lib5c Documentation

Release 0.6.1

Thomas Gilgenast

May 16, 2020

CONTENTS

A library for Chromosome Conformation Capture Carbon Copy (5C) analysis.

Contents:

CHAPTER

ONE

INSTALLATION

`lib5c` can be installed directly via `pip`

```
$ pip install lib5c
```

If you are having trouble with this step, make sure `pip` is up to date by running

```
$ pip install --upgrade pip
```

1.1 Prerequisites

`lib5c` currently only supports Python 2.7.11+ or Python 3.6+.

`pip` will install all required prerequisites automatically.

The complete list of required dependencies is listed in `install_requires` in `setup.py`.

1.2 Virtualenv installation walkthrough

It is highly recommended to install `lib5c` inside a fresh virtual environment to avoid package version conflicts.

If you don't have `virtualenv` you may install it via

```
$ pip install virtualenv
```

The step-by-step procedure for installing `lib5c` inside a fresh virtual environment is then:

```
$ deactivate                # deactivate any existing virtualenv
$ virtualenv venv           # create new python2 virtualenv
$ source venv/bin/activate  # activate the new virtualenv
(venv)$ pip install -U pip  # make sure pip is up to date
(venv)$ pip install lib5c   # install lib5c
(venv)$ lib5c -v            # check version number
```

On Windows, the virtualenv can be activated by running:

```
> venv\Scripts\activate
```

instead.

1.3 Docker image

A Docker image for `lib5c` is also provided and can be used as shown here:

```
$ docker pull creminslab/lib5c:latest
$ docker run -it creminslab/lib5c:latest
root@<container_id>:/# lib5c -v
```

We also provide a `creminslab/lib5c:slim` Docker image which is based on `python:2.7-slim`.

It is recommended to bind a directory containing input files. For example:

```
$ docker run -it -v c:/data:/data creminslab/lib5c:latest
root@<container_id>:/# cd /data
root@<container_id>:/data# lib5c pipeline
```

1.4 Optional dependencies

You can install `lib5c` with all optional dependencies included by running

```
$ pip install lib5c[complete]
```

The individual optional dependencies can also be installed one-at-a-time; they are described in detail in the following sub-sections.

1.4.1 `bsub` integration

If you use the LSF job scheduling system (also known as bsub), you should also install the `bsub` package to enable `lib5c` to leverage the job scheduler.

```
$ pip install bsub>=0.3.5
```

1.4.2 ICE matrix balancing

If you want to use the ICE matrix balancing algorithm in your analyses, you should also install the `iced` package to enable it in `lib5c`.

```
$ pip install iced>=0.4.0
```

This package may be difficult or impossible to install on Windows systems, in which case you can use the Knight-Ruiz matrix balancing algorithm as an alternative.

1.4.3 BigWig file support

In order to support interaction with BigWig formatted files, you should also install the `pyBigWig` package to enable it in `lib5c`.

```
$ pip install pyBigWig>=0.3.4
```

This package may be difficult or impossible to install on Windows systems.

1.5 Special notes

`lib5c` plots output using the `matplotlib` Python package. Instead of forcing a particular backend to be used, `lib5c` will respect the backend set in your `matplotlibrc`. As an example, the following `matplotlibrc` sets the backend to 'agg':

```
$ cat ~/.config/matplotlib/matplotlibrc
backend: agg
```

On Windows systems the `matplotlibrc` is usually located at `C:\Users\<username>\.matplotlib\matplotlibrc`

Another way to set your default backend is to use the `MPLBACKEND` environment variable:

```
$ export MPLBACKEND=agg
```

You can add this line to your `~/.bashrc` to set this environment variable on every login.

1.6 Editable mode installation

To install lib5c in editable mode:

```
$ git clone https://bitbucket.org/creminslab/lib5c
$ pip install -e ./lib5c
```

We recommend doing this is in a clean virtualenv.

To ensure that version information (e.g., as displayed by running `lib5c -v` on the command line) of an editable install of lib5c remains correct, install setuptools-scm:

```
$ pip install setuptools-scm
```

CHAPTER

TWO

INTRODUCTION

What is `lib5c`? How is it laid out? What can I do with it?

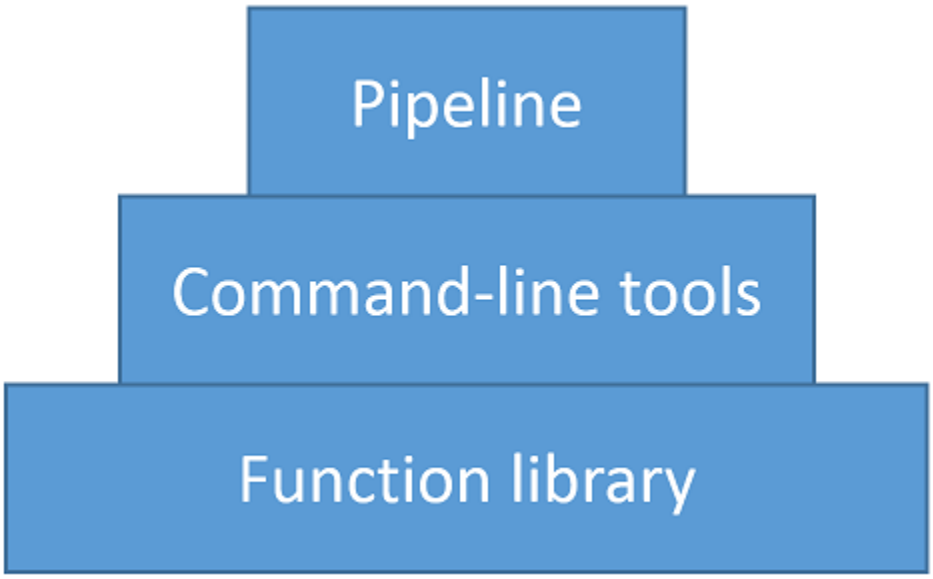
These are the questions that will be answered in this section.

2.1 What is `lib5c`?

`lib5c` is a comprehensive, modular library for analyzing the results of Chromosome Conformation Capture Carbon Copy (5C) experiments.

2.2 How is `lib5c` organized?

`lib5c` is organized into three distinct layers:



Each layer of the pyramid rests on the ones below it. At its base, `lib5c` is a library of modular, reusable functions that can be used for various tasks in 5C analysis. A set of command-line tools rests on top of these functions, making them easier to use for the most common use cases. The command-line tools in turn support a pipeline, which provides a customizable interface for performing entire sequences of analysis steps quickly and reproducibly.

Each of the three tutorials in the following sections of this documentation explain one of the three levels by example, starting from the top of the pyramid. You can start from the top, and go as deep as you feel comfortable going.

2.3 What can I do with `lib5c`?

This list shows just a few of the many things you can do with `lib5c`.

- Analyze:
  - Correct for locus-specific bias factors.
  - Bin or smooth fragment-level interaction data.
  - Construct distance-dependence models.
  - Perform statistical modeling.
  - Classify significant interactions from a two-condition experiment.
- Visualize:
  - Draw contact frequency heatmaps to see the architecture of the genome.
  - Draw bias factor heatmaps to idenfity covariates in the interaction data.
  - Visualize expected models and variance estimates.
  - Compare theoretical distributions to the real data.
  - Visualize enrichments between interaction classes and traditional genome annotations.

CHAPTER

THREE

API SPECIFICATION AND CONCEPTUAL DOCUMENTATION

This section describes the most important data structures and functions in more detail than the tutorial.

Contents:

3.1 Core data structures and file types

This section will introduce the core data structures and file types used ubiquitously throughout `lib5c`.

3.1.1 Core data structures

The core data structures used throughout `lib5c` are counts dicts and primermaps/pixelmaps, which serve as representations of contact matrices and locus information, respectively.

Representing contact matrices

The 5C assay attempts to quantify interactions between pairs of genomic loci. These genomic loci do not span the entire genome, as in Hi-C. Instead, they are restricted by the 5C primer design to contiguous blocks, which we will refer to as *regions*.

Throughout `lib5c`, we represent the pairwise *cis* (inter-region) interaction frequencies between loci as a square, symmetric matrix whose number of rows and columns is equal to the number of loci in the region. When there are multiple regions, we will put multiple matrices (each representing one region) into a dictionary whose keys are the region names as strings. Therefore, we end up with expressions like:

```
counts[region][i, j] = 23.5
```

where `region` is the name of the region as a string, `i` and `j` are integer indices corresponding to loci within the region, and `counts[region][i, j]` gives the value of the interaction frequency between the `i` th and the `j` th locus of the region, which may be an integer or floating-point number.

We will commonly call these data structures "counts dicts". More formally, the Python type annotation for a "counts dict" is:

```
Dict[str, np.ndarray]
```

Representing information about loci

A contract matrix is meaningless when it is separated from information about what specific genomic loci it describes. For every dictionary of contact matrices, we will usually also have a separate object that stores information about the genomic loci whose interactions are quantified in the contact matrices. This will have the form:

```
primermap[region][i] = {
    'chrom': 'chr3',
    'start': 34107373,
    'end': 34109022,
    'name': '5C_329_Sox2_REV_1',
    'strand': '-'
}
```

where `region` is the name of the region as a string, `i` is the index of the locus within the region, and `primermap[region][i]` is a dict storing information about the `i` th locus in the region. At a minimum, it must indicate the chromosome, start, and end of the locus. In practice, it can also include additional information such as the name of the locus or (for loci that are restriction fragments) the strand that the 5C primer for this fragment was designed to.

We will commonly call these data structures "primermaps" when the loci they describe are primers, and "pixelmaps" when the loci they describe are bins (in reference to the "pixels" on a 5C heatmap). More formally, the Python type annotation for either of these data structures is:

```
Dict[str, List[Dict[str, Any]]]
```

where the keys to the outer dict are region names and the inner dict must have at least the keys 'chrom', 'start', and 'end'.

3.1.2 Core file types

The core file types used as inputs and outputs throughout `lib5c` are countsfiles and primerfiles/bin bedfiles.

Representing contact matrices

Countsfiles are used to represent contact matrices.

Representing information about loci

Special bedfiles called primerfiles are used to represent the loci whose interactions are contained in countsfiles.

3.2 Parallelization across regions

Given that the most common data structure is a counts dict (whose keys are the region names in our dataset), we often want to call a function for each region in this dictionary:

```
>>> result = {region: fn(counts[region]) for region in counts}
```

This pattern may become even more complicated if `fn()` returns a tuple, for example. Furthermore, it is clear that overall operation is "embarrassingly parallel" with respect to the regions being processed. In order to simplify our code, reduce redundancy, and gain the benefits of parallel execution, we introduce a new decorator: `@parallelize_regions`, which can be found in the subpackage *lib5c.util.parallelization*. This

decorator allows you to write `fn()` just once, writing it as if it processes only one matrix, but then call it with one matrix or an entire counts dict as is convenient. For example, we can write

```
from lib5c.util.parallelization import parallelize_regions

@parallelize_regions
def fn(matrix):
    return matrix + 1
```

and then call this function via

```
result_counts = fn(counts)
```

or alternatively,

```
result_matrix = fn(counts['Sox2'])
```

as is convenient for us.

3.2.1 Mechanism and caveats

The following sections dig into the mechanics behind the `@parallelize_regions` decorator and highlight some important features and caveats.

First positional argument dependence

The `@parallelize_regions` decorator works by first checking to see if the first argument passed to the decorated function is a dict. If it is not, the decorator does nothing, and the function is executed as normal. If it is a dict, the execution of the function is parallelized across the keys of that dict. This means that if the non-parallelized version of `fn()` expects a dict as its first positional argument, you will not be able to use the same name for both the parallel and non-parallel versions of the function. To work around this, you can define

```
from lib5c.util.parallelization import parallelize_regions

def fn(somedict):
    return somedict

fn_parallel = parallelize_regions(fn)
```

and then you can call `fn(somedict)` when you want the non-parallelized version and `fn_parallel(doubledict)` when you want the parallelization.

Per-region args and kwargs

By default, `@parallelize_regions` will simply copy all the other args and kwargs to each region's invocation of `fn()`. In other words, when you call `fn(counts, arg_1, arg_2)`, the following will be executed:

```
fn(counts['region_1'], arg_1, arg_2)
fn(counts['region_2'], arg_1, arg_2)
...
```

However, if any arg or kwarg is a dict which has the same keys as the first positional argument (or, if the arg is a nested dict, if its second level has these same keys), the arg will be replaced with each region's entry in that dict. In

other words, if we call `fn(counts, primermap)`, where `primermap` is a dict whose keys match `counts`, the following will be executed:

```
fn(counts['region_1'], primermap['region_1'])
fn(counts['region_2'], primermap['region_2'])
...
```

This substitution is performed on an arg-by-arg basis, so you can use any mixture of normal and "regional dictionary" arguments when calling the fucnction.

Automatic result unpacking

Let's say `fn()` returns a tuple, for example:

```
from lib5c.util.parallelization import parallelize_regions

@parallelize_regions
def fn(matrix):
    return matrix + 1, matrix - 1
```

When we call `fn()` on a single matrix, we expect to see

```
bigger_matrix, smaller_matrix = fn(matrix)
```

The same thing will work when calling `fn()` on a counts dict:

```
bigger_counts_dict, smaller_counts_dict = fn(counts)
```

In this case `bigger_counts_dict` and `smaller_counts_dict` will each be dicts whose keys match the keys of `counts`.

Fallback to series execution

If an error is encountered during the parallel processing, the decorator will attempt to re-run the same job in series, in hopes that this will result in a more readable stack trace.

Signature preservation

`@parallelize_regions` is itself decorated by the `@pretty_decorator` meta-decorator, which can be found in *`lib5c.util.pretty_decorator`*. This allows the signature of the decorated function to be preserved through the decoration process.

3.3 Plotting

We often want to write a function which plots something. In order to be useful in a large variety of scenarios, a good library plotting function should be extremely flexible and support a large variety of options. We probably want to support basically the same set of options for virtually every plotting function we write. In order to avoid writing redundant code, we present the `@plotter` decorator, which is found in *`lib5c.util.plotting`*.

3.3.1 Design principles

We expect a generic plotting function `plot_fn()` should take in some data and plot it to the default `pyplot` axis. We expect to be able to override some details about the plot, including what axis should be plotted on, the plot title, whether or not to add a legend, etc. Ideally, the function should return the axis it plotted on, which would allow the caller to make further modifications to the plot before saving to disk. However, many clients will probably want to just save the figure to disk as fast as possible, so an "automatic save" option should exist as well.

3.3.2 Examples

We consider the plotting function `plot_fn()`:

```
import matplotlib.pyplot as plt

from lib5c.util.plotting import plotter

@plotter
def plot_fn(x, y, **kwargs):
    plt.scatter(x, y)
```

Importantly, the `@plotter` API requires client functions to accept arbitrary `**kwargs` in their signature.

A client may call

```
ax = plot_fn(x, y)
```

where the return value of `plot_fn()` will simply be the axis that was plotted on.

To plot to a specific axis called `ax`, the client may call

```
ax = plot_fn(x, y, ax=ax)
```

To automatically save the figure to disk, the client may call

```
plot_fn(x, y, outfile='plot.png')
```

Plots will be automatically saved at 300 dpi, but this can be overriden by calling

```
plot_fn(x, y, outfile='plot.png', dpi=800)
```

Plots will be styled with `seaborn`, using the `'ticks'` axis style. To use a different `seaborn` style, the client can call

```
plot_fn(x, y, style='darkgrid')
```

The `@plotter` decorator will always automatically remove the `seaborn` styles after the call completes, allowing future plotting calls to look "normal".

To avoid using the `seaborn` styles and stick with `matplotlib` defaults, the client can call

```
plot_fn(x, y, style=None)
```

The plot will be despined with `sns.despine()` by default. To disable this, the client can call

```
plot_fn(x, y, despine=False)
```

The decorated `plot_fn()` will accept a kwarg called `legend`. By default, this kwarg is `None`, which leaves all decisions about the legend in the hands of `plot_fn()`. If `plot_fn()` adds a legend to the plot, but you wish to remove it, call

```
plot_fn(x, y, legend=False)
```

If `plot_fn()` does not add a legend, but you wish to have one, call

```
plot_fn(x, y, legend=True)
```

If the legend is really large and you wish it to be added outside the plot area, you can call

```
plot_fn(x, y, legend='outside')
```

Other minor adjustments can be made to the plot with the kwargs illustrated in the following example call:

```
plot_fn(x, y, xlim=(0, 10), ylim=(-1, 1), xlabel='number of cows',
        ylabel='relative change in grass', xticks=11, yticks=[-1, 0, 1],
        title='cows vs grass')
```

Here we note that `xticks` and `yticks` can be passed as either an integer (in which case the specified axis will have that many evenly-spaced ticks) or as anything accepted by `plt.xticks()` (in our example, a list of tick locations).

3.4 Trimming

An important early preprocessing step is the removal of low-quality primers from the dataset.

3.4.1 Command-line interface

Primer trimming can be accomplished on the command line by running

```
$ lib5c trim
```

For complete details on the usage of this command, see the output of

```
$ lib5c trim -h
```

3.4.2 Exposed functionality

The algorithms which make up the primer trimming framework can be found in the *lib5c.algorithms.trimming* subpackage.

The core API is exposed in the following convenience functions:

- *lib5c.algorithms.trimming.trim_primers()*
- *lib5c.algorithms.trimming.wipe_counts()*
- *lib5c.algorithms.trimming.trim_counts()*

The functions `wipe_counts()` and `trim_counts()` also have convenience wrappers which apply them over a counts superdict (dict of counts dicts, whose first-level keys are replicate names), which are:

- *lib5c.algorithms.trimming.wipe_counts_superdict()*
- *lib5c.algorithms.trimming.trim_counts_superdict()*

Workflow

The general workflow is to trim primers first (based on the quality of the counts matrices in the dataset), and then either trim or wipe those counts matrices:

```
from lib5c.algorithms.trimming import trim_primers, trim_counts_superdict

trimmed_primermap, trimmed_indices = trim_primers(primermap, counts_superdict)
trimmed_counts_superdict = trim_counts_superdict(counts_superdict, trimmed_indices)
```

The call to `trim_primers()` does not modify the `counts_superdict`, leaving the client to decide what to do next.

Trimming versus wiping

`trim_counts()` removes rows and columns from the matrices in the counts dict, with the result that the dimensions of these matrices will match the lengths of the values of `trimmed_primermap`. This is the recommended way to treat removal of low-quality fragments.

