint` is expressed in terms of the dense spatial coordinates.

Parameters `footprint` (`np.ndarray`) – The footprint to roll.

Returns The rolled footprint.

Return type `np.ndarray`

hic3defdr.util.binning module

`hic3defdr.util.binning.equal_bin (data, n_bins)`

Bins data into `n_bins` bins, with an equal number of points in each bin.

<https://stackoverflow.com/a/40895507>

Parameters

- **data** (`np.ndarray`) – The data to bin.
- **n_bins** (`int`) – The number of bins to bin into.

Returns A vector of integers representing the bin index for each entry in `data`.

Return type `np.ndarray`

hic3defdr.util.classification module

`hic3defdr.util.classification.classify (row, col, value, clusters)`

Classifies pixels (specified by `row`, `col` coordinates) into classes (corresponding to the columns of `value`) where their value is highest, ignoring those pixels that don't belong to the specified clusters.

Parameters

- **col** (`row,`) – Row and column indices, respectively, identifying pixels which correspond to rows of `value`.
- **value** (`np.ndarray`) – Value to use to assign classes. Rows correspond to pixels, columns to conditions.
- **clusters** (*list of set of tuple*) – Only pixels in these clusters will be assigned classes.

Returns The outer list represents classes (one per condition). Its elements represent clusters of points with that class/condition.

Return type list of list of set of tuple

hic3defdr.util.cluster_table module

`hic3defdr.util.cluster_table.add_columns_to_cluster_table (cluster_table, name_pattern, row, col, data, labels=None, reducer='mean', chrom=None)`

Adds new data columns to an existing cluster table by evaluating a sparse dataset specified by `row`, `col`, `data` at the pixels in each cluster and combining the resulting values using `reducer`.

This function operates in-place.

Parameters

- **cluster_table** (*pd.DataFrame*) – Must contain a “cluster” column. If the values in this column are strings, they will be “corrected” to list of lists of int in-place.
- **name_pattern** (*str*) – The name of the column to fill in. If data contains more than one column, multiple columns will be added - include exactly one %s in the name_pattern, then the *i* th new column will be called name_pattern % labels[i].
- **col, data** (*row,*) – Sparse format data to use to determine the value to fill in for each cluster for each new column. row and col must be parallel to the first dimension of data. If data is two-dimensional, you must pass labels to label the columns and include a %s in name_pattern.
- **labels** (*list of str, optional*) – If data is two-dimensional, pass a list of strings labeling the columns of data.
- **reducer** ({'mean', 'max', 'min'}) – The function to use to combine the values for the pixels in each cluster.
- **chrom** (*str, optional*) – If the cluster_table contains data from multiple chromosomes, pass the name of the chromosome that row, col, data correspond to and only clusters for that chromosome will have their new column created/updated. If the cluster_table contains data from only one chromosome, pass None to update all clusters in the cluster_table.

Examples

```
>>> import numpy as np
>>> from hic3defdr.util.cluster_table import clusters_to_table, \
...     add_columns_to_cluster_table
>>> # basic test: clusters all on one chromosome
>>> clusters = [[(1, 2), (1, 1)], [(4, 4), (3, 4)]]
>>> res = 10000
>>> df = clusters_to_table(clusters, 'chrX', res)
>>> row, col = zip(*sum(clusters, []))
>>> data = np.array([[1, 2],
...                 [3, 4],
...                 [5, 6],
...                 [7, 8]], dtype=float)
>>> add_columns_to_cluster_table(df, '%s_mean', row, col, data,
...                             labels=['rep1', 'rep2'])
>>> df.iloc[0, :]
us_chrom      chrX
us_start      10000
us_end        20000
ds_chrom      chrX
ds_start      10000
ds_end        30000
cluster_size      2
cluster      [[1, 2], [1, 1]]
rep1_mean        2
rep2_mean        3
Name: chrX:10000-20000_chrX:10000-30000, dtype: object
>>> # advanced test: two chromosomes
>>> df1 = clusters_to_table(clusters, 'chr1', res)
>>> df2 = clusters_to_table(clusters, 'chr2', res)
```

(continues on next page)

(continued from previous page)

```

>>> df = pd.concat([df1, df2], axis=0)
>>> # add chr1 info
>>> add_columns_to_cluster_table(df, '%s_mean', row, col, data,
...                             labels=['rep1', 'rep2'], chrom='chr1')
>>> # chr1 cluster has data filled in
>>> df.loc[df.index[0], ['rep1_mean', 'rep2_mean']]
rep1_mean    2
rep2_mean    3
Name: chr1:10000-20000_chr1:10000-30000, dtype: object
>>> # chr2 cluster has nans
>>> df.loc[df.index[2], ['rep1_mean', 'rep2_mean']]
rep1_mean    NaN
rep2_mean    NaN
Name: chr2:10000-20000_chr2:10000-30000, dtype: object
>>> # add chr2 info, with different data (reversed row order)
>>> add_columns_to_cluster_table(df, '%s_mean', row, col, data[::-1, :],
...                             labels=['rep1', 'rep2'], chrom='chr2')
>>> # now the chr2 clusters have data
>>> df.loc[df.index[2], ['rep1_mean', 'rep2_mean']]
rep1_mean    6
rep2_mean    7
Name: chr2:10000-20000_chr2:10000-30000, dtype: object
>>> # edge case: data is a vector
>>> df = clusters_to_table(clusters, 'chrX', res)
>>> add_columns_to_cluster_table(df, 'value', row, col, data[:, 0])
>>> df.iloc[0, :]
us_chrom      chrX
us_start      10000
us_end        20000
ds_chrom      chrX
ds_start      10000
ds_end        30000
cluster_size    2
cluster      [[1, 2], [1, 1]]
value        2
Name: chrX:10000-20000_chrX:10000-30000, dtype: object

```

hic3defdr.util.cluster_table.**clusters_to_table**(clusters, chrom, res)

Creates a DataFrame which tabulates cluster information.

The DataFrame’s first column (and index) will be a “loop_id” in the form “chr:start-end_chr:start-end”. Its other columns will be “us_chrom”, “us_start”, “us_end”, and “ds_chrom”, “ds_start”, “ds_end”, representing the BED-style chromosome, start coordinate, and end coordinate of the upstream (“us”, smaller coordinate values) and downstream (“ds”, larger coordinate values) anchors of the loop, respectively. These anchors together form a rectangular “bounding box” that completely encloses the significant pixels in the cluster. The DataFrame will also have a “cluster_size” column representing the total number of significant pixels in the cluster. Finally, the exact indices of the significant pixels in the cluster will be recorded in a “cluster” column in a JSON-like format (using only square brackets).

Parameters

- **clusters** (*list of list of tuple*) – The outer list is a list of clusters. Each cluster is a list of (i, j) tuples marking the position of significant points which belong to that cluster.
- **chrom** (*str*) – The name of the chromosome these clusters are on.
- **res** (*int*) – The resolution of the contact matrix referred to by the row and column indices

in clusters, in units of base pairs.

Returns The table of loop information.

Return type pd.DataFrame

Examples

```
>>> from hic3defdr.util.cluster_table import clusters_to_table
>>> clusters = [[(1, 2), (1, 1)], [(4, 4), (3, 4)]]
>>> df = clusters_to_table(clusters, 'chrX', 10000)
>>> df.iloc[0, :]
```

us_chrom	chrX
us_start	10000
us_end	20000
ds_chrom	chrX
ds_start	10000
ds_end	30000
cluster_size	2
cluster	[[1, 2], [1, 1]]

```
Name: chrX:10000-20000_chrX:10000-30000, dtype: object
```

hic3defdr.util.cluster_table.**load_cluster_table**(*table_filename*)

Loads a cluster table from a TSV file on disk to a DataFrame.

This function will ensure that the “cluster” column of the DataFrame is converted from a string representation to a list of list of int to simplify downstream processing.

See the example below for details on how this function assumes the cluster table was saved.

Parameters *table_filename* (*str*) – String reference to the location of the TSV file.

Returns The loaded cluster table.