`wipe_counts()` does not change the dimensions of any matrix, and instead simply paints over the removed indices according to its kwarg `wipe_value`. This can be useful when removing low-quality regions from already-binned data, for example:

```
from lib5c.algorithms.trimming import trim_primers, wipe_counts_superdict

_, trimmed_indices = trim_primers(pixelmap, counts_superdict)
wiped_counts_superdict = wipe_counts_superdict(counts_superdict, trimmed_indices)
```

Notice that we discard the `trimmed_pixelmap` from the first function call, because this pixelmap's dimensions do not match any of the counts dicts.

Trimming options

There are two different ways to assess the quality of a primer: its total *cis* contact count (row sum in the counts matrix) or the fraction of its possible interactions which are nonzero. These two quality metrics are thresholded on by the two kwargs of `trim_primers()`: `min_sum` and `min_frac`.

3.5 Quantile normalization

Quantile normalization is a method for bringing disparate 5C libraries (of varying sequencing depth, library complexity, etc.) to a comparable scale.

3.5.1 Command-line interfaces

The subcommand is

```
$ lib5c qnorm
```

For detailed help, run

```
$ lib5c qnorm -h
```

3.5.2 Exposed functionality

The subpackage for quantile normalization is *lib5c.algorithms.qnorm*

Two functions are exposed:

- *lib5c.algorithms.qnorm.qnorm()*
- *lib5c.algorithms.qnorm.qnorm_counts_superdict()*

`qnorm()` performs quantile normalization on arbitrary data, and may be re-used in other contexts. It operates on a table of input data, not the counts dicts (and superdicts) which are commonly used in the library. Therefore, the wrapper function `qnorm_counts_superdict()` is provided to make this step easier for our 5C data.

3.6 Bias mitigation

5C read counts are strongly influenced by bias factors which are dictated by intrinsic properties of the restriction fragments and primers involved in the reactions. Moreover, the influence of these bias factors can vary from replicate to replicate. `lib5c` includes a wide variety of algorithms for mitigating the effects of these bias factors.

3.6.1 Approaches

A variety of approaches to bias mitigation are possible.

Explicit normalization (spline)

This approach involves fitting splines to the three-dimensional surfaces generated by plotting each entry of the contact matrix on the z-axis, and setting the x- and y-axis positions according to some property of the upsteam and downstream fragment involved in the ligation junction represented by that matrix entry. These fitted splines can then be simply subtracted from the experimentally observed data.

To perform this normalization on the command line, run

```
$ lib5c spline
```

Splines can be visualized by running

```
$ lib5c plot visualize-spline
```

The exposed function for performing the spline normalization is *lib5c.algorithms.spline_normalization.iterative_spline_normalization()*.

The exposed function for visualizing the spline is *lib5c.plotters.splines.visualize_spline()*.

Simple matrix balancing approaches (kr and iced)

Matrix balancing approaches attempt to equalize the row sums of the contact matrix, without knowing anything about the intrinsic properties of the restriction fragments.

The Knight-Ruiz matrix balancing algorithm can be used by running

```
$ lib5c kr
```

The exposed function is *lib5c.algorithms.knight_ruiz.kr_balance_matrix()*.

The ICED matrix balancing algorithm is implemented by iced, and an easy-to-use interface to this package is exposed in `lib5c`.

It can be used on the command line by running

```
$ lib5c iced
```

if `iced` has been installed by running

```
$ pip install iced
```

The exposed function is *lib5c.contrib.iced.balancing.iced_balance_matrix()*.

Advanced matrix balancing (express)

The Express matrix balancing algorithm takes into account a simple one-dimensional distance-dependent expected model when balancing, which can improve balancing performance given the wide dynamic range of interactions across any given row of the interaction matrix.

It can be used on the command line by running

```
$ lib5c express
```

The "Joint Express" variant first described in this library can be used by running

```
$ lib5c express -J
```

The exposed functions are:

- *lib5c.algorithms.express.express_normalize_matrix()*
- *lib5c.algorithms.express.joint_express_normalize()*

3.6.2 Assessing bias factor profiles

Bias factor profiles can be visualized by running

```
$ lib5c plot bias-heatmap
```

The exposed function is *lib5c.plotters.bias_heatmaps.plot_bias_heatmap()*

The overall balance of a contact matrix can be visualized by running

```
$ lib5c plot boxplot
```

The exposed function is *lib5c.plotters.boxplots.plot_regional_locus_boxplot()*

3.7 Binning and smoothing

To reduce spatial noise in 5C data, we can treat the 5C contact frequencies as a 2-D signal that can be smoothed with various filtering functions. Depending on the original coordinates of the contact matrix and the coordinates we choose to evaluate the filtered signal at, this process can be referred to as "binning" or "smoothing".

3.7.1 Theoretical overview

We will create "filtering functions" to pass over the contact matrices. For generality in terms of the level of data the filtering functions can be applied to, we will require that filtering functions compute values on "neighborhoods" of points defined by spatial proximity to the point we want to evaluate the filtered signal at.

3.7.2 Command-line interfaces

Command-line interfaces for binning and smoothing countsfiles are provided directly in `lib5c`.

Binning

If we have a fragment-level countsfile called `fragment_level.counts`, a primer bedfile called `primers.bed`, and a binned bedfile called `bins.bed`, we can run

```
$ lib5c bin -w 20000 -p primers.bed -b bins.bed fragment_level.counts binned.counts
```

to bin the counts using a 20 kb window width.

For a complete list of command-line flags for the `lib5c bin` subcommand, we can run

```
$ lib5c bin -h
```

Smoothing

If we have a countsfile called `unsmoothed.counts` and a bedfile called `loci.bed`, we can run

```
$ lib5c smooth -w 20000 -p loci.bed -b bins.bed unsmoothed.counts smoothed.counts
```

to smooth the counts using a 20 kb window width.

For a complete list of command-line flags for the `lib5c smooth` subcommand, we can run

```
$ lib5c smooth -h
```

3.7.3 Exposed functionality

The algorithms which make up the filtering can be found in the *lib5c.algorithms.filtering* subpackage.

Core API

The core API for filtering is in the form of three convenience functions:

- *lib5c.algorithms.filtering.bin_bin_filtering.bin_bin_filter()*
- *lib5c.algorithms.filtering.fragment_fragment_filtering.fragment_fragment_filter()*
- *lib5c.algorithms.filtering.fragment_bin_filtering.fragment_bin_filter()*

These functions take in a counts matrix and return a filtered counts matrix. They also take in a filter function, which will be described in detail below. They also require information about the loci involved in the filtering. Finally, they require a distance threshold for defining the neighborhood of points that will be passed to the filter function.

Filter function API

Filter functions must take in a representation of a "neighborhood" around a point and return a scalar value representing the evaluation of the filtered signal at that point. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

More formally, the Python type annotation for a filter function is:

```
Callable[[List[Dict[str, Any]]], float]
```

The *lib5c.algorithms.filtering.filter_functions* module provides a framework for the construction of filter functions with desired properties. The key function exposed there is

lib5c.algorithms.filtering.filter_functions.make_filter_function()

which constructs a filter function according to the specification in the kwargs and returns it.

3.8 Expected modeling

5C data exhibit a strong distance-dependent background signal which is also influenced by local contact domain structure. `lib5c` includes a variety of algorithms for modeling this background expected interaction frequency.

3.8.1 Command-line interfaces

The subcommand is

```
$ lib5c expected
```

For detailed help, run

```
$ lib5c expected -h
```

3.8.2 Exposed functionality

The subpackage responsible for expected modeling is *lib5c.algorithms.expected*.

The most important exposed function is *lib5c.algorithms.expected.make_expected_matrix()*.

To use a global expected model, you must also use *lib5c.algorithms.expected.get_global_distance_expected()*, leading to the following workflow:

```
from lib5c.algorithms.expected import make_expected_matrix, get_global_distance_
↪expected
```

(continues on next page)

(continued from previous page)

```
distance_expected = get_global_distance_expected(observed_counts)
expected_counts = make_expected_matrix(observed_counts, distance_expected=distance_
↪expected)
```

The following functions are provided for visualizing one-dimensional expected models overlayed over real data:

- *lib5c.plotters.expected.plot_bin_expected()*
- *lib5c.plotters.expected.plot_fragment_expected()*

3.9 Variance modeling

In order to obtain measures of statistical significance for 5C interactions, we need a quantitative measure of statistical noise for each interaction. `lib5c` provides a variety of methods for estimating this variance.

3.9.1 Command-line interfaces

To model the variance, run

```
$ lib5c variance
```

To visualize variance estimates, run

```
$ lib5c plot visualize-variance
```

3.9.2 Exposed functionality

The subpackage responsible for variance modeling is *lib5c.algorithms.variance*.

The top-level convenience function exposed is *lib5c.algorithms.variance.estimate_variance()*

`estimate_variance()` provides easy access to all available variance estimation methods.

For an example of how to visualize variance estimates using *lib5c.plotters.scatter.scatter()* or *lib5c.plotters.curve_fits.plot_fit()*, take a look at *lib5c.tools.visualize_variance.visualize_variance_tool()*.

3.10 Distributions

In order to call p-values for individual 5C interactions, we simply compare the observed value of the interaction to its expected value (see *Expected modeling*) and its variance estimate (see *Variance modeling*) by using a statistical distribution parameterized to have mean and variance equal to the predictions of the expected and variance models, respectively.

3.10.1 Conceptual overview

We may wish to call p-values using a variety of different statistical distributions, each of which has a unique "native" parameterization. To keep things from becoming too complicated, we ignore these "native" parameterizations and instead always parameterize distributions with the same two parameters: mean and variance.

3.10.2 Command-line interfaces

To call p-values, run

```
$ lib5c pvalues
```

To plot the parameterized distributions over the real data, run

```
$ lib5c plot visualize-fits
```

3.10.3 Exposed functionality

The exposed function is *lib5c.util.distributions.call_pvalues()*.

Additional utility functions are provided in *lib5c.util.distributions* to ease parameter conversion, etc.

The following functions are exposed for visualization of distributions:

- *lib5c.plotters.fits.plot_fit()*
- *lib5c.plotters.fits.plot_group_fit()*

`plot_fit()` is a reusable function for overlaying distributions.

`plot_group_fit()` is a convenience function for overlaying parametrized distributions over groups of real data points.

3.11 Thresholding

After calling p-values for interactions in each of several replicates, we want to find a way to combine this information across all replicates to identify condition-specific interactions. `lib5c` provides a simple implementation of such a procedure.

3.11.1 Conceptual background

We assume each replicate belongs to one of exactly two conditions, call them "A" and "B". This implies that there are three important classes of interactions we wish to recover:

- "A-only" (looping in condition A but not in condition B)
- "B-only" (looping in condition B but not in condition A)
- "constitutive" (looping in both conditions)

We will also recover a "background" class, which will be convenient when assessing *enrichments*.

3.11.2 Command-line interface

To perform the simple two-way thresholding, run

```
$ lib5c threshold
```

3.11.3 Exposed functionality

The subpackage for thresholding is *lib5c.algorithms.thresholding*.

The main exposed function is *lib5c.algorithms.thresholding.two_way_thresholding()*.

The following functions are exposed to compute various useful metrics from the results of `two_way_thresholding()`:

- *lib5c.algorithms.thresholding.kappa()*
- *lib5c.algorithms.thresholding.concordance_confusion()*
- *lib5c.algorithms.thresholding.color_confusion()*
- *lib5c.algorithms.thresholding.count_clusters()*

3.12 Enrichments

An important endgame step in 5C analysis is the quantification of overlap between specific interaction classes and traditional genomic annotations. `lib5c` provides a system for performing such an enrichment analysis.

3.12.1 Command-line interface

To plot traditional enrichments, run

```
$ lib5c plot enrichment
```

To plot the enrichment of occupied motifs with a specific orientation, run

```
$ lib5c plot convergency
```

3.12.2 Exposed functionality

The exposed functions for computing enrichments are:

- *lib5c.algorithms.enrichment.count_intersections_all()*
- *lib5c.algorithms.enrichment.get_annotation_percentage_all()*
- *lib5c.algorithms.enrichment.get_fold_change_all()*
- *lib5c.algorithms.enrichment.get_fisher_exact_pvalue_all()*
- *lib5c.algorithms.convergency.compute_convergency()*

The exposed functions for visualization are:

- *lib5c.plotters.enrichment.plot_looptype_vs_annotation_heatmap()*
- *lib5c.plotters.enrichment.plot_annotation_vs_annotation_heatmap()*
- *lib5c.plotters.enrichment.plot_stack_bargraph()*
- *lib5c.plotters.convergency.plot_convergency()*

3.13 Clustering

Sometimes, we want to group 5C interactions into *clusters* according to some perceived spatial and architectural related-ness. This can happen when we see several nearby "pixels" on a 5C heatmap with high intensities. This section explains the functionality of the `lib5c` clustering framework, which can be used to identify these clusters using any of a suite of clustering algorithms.

3.13.1 Command-line interfaces

A command-line interface to this clustering framework exists as a separate repository, which can be found at https://bitbucket.org/creminslab/3d-clusterbot

3.13.2 Exposed functionality

The algorithms which make up the clustering framework can be found in the *`lib5c.algorithms.clustering`* subpackage.

Core data structures

peak (`Dict[str, numeric]`) Withing the clustering framework, the term *peak* is used to refer to an object that represents a single interaction. The data structure used to represent a peak is a dictionary of the form:

```
{
    'x': 4,
    'y': 7,
    'value': 12.43
}
```

we can think of peaks as an unfolded contact matrix, where each entry in the contact matrix gets represented as a separate dictionary. This representation may seem overly verbose, but it has the dual advantage of becoming sparse when interactions when peaks with values below some threshold are excluded from clustering, and of providing a simple representation of clusters as lists of peaks (see below).

cluster (`List[Dict[str, numeric]]`) Within the clustering framework, the term *cluster* is used to refer to a list of peaks that have been determined to belong together.

Core API

Many different paradigms for clustering operations exist. There are three broadly-defined clustering operations:

1. cluster assembly (the initial construction of clustered from a flat list of candidate peaks)
2. cluster merging (the merging of two clusters into one cluster)
3. cluster splitting (the splitting of clusters into subclusters)

Cluster assembly

The API for cluster assembly is to define a single function

```
make_clusters(peaks, **kwargs)
```

that takes in a list of candidate peaks and returns a list of clusters.

The cluster assembly paradigms available are:

knn, via *lib5c.algorithms.clustering.knn.make_clusters()* Assembles clusters by classifying each peak into a cluster according to the cluster membership of its nearest neighbors.

adjacency, via *lib5c.algorithms.clustering.adjacency.make_clusters()* Assumes the clusters are simply the connected components made up of peaks, as judged by spatial adjacency.

greedy, via *lib5c.algorithms.clustering.greedy.make_clusters()* Attempts to grow clusters in a greedy fashion by absorbing all nearby peaks and clusters.

Cluster merging

The API for cluster merging is to define a single function

```
merge_to_which(clusters)
```

which takes in a list of clusters and returns the index of the cluster that `clusters[0]` should be merged into, or -1 if `clusters[0]` shouldn't be merged.

The cluster merging paradigms available are:

adjacency, via *lib5c.algorithms.clustering.adjacency.merge_to_which()* Merges adjacent clusters together.

enclave, via *lib5c.algorithms.clustering.enclave.merge_to_which()* Merges clusters together if one cluster completely surrounds another.

To recursively merge a list of clusters using a `merge_to_which()` function, a utility function is provided as

lib5c.algorithms.clustering.util.merge_clusters()

which takes in the list of clusters and a reference to the `merge_to_which()` function that defines the merge rule to use, and returns a list of recursively-merged clusters.

Cluster splitting

The API for cluster splitting is to define a single function

```
split_clusters(clusters, **kwargs)
```

which takes in a list of clusters and returns a list of clusters that have been recursively split according to some splitting rule. By convention, the splitting rule should be defined in a function called `split_cluster()`, but different splitting paradigms may use different signatures for this function.

The cluster splitting paradigms available are:

valley, via *lib5c.algorithms.clustering.valley.split_clusters()* Splits clusters by identifying "valleys" between the "mountains" that correspond to true clusters.

quasicontiguity, via *lib5c.algorithms.clustering.quasicontiguity.split_clusters()* Splits clusters according to a "quasicontiguity" criterion where clusters that are physically separated into two sufficiently large and sufficiently distant subcomponents get split into separate clusters.

CHAPTER

FOUR

TESTING, BUILDING, AND RELEASING

This section explains our testing, building, and release process.

4.1 Maintenance automation

We use tox to orchestrate our maintenance tasks. To use tox, all you need to do is install it (`pip install tox`). All other dependencies will be managed by tox.

4.2 Linting and testing

To run linting and testing across all supported Python versions, run

```
$ tox
```

To lint, run

```
$ tox -e lint
```

To run tests for a specific Python version (e.g., 2.7), run

```
$ tox -e py27-unpinned
```

We lint with flake8. and run tests with nosetests.

We generally follow PEP 8, with a few exceptions which are listed in `setup.cfg`.

Tests are primarily set up via doctest, though a few are set up using unittest and live in `lib5c.<some_module>.tests.test_<some_feature>.py`.

Our CI pipeline (run in Bitbucket Pipelines right inside the repo) runs tox under all our test environments for every commit.

We pin one specific set of dependency versions in `requirements.txt`. This is the set of pinned versions used for our Docker image builds as well as special `pyXX-pinned` test environments in tox. The `pyXX-unpinned` test environments do not pin any package versions; therefore, they closely resemble the experience of a new user attempting to run `pip install lib5c`.

The `pyXX-pinned` test environments are skipped on Windows, since our pinned package versions in `requirements.txt` include some packages which are not installable on Windows.

4.3 Committing

The git repository for `lib5c` is https://bitbucket.org/creminslab/lib5c/.

We roughly try to follow GitHub Flow when committing, trying to never allow the `master` branch to contain breaking code.

4.4 Building

We build Python wheels and source distributions, as well as Docker images.

4.4.1 Versioning

Before building, you can optionally tag the release with a version number. We try to follow semantic versioning whenever possible. Our version tags do not include a leading "v" (e.g., we would use `0.5.3` as a tag rather than `v0.5.3`). To tag, run

```
$ git tag 0.5.1
$ git push --tags
```

We use setuptools-scm to obtain information about the current version directly from git.

4.4.2 Python wheel and source distribution

To build the wheel for `lib5c`, run

```
$ python setup.py bdist_wheel
```

To build the source distribution for `lib5c`, run

```
$ python setup.py sdist
```

4.4.3 Docker image

To build both normal and slim docker images, run

```
$ tox -e docker build
```

To promote the current images (normal and slim) to the `latest` and `slim` tags, respectively, run

```
$ tox -e docker promote
```

To build the `lib5c` Docker images, we pass the version name into the `Dockerfile` as a `build-arg` called `VERSION`. To get the version in a cross-platform way, we provide a utility script `lib5c/_version.py` which, when run, simply prints the current version. Running this script requires that either lib5c or setuptools-scm (`pip install setuptools_scm`) be installed. setuptools-scm is likely to be the easier option since it is much lighter to install (it has no dependencies), but developers who have already installed lib5c (e.g., in dev mode) do not need to install setuptools-scm. The "docker" tox testenv installs setuptools-scm in its own isolated environment.

We supply two images for each tag: one based on `python:2.7` (about 1.3 GB total) and a second with a `-slim` suffix based on `python:2.7-slim` (about 600 MB total). The main Dockerfile is located in the root directory of

the project and is recommended if you already use the `python:2.7` base image anywhere else on your machine. The `-slim` Dockerfile is located in `docker-slim/` and is recommended if you don't plan to use `python:2.7` as a base image for any other work on your machine.

At the current time, we only supply these Python 2.7 based images. In the future, we may also supply Python 3 based images.

Note that we tag the Docker image with the direct output of `git describe`, since Docker image tags can't contain plus signs.

4.5 Releasing

We release wheel and source distributions to PyPI and the Docker images to Docker Hub.

4.5.1 PyPI

To build both the wheel and source distribution and upload them to PyPI, run

```
$ python setup.py sdist bdist_wheel upload
```

4.5.2 Docker Hub

To push the Docker images to Docker Hub, run

```
$ tox -e docker push
$ tox -e docker pushlatest  # pushes the latest and slim tags
```

These commands assume you've logged in with Docker by running `docker login`.

4.6 Documentation

To build docs, run

```
$ tox -e docs
```

Documentation pages are stored in `docs/` and are built using Sphinx.

We use the sphinxcontrib.apidoc extension to run `sphinx-apidoc` on every doc build. This means that the apidoc-generated `lib5c*.rst` files should not be checked into git. The tox testenv deletes these files for you after building the docs.

Docs are built automatically on every commit to `dev`, publishing the results to https://lib5c.readthedocs.io/en/latest/.

Tagged versions can be added to the list of stable versions via the readthedocs website. The latest tagged version on `master` will be published to https://lib5c.readthedocs.io/en/stable/.

4.7 Tutorials

To test the tutorials, run

```
$ tox -e tutorials
```

Tutorial source notebooks (containing no outputs) are stored in the `tutorials/` directory under the project root.

The tutorials are run by Bitbucket Pipelines but need to be triggered manually, which results in the creation of new notebooks that include outputs. These resulting notebooks are pushed to a separate repository.

To strip outputs from the notebooks before committing them back to the lib5c repo, run:

```
$ python tutorials/clean.py
```

4.8 Cheat sheet

The commands explained above are collected all in one place in the cheat sheet below:

```
# lint and test
tox

# git commit
git commit -m 'commit message'
git push

# git tag (optional)
git tag 0.5.1
git push --tags

# build wheel
python setup.py bdist_wheel

# build and upload to pypi
python setup.py sdist bdist_wheel upload

# local Docker build and tag
tox -e docker build
tox -e docker promote

# push Docker images
tox -e docker push
tox -e docker pushlatest

# build docs
tox -e docs

# run tutorials
tox -e tutorials

# cleanup tutorials
python tutorials/clean.py
```

CHAPTER

FIVE

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.

5.1 0.6.1 2020-05-16

5.1.1 Added

- A license (MIT).

5.1.2 Changed

- Reduced the usage of shortcut imports to improve compatibility with clctools, see clctools#4.

5.1.3 Fixed

- Minor bugs: #77, #78, #50, #81.

5.2 0.6.0 - 2020-03-11

This version finally brings Python 3 support to lib5c!

Currently all tests and all tutorials pass under both Python 2.7 and Python 3.6, but it's possible that a few bugs exist in areas with low test coverage.

5.2.1 Added

- Experimental Python 3 support, see #54.
- New functionality for getting data for the tutorials in `lib5c.util.demo_data`, see #75.
- New utility function for loading config files using ConfigParser: `lib5c.util.config.parse_config()`.

5.2.2 Changed

- numpy arrays containing strings are now always created with a "U" dtype.

5.2.3 Fixed

- You can now pass a `(positions, labels)` tuple to the `xticks` and `yticks` kwargs of a function decorated with `@plotter`, see #25.
- When running in Python 3 environments where ConfigParser is not available, the default pipeline config will have all its "%" symbols escaped when it is dropped to disk, see #73.
- Y-axis gene tracks now appear correctly on trans heatmaps, see #69.
- Two-way thresholding now works even when there is only one region, see #71.
- Quoting paths with wildcards when passing them to lib5c commands no longer prevents glob expansion in Windows, see #76.

5.2.4 Updates/maintenance

- Bumped minimum `matplotlib` dependency, see #59.
- Minor documentation improvements, see #63, #64, and 18a7f6.
- Changed testing/linting to use tox. tox now serves as a centralized place to control maintenance-related actions like running the tutorials, building Docker images, testing the doc build, etc.
- Since tox doesn't support shell redirection/expansion, we moved the Docker one-liners into a new utility script `_docker.py` in the project root which is called by the new docker testenv (`tox -e docker`).
- Overhauled doc build process. We now use sphinxcontrib-apidoc to automatically run `sphinx-apidoc` on every doc build, avoiding the need to commit the per-module apidoc-generated .rst files to git. We also use the readthedocs config file to configure the doc build on readthedocs. Both the local (`tox -e docs`) and readthedocs builds are now configured to exit when they encounter a warning.
- Reduced the fragility of many doctest cases that relied on the ordering of dictionary keys.
- Streamlined Docker image build so that building the wheel beforehand is no longer necessary, see #67. This also fixes #68.
- We now use setuptools-scm to manage version information instead of versioneer. We repurposed the existing `lib5c._version` module to provide setuptools-scm specific functionality.
- Added dependency on configparser to make config file parsing more consistent across Python versions.
- Relaxed maximum version constraint on python-daemon, since luigi seems to be handling this now.
- Replaced references to pandas's deprecated `get_matrix()` bound method with the now-recommended `values` property.
- Thanks to #75 and #76, tutorials have been updated to use `lib5c.util.demo_data` and now run on Windows (`tox -e tutorials`). Tutorials now run on Python 3.6.

5.3 0.5.5 - 2020-02-04

5.3.1 Changed

- Enabled extrapolation in `lib5c.util.lowess.lowess_fit()`, see #60.

5.3.2 Updates/maintenance

- Enforce maximum supported version of `statsmodels` since they are no longer building Python 2 wheels.

5.4 0.5.4 - 2019-06-07

First wave of major heatmap plotting upgrades plus streamlining of the release process via Bitbucket Pipelines.

5.4.1 Added

- A changelog, see #47.

5.4.2 Changed

- ExtendableHeatmap ChIP-seq track and gene track plotting are now faster thanks to the use of PolyCollection and LineCollection, see #28.

5.4.3 Fixed

- Bug #56.

5.4.4 Updates/maintenance

- Updated `flake8` configuration (to match previous behavior when using latest version of `flake8`).
- Simplified `.gitignore`.
- Tests are now performed by Bitbucket Pipelines on every push.
- Reworked documentation to use Read the Docs.
- Moved tutorials to new `tutorials/` directory under project root.
- Tutorials are now built and published as an independent step in Bitbucket Pipelines, separate from the documentation build process.
- Deployment to PyPI and Docker Hub is now performed on tag push by Bitbucket Pipelines.

5.5 0.5.3 - 2018-10-15

First official release, corresponds to what was used in the final version of this paper.

5.6 Diffs

- 0.6.1
- 0.6.0
- 0.5.5
- 0.5.4
- 0.5.3

CHAPTER

SIX

LIB5C

6.1 lib5c package

6.1.1 Subpackages

lib5c.algorithms package

Subpackages

lib5c.algorithms.clustering package

Submodules

lib5c.algorithms.clustering.adjacency module

Module for assembling or merging clusters using a simple adjacency heuristic.

lib5c.algorithms.clustering.adjacency.**make_clusters**(*peaks*)
Clusters peaks by adjacency.

Parameters **peaks** (*list of peaks*) – The peaks to cluster.

Returns The clustered peaks.

Return type list of clusters

lib5c.algorithms.clustering.adjacency.**merge_to_which**(*clusters*)
Determines which other cluster, if any, the first cluster in a list of clusters should be merged into.

Parameters **clusters** (*list of clusters*) – The list of clusters to consider. Ideally, this list should be sorted in ascending order of cluster size.

Returns The index of the cluster that the first cluster should be merged into. If the cluster should not be merged, the value will be -1.

Return type int

Notes

Under the adjacency heuristic, the condition for merging two clusters is that they must contain peaks that are immediately adjacent to each other in 2-D space.

lib5c.algorithms.clustering.enclave module

Module for merging clusters using an enclave-swallowing heuristic.

lib5c.algorithms.clustering.enclave.**merge_to_which**(*clusters*)

Determines which other cluster, if any, the first cluster in a list of clusters should be merged into.

Parameters **clusters** (*list of clusters*) – The list of clusters to consider. Ideally, this list should be sorted in ascending order of cluster size.

Returns The index of the cluster that the first cluster should be merged into. If the cluster should not be merged, the value will be -1.

Return type int

Notes

Under the enclave heuristic, the condition for merging an orphan cluster into a parent cluster is that the orphan cluster's peaks must have more adjacent neighbors among the parent cluster's peaks than among the orphan cluster's peaks.

lib5c.algorithms.clustering.greedy module

Module for assembling clusters using a greedy heuristic.

lib5c.algorithms.clustering.greedy.**make_clusters**(*peaks*)

Merges peaks using a greedy merge criterion that grows clusters by iteratively assimilating all peaks within r + 2 units of the existing cluster's centroid where r is thre cluster's current radius.

Parameters **peaks** (*list of peaks*) – The peaks to be clustered.

Returns The clustered peaks.

Return type list of clusters

lib5c.algorithms.clustering.knn module

Module for assembling clusters using an unassisted k-nearest neighbors heuristic.

lib5c.algorithms.clustering.knn.**classify_peak**(*peak*, *neighbors*, *clusters*, *peak_to_clusters*, *automove=True*, *weighted=False*)

Assigns the most fitting cluster to a peak given a list of its neighbors.

Parameters

- **peak** (*peak*) – The peak to classify.
- **neighbors** (*list of peaks*) – The peaks that should determine the query peak's classification.
- **clusters** (*list of clusters*) – A list of clusters to classify the query peak into.
- **peak_to_clusters** (*dict of (int, int) tuples*) – The keys are (x, y) tuples that represent peak locations. The values are the indices of the cluster for that peak. If the value is -1, it indicates that the peak does not belong to any cluster.

- **automove** (*bool*) – If True, the peak will automatically be moved to the appropriate cluster, or a new cluster will be appended to clusters. If False, the index of the appropriate target cluster will be returned. If there is no appropriate target cluster, -1 will be returned.
- **weighted** (*bool*) – If True, weigh the votes using peak weight and distance. If False, treat the votes from each peak as equal.

Returns The index of the appropriate target cluster. This function has no return value unless automove is False.

Return type int (sometimes)

Notes

This function uses a simple unweighted voting heuristic to determine which existing cluster in clusters the query peak should be classified into. If no suitable existing cluster is found, a new cluster is created containing only the query peak. This cluster is then appended to clusters. Pass the kwarg weighted=True to use a weighted voting heuristic.

lib5c.algorithms.clustering.knn.**direction_score**(*peak*, *neighbors*)

Calculates a direction-score for a peak given its neighbors.

Parameters

- **peak** (*peak*) – The query peak.
- **neighbors** (*list of peaks*) – The query peak's neighbors.

Returns The direction-score.

Return type float

Notes

Higher direction-scores are better.

lib5c.algorithms.clustering.knn.**distance_score**(*neighbors*)

Calculates a distance-score for a peak given its neighbors.

Parameters **neighbors** (*list of peaks*) – The query peak's neighbors.

Returns The distance-score.

Return type float

Notes

Lower distance-scores are better.

lib5c.algorithms.clustering.knn.**get_knn**(*peak*, *peaks*, *k*)

Given a list of peaks and a query peak, returns the k nearest neighbors of the query peak.

Parameters

- **peak** (*peak*) – The query peak to for which nearest neighbors should be identified.
- **peaks** (*list of peaks*) – The peaks that are candidates to be nearest neighbors.
- **k** (*int*) –

Returns The k nearest neighbors of the query peak among peaks. If fewer than k peaks were provided, the length of this list will be shorter than k.

Return type list of peaks

Notes

peak may be present in peaks, but it will not be returned as a neighbor.

lib5c.algorithms.clustering.knn.**make_clusters**(*peaks*, *k=8*, *dist_score=5*, *dist_k=8*, *dir_score=0.3*, *dir_k=8*)

Performs k-nearest neighbors clustering of peaks.

Parameters

- **k** (*int*) – The number of nearest neighbors to consider when clustering.
- **dist_score** (*float*) – The distance-score threshold to use when clustering.
- **dist_k** (*int*) – The number of nearest neighbors to consider when calculating the distance score.
- **dir_score** (*float*) – The direction-score threshold to use when clustering.
- **dir_k** (*int*) – The number of nearest neighbors to consider when calculating the direction score.

Returns A tuple whose first element is the list of merged clusters, whose second element is the list of peaks that did not pass the distance score threshold, and whose third element is the list of peaks that did not pass the direction score threshold.

Return type list of clusters, list of peaks, list of peaks

lib5c.algorithms.clustering.quasicontiguity module

Module for splitting clusters using a quasicontiguity heuristic.

lib5c.algorithms.clustering.quasicontiguity.**split_cluster**(*cluster*, *distance_threshold=3*, *size_threshold=2*)

Identifies the subclusters of a cluster, as determined by quasicontiguity and a size threshold.

Parameters

- **cluster** (*cluster*) – The cluster to determine the subclusters of.
- **distance_threshold** (*float*) – If two peaks are separated by a distance less than this threshold, they are considered "quasicontiguous".
- **size_threshold** (*int*) – If the size of a subcluster would be smaller than this threshold, that subcluster is not split from its parent.

Returns The subclusters of the query cluster.

Return type list of clusters

lib5c.algorithms.clustering.quasicontiguity.**split_clusters**(*clusters*, *distance_threshold=3*, *size_threshold=2*)

Iteratively splits all clusters in a list by quasicontiguity (as determined by a distance and size threshold) and returns the resulting subclusters.

Parameters

- **clusters** (*list of clusters*) – The clusters to split.
- **distance_threshold** (*float*) – If two peaks are separated by a distance less than this threshold, they are considered "quasicontiguous".
- **size_threshold** (*int*) – If the size of a subcluster would be smaller than this threshold, that subcluster is not split from its parent.

Returns The list of split clusters.

Return type list of clusters

lib5c.algorithms.clustering.util module

Module containing utility functions for clustering 5C interactions.

lib5c.algorithms.clustering.util.**array_index_to_peaks**(*idx*)
Convert a dense boolean array to a sparse list of peaks.

Parameters **idx** (*np.ndarray*) – Boolean array to convert.

Returns The peaks.

Return type list of peaks

lib5c.algorithms.clustering.util.**belongs_to**(*peak*, *cluster*)
Checks if a peak belongs to a cluster.

Parameters

- **peak** (*peak*) – The query peak.
- **cluster** (*cluster*) – The cluster to search for it in.

Returns True if peak belongs to cluster, False otherwise.

Return type bool

lib5c.algorithms.clustering.util.**belongs_to_which**(*peak*, *clusters*)
Identifies which cluster out of a list of clusters, if any, a peak belongs to.

Parameters

- **peak** (*peak*) – The query peak to consider.
- **clusters** (*list of clusters*) – The clusters to look for the query peak in.

Returns The index of the cluster within the list of clusters which contains the query peak, or -1 if no cluster in the list of clusters contains the query peak.

Return type int

lib5c.algorithms.clustering.util.**center_of_mass**(*cluster*)
Computes the center of mass, or centroid, of a cluster.

Parameters **cluster** (*cluster*) – The cluster to consider.

Returns The centroid of the cluster.

Return type 1D numpy array of length 2

Notes

For the purpose of this calculation, the mass of a peak is taken to be its value.

lib5c.algorithms.clustering.util.**clusters_to_array**(*clusters*, *size*)
Assembles clusters into a 2-D array for plotting on a heatmap.

Parameters

- **clusters** (*list of clusters*) – The clusters to be converted to a 2-D array.
- **size** (*int*) – The height and width of the array to generate. This should be equal to the number of bins in the region.

Returns A 2-D array with each clusters having been assigned a different sequential integer value for all its pixels. The next consecutive integer is a gap value, and then the one after that is a default value for all pixels not in a cluster.

Return type 2-D array

Notes

It is recommended to plot the resulting array using a rapidly-changing colorscale such as plt.get_cmap('gist_ncar').

lib5c.algorithms.clustering.util.**flatten_clusters**(*clusters*)
Flattens a list of clusters to a flat list of peaks.

Parameters **clusters** (*list of list of peaks*) – The clusters to flatten.

Returns The flattened peaks.

Return type list of peaks

lib5c.algorithms.clustering.util.**get_cluster**(*x*, *y*, *clusters*)
Identifies which cluster, if any, a specified point belongs to.

Parameters

- **x** (*int*) – x-coordinate of the point to consider.
- **y** (*int*) – y-coordinate of the point to consider.
- **clusters** (*list of clusters*) – List of clusters to search.

Returns The index of the cluster which contains the point (x,y). If no cluster in the list contains the point, the value is -1.

Return type int

lib5c.algorithms.clustering.util.**get_vector**(*peak*)
Gets an array representing a peak's location.

Parameters **peak** (*peak*) –

Returns The peak's location as an (x, y) ordered pair.

Return type 1D numpy array of length 2

lib5c.algorithms.clustering.util.**ident**(*peak1*, *peak2*)
Checks whether two peaks are the same peak.

Parameters

- **peak1** (*peak*) –

- **peak2** (*peak*) –

Returns True if the peaks are the same peak, False otherwise.

Return type bool

lib5c.algorithms.clustering.util.**identify_nearby_clusters**(*cluster*, *clusters*)
Figures out which other clusters from a list of clusters are adjacent to a query cluster.

Parameters

- **cluster** (*cluster*) – The query cluster to consider.
- **clusters** (*list of clusters*) – The clusters to check for adjacency to the query cluster.

Returns The indices of clusters within the list of clusters that were found to be adjacent to the query cluster.

Return type list of int

Notes

This may include duplicates. To get rid of them, just use:

```
set(identify_nearby_clusters(cluster, clusters))
```

To identify one nearby cluster at random, use:

```
nearby_clusters = identify_nearby_clusters(cluster, clusters)
if nearby_clusters:
    nearby_clusters[0]
```

To see what clusters are near a single peak, use:

```
identify_nearby_clusters([peak], clusters)
```

If cluster is in clusters, the cluster will be reported as adjacent to itself. As an example of how to avoid this in cases where it is undesirable, use:

```
filter(lambda x: x > 0, identify_nearby_clusters(clusters[0], clusters))
```

lib5c.algorithms.clustering.util.**merge_clusters**(*clusters*, *merge_to_which*)
Recursively merges clusters together from smallest to largest according to a specified merge function.