Return type pd.DataFrame

Examples

```
>>> from tempfile import TemporaryFile
>>> from hic3defdr.util.cluster_table import clusters_to_table, \
...     load_cluster_table
>>> clusters = [[(1, 2), (1, 1)], [(4, 4), (3, 4)]]
>>> df = clusters_to_table(clusters, 'chrX', 10000)
>>> f = TemporaryFile(mode='w+') # simulates a file on disk
>>> df.to_csv(f, sep='\t')
>>> position = f.seek(0)
>>> loaded_df = load_cluster_table(f)
>>> df.equals(loaded_df)
True
>>> loaded_df['cluster'][0]
[[1, 2], [1, 1]]
```

hic3defdr.util.cluster_table.**sort_cluster_table**(*cluster_table*)

Sorts the rows of a cluster table in the expected order.

This function does not operate in-place.

We expect this to get a lot easier after this pandas issue is fixed: <https://github.com/pandas-dev/pandas/issues/3942>

Parameters `cluster_table` (*pd.DataFrame*) – The cluster table to sort. Must have all the expected columns.

Returns The sorted cluster table.

Return type *pd.DataFrame*

Examples

```
>>> from hic3defdr.util.cluster_table import clusters_to_table, \
...     sort_cluster_table
>>> clusters = [[(4, 4), (3, 4)], [(1, 2), (1, 1)]]
>>> res = 10000
>>> df1 = clusters_to_table(clusters, 'chr1', res)
>>> df2 = clusters_to_table(clusters, 'chr2', res)
>>> df3 = clusters_to_table(clusters, 'chr11', res)
>>> df4 = clusters_to_table(clusters, 'chrX', res)
>>> df = pd.concat([df4, df3, df2, df1], axis=0)
>>> sort_cluster_table(df).index
Index(['chr1:10000-20000_chr1:10000-30000',
      'chr1:30000-50000_chr1:40000-50000',
      'chr2:10000-20000_chr2:10000-30000',
      'chr2:30000-50000_chr2:40000-50000',
      'chr11:10000-20000_chr11:10000-30000',
      'chr11:30000-50000_chr11:40000-50000',
      'chrX:10000-20000_chrX:10000-30000',
      'chrX:30000-50000_chrX:40000-50000'],
      dtype='object', name='loop_id')
```

hic3defdr.util.clusters module

class `hic3defdr.util.clusters.DirectedDisjointSet`

Bases: *object*

Based on <https://stackoverflow.com/a/3067672> but supporting directed edges.

The overall effect is like a directed sparse graph - `DDS.add(a, b)` is like adding an edge from *a* to *b*. *a* gets marked as a source, *b* does not (anything not in the set `DDS.sources` is assumed to be a destination). If *b* is in an existing group, but isn't also the source of any other edge, then the groups won't be merged. Finally, the groups returned by `DDS.get_groups()` will be filtered to include only source nodes.

This is an “improved” or “streamlined” version where destination nodes are not stored anywhere if they haven't previously been seen as a source.

add (*a, b*)

get_groups ()

class `hic3defdr.util.clusters.NumpyEncoder` (*, *skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: *json.encoder.JSONEncoder*

Pass this to `json.dump()` to correctly serialize numpy values.

Credit: <https://stackoverflow.com/a/27050186>

default (*obj*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement `default` like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

`hic3defdr.util.clusters.cluster_from_string` (*cluster_string*)

If a cluster gets converted to a string (e.g., when the cluster is written to a text file), this function allows you to recover the cluster as a normal Python object (a list of pairs of integers).

Parameters `cluster_string` (*str*) – The string representation of the cluster.

Returns The inner lists are pairs of integers specifying the row and column indices of the pixels in the cluster.

Return type list of list of int

Examples

```
>>> from hic3defdr.util.clusters import cluster_from_string
>>> cluster = [(4, 5), (3, 4), (3, 5), (3, 6)]
>>> cluster_string = str(cluster)
>>> cluster_string
'[(4, 5), (3, 4), (3, 5), (3, 6)]'
>>> cluster_from_string(cluster_string)
[[4, 5], [3, 4], [3, 5], [3, 6]]
```

`hic3defdr.util.clusters.cluster_to_loop_id` (*cluster*, *chrom*, *resolution*)

Makes a cluster into a loop id of the form “chr:start-end_chr:start-end”.

This is a copy of `hiclite.util.clusters.make_loop_id_for_cluster()`.

Parameters

- **cluster** (*set of tuple of int*) – The tuples should be (*row_index*, *col_index*) tuples specifying which entries of the chromosomal contact matrix belong to this cluster.
- **chrom** (*str*) – The chromosome name, e.g. ‘chr21’.
- **resolution** (*int*) – The resolution of the contact matrix referred to by *cluster*.

Returns The loop id, a string of the form “chr:start-end_chr:start-end”.

Return type str

Examples

```
>>> from hic3defdr.util.clusters import cluster_to_loop_id
>>> cluster = [(4, 5), (3, 4), (3, 5), (3, 6)]
>>> cluster_to_loop_id(cluster, 'chrX', 10000)
'chrX:30000-50000_chrX:40000-70000'
```

hic3defdr.util.clusters.**cluster_to_slices**(cluster, width=21)

Computes a square row and column slice of a specified width centered on a given cluster.

Parameters

- **cluster** (*list of tuple*) – A list of (i, j) tuples marking the position of significant points which belong to the cluster.
- **width** (*int*) – Should be odd. Specifies the side length of the square slice.

Returns The row and column slice, respectively.

Return type slice, slice

Examples

```
>>> from hic3defdr.util.clusters import cluster_to_slices
>>> cluster = [(4, 5), (3, 4), (3, 5), (3, 6)]
>>> width = 5
>>> slices = cluster_to_slices(cluster, width=width)
>>> slices
(slice(1, 6, None), slice(3, 8, None))
>>> slices[0].stop - slices[0].start == width
True
>>> slices[1].stop - slices[1].start == width
True
```

hic3defdr.util.clusters.**clusters_to_coo**(clusters, shape)

Converts clusters (list of list of tuple) to a COO sparse matrix.

Parameters

- **clusters** (*list of list of tuple*) – The outer list is a list of clusters. Each cluster is a list of (i, j) tuples marking the position of significant points which belong to that cluster.
- **shape** (*tuple*) – The shape with which to construct the COO matrix.

Returns The sparse matrix of significant points.

Return type scipy.sparse.coo_matrix

Examples

```
>>> from hic3defdr.util.clusters import clusters_to_coo
>>> coo = clusters_to_coo([(1, 2), (1, 1)], [(4, 4), (3, 4)], (5, 5))
>>> coo.toarray()
array([[False, False, False, False, False],
       [False, True, True, False, False],
       [False, False, False, False, False],
       [False, False, False, False, True],
       [False, False, False, False, True]])
```


`hic3defdr.util.clusters.clusters_to_pixel_set(clusters)`

Converts a list of clusters to a set of pixels.

This function has no callers and is usually used as a one-liner.

Parameters `clusters` (*list of list of tuple*) – The outer list is a list of clusters. Each cluster is a list of (i, j) tuples marking the position of significant points which belong to that cluster.

Returns Each tuple is of the form (i, j) and marks the position of a significant point in the clustering.

Return type set of tuple

`hic3defdr.util.clusters.convert_cluster_array_to_sparse(cluster_array)`

Converts an array of cluster information to a sparse, JSON-friendly format.

Parameters `cluster_array` (*np.ndarray or scipy.sparse.spmatrix*) – Square, triangular, int dtype. Entries should be the cluster id for points which belong to that cluster, zero everywhere else.

Returns The sets are clusters, tuples are the matrix indices of the pixels in that cluster.

Return type list of sets of tuples of int

Notes

Since the introduction of `hiclite.util.clusters.find_clusters()`, this function is no longer used.

`hic3defdr.util.clusters.filter_clusters_by_distance(clusters, min_dist, max_dist)`

Filters a list of clusters by distance.

Parameters

- **clusters** (*list of list of tuple*) – The outer list is a list of clusters. Each cluster is a list of (i, j) tuples marking the position of significant points which belong to that cluster.
- **max_dist** (*min_dist,*) – Specify a range of distances in bin units to filter by (inclusive). If either `min_dist` or `max_dist` is `None`, the distance bin will be considered unbounded on that end.

Returns The clusters that are within the distance range requested.

Return type list of list of tuple

`hic3defdr.util.clusters.find_clusters(sig_points, connectivity=1)`

Finds clusters of adjacent True points in a boolean matrix.