Parameters

- **clusters** (*list of clusters*) – The clusters to be merged. All elements will be removed from this list when this function is called.
- **merge_to_which** (*function*) – Function that takes in a list of clusters and returns the index of the cluster the first cluster in the list should be merged into. If the first cluster in the list should not be merged, this function should return -1.

Returns The list of merged clusters.

Return type list of clusters

lib5c.algorithms.clustering.util.**peaks_to_array_index**(*peaks*, *shape*)
Convert a sparse list of peaks to a dense boolean array.

Parameters

- **peaks** (*list of peaks*) – The peaks to convert.
- **shape** (*tuple of int*) – The shape of the resulting array.

Returns The dense boolean array.

Return type np.ndarray

lib5c.algorithms.clustering.util.**reshape_cluster_array_to_dict**(*cluster_array*, *ignored_values=None*)

Reshapes loops dict structure into a nested dict structure.

Parameters

- **cluster_array** (*np.ndarray*) – The entries of this array are cluster ID's. Values that will be ignored include '', 'n.s.', 'NA', 'NaN', np.nan.
- **ignored_values** (*set, optional*) – Set of values in cluster_array that should not be treated as cluster ID's. By default this will be {'', 'n.s.', 'NA', 'NaN', np.nan}

Returns

The outer dict's keys are cluster ID's, its values are lists of points belonging to that cluster, with the points being provided as dicts with the following strucure:

```
{
    'x': int,
    'y': int,
    'value': 0
}
```

Return type Dict[Any, List[Dict[str, Any]]]

Notes

To rectangularize the returned data structure against a full list of cluster ID's, use something like:

```
cluster_dict = reshape_cluster_array_to_dict(cluster_array)
for cluster_id in all_cluster_ids:
    if cluster_id not in cluster_dict:
        cluster_dict[cluster_id] = []
```

Examples

```
>>> import numpy as np
>>> cluster_array = np.array([['', 'cow'], ['cow', 'grass']])
>>> reshape_cluster_array_to_dict(cluster_array) == \
...     {'grass': [{'x': 1, 'y': 1, 'value': 0}],
...      'cow': [{'x': 0, 'y': 1, 'value': 0},
...              {'x': 1, 'y': 0, 'value': 0}]}
True
```

lib5c.algorithms.clustering.valley module

Module for splitting clusters using a valley heuristic.

lib5c.algorithms.clustering.valley.**split_cluster**(*parent_cluster*, *guides*, *reassign=1.0*)

Splits a cluster using the valley heuristic.

Parameters

- **parent_cluster** (*cluster*) – The cluster to split.
- **guides** (*list of clusters*) – The guides used to seed the child clusters. See lib5c.algorithms.clustering.valley.split_clusters().
- **reassign** (*float between 0.0 and 1.0*) – When a cluster is split, the peaks in the parent cluster with pvalues above this threshold may be discarded instead of being assigned to one or the other of the child clusters. When the value of this kwarg is 1.0, no peaks are discarded and all peaks are reassigned to one or the other of the child clusters. When the value of this kwarg is 0.0, all peaks are discarded and no peaks are reassigned to the child clusters.

Returns The child clusters of the parent cluster.

Return type list of clusters

Notes

See lib5c.algorithms.clustering.valley.split_clusters().

lib5c.algorithms.clustering.valley.**split_clusters**(*clusters*, *reassign=1.0*, *size_threshold=3*)

Splits all clusters in a list of clusters using a recursive valley-splitting heuristic.

Parameters

- **clusters** (*list of clusters*) – The list of clusters to split.
- **reassign** (*float between 0.0 and 1.0*) – When a cluster is split, the peaks in the parent cluster with pvalues above this threshold may be discarded instead of being assigned to one or the other of the child clusters. When the value of this kwarg is 1.0, no peaks are discarded and all peaks are reassigned to one or the other of the child clusters. When the value of this kwarg is 0.0, all peaks are discarded and no peaks are reassigned to the child clusters.
- **size_threshold** (*int*) – The minimum size of child clusters that will be created by the splitting. If a splitting operation would result in clusters smaller than this number, that splitting operation will not be performed.

Returns The split clusters.

Return type list of clusters

Notes

A cluster will get split if there exists a p-value such that thresholding the cluster at that p-value results in the creation of at least two contiguous groups of high-confidence peaks (p-value below the threshold) larger than the size_threshold separated by a "valley" of low-confidence peaks (p-value above the threshold). This rule is applied recursively until only unsplittable, or "atomic", clusters remain.

Module contents

Subpackage for clustering 5C interactions.

lib5c.algorithms.filtering package

Submodules

lib5c.algorithms.filtering.bin_bin_filtering module

Module for smoothing bin-level 5C interaction matrices.

lib5c.algorithms.filtering.bin_bin_filtering.**bin_bin_filter**(*array*, *filter_function*, *regional_pixelmap*, *threshold*, *filter_kwargs=None*)

Convenience function for filtering a bin-level matrix to a bin-level matrix.

Parameters

- **array** (*np.ndarray*) – The matrix to filter.
- **filter_function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use when filtering. This function should take in a "neighborhood" and return the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

  where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs. See `lib5c.algorithms.filtering.filter_functions` for examples of filter functions and how they can be created.
- **regional_pixelmap** (*List[Dict[str, Any]]*) – The list of bins in this region.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.
- **filter_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the `filter_function`.

Returns The filtered matrix.

Return type np.ndarray

lib5c.algorithms.filtering.bin_bin_filtering.**bin_bin_filter_counts**(*counts*, *function*, *pixelmap*, *threshold*, *function_kwargs=None*)

Non-parallel wrapper for `bin_bin_filter()`. Deprecated now that `bin_bin_filter()` is decorated with `@parallelize_regions`.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The counts dict to filter.

- **function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use for filtering. See the description of the `filter_function` arg in `bin_bin_filter()`.
- **pixelmap** (*Dict[str, List[Dict[str, Any]]]*) – The pixelmap describing the bins for `counts`.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.
- **function_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the `function`.

Returns The dict of filtered counts.

Return type Dict[str, np.ndarray]

lib5c.algorithms.filtering.bin_bin_filtering.**find_nearby_bins**(*index*, *regional_pixelmap*, *threshold*)

Finds the bins near a target bin as specified by an index.

Parameters

- **index** (*int*) – The index of the bin to look near.
- **regional_pixelmap** (*List[Dict[str, Any]]*) – The list of bins in this region.
- **threshold** (*int*) – The threshold for deciding if a bin is "nearby" or not, as a distance in base pairs.

Returns

A list of nearby bins, where each nearby bin is represented as a dict of the following form:

```
{
    'index': int,
    'distance': int
}
```

where 'index' is the index of the bin within the region and 'distance' is the distance between this bin and the target bin.

Return type List[Dict[str, int]]

lib5c.algorithms.filtering.filter_functions module

Module providing utilities for defining and constructing filter functions.

lib5c.algorithms.filtering.filter_functions.**amean_gaussian**(*sigma=1000.0*, *norm_ord=1*, *check_threshold=0.2*)

Constructs a filter function that uses the arithmetic mean with Gaussian weights as the aggregating function and a p-norm as the norm function.

Parameters

- **sigma** (*float*) – The standard deviation to use for the Gaussian when assigning weights.
- **norm_ord** (*int*) – The order of the p-norm to use to convert (x-dist, y-dist) vectors to scalar distances.

- **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**amean_inverse**(*bin_width=4000*, *norm_ord=1*, *check_threshold=0.2*)

Constructs a filter function that uses the arithmetic mean with "inverse" weights as the aggregating function and a p-norm as the norm function.

Parameters

- **bin_width** (*int*) – The bin width in base pairs.
- **norm_ord** (*int*) – The order of the p-norm to use to convert (x-dist, y-dist) vectors to scalar distances.
- **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**arithmetic_mean**(*check_threshold=0.2*)

Constructs a filter function that uses the unweighted arithmetic mean as the aggregating function.

Parameters **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**check_neighboorhood_nonnan**(*neighborhood*, *threshold*)

Check to see if a neighborhood clears as specified non-nan fraction threshold.

Parameters

- **neighborhood** (*List[Dict[str, Any]]*) – A list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

- **threshold** (*float*) – If less than this fraction of the values in the neighborhood are non-infinite, the neighborhood fails the check.

Returns True if this neighborhood clears the threshold, otherwise False.

Return type bool

lib5c.algorithms.filtering.filter_functions.**check_neighboorhood_positive**(*neighborhood*, *threshold*)

Check to see if a neighborhood clears as specified positive fraction threshold.

Parameters

- **neighborhood** (*List[Dict[str, Any]]*) – A list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

- **threshold** (*float*) – If less than this fraction of the values in the neighborhood are positive, the neighborhood fails the check.

Returns True if this neighborhood clears the threshold, otherwise False.

Return type bool

lib5c.algorithms.filtering.filter_functions.**geometric_mean**(*check_threshold=0.2*)

Constructs a filter function that uses the unweighted geometric mean as the aggregating function.

Parameters **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**gmean_gaussian**(*sigma=1000.0*, *norm_ord=1*, *check_threshold=0.2*)

Constructs a filter function that uses the geometric mean with Gaussian weights as the aggregating function and a p-norm as the norm function.

Parameters

- **sigma** (*float*) – The standard deviation to use for the Gaussian when assigning weights.
- **norm_ord** (*int*) – The order of the p-norm to use to convert (x-dist, y-dist) vectors to scalar distances.
- **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**gmean_inverse**(*bin_width=4000*, *norm_ord=1*, *check_threshold=0.2*)

Constructs a filter function that uses the geometric mean with "inverse" weights as the aggregating function and a p-norm as the norm function.

Parameters

- **bin_width** (*int*) – The bin width in base pairs.
- **norm_ord** (*int*) – The order of the p-norm to use to convert (x-dist, y-dist) vectors to scalar distances.
- **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**inverse_weighting_function**(*distance*, *bin_width=None*)

The "inverse" weighting function used in Yaffe and Tanay 2011.

Parameters

- **distance** (*float*) – The distance to compute a weight for, in base pairs.
- **bin_width** (*Optional[int]*) – The bin width in base pairs. Used to make results equivalent to Yaffe and Tanay 2011 by scaling `distance` to units of bins. Pass None to simply leave the distance in units of base pairs

Returns A weight appropriate for this distance.

Return type float

lib5c.algorithms.filtering.filter_functions.**make_filter_function**(*function='gmean'*, *threshold=0.0*, *norm_order=1*, *bin_width=4000*, *sigma=12000.0*, *inverse=False*, *gaussian=False*)

Convenience function for quickly constructing filtering functions with desired properties.

Parameters

- **function** (*{'sum', 'median', 'amean', 'gmean'}*) – The aggregation function to use. This is the operation that will be applied to all points in the neighborhood, after weighting their values if appropriate.

- **threshold** (*float*) – If less than this fraction of the values in a neighborhood are non-infinite, the filter function will return nan for that neighborhood.
- **norm_order** (*int*) – The order of p-norm to use when computing distances.
- **bin_width** (*int*) – The width of each bin in base pairs. This value is used to scale certain weights.
- **sigma** (*float*) – The value to use for the standard deviation of the Gaussian when using Gaussian weights.
- **inverse** (*bool*) – Pass True to use "inverse" weights as in Yaffe and Tanay 2011.
- **gaussian** (*bool*) – Pass True to use Gaussian weights with standard deviation `sigma`.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**median**(*check_threshold=0.2*)

Constructs a filter function that uses the median as the aggregating function.

Parameters **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**norm_filter_function**(*weighted_function*, *norm_function*, *weighted_kwargs=None*, *norm_kwargs=None*, *pseudocount=0*, *check_function=None*, *check_threshold=None*)

Constructs a filter function that passes the value and some distance norm (as specified by `norm_function`) for each point in the neighborhood to a special aggregation function capable of performing weighted aggregation based on these distances.

Parameters

- **weighted_function** (*Callable[[List[Dict[str, float]], float]]*) – A special aggregation function that takes in a list of points represented as dicts with the following structure:

```
{
    'value': float,
    'dist': float
}
```

  where 'value' is the interaction value at that point and 'dist' is its scalar distance from the neighborhood. This function should then return a float representing the aggregate value of the neighborhood, weighted using the distances.

- **norm_function** (*Callable[[Tuple[int]], float]*) – A function that takes in a tuple of ints representing the x- and y-axis distances of a point to the neighborhood and returns a scalar value representing the distance.
- **weighted_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to `weighted_function`.
- **norm_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to `norm_function`.
- **pseudocount** (*float*) – A pseudocount to be added to the values before applying the aggregation function. Useful if the aggregation function has catastrophic behavior when one input value is zero.
- **check_function** (*Optional[Callable[[List[Dict[str, Any]], float], bool]]*) – A function that takes in a neighborhood and a threshold value and performs some sort of test on the neighborhood, returning False if the filter function should return NaN for the neighborhood because it fails some critical condition.
- **check_threshold** (*float*) – The threshold to pass as the second arg to `check_function`.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

`lib5c.algorithms.filtering.filter_functions.`**simple_sum**(*check_threshold=0.2*)

Constructs a filter function that uses a simple sum as the aggregating function.

Parameters **check_threshold** (*float*) – If less than this fraction of the values in a neighborhood are positive, the filter function will return NaN.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**value_filter_function**(*function*, *function_kwargs=None*, *pseudocount=0*, *check_function=None*, *check_threshold=None*)

Constructs a filter function that passes the values in the neighborhood to an aggregation function.

Parameters

- **function** (*Callable[Sequence[float], float]*) – The aggregation function to use on the values in each neighborhood.
- **function_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to `function`.
- **pseudocount** (*float*) – A pseudocount to be added to the values before applying the aggregation function. Useful if the aggregation function has catastrophic behavior when one input value is zero.
- **check_function** (*Optional[Callable[[List[Dict[str, Any]], float], bool]]*) – A function that takes in a neighborhood and a threshold value and performs some sort of test on the neighborhood, returning False if the filter function should return NaN for the neighborhood because it fails some critical condition.
- **check_threshold** (*float*) – The threshold to pass as the second arg to `check_function`.

Returns

The constructed filter function. This function takes in a "neighborhood" and returns the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs.

Return type Callable[[List[Dict[str, Any]]], float]

lib5c.algorithms.filtering.filter_functions.**weighted_amean**(*values*, *weights*)
Weighted version of the arithmetic mean.

Parameters

- **values** (*Sequence[float]*) – The values to aggregate.
- **weights** (*Sequence[float]*) – The weights for each value.

Returns The weighted arithmetic mean of the values given the weights.

Return type float

lib5c.algorithms.filtering.filter_functions.**weighted_gmean**(*values*, *weights*)
Weighted version of the geometric mean.

Parameters

- **values** (*Sequence[float]*) – The values to aggregate.
- **weights** (*Sequence[float]*) – The weights for each value.

Returns The weighted geometric mean of the values given the weights.

Return type float

lib5c.algorithms.filtering.filter_functions.**weighted_values_distances_function**(*weighting_functi*
ag-
gre-
gat-
ing_function,
weight-
ing_kwargs=Nor
ag-
gre-
gat-
ing_kwargs=Nor
cache=True)
Constructs a weighted aggregation function appropriate for use with `norm_filter_function()`.

Parameters

- **weighting_function** (*Callable[[float], float]*) – A function that takes in a distance and returns a weight.
- **aggregating_function** (*Callable[[Sequence[float], Sequence[float]], float]*) – A special aggregating function that takes in the values and the weights as parallel vectors and returns the aggregated value.
- **weighting_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to `weighting_function`.
- **aggregating_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to `aggregating_function`.
- **cache** (*bool*) – Pass True to make the returned function use a cache to avoid recomputing expensive weighting function calls.

Returns

A special aggregation function that takes in a list of points represented as dicts with the following structure:

```
{
    'value': float,
    'dist': float
}
```

where 'value' is the interaction value at that point and 'dist' is its scalar distance from the neighborhood. This function returns a float representing the aggregate value of the neighborhood, weighted using the distances.

Return type Callable[[List[Dict[str, float]], float]

lib5c.algorithms.filtering.fragment_bin_filtering module

lib5c.algorithms.filtering.fragment_bin_filtering.**find_nearby_fragments**(*index*, *regional_pixelmap*, *regional_primermap*, *upstream_primer_mapping*, *threshold*, *midpoint=False*)

Finds the primers near a target bin as specified by an index.

Parameters

- **index** (*int*) – The index of the bin to look near.
- **regional_pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap describing the primers for this region.
- **upstream_primer_mapping** (*Dict[int, int]*) – A mapping from each bin index to the index of its nearest upstream primer. See `lib5c.algorithms.filtering.fragment_bin_filtering .find_upstream_primers()`.
- **threshold** (*int*) – The threshold for deciding if a fragment is "nearby" or not, as a distance in base pairs.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns

A list of nearby fragments, where each nearby fragment is represented as a dict of the following form:

```
{
    'index': int,
    'distance': int
}
```

where 'index' is the index of the fragment within the region and 'distance' is the distance between this fragment and the target bin.

Return type List[Dict[str, int]]

lib5c.algorithms.filtering.fragment_bin_filtering.**find_upstream_primers**(*regional_pixelmap*, *regional_primermap*)

Creates a mapping from a bin index to the index of its nearest upstream primer.

Parameters

- **regional_pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap describing the primers for this region.

Returns A map from each bin index to the index of its nearest upstream primer.

Return type Dict[int, int]

lib5c.algorithms.filtering.fragment_bin_filtering.**fragment_bin_filter**(*array*, *filter_function*, *regional_pixelmap*, *regional_primermap*, *threshold*, *filter_kwargs=None*, *midpoint=False*)

Convenience function for filtering a fragment-level matrix to a bin-level matrix.

Parameters

- **array** (*np.ndarray*) – The matrix to filter.
- **filter_function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use when filtering. This function should take in a "neighborhood" and return the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs. See lib5c.

algorithms.filtering.filter_functions for examples of filter functions and how they can be created.

- **regional_pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap describing the primers for this region.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.
- **filter_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the filter_function.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns The filtered matrix.