Parameters

- **sig_points** (*scipy.sparse.spmatrix or np.ndarray*) – A boolean matrix indicating which points are significant.
- **connectivity** (*int*) – The connectivity to use when clustering.

Returns The clusters.

Return type list of set of tuple of int

`hic3defdr.util.clusters.hiccups_to_clusters(hiccups_txt, resolution)`

Loads HiCCUPS-format loop calls as clusters, approximating each loop as a cluster with just one pixel.

Parameters

- **hiccups_txt** (*str*) – The HiCCUPS-format loop call file to load.

- **resolution** (*int*) – The resolution to use for the clusters.

Returns strings. The values are lists of clusters on that chromosome. Each clusters is a list of [x, y] pairs representing the row and column indices of the pixels in that cluster.

Return type dict of list of clusters The keys of the dict are chromosome names as

`hic3defdr.util.clusters.load_clusters(infile)`

Loads clusters in a sparse format from a JSON file.

Parameters **infile** (*str*) – The JSON file containing sparse cluster information.

Returns The sets are clusters, the tuples are the indices of entries in that cluster.

Return type list of set of tuple of int

`hic3defdr.util.clusters.save_clusters(clusters, outfile)`

Saves cluster information to disk in sparse JSON format.

Parameters

- **clusters** (*np.ndarray or list of set of tuple of int*) – If an *np.ndarray* is passed, it should be square and triangular and have int dtype. Entries should be the cluster id for points which belong to that cluster, zero everywhere else. If a list of sets is passed, the sets are clusters, the tuples are the indices of entries in that cluster.
- **outfile** (*str*) – File to write JSON output to.

hic3defdr.util.demo_data module

`hic3defdr.util.demo_data.check_demo_data(dest_dir='~/hic3defdr-demo-data')`

Checks to see if all demo files are present.

Parameters **dest_dir** (*str*) – Path to destination directory to look for demo data files in.

Returns True if all demo files are present, False otherwise.

Return type bool

`hic3defdr.util.demo_data.ensure_demo_data(dest_dir='~/hic3defdr-demo-data')`

Checks that all the demo data files are present to dest_dir, and downloads them if any are missing.

Parameters **dest_dir** (*str*) – Path to destination directory to download files to.

`hic3defdr.util.demo_data.main()`

hic3defdr.util.dispersion module

`hic3defdr.util.dispersion.cml(data, f=None)`

Estimates the dispersion parameter of a NB distribution given the data via conditional maximum likelihood.

One common dispersion will be estimated across all pixels in the data, though the individual pixels in the data may have distinct means.

Parameters

- **data** (*np.ndarray*) – Rows should correspond to pixels, columns to replicates.
- **f** (*np.ndarray*) – Matrix of scaling factors parallel to data. Pass None to skip scaling.

Returns The estimated dispersion.

Return type float

`hic3defdr.util.dispersion.estimate_dispersion` (*data*, *cov*, *estimator*='qcml', *n_bins*=100, *logx*=False)

Estimates trended dispersion for each point in *data* with respect to a covariate *cov*, using *estimator* to estimate the dispersion within each of *n_bins* equal-number bins and fitting a curve through the per-bin dispersion estimates using lowess smoothing.

This function is deprecated and has no callers.

Parameters

- **data** (*np.ndarray*) – Rows should correspond to pixels, columns to replicates.
- **cov** (*np.ndarray*) – A vector of covariate values per pixel.
- **estimator** ('cml', 'qcml', 'mme', or a function) – Pass 'cml', 'qcml', 'mme' to use conditional maximum likelihood (CML), qnorm-CML (qCML), or method of moments estimation (MME) to estimate the dispersion within each bin. Pass a function that takes in a (pixels, replicates) shaped array of data and returns a dispersion value to use that instead.
- **n_bins** (*int*) – The number of bins to use when binning the pixels according to *cov*.
- **logx** (*bool*) – Whether or not to perform the lowess fit in log x space.

Returns

- **smoothed_disp** (*np.ndarray*) – Vector of smoothed dispersion estimates for each pixel.
- **cov_per_bin, disp_per_bin** (*np.ndarray*) – Average covariate value and estimated dispersion value, respectively, per bin.
- **disp_smooth_func** (*function*) – Vectorized function that returns a smoothed dispersion given the value of the covariate.

`hic3defdr.util.dispersion.mme` (*data*, *f*=None)

Estimates the dispersion parameter of a NB distribution given the data using a method of moments approach.

One common dispersion will be estimated across all pixels in the data, though the individual pixels in the data may have distinct means.

Parameters

- **data** (*np.ndarray*) – Rows should correspond to pixels, columns to replicates.
- **f** (*np.ndarray*) – Matrix of scaling factors parallel to *data*. Pass None to skip scaling.

Returns The estimated dispersion.

Return type float

`hic3defdr.util.dispersion.mme_per_pixel` (*data*, *f*=None)

Estimates the dispersion parameter of a separate NB distribution for each pixel given the data using a method of moments approach.

Parameters

- **data** (*np.ndarray*) – Rows should correspond to pixels, columns to replicates.
- **f** (*np.ndarray*) – Matrix of scaling factors parallel to *data*. Pass None to skip scaling.

Returns The estimated dispersion for each pixel.

Return type *np.ndarray*

`hic3defdr.util.dispersion.qcml` (*data*, *f*=None, *max_iter*=10, *tol*=0.0001)

Estimates the dispersion parameter of a NB distribution given the data via quantile-adjusted conditional maximum likelihood.

One common dispersion will be estimated across all pixels in the data, though the individual pixels in the data may have distinct means.

Parameters

- **data** (*np.ndarray*) – Rows should correspond to pixels, columns to replicates.
- **f** (*np.ndarray*) – Matrix of scaling factors parallel to **data**. Pass **None** to skip scaling.

Returns The estimated dispersion.

Return type float

hic3defdr.util.evaluation module

`hic3defdr.util.evaluation.compute_fdr(y_true, y_pred)`

Computes the observed false discovery rate from boolean vectors of true and predicted labels.

Parameters **y_pred** (*y_true*,) – Boolean vectors of the true and predicted labels, respectively.

Returns The false discovery rate.

Return type float

`hic3defdr.util.evaluation.evaluate(y_true, qvalues, n_fdr_points=100)`

Evaluates how good a vector of q-values (or p-values) is at predicting the vector of true labels.

Parameters

- **y_true** (*np.ndarray*) – The boolean vector of true labels.
- **qvalues** (*np.ndarray*) – Vector of q-values or p-values which are supposed to predict the boolean label in **y_true**.
- **n_fdr_points** (*int*) – The maximum number of points at which to compute FDR. The FDR computation is not parallelized so increasing this number will slow down the evaluation. The default value of 100 should be sufficient to visualize the FDR control curve.

Returns **fdr, fpr, tpr, thresh** – Parallel arrays of the FDR, FPR, TPR, and thresholds (in 1 – qvalue) space which specify the FDR, FPR, and and TPR at each threshold. The thresholds are selected to represent the convex edge of the ROC curve. The FDR will only be evaluated at about 100 selected thresholds and will be set to `np.nan` at the un-evaluated thresholds.

Return type `np.ndarray`

`hic3defdr.util.evaluation.make_y_true(row, col, clusters, labels)`

Makes a boolean vector of the true labels for each pixel, given a list of clusters and the true label for each cluster.

Parameters

- **col** (*row*,) – The row and column indices of pixels to be labeled.
- **clusters** (*list of list of tuple*) – The outer list is a list of clusters. Each cluster is a list of (i, j) tuples marking the position of significant points which belong to that cluster.
- **labels** (*list of str*) – List of labels for each cluster, parallel to **clusters**.

Returns Boolean vector with the same length as **row/col**. It's *i*'th element is *False* when the pixel at `(row[i], col[i])` is in a cluster with label 'constit' and is *True* otherwise.

Return type `np.ndarray`

hic3defdr.util.filtering module

hic3defdr.util.filtering.**filter_sparse_rows_count** (*matrix*, *min_nnz*=25, *k*=300)

Wipes the sparse bins (both rows and columns) of a matrix, where “sparse” means that the bin has less than *min_nnz* nonzero interactions with both the *k* nearest upstream and downstream bins.

This function wipes with zeros and will eliminate the wiped positions if *matrix* is a CSR format sparse matrix.

Parameters

- **matrix** (*np.ndarray*, *scipy.sparse.csr_matrix*, or *BandedMatrix*) – The matrix to wipe sparse bins from.
- **min_nnz** (*int*) – The minimum number of nonzero contacts a bin has to make in either direction (upstream or downstream) to escape being wiped.
- **k** (*int*) – How many bins upstream or downstream to look when counting the number of nonzero contacts a bin makes in a given direction.

Returns The filtered matrix.

Return type *np.ndarray*, *scipy.sparse.csr_matrix*, or *BandedMatrix*

hic3defdr.util.lowess module

hic3defdr.util.lowess.**lowess_fit** (*x*, *y*, *logx*=False, *logy*=False, *left_boundary*=None, *right_boundary*=None, *frac*=0.3, *delta*=0.01)

Opinionated convenience wrapper for lowess smoothing.

Parameters

- **y** (*x*,) – The *x* and *y* values to fit, respectively.
- **logy** (*logx*,) – Pass True to perform the fit on the scale of $\log(x)$ and/or $\log(y)$, respectively.
- **right_boundary** (*left_boundary*,) – Allows specifying boundaries for the fit, in the original *x* space. If a float is passed, the returned fit will return the farthest left or farthest right lowess-estimated *y_hat* (from the original fitting set) for all points which are left or right of the specified left or right boundary point, respectively. Pass None to use linear extrapolation for these points instead.
- **frac** (*float*) – The lowess smoothing fraction to use.
- **delta** (*float*) – Distance (on the scale of *x* or $\log(x)$) within which to use linear interpolation when constructing the initial fit, expressed as a fraction of the range of *x* or $\log(x)$.

Returns This function takes in *x* values on the original *x* scale and returns estimated *y* values on the original *y* scale (regardless of what is passed for *logx* and *logy*). This function will still return sane estimates for *y* even at points not in the original fitting set by performing linear interpolation in the space the fit was performed in.

Return type function

Notes

No filtering of input values is performed; clients are expected to handle this if desired. NaN values should not break the function, but *x* points with zero values passed when *logx* is True are expected to break the function.

The default value of the `delta` parameter is set to be non-zero, matching the behavior of lowess smoothing in R and improving performance.

Linear interpolation between x-values in the original fitting set is used to provide a familiar functional interface to the fitted function.

Boundary conditions on the fitted function are exposed via `left_boundary` and `right_boundary`, mostly as a convenience for points where $x == 0$ when fitting was performed on the scale of $\log(x)$.

When `left_boundary` or `right_boundary` are `None` (this is the default) the fitted function will be linearly extrapolated for points beyond the lowest and highest x-values in x .

```
hic3defdr.util.lowess.weighted_lowess_fit(x, y, logx=False, logy=False,
                                          left_boundary=None, right_boundary=None,
                                          frac=None, auto_frac_factor=15.0,
                                          delta=0.01, w=20, power=0.25, interpolate_before_increase=True)
```

Performs lowess fitting as in `lowess_fit()`, but weighting the data points automatically according to the precision in the y values as estimated by a rolling window sample variance.

Points are weighted proportionally to a specified power `power` of their precision by adding duplicated points to the dataset. This should approximate the effects of a true weighted lowess fit, with the caveat that the weights are rounded a bit.

Weighting the data points according to this rolling window sample variance is probably only a good idea if the marginal distribution of x values is uniform.

Parameters

- **`y`** (x ,) – The x and y values to fit, respectively.
- **`logy`** ($\log x$,) – Pass `True` to perform the fit on the scale of $\log(x)$ and/or $\log(y)$, respectively.
- **`right_boundary`** (`left_boundary`,) – Allows specifying boundaries for the fit, in the original x space. If a float is passed, the returned fit will return the farthest left or farthest right lowess-estimated \hat{y} (from the original fitting set) for all points which are left or right of the specified left or right boundary point, respectively. Pass `None` to use linear extrapolation for these points instead.
- **`frac`** (`float`, *optional*) – The lowess smoothing fraction to use. Pass `None` to use the default: `auto_frac_factor` divided by the product of the average of the unscaled weights and the largest scaled weight.
- **`auto_frac_factor`** (`float`) – When `frac` is `None`, this factor scales the automatically determined fraction parameter.
- **`delta`** (`float`) – Distance (on the scale of x or $\log(x)$) within which to use linear interpolation when constructing the initial fit, expressed as a fraction of the range of x or $\log(x)$.
- **`w`** (`int`) – The size of the rolling window to use when estimating the precision of the y values.
- **`power`** (`float`) – Precisions will be taken to this power to obtain unscaled weights.
- **`interpolate_before_increase`** (`bool`) – Hacky flag introduced to handle quirk of Hi-C dispersion vs distance relationships in which dispersion is elevated at extremely short distances. When `True`, this function will identify a group of points with the lowest x -values across which the y -value is monotonically decreasing. These points will be included in the variance estimation, but will be excluded from lowess fitting. Linear interpolation will be used at these x -values instead, since it is hard to convince lowess to follow a sharp change

in the trend that is only supported by 3-4 data points out of 200-500 total data points, even with our best attempts at weighting. Pass `False` to perform a simple weighted lowess fit with no linear interpolation.

Returns This function takes in x values on the original x scale and returns estimated y values on the original y scale (regardless of what is passed for `logx` and `logy`). This function will still return sane estimates for y even at points not in the original fitting set by performing linear interpolation in the space the fit was performed in.

Return type function

hic3defdr.util.lrt module

`hic3defdr.util.lrt.lrt` (*raw, f, disp, design, refit_mu=True*)

Performs a likelihood ratio test on raw data *raw* given scaling factors *f* and dispersion *disp*.

Parameters

- **f**, **disp** (*raw*,) – Matrices of raw values, combined scaling factors, and dispersions, respectively. Rows correspond to pixels, columns correspond to replicates.
- **design** (*np.ndarray*) – Describes the grouping of replicates into conditions. Rows correspond to replicates, columns correspond to conditions, and values should be `True` where a replicate belongs to a condition and `False` otherwise.

Returns

- **pvalues** (*np.ndarray*) – The LRT p-values per pixel.
- **llr** (*np.ndarray*) – The log likelihood ratio per pixel.
- **mu_hat_null**, **mu_hat_alt** (*np.ndarray*) – The fitted mean parameters under the null and alt models, respectively, per pixel.

hic3defdr.util.matrices module

`hic3defdr.util.matrices.deconvolute` (*matrix, bias, invert=False*)

Applies bias factors to a sparse matrix.

Parameters

- **matrix** (*scipy.sparse.spmatrix*) – The matrix to bias. Will be converted to CSR by this function.
- **bias** (*np.ndarray*) – The dense bias vector.
- **invert** (*bool*) – Whether or not to invert the bias before applying it. By default this function multiplies the matrix by the bias. Infinite bias (1 / 0) will be converted to zero bias to allow rows with infinite bias to be dropped from the resulting sparse matrix.

Returns The deconvoluted matrix.

Return type `scipy.sparse.csr_matrix`

`hic3defdr.util.matrices.dilate` (*matrix, doublings*)

Doubles the “resolution” of a matrix using nearest-neighbor interpolation.

Parameters

- **matrix** (*np.ndarray*) – Input matrix.

- **doublings** (*int*) – The number of times the resolution will be doubled.

Returns The dilated matrix.

Return type np.ndarray

hic3defdr.util.matrices.**select_matrix**(*row_slice*, *col_slice*, *row*, *col*, *data*, *symmetrize=True*)

Slices out a dense matrix from COO-formatted sparse data, filling empty values with np.nan.

Parameters

- **col_slice** (*row_slice*,) – Row and column slice to use, respectively.
- **col**, **data** (*row*,) – COO-format sparse matrix to be sliced.
- **symmetrize** (*bool*) – Pass True to fill in lower-triangle points of the matrix.

Returns Dense matrix.

Return type np.ndarray

hic3defdr.util.matrices.**sparse_intersection**(*fnames*, *bias=None*)

Computes the intersection set of (row, col) pairs across multiple sparse matrices.