Return type np.ndarray

lib5c.algorithms.filtering.fragment_bin_filtering.**fragment_bin_filter_counts**(*counts*, *function*, *pixelmap*, *primermap*, *threshold*, *function_kwargs=None*, *midpoint=False*)

Non-parallel wrapper for fragment_bin_filter(). Deprecated now that fragment_bin_filter() is decorated with @parallelize_regions.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The counts dict to filter.
- **function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use for filtering. See the description of the filter_function arg in fragment_bin_filter().
- **pixelmap** (*Dict[str, List[Dict[str, Any]]]*) – The pixelmap describing the bins.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap describing the fragments.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.
- **function_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the function.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns The dict of filtered counts.

Return type Dict[str, np.ndarray]

lib5c.algorithms.filtering.fragment_fragment_filtering module

Module for smoothing fragment-level 5C interaction matrices.

lib5c.algorithms.filtering.fragment_fragment_filtering.**find_nearby_fragments**(*index*, *regional_primermap*, *threshold*, *midpoint=False*)

Finds the fragments near a target fragment as specified by an index.

Parameters

- **index** (*int*) – The index of the bin to look near.
- **regional_primermap** (*List[Dict[str, Any]]*) – The list of fragments in this region.
- **threshold** (*int*) – The threshold for deciding if a fragment is "nearby" or not, as a distance in base pairs.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns

A list of nearby fragments, where each nearby bin is represented as a dict of the following form:

```
{
    'index': int,
    'distance': int
}
```

where 'index' is the index of the fragment within the region and 'distance' is the distance between this fragment and the target fragment.

Return type List[Dict[str, int]]

lib5c.algorithms.filtering.fragment_fragment_filtering.**fragment_fragment_filter**(*array*, *filter_function*, *regional_primern*, *threshold*, *filter_kwargs=No*, *midpoint=False*)

Convenience function for filtering a fragment-level matrix to a fragment-level matrix.

Parameters

- **array** (*np.ndarray*) – The matrix to filter.

- **filter_function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use when filtering. This function should take in a "neighborhood" and return the filtered value given that neighborhood. A neighborhood is represented as a list of "nearby points" where each nearby point is represented as a dict of the following form:

```
{
    'value': float,
    'x_dist': int,
    'y_dist': int
}
```

  where 'value' is the value at the point and 'x_dist' and 'y_dist' are its distances from the center of the neighborhood along the x- and y-axis, respectively, in base pairs. See `lib5c.algorithms.filtering.filter_functions` for examples of filter functions and how they can be created.

- **regional_primermap** (*List[Dict[str, Any]]*) – The list of fragments in this region.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.
- **filter_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the `filter_function`.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns The filtered matrix.

Return type np.ndarray

lib5c.algorithms.filtering.fragment_fragment_filtering.**fragment_fragment_filter_counts**(*counts*, *function*, *primermap*, *threshold*, *function_kwargs*, *midpoint*)

Non-parallel wrapper for `fragment_fragment_filter()`. Deprecated now that `fragment_fragment_filter()` is decorated with `@parallelize_regions`.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The counts dict to filter.
- **function** (*Callable[[List[Dict[str, Any]]], float]*) – The filter function to use for filtering. See the description of the `filter_function` arg in `fragment_fragment_filter()`.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap describing the fragments for `counts`.
- **threshold** (*int*) – The threshold for defining the size of the neighborhood passed to the filter function, in base pairs.

- **function_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to be passed to the `function`.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns The dict of filtered counts.

Return type Dict[str, np.ndarray]

lib5c.algorithms.filtering.unsmoothable_columns module

Module for identifying "unsmoothable columns" - sets of bins that don't contain any non-zero fragments and are too wide to smooth over.

lib5c.algorithms.filtering.unsmoothable_columns.**find_prebinned_unsmoothable_columns**(*regional_re-gional_pi win-dow_widt*

Identifies the unsmoothable columns in a region assuming that the smoothing was a filtering operation applied on data that was already bin-level.

Parameters

- **regional_counts** (*np.ndarray*) – The matrix of counts for this region.
- **regional_pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **window_width** (*int*) – The width of the filtering window in base pairs.

Returns A list of boolean values with length equal to the number of bins in the region. The `i` th element of this list is True if the `i` th bin in the region is an "unsmoothable column".

Return type List[bool]

lib5c.algorithms.filtering.unsmoothable_columns.**find_unsmoothable_columns**(*regional_primermap*, *re-gional_pixelmap*, *win-dow_width*, *up-stream_primer_mapping mid-point=False*)

Identifies the unsmoothable columns in a region assuming that the smoothing was a filtering operation applied on fragment-level data.

Parameters

- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap describing the primers for this region.
- **regional_pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **window_width** (*int*) – The width of the filtering window in base pairs.

- **upstream_primer_mapping** (*Dict[int, int]*) – A mapping from each bin index to the index of its nearest upstream primer. See `lib5c.algorithms.filtering.fragment_bin_filtering .find_upstream_primers()`.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns A list of boolean values with length equal to the number of bins in the region. The `i` th element of this list is True if the `i` th bin in the region is an "unsmoothable column".

Return type List[bool]

lib5c.algorithms.filtering.unsmoothable_columns.**unsmoothable_column_threshold_heuristic**(*win bin*

This function defines the heuristic that determines how long a run of fragment-less bins must be before it is considered "unsmoothable".

Parameters

- **window_width** (*int*) – The width of the filtering window in base pairs.
- **bin_step** (*int*) – The "sampling rate" or "bin step".

Returns The maximum length of a run of fragment-less bins must be before it is considered "unsmoothable".

Return type int

lib5c.algorithms.filtering.unsmoothable_columns.**wipe_prebinned_unsmoothable_columns**(*smoothed pre- binned_c pix- elmap, win- dow_widt*

Convenience function for wiping the unsmoothable columns in a binned counts matrix assuming that the smoothing was a filtering operation applied on bin-level data.

Parameters

- **smoothed_counts** (*np.ndarray*) – The matrix of smoothed counts to wipe unsmoothable columns from.
- **prebinned_counts** (*np.ndarray*) – The original binned counts matrix to use to identify zero-count columns.
- **pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **window_width** (*int*) – The width of the filtering window in base pairs.

Returns The wiped matrix of binned counts.

Return type np.ndarray

lib5c.algorithms.filtering.unsmoothable_columns.**wipe_unsmoothable_columns**(*binned_counts, primermap, pix- elmap, win- dow_width, mid- point=False*)

Convenience function for wiping the unsmoothable columns in a binned counts matrix assuming that the smoothing was a filtering operation applied on fragment-level data.

Parameters

- **binned_counts** (*np.ndarray*) – The matrix of binned counts to wipe unsmoothable columns from.
- **primermap** (*List[Dict[str, Any]]*) – The primermap describing the primers for this region.
- **pixelmap** (*List[Dict[str, Any]]*) – The pixelmap describing the bins for this region.
- **window_width** (*int*) – The width of the filtering window in base pairs.
- **midpoint** (*bool*) – Pass True to restore legacy behavior when distances to fragments were based on their midpoints. The new behavior (with this kwarg set to False) is to compute distances to fragments based on their closest endpoint.

Returns The wiped matrix of binned counts.

Return type np.ndarray

lib5c.algorithms.filtering.util module

Module containing utility functions for filtering 5C interaction matrices.

lib5c.algorithms.filtering.util.**filter_selector**(*array*, *nearby_x*, *nearby_y*)

Create a list of dicts that describes the "neighborhood" around a point given an array of values and lists of the nearby entities along both the x- and y-axes.

Parameters

- **array** (*np.ndarray*) – The array of values at each point in the region.
- **nearby_y** (*nearby_x,*) – A list of nearby entities (bins or fragments) along the x- or y-axis, respectively, represented as dicts of the form:

```
{
    'index': int,
    'distance': int
}
```

where 'index' is the index of the entity within the region, and 'distance' is the distance from this entity to the query point in base pairs. A list of this form can be created by functions like `find_nearby_bins()` or `find_nearby_fragments()`.

Module contents

Subpackage for "smoothing", "binning", or "filtering" 5C contact frequencies.

lib5c.algorithms.variance package

Submodules

lib5c.algorithms.variance.combined module

lib5c.algorithms.variance.combined.**cross_rep_plus_deviation_variance**(*obs*, *exp*, *rep*, *model='lognorm'*, *min_disp=1e-08*)

Estimates pixel-wise variance as the squared deviation between observed and expected values.

Parameters

- **obs** (*dict or list of np.ndarray*) – Dict values or list entries are are square, symmetric count matrices across replicates.
- **exp** (*np.ndarray*) – Square, symmetric matrix of expected values.
- **rep** (*int or str*) – The index into `obs` identifying which replicate to compute variance estimates for.
- **model** (*{'lognorm', 'norm'}*) – Statistical model to use.
- **min_disp** (*float*) – Force a minimum value of the dispersion parameter.

Returns The first three elements are the mean parameter estimate, dispersion estimate, and variance estimate, respectively, for each pixel. The fourth element is a boolean matrix showing which pixels are considered to be overdispersed.

Return type tuple of np.ndarray

lib5c.algorithms.variance.cross_rep module

lib5c.algorithms.variance.cross_rep.**cross_rep_variance**(*obs*, *model='lognorm'*, *min_disp=1e-08*, *method='mme'*)

Estimates pixel-wise variance across replicates.

Parameters

- **obs** (*dict or list of np.ndarray*) – Dict values or list entries are are square, symmetric count matrices across replicates.
- **model** (*{'lognorm', 'nbinom', 'norm'}*) – Statistical model to use.
- **min_disp** (*float*) – Force a minimum value of the dispersion parameter.
- **method** (*{'mme', 'mle'}*) – When model='nbinom', pass 'mle' to run maximum likelihood estimation for each pixel independently. Pass 'mme' to use method-of-moments variance estimation. Has no effect if model='lognorm'.

Returns The first three elements are the mean parameter estimate, dispersion estimate, and variance estimate, respectively, for each pixel. The fourth element is a boolean matrix showing which pixels are considered to be overdispersed.

Return type tuple of np.ndarray

lib5c.algorithms.variance.deviation module

lib5c.algorithms.variance.deviation.**deviation_variance**(*obs*, *exp*, *model='lognorm'*, *min_disp=1e-08*)

Estimates pixel-wise variance as the squared deviation between observed and expected values.

Parameters

- **exp** (*obs,*) – Square, symmetric matrix of observed and expected values, respectively.
- **model** (*{'lognorm', 'nbinom', 'norm'}*) – Statistical model to use.
- **min_disp** (*float*) – Force a minimum value of the dispersion parameter.

Returns The first three elements are the mean parameter estimate, dispersion estimate, and variance estimate, respectively, for each pixel. The fourth element is a boolean matrix showing which pixels are considered to be overdispersed.

Return type tuple of np.ndarray

lib5c.algorithms.variance.estimate_variance module

lib5c.algorithms.variance.estimate_variance.**estimate_variance**(*obs_counts*, *exp_counts*, *key_rep=None*, *model='lognorm'*, *source='deviation'*, *source_kwargs=None*, *fitter='lowess'*, *fitter_agg='lowess'*, *fitter_kwargs=None*, *x_unit='dist'*, *y_unit='disp'*, *logx=False*, *logy=False*, *min_disp=1e-08*, *min_obs=2*, *min_dist=6*, *regional=False*)

Convenience function for computing variance estimates.

Parameters

- **obs_counts** (*dict of np.ndarray or dict of dict of np.ndarray*) – Counts dict of observed values (keys are region names, values are square symmetric matrices), or superdict (outer keys are replicate names, inner keys are region names, values are square symmetric matrices) if `source='cross_rep'`.
- **exp_counts** (*dict of np.ndarray*) – Counts dict of expected values.
- **key_rep** (*str*) – If `obs_counts` is a dict of dict of np.ndarray, pass a string naming the specific replicate to compute variance estimates for.
- **model** (*{'lognorm', 'loglogistic', 'nbinom', 'poisson'}*) – Statistical model to use.

- **source** (*{'local', 'cross_rep', 'deviation', 'mle'}*) – Specify the source of the variance estimates.
- **source_kwargs** (*dict*) – Kwargs to pass through to the variance source function.
- **fitter** (*{'constant', 'group', 'lowess', 'none'}*) – Select fitting method to use for trend fitting. Pass 'none' to skip trend fitting and simply return unfiltered point-wise estimates.
- **fitter_agg** (*{'median', 'mean', 'lowess'}*) – If `fitter` is 'group' or 'constant', select what function to use to aggregate values (within groups for group fitting or across the whole dataset for constant fitting).
- **fitter_kwargs** (*dict*) – Kwargs to pass through to the fitting function.
- **x_unit** (*{'dist', 'exp'}*) – The x-unit to fit the variance relationship against.
- **y_unit** (*{'disp', 'var'}*) – The y-unit to fit the variance relationship against. When `model='nbinom'`, "disp" refers to the negative binomial dispersion parameter. When `model='lognorm'`, "disp" refers to the variance parameter of the normal distribution describing the logarithm of the observed counts.
- **logy** (*logx,*) – Pass True to fit the variance relationship on the scale of `log(x)` and/or `log(y)`.
- **min_disp** (*float*) – When `model='nbinom'`, this sets the minimum value of the negative binomial dispersion parameter. When `model='lognormal'`, this sets the minimum value of the variance of logged observed counts.
- **min_obs** (*float*) – Points with observed values below this threshold in any replicate will be excluded from MLE estimation and relationship fitting.
- **min_dist** (*int*) – Points with interaction distances (in bin units) below this threshold will be excluded from MLE estimation and relationship fitting.
- **regional** (*bool*) – Pass True to perform MLE estimation and relationship fitting on a per-region basis.

Returns The variance estimates as a counts dict.

Return type dict of np.ndarray

lib5c.algorithms.variance.local module

lib5c.algorithms.variance.local.**local_variance**(*matrix*, *model='lognorm'*, *w=1*, *min_finite=3*, *min_disp=1e-08*)

Estimates pixel-wise variance by treating nearby matrix entries as replicates.

Parameters

- **matrix** (*np.ndarray*) – Square, symmetric matrix of count values.
- **model** (*{'lognorm', 'nbinom'}*) – Statistical model to use.
- **w** (*int or np.ndarray*) – Size of footprint to use. Footprint will be `np.eye(2*w+1)`. To use a different footprint, pass it as an `np.ndarray`.
- **min_finite** (*int*) – Points with fewer than this many finite entries inside their footprint will have their variance estimate set to nan.
- **min_disp** (*float*) – Force a minimum value of the dispersion parameter.

Returns The first three elements are the mean parameter estimate, dispersion estimate, and variance estimate, respectively, for each pixel. The fourth element is a boolean matrix showing which pixels are considered to be overdispersed.

Return type tuple of np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.variance.local import local_variance
>>> local_variance(np.array([[1, 4, 1],
...                          [4, 1, 1],
...                          [1, 1, 1]]), model='norm', min_finite=2)
(array([[1. , 2.5, 1. ],
        [2.5, 1. , 2.5],
        [1. , 2.5, 1. ]]),
 array([[1.0e-08, 4.5e+00,     nan],
        [4.5e+00, 1.0e-08, 4.5e+00],
        [    nan, 4.5e+00, 1.0e-08]]),
 array([[1.0e-08, 4.5e+00,     nan],
        [4.5e+00, 1.0e-08, 4.5e+00],
        [    nan, 4.5e+00, 1.0e-08]]),
 array([[False,  True, False],
        [ True, False,  True],
        [False,  True, False]]))
```

lib5c.algorithms.variance.lognorm_dispersion module

Module for estimating lognormal dispersion parameters for 5C interaction data.

lib5c.algorithms.variance.lognorm_dispersion.**dispersion_to_variance**(*disp*, *exp*)

Converts a dispersion estimate to a variance by applying it to the expected value of unlogged counts.

Parameters

- **disp** (*float or np.ndarray*) – The dispersion (variance of logged values).
- **exp** (*float or np.ndarray*) – The expected value (of unlogged values).

Returns The variance.

Return type float or np.ndarray

lib5c.algorithms.variance.lognorm_dispersion.**dispersion_to_variance_direct**(*disp*, *mu*)

Converts a dispersion estimate to a variance by applying it to the expected value of logged counts.

Parameters

- **disp** (*float or np.ndarray*) – The dispersion (variance of logged values).
- **mu** (*float or np.ndarray*) – The expected value (of logged values).

Returns The variance.

Return type float or np.ndarray

lib5c.algorithms.variance.lognorm_dispersion.**variance_to_dispersion**(*var*, *exp*)

Converts a variance estimate to a dispersion by applying it to the expected value of unlogged counts.

Parameters

- **var** (*float or np.ndarray*) – The variance (of unlogged values).
- **exp** (*float or np.ndarray*) – The expected value (of unlogged values).

Returns The dispersion (variance of logged values).

Return type float or np.ndarray

lib5c.algorithms.variance.mle module

lib5c.algorithms.variance.mle.**mle_variance**(*obs*, *exp*, *model='lognorm'*, *min_obs=2*, *min_dist=6*, *regional=False*)

Fits a single point estimate of the dispersion across each or all regions under the selected model, and returns the converted variance estimates.

Parameters

- **exp** (*obs,*) – The counts dicts of observed and expected data. Keys are region names, values are square, symmetric count matrices.
- **model** (*{'lognorm', 'loglogistic', 'nbinom'}*) – The statistical model to use for MLE point estimation.
- **min_obs** (*float*) – Fit only points with at least this many observed counts.
- **min_dist** (*int*) – Fit only points with at least this interaction distance in bin units.
- **regional** (*bool*) – Pass True to fit a separate point estimate for each region.

Returns The variance estimates.

Return type dict of np.ndarray

lib5c.algorithms.variance.nbinom_dispersion module

Module for estimating negative binomial dispersion parameters for 5C interaction data.

lib5c.algorithms.variance.nbinom_dispersion.**dispersion_to_variance**(*disp*, *exp*)

Converts a dispersion estimate to a variance by applying it to the expected value via the relationship `var = exp + disp * exp**2`.

Parameters

- **disp** (*float or np.ndarray*) – The dispersion.
- **exp** (*float or np.ndarray*) – The expected value.

Returns The variance.

Return type float or np.ndarray

lib5c.algorithms.variance.nbinom_dispersion.**nb_nll**(*disp*, *obs*, *exp*)

The negative log likelihood of observed data `obs` given mean/expected value `exp` and dispersion parameter `disp`.

Parameters

- **disp** (*float or np.ndarray*) – The dispersion parameter.
- **obs** (*int or np.ndarray*) – The observed data.
- **exp** (*float or np.ndarray*) – The mean/expected value.

Returns The negative log likelihood.

Return type float

lib5c.algorithms.variance.nbinom_dispersion.**nb_nll_derivative**(*disp*, *obs*)
Derivative of the negative binomial log-likelihood function with respect to the dispersion parameter, given observed data.

This function is vectorized. Pass one dispersion and a vector of observed values to evaluate the derivative with just that one dispersion on the collection of all the observed values passed. Pass a vector of dispersions and a matrix of observed values to compute a vector of derivative evaluations, using the `i` th element of the dispersion vector and the `i` th row of the observed matrix to compute the `i` th derivative evaluation.

Parameters

- **disp** (*float or np.ndarray*) – The negative binomial dispersion parameter.
- **obs** (*np.ndarray*) – The observed values. If `disp` is a vector, this should be a matrix whose number of rows equals the length of `disp`.

Returns The derviative evaluation(s).

Return type float or np.ndarray

lib5c.algorithms.variance.nbinom_dispersion.**nb_pmf**(*k*, *m*, *alpha*)
The negative binomial PMF, parametrized in terms of a mean `m` and a dispersion `alpha`.

Parameters

- **k** (*int or np.ndarray*) – The observed value.
- **m** (*float or np.ndarray*) – The expected or mean value.
- **alpha** (*float or np.ndarray*) – The dispersion parameter.

Returns The value of the PMF.

Return type float or np.ndarray

lib5c.algorithms.variance.nbinom_dispersion.**variance_to_dispersion**(*var*, *exp*, *min_disp=None*)
Converts a variance estimate to a dispersion estimate by reversing the relationship in `dispersion_to_variance()`. Only defined for points where `var > exp`. If `var` is the sample variance and `exp` is the sample mean, this is the equivalent to the method-of-moments estimate of the dispersion parameter.

Parameters

- **var** (*float or np.ndarray*) – The variance.
- **exp** (*float or np.ndarray*) – The expected value.
- **min_disp** (*float, optional*) – Pass a value to enter a lenient mode where underdispersed points will be allowed, but will not be assigned a dispersion value from the statistical relationship. Underdispersed points will instead be assigned the `min_disp` value.

Returns The dispersion.

Return type float or np.ndarray

Module contents

Submodules

lib5c.algorithms.convergency module

Module containing functions for assisting in assessing the degree of convergency in orientation of transcription factors.

lib5c.algorithms.convergency.**compute_convergency**(*loops*, *pixelmap*, *peaks*, *motifs*, *loop_classes=('constitutive',)*, *margin=0*)

lib5c.algorithms.convergency.**prepare_convergency_annotations**(*pixelmap*, *peaks*, *motifs*)

lib5c.algorithms.correlation module

Module for computing correlations between 5C replicates.

lib5c.algorithms.correlation.**make_pairwise_correlation_matrix**(*counts_superdict*, *correlation='pearson'*, *rep_order=None*)

Computes a matrix of pairwise correlation coefficients among a set of 5C replicates.

Parameters

- **counts_superdict** (*Dict[str, Dict[str, np.ndarray]]*) – The keys to the outer dict are replicate names as strings. The values are standard "counts dicts" whose keys are region names as strings and whose values are square symmetric matrices of counts.
- **correlation** (*{'pearson', 'spearman'}*) – Controls which correlation will be used.
- **rep_order** (*Optional[List[str]]*) – Pass a list of strings to specify the order of the replicates in the rows and columns of the returned correlation matrix. If this kwarg is omitted the columns and rows of the returned correlation matrix will be arranged in the iteration order of the keys of `counts_superdict`.

Returns The square, symmetric pairwise correlation matrix.

Return type np.ndarray

lib5c.algorithms.correlation.**make_pairwise_correlation_matrix_from_counts_matrix**(*counts_matrix*, *correlation='pearson*

Computes a matrix of pairwise correlation coefficients among a set of 5C replicates.

Parameters

- **counts_matrix** (*np.ndarray*) – The rows are replicates, the columns are FFLJs.
- **correlation** (*{'pearson', 'spearman'}*) – Controls which correlation will be used.

Returns The square, symmetric pairwise correlation matrix.

Return type np.ndarray

lib5c.algorithms.determine_bins module

Module for computing sets of evenly-spaced bins for tiling 5C regions.

lib5c.algorithms.determine_bins.**default_bin_namer**(*bin_index*, *region_name=None*)
Names a bin given its index and, optionally, the name of the region.

Parameters

- **bin_index** (*int*) – The index of this bin, within the region if appropriate.
- **region_name** (*Optional[str]*) – The name of the region this bin is in.

Returns The name for this bin.

Return type str

Examples