Parameters

- **fnames** (*list of str*) – File paths to sparse matrices loadable by `scipy.sparse.load_npz()`. Will be converted to COO by this function.
- **bias** (*np.ndarray, optional*) – Pass the bias matrix to drop rows and columns with bias factors of zero in any replicate.

Returns **row**, **col** – The intersection set of (row, col) pairs.

Return type np.ndarray

hic3defdr.util.matrices.**sparse_union**(*fnames*, *dist_thresh=1000*, *bias=None*, *size_factors=None*, *mean_thresh=0.0*)

Computes the intersection set of (row, col) pairs across multiple sparse matrices.

Parameters

- **fnames** (*list of str*) – File paths to sparse matrices loadable by `scipy.sparse.load_npz()`. Will be converted to COO by this function.
- **dist_thresh** (*int*) – The maximum distance allowed, respectively, in bin units.
- **bias** (*np.ndarray, optional*) – Rectangular matrix containing bias factors for each bin (rows) and each replicate (columns).
- **size_factors** (*np.ndarray, optional*) – Size factors for each replicate.
- **mean_thresh** (*float*) – Minimum mean value (in normalized space) to keep pixels for.

Returns **row**, **col** – The union set of (row, col) pairs.

Return type np.ndarray

hic3defdr.util.matrices.**wipe_distances**(*matrix*, *min_dist*, *max_dist*)

Eliminates entries from a sparse matrix outside of a specified distance range.

Parameters

- **matrix** (*scipy.sparse.spmatrix*) – The matrix to wipe. Will be converted to COO by this function.

- **max_dist** (*min_dist*,) – The minimum and maximum distance allowed, respectively, in bin units.

Returns The wiped matrix.

Return type `scipy.sparse.coo_matrix`

hic3defdr.util.parallelization module

`hic3defdr.util.parallelization.parallel_apply` (*fn*, *kwargs_list*, *n_threads=None*)

Applies a function in parallel over a list of kwarg dicts, using a specified number of threads.

Parameters

- **fn** (*function*) – The function to parallelize.
- **kwargs_list** (*list of dict*) – The function will be called `len(kwargs_list)` times. Each time it is called it will use a different element of this list to determine the kwargs the function will be called with.
- **n_threads** (*int*) – Specify the number of subprocesses to use for parallelization. Pass -1 to use as many subprocesses as there are CPUs.

`hic3defdr.util.parallelization.parallel_map` (*fn*, *kwargs_list*, *n_threads=None*)

Maps a function in parallel over a list of kwarg dicts, using a specified number of threads.

Parameters

- **fn** (*function*) – The function to parallelize.
- **kwargs_list** (*list of dict*) – The function will be called `len(kwargs_list)` times. Each time it is called it will use a different element of this list to determine the kwargs the function will be called with.
- **n_threads** (*int*) – Specify the number of subprocesses to use for parallelization. Pass -1 to use as many subprocesses as there are CPUs.

hic3defdr.util.printing module

`hic3defdr.util.printing.eprint` (**args*, ***kwargs*)

Drop-in replacement for `print()` that prints to `stderr` instead of `stdout`.

Parameters **kwargs** (*args*,) – All args and kwargs will be passed through to `print()` except for the special kwarg `skip`; if this kwarg is present and set to `True`, nothing will be printed.

hic3defdr.util.progress module

`hic3defdr.util.progress.context` ()

Infers the context in which the current program is being executed.

<https://stackoverflow.com/a/47428575>

Returns The context, either 'colab', 'jupyter', 'ipython', or 'terminal'.

Return type `str`

`hic3defdr.util.progress.tqdm_maybe` (*iter*, ***kwargs*)

Drop-in replacement from `tqdm.tqdm()` except will simply do nothing if `tqdm` is not present and will use `tqdm.tqdm_notebook()` if run in a notebook.

Parameters

- **iter** (*iterable*) – The iterable to wrap.
- **kwargs** (*kwargs*) – Will be passed through to `tqdm.tqdm()`.

Returns Wrapped by `tqdm` if it was installed, or just `iter` otherwise.

Return type iterator

hic3defdr.util.scaled_nb module

`hic3defdr.util.scaled_nb.equalize(data, f, alpha)`

Given known scaling factors `f` and a known dispersion `alpha`, creates common-scale pseudodata from raw values `data`.

See <https://rdrr.io/bioc/edgeR/src/R/equalizeLibSizes.R>

Parameters

- **data** (*np.ndarray*) – Matrix of raw data to equalize. Rows are pixels, columns are replicates.
- **f** (*np.ndarray*) – Matrix of combined scaling factors for each pixel.
- **alpha** (*float*) – Single fixed dispersion to use during equalization.

Returns Matrix of equalized data.

Return type `np.ndarray`

`hic3defdr.util.scaled_nb.fit_mu_hat(x, b, alpha, verbose=True)`

Numerical MLE fitter for the mean parameter of the scaled NB model under fixed dispersion. Vectorized.

See the following colab notebook for background and derivation: <https://colab.research.google.com/drive/1SgMMvc3XhflXoBx8tsyJt-yyBDIRFQCJ>

Parameters

- **x** (*np.ndarray*) – The vector of observed counts.
- **b** (*np.ndarray*) – The vector of scaling factors, parallel to `x`.
- **alpha** (*np.ndarray*) – The vector of dispersions, parallel to `x`.
- **verbose** (*bool*) – Pass `False` to silence reporting of progress to `stderr`.

Returns The MLE of the mean parameter.

Return type `float`

Examples

```
>>> import numpy as np
>>> from hic3defdr.util.scaled_nb import fit_mu_hat
```

3 pixels, 2 reps (matrices): >>> `x = np.array([[1, 2], ... [3, 4], ... [5, 6]])` >>> `b = np.array([[0.9, 1.1], ... [0.8, 1.2], ... [0.7, 1.3]])` >>> `alpha = np.array([[0.1, 0.2], ... [0.3, 0.4], ... [0.5, 0.6]])` >>> `fit_mu_hat(x, b, alpha)`
`array([1.47251127, 3.53879843, 5.86853465])`

broadcast dispersion down the pixels: >>> `fit_mu_hat(x, b, np.array([0.1, 0.2]))` `array([1.47251127, 3.53749833, 5.85554075])`

broadcast dispersion across the reps: >>> fit_mu_hat(x, b, np.array([0.1, 0.2, 0.3])[:, None]) array([1.49544092, 3.51679438, 5.73129492])

1 pixel, two reps (vectors): >>> fit_mu_hat(np.array([1, 2]), np.array([0.9, 1.1]), np.array([0.1, 0.2])) array([1.47251127])

broadcast dispersion across reps: >>> fit_mu_hat(np.array([1, 2]), np.array([0.9, 1.1]), 0.1) array([1.49544092])

one pixel is fitted with newton, the second is fitted with brentq >>> x = np.array([2, 3, 4, 2], ... [6, 9, 3, 1]) >>> b = np.array([[0.45, 0.53, 0.088, 0.091], ... [0.70, 0.83, 0.14, 0.15]]) >>> alpha = np.array([[0.0071, 0.0071, 0.0073, 0.0073], ... [0.0070, 0.0070, 0.0072, 0.0072]]) >>> fit_mu_hat(x, b, alpha) array([9.5900971, 10.45962955])

hic3defdr.util.scaled_nb.inverse_mvr(mean, var)

Inverse function of the negative binomial fixed-dispersion mean-variance relationship. Vectorized.

Parameters **var** (mean,) – The mean and variance of a NB distribution, respectively.

Returns The dispersion of that NB distribution.

Return type float

hic3defdr.util.scaled_nb.logpmf(k, m, phi)

Log of the PMF of the negative binomial distribution, parameterized by its mean m and dispersion phi. Vectorized.

Parameters

- **k** (int) – The number of counts observed.
- **m** (float) – The mean parameter.
- **phi** (float) – The dispersion parameter.

Returns The log of the probability of observing k counts.

Return type float

hic3defdr.util.scaled_nb.mvr(mean, disp)

Negative binomial fixed-dispersion mean-variance relationship. Vectorized.

Parameters **disp** (mean,) – The mean and dispersion of a NB distribution, respectively.

Returns The variance of that NB distribution.

Return type float

hic3defdr.util.scaled_nb.q2qnbinom(x, mu_in, mu_out, alpha)

Converts values between two NB distributions with different means but the same dispersion.

See <https://rdrr.io/bioc/edgeR/src/R/q2qnbinom.R>

Parameters

- **x** (np.ndarray) – Vector of values to convert.
- **mu_out** (mu_in,) – Vectors of means to convert between.
- **alpha** (np.ndarray or float) – Single dispersion (to use for all x) or vector of dispersions (one per x) to hold constant during conversion.

Returns x converted from mu_in to mu_out.

Return type np.ndarray

hic3defdr.util.scaling module

hic3defdr.util.scaling.**conditional** (*data, dist, fn, n_bins=None*)

Applies a size factor computing function *fn* to *data* conditioning on *dist*, optionally binning *dist* into *n_bins* equal-number bins.

If *n_bins* is not *None*, the final size factors will be interpolated for interaction distances falling between the bins to avoid bin edge effects.

Parameters

- **data** (*np.ndarray*) – Rows correspond to pixels, columns correspond to replicates.
- **dist** (*np.ndarray*) – The distance of each pixel in *data*.
- **fn** (*function*) – A function that computes a size factor given some data.
- **n_bins** (*int, optional*) – Pass an int to bin distance into this many equal-number bins.

hic3defdr.util.scaling.**conditional_mor** (*data, dist, n_bins=None*)

Computes size factors for a dataset using median of ratios normalization, conditioning on distance.

Parameters

- **data** (*np.ndarray*) – Rows correspond to pixels, columns correspond to replicates.
- **dist** (*np.ndarray*) – The distance of each pixel in *data*
- **n_bins** (*int, optional*) – Pass an int to bin distance into this many equal-number bins.

Returns Matrix of size factors, per pixel (rows) and per replicate (columns).

Return type *np.ndarray*

hic3defdr.util.scaling.**conditional_scaling** (*data, dist, n_bins=None*)

Computes size factors for a dataset using simple scaling normalization, conditioning on distance.

Parameters

- **data** (*np.ndarray*) – Rows correspond to pixels, columns correspond to replicates.
- **dist** (*np.ndarray*) – The distance of each pixel in *data*.
- **n_bins** (*int, optional*) – Pass an int to bin distance into this many equal-number bins.

Returns Matrix of size factors, per pixel (rows) and per replicate (columns).

Return type *np.ndarray*

hic3defdr.util.scaling.**median_of_ratios** (*data, filter_zeros=True*)

Computes size factors for a dataset using the median of ratios method.

Parameters

- **data** (*np.ndarray*) – Rows correspond to pixels, columns correspond to replicates.
- **filter_zeros** (*bool*) – Pass True to filter out pixels with a zero value in any replicate. Pass False to include all data.

Returns Vector of size factors, per replicate.

Return type *np.ndarray*

`hic3defdr.util.scaling.no_scaling(data)`

Computes dummy size factors of 1 for each replicate.

Parameters `data` (`np.ndarray`) – Rows correspond to pixels, columns correspond to replicates.

Returns Vector of size factors, per replicate.

Return type `np.ndarray`

`hic3defdr.util.scaling.simple_scaling(data)`

Computes size factors for a dataset using a simple scaling method.

Parameters `data` (`np.ndarray`) – Rows correspond to pixels, columns correspond to replicates.

Returns Vector of size factors, per replicate.

Return type `np.ndarray`

hic3defdr.util.simulation module

`hic3defdr.util.simulation.perturb_cluster(matrix, cluster, effect, respect_zeros=True)`

Perturbs a specific cluster in a contact matrix with a given effect.

Operates in-place.

Based on a notebook linked here: https://colab.research.google.com/drive/1dk9kX57ZtlxQ3jubrKL_q2r8LZnSIVwY

Parameters

- **matrix** (`scipy.sparse.spmatrix`) – The contact matrix. Must support slicing.
- **cluster** (*list of tuple of int*) – A list of (i, j) tuples marking the position of points which belong to the cluster which we want to perturb.
- **effect** (`float`) – The effect to apply to the cluster. Values in `matrix` under the cluster footprint will be shifted by this proportion of their original value.
- **respect_zeros** (`bool`) – Pass True to preserve the sparsity structure of `matrix` if it is sparse. Has no effect if `matrix` is dense.

`hic3defdr.util.simulation.simulate(row, col, mean, disp_fn, bias, size_factors, clusters, beta=0.5, p_diff=0.4, trend='mean', verbose=True)`

Simulates raw contact matrices based on `mean` and `disp_fn` using `bias` and `size_factors` per simulated replicate and perturbing the loops specified in `clusters` with an effect size of `beta` and direction chosen at random for `p_diff` fraction of clusters.

Parameters

- **col** (`row,`) – Row and column indices identifying the location of pixels in `mean`.
- **mean** (`np.ndarray`) – Vector of mean values for each pixel to use as a base to simulate from.
- **disp_fn** (*function*) – Function that returns a dispersion given a mean or distance (as specified by `trend`). Will be used to determine the dispersion values to use during simulation.
- **bias** (`np.ndarray`) – Rows are bins of the full contact matrix, columns are to-be-simulated replicates. Each column represents the bias vector to use for simulating that replicate.

- **size_factors** (*np.ndarray*) – Vector of size factors to use for simulating for each to-be-simulated replicate. To use a different size factor at different distance scales, pass a matrix whose rows correspond to distance scales and whose columns correspond to replicates.
- **clusters** (*list of list of tuple*) – The outer list is a list of clusters which represent the locations of loops. Each cluster is a list of (i, j) tuples marking the position of pixels which belong to that cluster.
- **beta** (*float*) – The effect size of the loop perturbations to use when simulating. Perturbed loops will be strengthened or weakened by this fraction of their original strength.
- **p_diff** (*float or list of float*) – Pass a single float to specify the probability that a loop will be perturbed across the simulated conditions. Pass four floats to specify the probabilities of all four specific perturbations: up in A, down in A, up in B, down in B. The remaining loops will be constitutive.
- **trend** (*'mean' or 'dist'*) – Whether `disp_fn` returns the smoothed dispersion as a function of mean or of interaction distance.
- **verbose** (*bool*) – Pass False to silence reporting of progress to stderr.

Returns

- **classes** (*np.ndarray*) – Vector of ground-truth class labels used for simulation with ‘U7’ dtype.
- **gen** (generator of *scipy.sparse.csr_matrix*) – Generates the simulated raw contact matrices for each simulated replicate, in order.

hic3defdr.util.thresholding module

`hic3defdr.util.thresholding.size_filter` (*clusters, cluster_size*)

Filters out clusters which are smaller than `cluster_size`.

Parameters

- **clusters** (*list of set of tuple of int*) – The clusters to filter.
- **cluster_size** (*int*) – The minimum size of a cluster needed to pass this filter.

Returns The filtered clusters.

Return type list of set of tuple of int

`hic3defdr.util.thresholding.threshold_and_cluster` (*qvalues, row, col, fdr*)

Thresholds pixels by comparing their q-values to a target FDR and clusters the significant and insignificant pixels.

Parameters

- **qvalues** (*np.ndarray*) – The qvalue for each pixel under consideration.
- **col** (*row,*) – The row and column indices corresponding to the qvalues.
- **fdr** (*float*) – The FDR to threshold on.

Returns **sig_clusters**, **insig_clusters** – Lists of the significant and insignificant clusters, respectively.