```
>>> default_bin_namer(3)
'BIN_003'
>>> default_bin_namer(123, region_name='Sox2')
'Sox2_BIN_123'
```

lib5c.algorithms.determine_bins.**determine_regional_bins**(*regional_primermap*, *bin_width*, *region_name=None*, *bin_namer=<function default_bin_namer>*, *bin_namer_kwargs=None*, *region_span='mid-to-mid'*, *bin_number='n'*)
Determines a set of bins of a specified width that will tile a set of primers within a region.

Parameters

- **regional_primermap** (*List[Dict[str, Any]]*) – An ordered list of fragments in this region. The elements of the list are dicts (representing fragments) with at least the following structure:

```
{
    'chrom': str
    'start': int,
    'end': int
}
```

  See lib5c.parsers.primers.get_primermap().
- **bin_width** (*int*) – The width of the bins, in bp.
- **region_name** (*Optional[str]*) – The name of the region as a string. If this value is provided, it will also be passed on to the bin_namer as a kwarg.
- **bin_namer** (*Callable[[int, ..], str]*) – A function mapping bin indices to bin names. This function will be used to name the resulting bins. If region_name is passed, it will be passed on to this function as a kwarg.
- **bin_namer_kwargs** (*Optional[Dict[Any, Any]]*) – Additional kwargs to be passed to the bin_namer.

- **region_span** (*Optional[str]*) – Describes whether the span of the region is considered to be stretching from the midpoint of the first fragment to the midpoint of the last fragment ('mid-to-mid') or from the beginning of the first fragment to the end of the last fragment ('start-to-end').
- **bin_number** (*Optional[str]*) – Describes how many bins to fit in the region, given that 'n' is the largest number of full bins that will fit in the region. Use 'n' to reproduce traditional pipeline output, at the risk of leaving some fragment midpoints outside of the range of the bins. Use 'n+1' for a more conservative binning strategy that is guaranteed to not leave any fragment midpoints outside of the region if region_span is 'mid-to-mid'.

Returns

An ordered list of bins tiling the region. The elements of the list are dicts (representing bins) with the following structure:

```
{
    'name': str,
    'chrom': str,
    'start': int,
    'end': int,
    'index': int,
    'region': str (present only if region_name was passed)
}
```

Return type List[Dict[str, Any]]

Examples

```
>>> # single fragment results in single bin centered on the fragment
>>> regional_primermap = [{'chrom': 'chr1', 'start': 2000, 'end': 4000}]
>>> (determine_regional_bins(regional_primermap, 4000,
...                          region_name='Sox2') ==
... [{'name': 'Sox2_BIN_000', 'chrom': 'chr1', 'start': 1000, 'end': 5000,
...   'index': 0, 'region': 'Sox2'}])
True
```

```
>>> # examples for region_span='mid-to-mid'
>>> regional_primermap = [{'chrom': 'chr1', 'start': 2000, 'end': 4000},
...                       {'chrom': 'chr1', 'start': 9500, 'end': 10500}]
>>> (determine_regional_bins(regional_primermap, 5000) ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 4000, 'end': 9000,
...   'index': 0}])
True
>>> (determine_regional_bins(regional_primermap, 3000) ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 3500, 'end': 6500,
...   'index': 0},
...  {'name': 'BIN_001', 'chrom': 'chr1', 'start': 6500, 'end': 9500,
...   'index': 1}])
True
>>> (determine_regional_bins(regional_primermap, 3000,
...                          bin_number='n+1') ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 2000, 'end': 5000,
...   'index': 0},
...  {'name': 'BIN_001', 'chrom': 'chr1', 'start': 5000, 'end': 8000,
...   'index': 1},
```

(continues on next page)

(continued from previous page)

```
...  {'name': 'BIN_002', 'chrom': 'chr1', 'start': 8000, 'end': 11000,
...   'index': 2}])
True
```

```
>>> # examples for region_span='start-to-end'
>>> regional_primermap = [{'chrom': 'chr1', 'start': 2000, 'end': 4000},
...                       {'chrom': 'chr1', 'start': 9000, 'end': 10000}]
>>> (determine_regional_bins(regional_primermap, 5000,
...                          region_span='start-to-end') ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 3500, 'end': 8500,
...   'index': 0}])
True
>>> (determine_regional_bins(regional_primermap, 3000,
...                          region_span='start-to-end') ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 3000, 'end': 6000,
...   'index': 0},
...  {'name': 'BIN_001', 'chrom': 'chr1', 'start': 6000, 'end': 9000,
...   'index': 1}])
True
>>> (determine_regional_bins(regional_primermap, 3000,
...                          region_span='start-to-end',
...                          bin_number='n+1') ==
... [{'name': 'BIN_000', 'chrom': 'chr1', 'start': 1500, 'end': 4500,
...   'index': 0},
...  {'name': 'BIN_001', 'chrom': 'chr1', 'start': 4500, 'end': 7500,
...   'index': 1},
...  {'name': 'BIN_002', 'chrom': 'chr1', 'start': 7500, 'end': 10500,
...   'index': 2}])
True
```

lib5c.algorithms.donut_filters module

lib5c.algorithms.donut_filters.**apply_filter**(*obs_matrix*, *exp_matrix*, *footprint*, *max_percent=0.2*, *min_exp=0.1*)

Computes a corrected expected value by applying a footprint to observed and expected matrix.

Parameters

- **exp_matrix** (*obs_matrix,*) – The square, symmetric matrices of observed and expected counts, respectively.
- **footprint** (*np.ndarray*) – The footprint to convolve. Should contain 1's at positions included in the footprint and 0's everywhere else.
- **max_percent** (*float*) – If the proportion of nan's in the footprint for a pixel is greater than this value, the corrected expected at that point will be set to nan.
- **min_exp** (*float*) – If the sum of entries in `exp_matrix` under the footprint for a particular pixel is less than this value, set the output at this pixel to nan to avoid numerical instability related to division by small numbers.

Returns The corrected expected value.

Return type np.ndarray

lib5c.algorithms.donut_filters.**donut_filt**(*obs_matrix*, *exp_matrix*, *p*, *w*, *max_percent=0.2*, *min_exp=0.1*)

Computes the full donut filter.

Parameters

- **exp_matrix** (*obs_matrix,*) – The square, symmetric matrices of observed and expected counts, respectively.
- **w** (*p,*) – The inner and outer radii of the donut, respectively.
- **max_percent** (*float*) – If the proportion of nan's in the footprint for a pixel is greater than this value, the corrected expected at that point will be set to nan.
- **min_exp** (*float*) – If the sum of entries in `exp_matrix` under the footprint for a particular pixel is less than this value, set the output at this pixel to nan to avoid numerical instability related to division by small numbers.

Returns The corrected expected value.

Return type np.ndarray

`lib5c.algorithms.donut_filters.`**lower_left_filt**(*obs_matrix*, *exp_matrix*, *p*, *w*, *max_percent=0.2*, *min_exp=0.1*)

Computes the lower left donut filter.

Parameters

- **exp_matrix** (*obs_matrix,*) – The square, symmetric matrices of observed and expected counts, respectively.
- **w** (*p,*) – The inner and outer radii of the donut, respectively.
- **max_percent** (*float*) – If the proportion of nan's in the footprint for a pixel is greater than this value, the corrected expected at that point will be set to nan.
- **min_exp** (*float*) – If the sum of entries in `exp_matrix` under the footprint for a particular pixel is less than this value, set the output at this pixel to nan to avoid numerical instability related to division by small numbers.

Returns The corrected expected value.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.donut_filters import lower_left_filt
>>> from lib5c.algorithms.expected import empirical_binned
>>> from lib5c.algorithms.expected import make_expected_matrix_from_list
>>> obs = np.array([[10,  4,  1],
...                 [ 4,  8,  6],
...                 [ 1,  6, 12]]).astype(float)
>>> exp = make_expected_matrix_from_list(
...     empirical_binned(obs, log_transform=False))
>>> exp
array([[10.,  5.,  1.],
       [ 5., 10.,  5.],
       [ 1.,  5., 10.]])
>>> lower_left_filt(obs, exp, 0, 1, max_percent=0.0, min_exp=0.0)
array([[ nan,  4. ,  0.8],
       [ 4. , 10. ,  6. ],
       [ 0.8,  6. ,  nan]])
```

lib5c.algorithms.enrichment module

Module for computing enrichments of annotations within categories of categorized loops.

lib5c.algorithms.enrichment.**clear_enrichment_caches**()
Clear all caches related to enrichment computations.

This function is deprecated. Previously, it was necessary to call this function within a script whenever the content of annotationmaps or looping_classes changed. The current cache implementation does not need to be cleared when this happens.

Examples

```
>>> import numpy as np
>>> clear_enrichment_caches()
>>> annotationmaps = {'a': {'r1': [0, 0, 2, 1]},
...                   'b': {'r1': [1, 1, 0, 0]}}
>>> looping_classes = {'r1': np.array([['' , '' , 'es' , 'ips'],
...                                    ['' , '' , 'npc', 'npc'],
...                                    ['es' , 'npc', '' , '' ],
...                                    ['ips', 'npc', '' , '' ]],
...                                   dtype='U25')}
>>> count_intersections('a', 'b', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0)
1
>>> looping_classes = {'r1': np.array([['' , '' , 'ips', 'ips'],
...                                    ['' , '' , 'npc', 'npc'],
...                                    ['ips', 'npc', '' , '' ],
...                                    ['ips', 'npc', '' , '' ]],
...                                   dtype='U25')}
>>> count_intersections('a', 'b', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0)
0
>>> clear_enrichment_caches()
>>> count_intersections('a', 'b', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0)
0
```

lib5c.algorithms.enrichment.**count_intersections**(*annotation_a*, *annotation_b*, *region*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)
Counts the number of times one annotation intersects another at a particular category of called loops within a specified region.

Parameters

- **annotation_a** (*str*) – The annotation to look for on one side of the loop. Must be a key into annotationmaps.
- **annotation_b** (*str*) – The annotation to look for on the other side of the loop. Must be a key into annotationmaps.
- **region** (*str*) – The region to count intersections over.
- **category** (*str*) – The loop category to count intersections for.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The total number of intersections.

Return type int

Examples

```
>>> import numpy as np
>>> clear_enrichment_caches()
>>> annotationmaps = {'a': {'r1': [0, 0, 2, 1]},
...                   'b': {'r1': [1, 1, 0, 0]}}
>>> looping_classes = {'r1': np.array([['',  '',  'es' , 'ips'],
...                                    ['',  '',  'npc', 'npc'],
...                                    ['es' , 'npc', '',  ''  ],
...                                    ['ips', 'npc', '',  ''  ]],
...                                   dtype='U25')}
>>> count_intersections('a', 'b', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0)
1
>>> count_intersections('a', 'b', 'r1', 'npc', annotationmaps,
...                     looping_classes, margin=0)
2
>>> count_intersections('a', 'b', 'r1', 'npc', annotationmaps,
...                     looping_classes, margin=0)
2
>>> count_intersections.cache_info()
```

(continues on next page)

(continued from previous page)

```
CacheInfo(hits=1, misses=2, maxsize=None, currsize=2)
>>> count_intersections('a', 'b', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0, asymmetric=True)
0
>>> count_intersections('b', 'a', 'r1', 'es', annotationmaps,
...                     looping_classes, margin=0, asymmetric=True)
1
```

lib5c.algorithms.enrichment.**count_intersections_all**(*annotation_a*, *annotation_b*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Counts the number of times `annotation_a` and `annotation_b` are found on opposite ends of loops in a given category of loop type across all genomic regions.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **category** (*str*) – Only consider loops of this category.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The total number of intersections across all regions.

Return type int

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [0, 0, 2], 'r2': [1, 0]},
...                   'b': {'r1': [1, 1, 0], 'r2': [0, 1]}}
>>> looping_classes = {'r1': np.array([['npc', ''   , 'es' ],
...                                    ['' , ''   , 'npc'],
...                                    ['es' , 'npc', ''   ]],
...                                   dtype='U25'),
...                    'r2': np.array([['' , 'es' ],
...                                    ['es' , ''   ]],
...                                   dtype='U25')}
>>> count_intersections_all('a', 'b', 'es', annotationmaps,
...                         looping_classes, margin=0)
2
>>> count_intersections_all('a', 'b', 'npc', annotationmaps,
...                         looping_classes, margin=0)
1
```

lib5c.algorithms.enrichment.**get_annotation_percentage**(*annotation_a*, *annotation_b*, *region*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Computes the precentage of loops within a particular region categorized into a particular category that represent loops between `annotation_a` and `annotation_b`.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **region** (*str*) – The region to compute the percentage within.
- **category** (*str*) – The category of loops to consider.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The fraction of loops within the region of the specified category that represent loops between the indicated annotations.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [0, 0, 0, 1]},
...                   'b': {'r1': [1, 1, 0, 0]}}
>>> looping_classes = {'r1': np.array([['' , '' , 'es' , 'ips'],
...                                    ['' , '' , 'npc', 'npc'],
...                                    ['es' , 'npc', '' , '' ],
...                                    ['ips', 'npc', '' , '' ]],
...                                   dtype='U25')}
>>> get_annotation_percentage('a', 'b', 'r1', 'ips', annotationmaps,
...                           looping_classes, margin=0)
1.0
>>> get_annotation_percentage('a', 'b', 'r1', 'npc', annotationmaps,
...                           looping_classes, margin=0)
0.5
```

lib5c.algorithms.enrichment.**get_annotation_percentage_all**(*annotation_a*, *annotation_b*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Computes the precentage of loops across all regions categorized into a particular category that represent loops between `annotation_a` and `annotation_b`.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **category** (*str*) – The category of loops to consider.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
```

(continues on next page)

(continued from previous page)

```
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The fraction of loops across all regions of the specified category that represent loops between the indicated annotations.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [0, 0, 2], 'r2': [1, 0]},
...                   'b': {'r1': [1, 1, 0], 'r2': [0, 1]}}
>>> looping_classes = {'r1': np.array([['npc', ''   , 'es' ],
...                                    ['' , ''   , 'npc'],
...                                    ['es' , 'npc', ''   ]],
...                                   dtype='U25'),
...                    'r2': np.array([['npc', 'es' ],
...                                    ['es' , 'npc']],
...                                   dtype='U25')}
>>> get_annotation_percentage_all('a', 'b', 'es', annotationmaps,
...                               looping_classes, margin=0)
1.0
>>> get_annotation_percentage_all('a', 'b', 'npc', annotationmaps,
...                               looping_classes, margin=0)
0.25
```

lib5c.algorithms.enrichment.**get_fisher_exact_pvalue**(*annotation_a*, *annotation_b*, *region*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Use Fisher's exact test to compute a one-sided p-value against the null hypothesis that the selected loop category's overlap with selected annotations in a chosen region is the same as the special "background" reference loop category's overlap with the same annotations.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **region** (*str*) – The region to compute the p-value within
- **category** (*str*) – The category of loops to consider.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The p-value.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [1, 0, 0, 1]},
...                   'b': {'r1': [1, 1, 0, 0]}}
>>> looping_classes = {'r1': np.array([['' , '' , 'es' , 'ips'],
...                                    ['' , '' , 'npc', 'npc'],
```

(continues on next page)

(continued from previous page)

```
...                                     ['es' , 'npc', ''   , ''   ],
...                                     ['ips', 'npc', ''   , ''   ]],
...                                    dtype='U25')}
>>> looping_classes['r1'][looping_classes['r1'] == ''] = 'background'
>>> get_fisher_exact_pvalue('a', 'b', 'r1', 'ips', annotationmaps,
...                         looping_classes, margin=0)
0.428571428571428...
>>> get_fisher_exact_pvalue('a', 'b', 'r1', 'npc', annotationmaps,
...                         looping_classes, margin=0)
0.642857142857142...
```

lib5c.algorithms.enrichment.**get_fisher_exact_pvalue_all**(*annotation_a*, *annotation_b*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Use Fisher's exact test to compute a one-sided p-value against the null hypothesis that the selected loop category's overlap with selected annotations across all regions is the same as the special "background" reference loop category's overlap with the same annotations.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **category** (*str*) – The category of loops to consider.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.

- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The p-value.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [0, 1, 2], 'r2': [1, 0]},
...                   'b': {'r1': [1, 1, 0], 'r2': [0, 1]}}
>>> looping_classes = {'r1': np.array([['npc', ''   , 'es' ],
...                                    ['' , ''   , 'npc'],
...                                    ['es' , 'npc', ''   ]],
...                                   dtype='U25'),
...                    'r2': np.array([['ips', 'es' ],
...                                    ['es' , ''   ]],
...                                   dtype='U25')}
>>> looping_classes['r1'][looping_classes['r1'] == ''] = 'background'
>>> looping_classes['r2'][looping_classes['r2'] == ''] = 'background'
>>> round(get_fisher_exact_pvalue_all('a', 'b', 'es', annotationmaps,
...                                   looping_classes, margin=0), 14)
0.4
>>> round(get_fisher_exact_pvalue_all('a', 'b', 'npc', annotationmaps,
...                                   looping_classes, margin=0), 14)
0.8
>>> round(get_fisher_exact_pvalue_all('a', 'b', 'ips', annotationmaps,
...                                looping_classes, margin=0), 14)
0.6
```

lib5c.algorithms.enrichment.**get_fold_change**(*annotation_a*, *annotation_b*, *region*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Computes the fold enrichment of the percentage of loops of a particular category in a particular region connecting specified annotations relative to the special "background" reference category.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **region** (*str*) – The region to compute the fold enrichment within.
- **category** (*str*) – The category of loops to consider.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
```

(continues on next page)

(continued from previous page)

```
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The fold enrichment.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [1, 0, 0, 1]},
...                   'b': {'r1': [1, 1, 0, 0]}}
>>> looping_classes = {'r1': np.array([['' , '' , 'es' , 'ips'],
...                                    ['' , '' , 'npc', 'npc'],
...                                    ['es' , 'npc', '' , '' ],
...                                    ['ips', 'npc', '' , '' ]],
...                                   dtype='U25')}
>>> looping_classes['r1'][looping_classes['r1'] == ''] = 'background'
>>> get_fold_change('a', 'b', 'r1', 'ips', annotationmaps, looping_classes,
...                 margin=0)
3.0
>>> get_fold_change('a', 'b', 'r1', 'npc', annotationmaps, looping_classes,
...                 margin=0)
1.5
```

lib5c.algorithms.enrichment.**get_fold_change_all**(*annotation_a*, *annotation_b*, *category*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, *asymmetric=False*)

Computes the fold enrichment of the percentage of loops of a particular category across all regions connecting specified annotations relative to the special "background" reference category.

Parameters

- **annotation_a** (*str*) – Annotation to look for on one side of the loop.
- **annotation_b** (*str*) – Annotation to look for on the other side of the loop.
- **category** (*str*) – The category of loops to consider.

- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **asymmetric** (*bool*) – Pass True to only count situations when A is upstream of B. Pass False to count intersections regardless of order.

Returns The fold enrichment.

Return type float

Examples

```
>>> import numpy as np
>>> annotationmaps = {'a': {'r1': [0, 1, 2], 'r2': [1, 0]},
...                   'b': {'r1': [1, 1, 0], 'r2': [0, 1]}}
>>> looping_classes = {'r1': np.array([['npc', ''   , 'es' ],
...                                    ['' , ''   , 'npc'],
...                                    ['es' , 'npc', ''   ]],
...                                   dtype='U25'),
...                    'r2': np.array([['ips', 'es' ],
...                                    ['es' , ''   ]],
...                                   dtype='U25')}
>>> looping_classes['r1'][looping_classes['r1'] == ''] = 'background'
>>> looping_classes['r2'][looping_classes['r2'] == ''] = 'background'
>>> get_fold_change_all('a', 'b', 'es', annotationmaps, looping_classes,
...                     margin=0)
2.0
>>> get_fold_change_all('a', 'b', 'npc', annotationmaps, looping_classes,
...                     margin=0)
```

(continues on next page)

(continued from previous page)

```
1.0
>>> get_fold_change_all('a', 'b', 'ips', annotationmaps, looping_classes,
...                     margin=0)
0.0
```

lib5c.algorithms.enrichment.**process_annotations**(*annotation_label*, *region*, *annotationmaps*, *threshold=0*, *margin=1*)

Extracts one annotation and one region from a dict of annotationmaps and returns it in a vector form.

This function should be called from within the bodies of vectorized enrichment functions that accept standard annotationmaps as arguments.

Parameters

- **annotation_label** (*str*) – The annotation for which a vector should be created. Must be a key of `annotationmaps`.
- **region** (*str*) – The specific region for which a vector should be created. Must be a key of `annotationmaps[annotation_label]`.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.

Returns The processed vector representing the coverage of the selected annotation across the selected region, according to the definitions implied by the choise of `threshold` and `margin`.

Return type np.ndarray

Examples

```
>>> annotationmaps = {'a': {'r1': [0, 0, 2, 1]}}
>>> process_annotations('a', 'r1', annotationmaps)
array([0, 1, 1, 1])
```

(continues on next page)

(continued from previous page)

```
>>> process_annotations('a', 'r1', annotationmaps, threshold=1, margin=0)
array([0, 0, 1, 0])
```

lib5c.algorithms.expected module

Module for computing expected models for 5C interaction data.

lib5c.algorithms.expected.**empirical_binned**(*regional_counts*, *log_transform=True*)

Make a regional one-dimensional bin-level expected model by taking an average of the interaction values at each distance.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **log_transform** (*bool*) – Pass True to take the geometric mean instead of the arithmetic mean, which is equivalent to averaging log-transformed counts.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.expected.**force_monotonic**(*distance_expected*)

Force a one-dimensional distance expected to be monotonic.

Parameters **distance_expected** (*Union[List[float], Dict[int, float]]*) – The one-dimensional expected model to force to monotonicity. If the model describes bin-level data, this should be a list of floats, where the `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. If the model describes fragment-level data, this should be a dict mapping interaction distances in units of base pairs to the expected value at that distance.

Returns The forced-monotonic version of the input one-dimensional expected model.

Return type Union[List[float], Dict[int, float]]

lib5c.algorithms.expected.**get_distance_expected**(*obs_matrix*, *regional_primermap=None*, *level='bin'*, *powerlaw=False*, *regression=False*, *degree=1*, *lowess_smooth=False*, *lowess_frac=0.8*, *log_transform='auto'*, *exclude_near_diagonal=False*)

Convenience function for computing a regional one-dimensional expected model from a matrix of observed counts, with properties that can be customized by kwargs.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of observed counts to model.
- **regional_primermap** (*Optional[List[Dict[str, Any]]]*) – The primermap for this region. Required if `obs_matrix` is fragment-level.
- **level** (*{'bin', 'fragment'}*) – The level of `obs_matrix`.
- **powerlaw** (*bool*) – Whether or not to fit a discrete power law distribution to the data.
- **regression** (*bool*) – Whether or not to use a polynomial regression model.
- **degree** (*int*) – The degree of the regression model to use.

- **lowess_smooth** (*bool*) – Whether or not to use lowess smoothing to compute the model.
- **lowess_frac** (*float*) – The lowess smoothing fraction parameter.
- **log_transform** (*{'counts', 'both', 'none', 'auto'}*) –

  What to transform into log space.

  - counts: log-transform only the counts but not the distances. This results in semi-log models, which don't work on fragment-level data yet.
  - both: log-transform both the counts and the distances, resulting in log-log models.
  - none: don't log anything.
  - auto: automatically pick a reasonably choice based on the other kwargs.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. For bin-level data, this is a list of floats, where the `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. For fragment-level data, this is a dict mapping interaction distances in units of base pairs to the appropriate expected values.

Return type Union[List[float], Dict[int, float]]

lib5c.algorithms.expected.**get_global_distance_expected**(*counts*, *primermap=None*, *level='bin'*, *powerlaw=False*, *regression=False*, *degree=1*, *lowess_smooth=False*, *lowess_frac=0.8*, *log_transform='auto'*, *exclude_near_diagonal=False*)

Convenience function for computing a global one-dimensional expected model from a dict of observed counts, with properties that can be customized by kwargs.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The dict of observed counts to model.
- **primermap** (*Optional[Dict[str, List[Dict[str, Any]]]]*) – A primermap corresponding to `counts`.
- **level** (*{'bin', 'fragment'}*) – The level of `counts`.
- **powerlaw** (*bool*) – Whether or not to fit a discrete power law distribution to the data.
- **regression** (*bool*) – Whether or not to use a polynomial regression model.
- **degree** (*int*) – The degree of the regression model to use.
- **lowess_smooth** (*bool*) – Whether or not to use lowess smoothing to compute the model.
- **lowess_frac** (*float*) – The lowess smoothing fraction parameter.
- **log_transform** (*{'counts', 'both', 'none', 'auto'}*) –

  What to transform into log space.

  - counts: log-transform only the counts but not the distances. This results in semi-log models, which don't work on fragment-level data yet.

  - both: log-transform both the counts and the distances, resulting in log-log models.
  - none: don't log anything.
  - auto: automatically pick a reasonably choice based on the other kwargs.

- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. For bin-level data, this is a list of floats, where the `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. For fragment-level data, this is a dict mapping interaction distances in units of base pairs to the appropriate expected values.

Return type Union[List[float], Dict[int, float]]

lib5c.algorithms.expected.**global_empirical_binned**(*counts*, *log_transform=True*)

Make a global one-dimensional bin-level expected model by taking an average of the interaction values at each distance.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **log_transform** (*bool*) – Pass True to take the geometric mean instead of the arithmetic mean, which is equivalent to averaging log-transformed counts.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**global_lowess_binned**(*counts*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a global one-dimensional bin-level expected model by performing lowess regression in unlogged space, excluding the first third of the distance scales and only using the emprical arithmetic means there instead.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**global_lowess_binned_log_counts**(*counts*, *pseudocount=1*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a global one-dimensional bin-level expected model by performing lowess regression in log-counts space, excluding the first third of the distance scales and only using the emprical geometric means there instead.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.

- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**global_lowess_log_log_binned**(*counts*, *pseudocount=1*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a global one-dimensional bin-level expected model by performing lowess regression in log-log space.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**global_lowess_log_log_fragment**(*counts*, *distances*, *pseudocount=1*, *frac=0.8*)

Make a global one-dimensional fragment-level expected model by performing lowess regression in log-log space.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **distances** (*Dict[str, np.ndarray]*) – A dict of pairwise distance matrices describing the genomic distances between the elements of the matrices in `counts`. The keys and array dimensions should match the keys and array dimensions of `counts`.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.

Returns A mapping from interaction distances in units of base pairs to the expected value at that distance.

Return type Dict[int, float]

lib5c.algorithms.expected.**global_poly_log_log_binned**(*counts*, *degree=1*, *pseudocount=1*, *exclude_near_diagonal=False*)

Make a global one-dimensional bin-level expected model by fitting a polynomial in log-log space.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **degree** (*int*) – The degree of the polynomial to fit.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.

- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**global_poly_log_log_fragment**(*counts*, *distances*, *degree=1*, *pseudocount=1*)

Make a global one-dimensional fragment-level expected model by fitting a polynomial in log-log space.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **distances** (*Dict[str, np.ndarray]*) – A dict of pairwise distance matrices describing the genomic distances between the elements of the matrices in `counts`. The keys and array dimensions should match the keys and array dimensions of `counts`.
- **degree** (*int*) – The degree of the polynomial to fit.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.

Returns A mapping from interaction distances in units of base pairs to the expected value at that distance.

Return type Dict[int, float]

lib5c.algorithms.expected.**global_powerlaw_binned**(*counts*, *exclude_near_diagonal=False*)

Make a global one-dimensional bin-level expected model by fitting a polynomial in log-log space.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The observed counts dict to fit the model to.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins. The length of this list will match the size of the largest region in the input counts dict.

Return type List[float]

lib5c.algorithms.expected.**interpolate_expected**(*expected_matrix*, *regional_primermap*, *distance*)

Interpolate the value of an expected model (represented as a matrix) at an arbitrary distance scale.

Parameters

- **expected_matrix** (*np.ndarray*) – The expected matrix to use as a source for interpolation.
- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap for this region.
- **distance** (*int*) – The interaction distance at which to estimate the expected value, in base pairs.

Returns The interpolated expected value, or -1 if `distance` is outside of the range of the expected model.

Return type float

lib5c.algorithms.expected.**lowess_binned**(*regional_counts*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a regional one-dimensional bin-level expected model by performing lowess regression in unlogged space, excluding the first third of the region and only using the emprical geometric means there instead.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.expected.**lowess_binned_log_counts**(*regional_counts*, *pseudocount=1*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a regional one-dimensional bin-level expected model by performing lowess regression in log-counts space, excluding the first third of the region and only using the emprical geometric means there instead.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.expected.**lowess_log_log_binned**(*regional_counts*, *pseudocount=1*, *frac=0.8*, *exclude_near_diagonal=False*)

Make a regional one-dimensional bin-level expected model by performing lowess regression in log-log space.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.expected.**lowess_log_log_fragment**(*regional_counts*, *distances*, *pseudocount=1*, *frac=0.8*)

Make a regional one-dimensional fragment-level expected model by performing lowess regression in log-log space.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **distances** (*np.ndarray*) – The pairwise distance matrix for all fragments in this region in units of base pairs.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **frac** (*float*) – The lowess smoothing fraction parameter to use.

Returns A mapping from interaction distances in units of base pairs to the expected value at that distance.

Return type Dict[int, float]

lib5c.algorithms.expected.**make_distance_matrix**(*regional_primermap*)

Construct a pairwise distance matrix for the fragments in a region from the primermap describing those fragments.

Parameters **regional_primermap** (*List[Dict[str, Any]]*) – The primermap for this region.

Returns The pairwise distance matrix for all fragments in this region in units of base pairs.

Return type np.ndarray

lib5c.algorithms.expected.**make_expected_dict_from_matrix**(*expected_matrix*, *distance_matrix*)

Convert an expected matrix into a dict representation of the one-dimensional expected model it embodies.

Parameters

- **expected_matrix** (*np.ndarray*) – The expected matrix.
- **distance_matrix** (*np.ndarray*) – The pairwise distance matrix for the fragments in this region.

Returns A mapping from interaction distances in units of base pairs to the expected value at that distance.

Return type Dict[int, float]

lib5c.algorithms.expected.**make_expected_matrix**(*obs_matrix*, *regional_primermap=None*, *level='bin'*, *powerlaw=False*, *regression=False*, *degree=1*, *lowess_smooth=False*, *lowess_frac=0.8*, *log_transform='auto'*, *monotonic=False*, *donut=False*, *w=15*, *p=5*, *donut_frac=0.2*, *min_exp=0.1*, *log_donut=False*, *max_donut_ll=False*, *distance_expected=None*, *exclude_near_diagonal=False*)

Convenience function for computing a complete expected matrix given a matrix of observed counts that can be customized with a variety of kwargs.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of observed counts to make an expected matrix for.
- **regional_primermap** (*Optional[List[Dict[str, Any]]]*) – The primermap for this region. Required if `obs_matrix` is fragment-level.
- **level** (*{'bin', 'fragment'}*) – The level of `obs_matrix`.

- **powerlaw** (*bool*) – Whether or not to fit a discrete power law distribution to the data.
- **regression** (*bool*) – Whether or not to use a polynomial regression model.
- **degree** (*int*) – The degree of the regression model to use.
- **lowess_smooth** (*bool*) – Whether or not to use lowess smoothing to compute the model.
- **lowess_frac** (*float*) – The lowess smoothing fraction parameter.
- **log_transform** (*{'counts', 'both', 'none', 'auto'}*) –

  What to transform into log space.

  - counts: log-transform only the counts but not the distances. This results in semi-log models, which don't work on fragment-level data yet.
  - both: log-transform both the counts and the distances, resulting in log-log models.
  - none: don't log anything.
  - auto: automatically pick a reasonably choice based on the other kwargs.
- **monotonic** (*bool*) – Pass True to force the one-dimensional expected model to be monotonic.
- **donut** (*bool*) – Pass True to apply donut-filter local correction to the expected model. Not implemented for fragment-level input data.
- **w** (*int*) – The outer width of the donut when using donut correction. Should be an odd integer.
- **p** (*int*) – The inner width of the donut when using donut correction. Should be an odd integer.
- **donut_frac** (*float*) – If the fraction of possible elements in the donut that lie wihtin the region and have non-infinte values is lower than this fraction then the donut-corrected value at that point will be NaN.
- **min_exp** (*float*) – If the sum of the 1-D expected matrix under the donut or lower left footprint for a particular pixel is less than this value, set the output at this pixel to nan to avoid numerical instability related to division by small numbers.
- **log_donut** (*bool*) – Pass True to perform donut correction in log-counts space.
- **max_donut_ll** (*bool*) – If `donut` is True, pass True here too to make the donut correction use the maximum of the "donut" and "lower-left" regions.
- **distance_expected** (*Optional[Union[List[float], Dict[int, float]]]*) – Pass a one-dimensional expected model to use it instead of computing a new one from scratch according to the other kwargs.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns

- *Tuple[np.ndarray, Union[List[float], Dict[int, float]], Optional[*
- *np.ndarray]]* – The first element of the tuple is the expected matrix. The second element of the tuple is the one-dimensional expected model, which will be a list of expected values if `level` was 'bin' or a dict mapping integer distances to expected values if `level` was 'fragment'. The third element will be the pairwise distance matrix if `level` was 'fragment', but will simply be None if `level` was 'bin'.

lib5c.algorithms.expected.**make_expected_matrix_from_dict**(*distance_expected*, *distance_matrix*)

Converts a fragment-level one-dimensional expected model into an expected matrix.