Return type list of set of tuple of int

Module contents

5.1.2 Module contents

CHAPTER 6

Indices and tables

- `genindex`
- `modindex`
- `search`

h

- `hic3defdr`, 91
- `hic3defdr.analysis`, 58
 - `hic3defdr.analysis.alternatives`, 45
 - `hic3defdr.analysis.analysis`, 47
 - `hic3defdr.analysis.constructor`, 50
 - `hic3defdr.analysis.core`, 52
 - `hic3defdr.analysis.plotting`, 53
 - `hic3defdr.analysis.simulation`, 57
 - `hic3defdr.plotting`, 65
 - `hic3defdr.plotting.dispersion`, 58
 - `hic3defdr.plotting.distance_bias`, 60
 - `hic3defdr.plotting.distance_dependence`, 61
 - `hic3defdr.plotting.fdr`, 61
 - `hic3defdr.plotting.fn_vs_fp`, 62
 - `hic3defdr.plotting.grid`, 62
 - `hic3defdr.plotting.heatmap`, 63
 - `hic3defdr.plotting.histograms`, 64
 - `hic3defdr.plotting.ma`, 64
 - `hic3defdr.plotting.roc`, 65
- `hic3defdr.util`, 91
 - `hic3defdr.util.apa`, 65
 - `hic3defdr.util.balancing`, 66
 - `hic3defdr.util.banded_matrix`, 67
 - `hic3defdr.util.binning`, 70
 - `hic3defdr.util.classification`, 70
 - `hic3defdr.util.cluster_table`, 70
 - `hic3defdr.util.clusters`, 74
 - `hic3defdr.util.demo_data`, 78
 - `hic3defdr.util.dispersion`, 78
 - `hic3defdr.util.evaluation`, 80
 - `hic3defdr.util.filtering`, 81
 - `hic3defdr.util.lowess`, 81
 - `hic3defdr.util.lrt`, 83
 - `hic3defdr.util.matrices`, 83
 - `hic3defdr.util.parallelization`, 85
 - `hic3defdr.util.printing`, 85
 - `hic3defdr.util.progress`, 85
 - `hic3defdr.util.scaled_nb`, 86
 - `hic3defdr.util.scaling`, 88
 - `hic3defdr.util.simulation`, 89
 - `hic3defdr.util.thresholding`, 90

A

`add()` (*hic3defdr.util.clusters.DirectedDisjointSet* method), 74
`add_columns_to_cluster_table()` (in module *hic3defdr.util.cluster_table*), 70
`align()` (*hic3defdr.util.banded_matrix.BandedMatrix* class method), 67
`AnalyzingHiC3DeFDR` (class in *hic3defdr.analysis.analysis*), 47
`apply()` (*hic3defdr.util.banded_matrix.BandedMatrix* class method), 67

B

`BandedMatrix` (class in *hic3defdr.util.banded_matrix*), 67
`bh()` (*hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR* method), 47
`bias_patterns` (*hic3defdr.analysis.constructor.HiC3DeFDR* attribute), 51
`bias_thresh` (*hic3defdr.analysis.constructor.HiC3DeFDR* attribute), 51

C

`check_demo_data()` (in module *hic3defdr.util.demo_data*), 78
`chroms` (*hic3defdr.analysis.constructor.HiC3DeFDR* attribute), 51
`classify()` (*hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR* method), 47
`classify()` (in module *hic3defdr.util.classification*), 70
`cluster_from_string()` (in module *hic3defdr.util.clusters*), 75
`cluster_to_loop_id()` (in module *hic3defdr.util.clusters*), 75
`cluster_to_slices()` (in module *hic3defdr.util.clusters*), 76
`clusters_to_coo()` (in module *hic3defdr.util.clusters*), 76
`clusters_to_pixel_set()` (in module *hic3defdr.util.clusters*), 76
`clusters_to_table()` (in module *hic3defdr.util.cluster_table*), 72
`cml()` (in module *hic3defdr.util.dispersion*), 78
`collect()` (*hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR* method), 48
`compare_disp_fits()` (in module *hic3defdr.plotting.dispersion*), 58
`compute_fdr()` (in module *hic3defdr.util.evaluation*), 80
`conditional()` (in module *hic3defdr.util.scaling*), 88
`conditional_mor()` (in module *hic3defdr.util.scaling*), 88
`conditional_scaling()` (in module *hic3defdr.util.scaling*), 88
`context()` (in module *hic3defdr.util.progress*), 85
`convert_cluster_array_to_sparse()` (in module *hic3defdr.util.clusters*), 77
`copy()` (*hic3defdr.util.banded_matrix.BandedMatrix* method), 67
`CoreHiC3DeFDR` (class in *hic3defdr.analysis.core*), 52

D

`data_indices()` (*hic3defdr.util.banded_matrix.BandedMatrix* method), 67
`deconvolute()` (*hic3defdr.util.banded_matrix.BandedMatrix* method), 67
`deconvolute()` (in module *hic3defdr.util.matrices*), 83
`default()` (*hic3defdr.util.clusters.NumpyEncoder* method), 74
`design` (*hic3defdr.analysis.constructor.HiC3DeFDR* attribute), 51
`dilate()` (in module *hic3defdr.util.matrices*), 83
`DirectedDisjointSet` (class in *hic3defdr.util.clusters*), 74

E

`ensure_demo_data()` (in module

`hic3defdr.util.demo_data`), 78
`eprint()` (in module `hic3defdr.util.printing`), 85
`equal_bin()` (in module `hic3defdr.util.binning`), 70
`equalize()` (in module `hic3defdr.util.scaled_nb`), 86
`estimate_disp()` (`hic3defdr.analysis.alternatives.Global3DeFDR`
 method), 45
`estimate_disp()` (`hic3defdr.analysis.alternatives.Poisson3DeFDR`
 method), 46
`estimate_disp()` (`hic3defdr.analysis.alternatives.Unsmoothed3DeFDR`
 method), 47
`estimate_disp()` (`hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR`
 method), 48
`estimate_dispersion()` (in module `hic3defdr.util.dispersion`), 78
`evaluate()` (`hic3defdr.analysis.simulation.SimulatingHiC3DeFDR`
 method), 57
`evaluate()` (in module `hic3defdr.util.evaluation`), 80

F

`filter_clusters_by_distance()` (in module `hic3defdr.util.clusters`), 77
`filter_sparse_rows_count()` (in module `hic3defdr.util.filtering`), 81
`find_clusters()` (in module `hic3defdr.util.clusters`), 77
`fit_mu_hat()` (in module `hic3defdr.util.scaled_nb`), 86
`flatten()` (`hic3defdr.util.banded_matrix.BandedMatrix`
 method), 68
`from_dia_matrix()`
 (`hic3defdr.util.banded_matrix.BandedMatrix`
 class method), 68
`from_ndarray()` (`hic3defdr.util.banded_matrix.BandedMatrix`
 class method), 68
`from_sparse()` (`hic3defdr.util.banded_matrix.BandedMatrix`
 class method), 68

G

`get_groups()` (`hic3defdr.util.clustersDirectedDisjointSet`
 method), 74
`get_matrix()` (`hic3defdr.analysis.core.CoreHiC3DeFDR`
 method), 52
`Global3DeFDR` (class in `hic3defdr.analysis.alternatives`), 45

H

`HiC3DeFDR` (class in `hic3defdr.analysis.constructor`), 50
`hic3defdr` (module), 91
`hic3defdr.analysis` (module), 58
`hic3defdr.analysis.alternatives` (module), 45
`hic3defdr.analysis.analysis` (module), 47

`hic3defdr.analysis.constructor` (module), 50
`hic3defdr.analysis.core` (module), 52
`hic3defdr.analysis.plotting` (module), 53
`hic3defdr.analysis.simulation` (module), 57
`hic3defdr.plotting` (module), 65
`hic3defdr.plotting.dispersion` (module), 58
`hic3defdr.plotting.distance_bias` (mod-
 ule), 60
`hic3defdr.plotting.distance_dependence`
 (module), 61
`hic3defdr.plotting.fdr` (module), 61
`hic3defdr.plotting.fn_vs_fp` (module), 62
`hic3defdr.plotting.grid` (module), 62
`hic3defdr.plotting.heatmap` (module), 63
`hic3defdr.plotting.histograms` (module), 64
`hic3defdr.plotting.ma` (module), 64
`hic3defdr.plotting.roc` (module), 65
`hic3defdr.util` (module), 91
`hic3defdr.util.apa` (module), 65
`hic3defdr.util.balancing` (module), 66
`hic3defdr.util.banded_matrix` (module), 67
`hic3defdr.util.binning` (module), 70
`hic3defdr.util.classification` (module), 70
`hic3defdr.util.cluster_table` (module), 70
`hic3defdr.util.clusters` (module), 74
`hic3defdr.util.demo_data` (module), 78
`hic3defdr.util.dispersion` (module), 78
`hic3defdr.util.evaluation` (module), 80
`hic3defdr.util.filtering` (module), 81
`hic3defdr.util.lowess` (module), 81
`hic3defdr.util.lrt` (module), 83
`hic3defdr.util.matrices` (module), 83
`hic3defdr.util.parallelization` (module), 85
`hic3defdr.util.printing` (module), 85
`hic3defdr.util.progress` (module), 85
`hic3defdr.util.scaled_nb` (module), 86
`hic3defdr.util.scaling` (module), 88
`hic3defdr.util.simulation` (module), 89
`hic3defdr.util.thresholding` (module), 90
`hiccups_to_clusters()` (in module `hic3defdr.util.clusters`), 77

I

`inverse_mvr()` (in module `hic3defdr.util.scaled_nb`), 87
`is_bandedmatrix()`
 (`hic3defdr.util.banded_matrix.BandedMatrix`
 class method), 68
`is_upper()` (`hic3defdr.util.banded_matrix.BandedMatrix`
 method), 68

K

`kr_balance()` (in module `hic3defdr.util.balancing`), 66

L

`load()` (`hic3defdr.analysis.core.CoreHiC3DeFDR` class method), 52

`load()` (`hic3defdr.util.banded_matrix.BandedMatrix` class method), 68

`load_bias()` (`hic3defdr.analysis.core.CoreHiC3DeFDR` method), 52

`load_cluster_table()` (in module `hic3defdr.util.cluster_table`), 73

`load_clusters()` (in module `hic3defdr.util.clusters`), 78

`load_data()` (`hic3defdr.analysis.core.CoreHiC3DeFDR` method), 52

`load_disp_fn()` (`hic3defdr.analysis.core.CoreHiC3DeFDR` method), 53

`log()` (`hic3defdr.util.banded_matrix.BandedMatrix` method), 68

`logpmf()` (in module `hic3defdr.util.scaled_nb`), 87

`loop_patterns()` (`hic3defdr.analysis.constructor.HiC3DeFDR` attribute), 51

`lowess_fit()` (in module `hic3defdr.util.lowess`), 81

`lrt()` (`hic3defdr.analysis.alternatives.Poisson3DeFDR` method), 46

`lrt()` (`hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR` method), 48

`lrt()` (in module `hic3defdr.util.lrt`), 83

M

`main()` (in module `hic3defdr.util.demo_data`), 78

`make_apu_stack()` (in module `hic3defdr.util.apu`), 65

`make_mask()` (`hic3defdr.util.banded_matrix.BandedMatrix` static method), 68

`make_upper()` (`hic3defdr.util.banded_matrix.BandedMatrix` method), 68

`make_y_true()` (in module `hic3defdr.util.evaluation`), 80

`max()` (`hic3defdr.util.banded_matrix.BandedMatrix` class method), 69

`max_range()` (`hic3defdr.util.banded_matrix.BandedMatrix` method), 69

`mean_thresh()` (`hic3defdr.analysis.constructor.HiC3DeFDR` attribute), 51

`median_of_ratios()` (in module `hic3defdr.util.scaling`), 88

`mme()` (in module `hic3defdr.util.dispersion`), 79

`mme_per_pixel()` (in module `hic3defdr.util.dispersion`), 79

`mvr()` (in module `hic3defdr.util.scaled_nb`), 87

N

`no_scaling()` (in module `hic3defdr.util.scaling`), 88

`NumpyEncoder` (class in `hic3defdr.util.clusters`), 74

O

`outdir` (`hic3defdr.analysis.constructor.HiC3DeFDR` attribute), 51

P

`parallel_apply()` (in module `hic3defdr.util.parallelization`), 85

`parallel_map()` (in module `hic3defdr.util.parallelization`), 85

`perturb_cluster()` (in module `hic3defdr.util.simulation`), 89

`picklefile` (`hic3defdr.analysis.core.CoreHiC3DeFDR` attribute), 53

`plot_correlation_matrix()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 53

`plot_dd_curves()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 54

`plot_dd_curves()` (in module `hic3defdr.plotting.distance_dependence`), 61

`plot_ddd()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 54

`plot_ddd()` (in module `hic3defdr.plotting.dispersion`), 59

`plot_dispersion_fit()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 54

`plot_distance_bias()` (in module `hic3defdr.plotting.distance_bias`), 60

`plot_fdr()` (in module `hic3defdr.plotting.fdr`), 61

`plot_fn_vs_fp()` (in module `hic3defdr.plotting.fn_vs_fp`), 62

`plot_grid()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 55

`plot_grid()` (in module `hic3defdr.plotting.grid`), 62

`plot_heatmap()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 55

`plot_heatmap()` (in module `hic3defdr.plotting.heatmap`), 63

`plot_ma()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 56

`plot_ma()` (in module `hic3defdr.plotting.ma`), 64

`plot_mvr()` (in module `hic3defdr.plotting.dispersion`), 59

`plot_pvalue_distribution()` (`hic3defdr.analysis.plotting.PlottingHiC3DeFDR` method), 56

`plot_pvalue_histogram()` (in module `hic3defdr.plotting.histograms`), 64

`plot_qvalue_distribution()` (*hic3defdr.analysis.plotting.PlottingHiC3DeFDR* *method*), 57
`plot_roc()` (*in module hic3defdr.plotting.roc*), 65
`PlottingHiC3DeFDR` (*class in hic3defdr.analysis.plotting*), 53
`Poisson3DeFDR` (*class in hic3defdr.analysis.alternatives*), 46
`poisson_fit_mu_hat()` (*in module hic3defdr.analysis.alternatives*), 47
`poisson_logpmf()` (*in module hic3defdr.analysis.alternatives*), 47
`poisson_lrt()` (*in module hic3defdr.analysis.alternatives*), 47
`prepare_data()` (*hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR* *method*), 49
Q
`q2qnbino()` (*in module hic3defdr.util.scaled_nb*), 87
`qcml()` (*in module hic3defdr.util.dispersion*), 79
R
`ravel()` (*hic3defdr.util.banded_matrix.BandedMatrix* *method*), 69
`raw_npz_patterns` (*hic3defdr.analysis.constructor.HiC3DeFDR* *attribute*), 50
`res` (*hic3defdr.analysis.constructor.HiC3DeFDR* *attribute*), 51
`reshape()` (*hic3defdr.util.banded_matrix.BandedMatrix* *method*), 69
`roll_footprint()` (*in module hic3defdr.util.banded_matrix*), 69
`roll_matrix()` (*hic3defdr.util.banded_matrix.BandedMatrix* *static method*), 69
`run_to_qvalues()` (*hic3defdr.analysis.analysis.AnalyzingHiC3DeFDR* *method*), 49
S
`save()` (*hic3defdr.util.banded_matrix.BandedMatrix* *method*), 69
`save_clusters()` (*in module hic3defdr.util.clusters*), 78
`save_data()` (*hic3defdr.analysis.core.CoreHiC3DeFDR* *method*), 53
`save_disp_fn()` (*hic3defdr.analysis.core.CoreHiC3DeFDR* *method*), 53
`select_matrix()` (*in module hic3defdr.util.matrices*), 84
`simple_scaling()` (*in module hic3defdr.util.scaling*), 89
`simulate()` (*hic3defdr.analysis.simulation.SimulatingHiC3DeFDR* *method*), 57
`simulate()` (*in module hic3defdr.util.simulation*), 89
`SimulatingHiC3DeFDR` (*class in hic3defdr.analysis.simulation*), 57
`size_filter()` (*in module hic3defdr.util.thresholding*), 90
`sort_cluster_table()` (*in module hic3defdr.util.cluster_table*), 73
`sparse_intersection()` (*in module hic3defdr.util.matrices*), 84
`sparse_union()` (*in module hic3defdr.util.matrices*), 84
`symmetrize()` (*hic3defdr.util.banded_matrix.BandedMatrix* *method*), 69
T
`threshold_and_cluster()` (*in module hic3defdr.util.thresholding*), 90
`tqdm_maybe()` (*in module hic3defdr.util.progress*), 85
U
`Unsmoothed3DeFDR` (*class in hic3defdr.analysis.alternatives*), 47
W
`weighted_lowess_fit()` (*in module hic3defdr.util.lowess*), 82
`where()` (*hic3defdr.util.banded_matrix.BandedMatrix* *method*), 69
`wipe_distances()` (*in module hic3defdr.util.matrices*), 84