Parameters

- **distance_expected** (*Dict[int, float]*) – A mapping from interaction distances in units of base pairs to the expected value at that distance.
- **distance_matrix** (*np.ndarray*) – The pairwise distance matrix for the fragments in this region.

Returns The expected matrix.

Return type np.ndarray

lib5c.algorithms.expected.**make_expected_matrix_from_list**(*distance_expected*)

Converts a bin-level one-dimensional expected model into an expected matrix.

Parameters **distance_expected** (*List[float]*) – The one-dimensional distance expected model to make a matrix out of.

Returns The expected matrix.

Return type np.ndarray

lib5c.algorithms.expected.**make_poly_log_log_binned_expected_matrix**(*obs_matrix*, *exclude_near_diagonal=False*)

Convenience function for quickly making an expected matrix for a bin-level observed counts matrix based on a simple power law relationship.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of observed counts.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The expected matrix.

Return type np.ndarray

lib5c.algorithms.expected.**make_poly_log_log_fragment_expected_matrix**(*obs_matrix*, *regional_primermap*)

Convenience function for quickly making an expected matrix for a fragment-level observed counts matrix based on a simple power law relationship.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of observed counts.
- **regional_primermap** (*List[Dict[str, Any]]*) – Primermap describing the loci in the region represented by `obs_matrix`. Necessary to figure out distances between elements in the contact matrix.

Returns The expected matrix.

Return type np.ndarray

lib5c.algorithms.expected.**make_powerlaw_binned_expected_matrix**(*obs_matrix*, *exclude_near_diagonal=False*)

Convenience function for quickly making an expected matrix for a bin-level observed counts matrix based on a simple power law relationship.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of observed counts.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The expected matrix.

Return type np.ndarray

lib5c.algorithms.expected.**poly_log_log_binned**(*regional_counts*, *degree=1*, *pseudocount=1*, *exclude_near_diagonal=False*)

Make a regional one-dimensional bin-level expected model by fitting a polynomial in log-log space.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **degree** (*int*) – The degree of the polynomial to fit.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.expected.**poly_log_log_fragment**(*regional_counts*, *distances*, *degree=1*, *pseudocount=1*)

Make a regional one-dimensional fragment-level expected model by fitting a polynomial in log-log space.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **distances** (*np.ndarray*) – The pairwise distance matrix for all fragments in this region in units of base pairs.
- **degree** (*int*) – The degree of the polynomial to fit.
- **pseudocount** (*int*) – The pseudocount to add to the counts before logging.

Returns A mapping from interaction distances in units of base pairs to the expected value at that distance.

Return type Dict[int, float]

lib5c.algorithms.expected.**powerlaw_binned**(*regional_counts*, *exclude_near_diagonal=False*)

Make a regional one-dimensional bin-level expected model by fitting a polynomial in log-log space.

Parameters

- **regional_counts** (*np.ndarray*) – The observed counts matrix for this region.
- **exclude_near_diagonal** (*bool*) – If regression or lowess_smooth are True, set this kwarg to True to ignore the first third of the distance scales when fitting the model.

Returns The one-dimensional expected model. The `i` th element of the list corresponds to the expected value for interactions between loci separated by `i` bins.

Return type List[float]

lib5c.algorithms.express module

Module for implementation of the "Express" algorithm from Sauria et al. 2015.

lib5c.algorithms.express.**express_normalize_matrix**(*obs_matrix*, *exp_matrix*, *max_iter=1000*, *eps=0.0001*)

Express balance a matrix given a corresponding expected matrix.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix to normalize.
- **exp_matrix** (*np.ndarray*) – The expected matrix corresponding to the obs_matrix.
- **max_iter** (*int*) – The maximum number of iterations.
- **eps** (*float*) – When the fractional change in the residual is less than this number, the algorithm is considered to have converged and will stop iterating.

Returns The first element of the tuple is the normalized matrix. The second element is the multiplicative bias vector. The third element is a list containing the L1 norm of the residual at every iteration.

Return type Tuple[np.ndarray, np.ndarray, List[float]]

lib5c.algorithms.express.**joint_express_normalize**(*obs_matrices*, *exp_matrices*, *max_iter=1000*, *eps=0.0001*)

Express balance a set of matrices given a set of corresponding expected matrices, using a single shared bias vector.

Parameters

- **obs_matrices** (*List[np.ndarray]*) – The matrix to normalize.
- **exp_matrices** (*List[np.ndarray]*) – The expected matrix corresponding to the obs_matrix.
- **max_iter** (*int*) – The maximum number of iterations.
- **eps** (*float*) – When the fractional change in the residual is less than this number, the algorithm is considered to have converged and will stop iterating.

Returns The first element of the tuple is the list of normalized matrices. The second element is the multiplicative bias vector. The third element is a list containing the L1 norm of the residual at every iteration.

Return type Tuple[List[np.ndarray], np.ndarray, List[float]]

lib5c.algorithms.knight_ruiz module

Knight-Ruiz algorithm.

Transcribed from the MATLAB source provided in Rao et al. 2014 by Dan Gillis.

lib5c.algorithms.knight_ruiz.**balance_matrix**(*matrix*, *bias_vector*, *invert=False*)

Balance a matrix given the appropriate multiplicative bias vector.

Parameters

- **matrix** (*np.ndarray*) –
- **bias_vector** (*np.ndarray*) –
- **invert** (*Optional[bool]*) – Pass True to invert the bias vector before balancing.

Returns The balanced matrix.

Return type np.ndarray

lib5c.algorithms.knight_ruiz.**kr_balance**(*array*, *tol=1e-06*, *x0=None*, *delta=0.1*, *ddelta=3*, *fl=0*, *max_iter=3000*)

Performs Knight-Ruiz matrix balancing algorithm on a 2D symmetric numpy array.

Note that this function does not check for symmetry of the array - this function may not converge if given non-symmetric matrix.

Note also that this function does not return balanced matrix - it returns the entries of the diagonal matrix that should be multiplied on either side of `array` to get balanced matrix.

Parameters

- **array** (*np.ndarray*) – Matrix to balance. Should be square and symmetric.
- **tol** (*float*) – Parameter related to tolerance
- **x0** (*Optional[np.ndarray]*) – The initial guess to use for the bias vector. If not passed, a vector of all 1's will be used.
- **delta** (*float*) – Parameter related to learning rate.
- **ddelta** (*float*) – Parameter related to learning rate.
- **fl** (*int*) – Adjusts the verbosity of command line output.
- **max_iter** (*Optional[int]*) – The maximum number of iterations. Pass None to set no limit.

Returns The first element is the bias vector, the second is the residual.

Return type Tuple[np.ndarray]

Examples

```
>>> import numpy as np
>>> X = np.reshape(range(16), (4, 4)).astype(float)
>>> counts = X + X.T
>>> counts
array([[ 0.,  5., 10., 15.],
       [ 5., 10., 15., 20.],
       [10., 15., 20., 25.],
       [15., 20., 25., 30.]])
>>> counts.sum(axis=1)
array([30., 50., 70., 90.])
>>> x, res = kr_balance(counts)
>>> balanced = x.T * counts * x
>>> balanced
array([[0.        , 0.26604444, 0.34729636, 0.3866592 ],
       [0.26604444, 0.2489703 , 0.24375574, 0.24122952],
       [0.34729636, 0.24375574, 0.21213368, 0.19681423],
       [0.3866592 , 0.24122952, 0.19681423, 0.17529705]])
>>> balanced.sum(axis=1)
array([1., 1., 1., 1.])
>>> for i in range(len(counts)):
...     for j in range(i+1):
...         if i % 2 == j % 2:
...             counts[i, j] = 0.0
```

(continues on next page)

(continued from previous page)

```
...             counts[j, i] = 0.0
>>> counts
array([[ 0.,  5.,  0., 15.],
       [ 5.,  0., 15.,  0.],
       [ 0., 15.,  0., 25.],
       [15.,  0., 25.,  0.]])
>>> x, res = kr_balance(counts)
>>> balanced = x.T * counts * x
>>> balanced
array([[0.        , 0.42705098, 0.        , 0.57294902],
       [0.42705098, 0.        , 0.57294902, 0.        ],
       [0.        , 0.57294902, 0.        , 0.42705098],
       [0.57294902, 0.        , 0.42705098, 0.        ]])
>>> balanced.sum(axis=1)
array([1., 1., 1., 1.])
```

lib5c.algorithms.knight_ruiz.**kr_balance_matrix**(*matrix*, *max_iter=3000*, *retain_scale=True*, *imputation_size=0*)

Convenience function for applying KR balancing to a counts matrix.

Parameters

- **matrix** (*np.ndarray*) – The matrix to balance.
- **max_iter** (*int*) – The maximum number of iterations to try.
- **retain_scale** (*bool*) – Pass True to rescale the results to the scale of the original matrix using a ratio of geometric means.
- **imputation_size** (*int*) – Pass an int greater than 0 to replace NaN's in the matrix with a local median approximation. Pass 0 to skip imputation.

Returns The first array contains the balanced matrix. The second contains the bias vector. The third contains the residual.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray]

lib5c.algorithms.knight_ruiz.**strip_zero_rows_columns_sym_mat**(*sym_mat*)

Given symmetric 2D numpy array sym_mat, removes rows and columns that have no non-zero entries

lib5c.algorithms.outliers module

Module for identifying and removing high spatial outliers from 5C contact matrices.

lib5c.algorithms.outliers.**flag_array_high_spatial_outliers**(*array*, *size=5*, *fold_threshold=8.0*)

Identifies which elements of an array are high spatial outliers.

Parameters

- **array** (*np.ndarray*) – The array to look for outliers in.
- **size** (*int*) – The size of the window to look in around each element when deciding if it is an outlier. Should be an odd integer.
- **fold_threshold** (*float*) – Elements will be flagged as outliers if they are greater than this number or greater than this many times the local median (as estimated using the window size in `size`).

Returns A matrix of the same size and shape as the input matrix, with 1's at positions flagged as high spatial outliers and 0's everywhere else.

Return type np.ndarray

lib5c.algorithms.outliers.**remove_high_spatial_outliers**(*counts*, *size=5*, *fold_threshold=8.0*, *overwrite_value='nan'*, *primermap=None*, *level='fragment'*)

Convenience function for removing high spatial outliers from counts matrices.

Parameters

- **counts** (*np.ndarray*) – The matrix to remove outliers from.
- **size** (*int*) – The size of the window to look in around each element when deciding if it is an outlier. Should be an odd integer.
- **fold_threshold** (*float*) – Elements will be flagged as outliers if they are greater than this number or greater than this many times the local median (as estimated using the window size in `size`).
- **overwrite_value** (*{'nan', 'zero', 'median'}*) – The value to overwrite elements flagged as outliers with.
- **primermap** (*List[Dict[str, Any]]*) – The list of fragments for this region corresponding to `counts`.
- **level** (*{'fragment', 'bin'}*) – Whether to interpret `counts` as bin- or fragment-level. The difference is that bin-level matrices are assumed to have equal distance between elements.

Returns The input matrix with all spatial outliers overwritten.h

Return type np.ndarray

lib5c.algorithms.outliers.**remove_primer_primer_pairs**(*counts_superdict*, *primermap*, *threshold=5.0*, *num_reps=None*, *fraction_reps=None*, *all_reps=False*, *inplace=True*)

Removes primer-primer pairs from a set of replicates according to criteria specified by the kwargs.

Legacy code inherited from https://bitbucket.org/creminslab/primer-primer-pair-remover

Parameters

- **counts_superdict** (*Dict[str, Dict[str, np.ndarray]]*) – The counts superdict data structure to remove primer-primer pairs from.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap or pixelmap describing the loci whose interaction frequencies are quantified in the `counts_superdict`.
- **threshold** (*float*) – Sets the threshold. A rep passes the threshold if it is greater than or equal to this number.
- **num_reps** (*Optional[int]*) – Pass an int to make the condition be that this many reps must clear the threshold.
- **fraction_reps** (*Optional[float]*) – Pass a fraction (between 0 and 1) as a float to make the condition be that this fraction of the reps must clear the threshold.

- **all_reps** (*bool*) – Pass True to make the condition be that the sum across all replicates must clear the threshold. This is the default mode if niether `num_reps` nor `percentage_reps` is passed.
- **inplace** (*bool*) – Pass True to operate in-place on the passed `counts_superdict`; pass False to return a new counts superdict.

Returns The result of the primer-primer pair removal, in the form of a counts superdict data structure analagous to the `counts_superdict` passed to this function.

Return type Dict[str, Dict[str, np.ndarray]]

lib5c.algorithms.pca module

`lib5c.algorithms.pca.`**compute_pca**(*matrix*, *scaled=True*, *logged=False*, *kernel=None*, *kernel_kwargs=None*, *variant='pca'*, *pf=1*)

Performs PCA on a matrix.

Parameters

- **matrix** (*np.ndarray*) – The design matrix, whose rows are observations (replicates) and whose columns are features (interaction values at each position).
- **scaled** (*bool*) – Pass True to scale the features to unit variance.
- **logged** (*bool*) – Pass True to log the features before PCA.
- **kernel** (*Optional[str]*) – Pass a kernel accepted by `sklearn.decomposition.KernelPCA()` to perform KPCA.
- **kernel_kwargs** (*Optional[Dict[str, Any]]*) – Kwargs to use for the kernel.
- **variant** (*{'pca', 'ica', 'fa', 'mds'}*) – Select which variant of PCA to use.
- **pf** (*int*) – Specify an integer number of pure polynomial features to use in the PCA.

Returns The first element is the matrix of PCA-projected replicates. The second element is the PVE for each component, or None if the PCA method selected doesn't provide a PVE estimate. The third element is a matrix of the principle component vectors, or None if the PCA method selected doesn't provide a set of principle component vectors.

Return type Tuple[np.ndarray]

`lib5c.algorithms.pca.`**compute_pca_from_counts_superdict**(*counts_superdict*, *rep_order=None*, ***kwargs*)

Convenience function for performing PCA on a counts superdict data structure.

Parameters

- **counts_superdict** (*Dict[str, Dict[str, np.ndarray]]*) – The counts superdict structure to compute PCA on.
- **rep_order** (*Optional[List[str]]*) – The order in which the replicates in `counts_superdict` should be considered when filling in the rows of the design matrix.
- **kwargs** (*Dict[str, Any]*) – Additional kwargs to be passed to `compute_pca()`.

Returns The first element is the matrix of PCA-projected replicates. The second element is the PVE for each component, or None if the PCA method selected doesn't provide a PVE estimate. The third element is a matrix of the principle component vectors, or None if the PCA method selected doesn't provide a set of principle component vectors.

Return type Tuple[np.ndarray]

lib5c.algorithms.qnorm module

Module for quantile normalization.

Original author of *qnorm()*, `_rank_data()`, `_average_rows()`, and `_sub_in_normed_val()`: Dan Gillis

Note: data matrices in these functions are typically expected to be arranged with each column representing one replicate, except for the functions `_rank_data()`, `_average_rows()`, and `_sub_in_normed_val()`, which expect them to be arranged with each row representing one replicate.

The exposed functions are *qnorm()*, *qnorm_parallel()*, *qnorm_fast()*, *qnorm_fast_parallel()*, and the convenience function *qnorm_counts_superdict()*.

`lib5c.algorithms.qnorm.`**qnorm**(*data*, *tie='lowest'*, *reference_index=None*)

Quantile normalizes a data set.

Parallelizable if `data` is a 2d np.ndarray; see `lib5c.algorithms.qnorm.qnorm_parallel()`.

Parameters

- **data** (*2d numeric structure, or dict of 1d numeric structure*) – Anything that can be cast to array. Should be passed as row-major. Quantile normalization will performed on the columns of `data`.
- **tie** (*{'lowest', 'average'}, optional*) – Pass `'lowest'` to set all tied entries to the value of the lowest rank. Pass `'average'` to set all tied entries to the average value across the tied ranks.
- **reference_index** (*int or str, optional*) – If `data` is a row-major array, pass a column index to serve as a reference distribution. If `data` is a dict, pass a key of that dict that should serve as the reference distribution. Pass None to use the average of all distributions as the target distribution.

Returns The quantile normalized data. If `data` was passed as a dict, then a dict with the same keys is returned.

Return type 2d numpy array, or dict of 1d numpy array

Notes

This function is nan-safe. As long as each column of the input data contains the same number of nan's, nan's will only get averaged with other nan's, and they will get substituted back into their original positions. See the Examples section for an example of this.

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.qnorm import qnorm
>>> qnorm(np.array([[5,    4,    3],
...                 [2,    1,    4],
...                 [3,    4,    6],
...                 [4,    2,    8]]))
...
array([[5.66666667, 4.66666667, 2.        ],
```

(continues on next page)

(continued from previous page)

```
       [2.        , 2.        , 3.        ],
       [3.        , 4.66666667, 4.66666667],
       [4.66666667, 3.        , 5.66666667]])
>>> qnorm(np.array([[     5, np.nan,      3],
...                 [     2,      1,      4],
...                 [np.nan,      4,      6],
...                 [     4,      2, np.nan]]))
...
array([[5.        ,        nan, 2.        ],
       [2.        , 2.        , 3.33333333],
       [       nan, 5.        , 5.        ],
       [3.33333333, 3.33333333,        nan]])
>>> qnorm(np.array([[     5, np.nan,      3],
...                 [     2,      1,      4],
...                 [np.nan,      4,      6],
...                 [     4,      2, np.nan]]), reference_index=1)
...
array([[ 4., nan,  1.],
       [ 1.,  1.,  2.],
       [nan,  4.,  4.],
       [ 2.,  2., nan]])
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]})
>>> list(sorted(res.items()))
[('A', array([5.66666667, 2.        , 3.        , 4.66666667])),
 ('B', array([4.66666667, 2.        , 4.66666667, 3.        ])),
 ('C', array([2.        , 3.        , 4.66666667, 5.66666667]))]
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]}, reference_index='C')
>>> list(sorted(res.items()))
[('A', array([8., 3., 4., 6.])),
 ('B', array([6., 3., 6., 4.])),
 ('C', array([3., 4., 6., 8.]))]
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]}, reference_index='C', tie='average')
>>> list(sorted(res.items()))
[('A', array([8., 3., 4., 6.])),
 ('B', array([7., 3., 7., 4.])),
 ('C', array([3., 4., 6., 8.]))]
```

lib5c.algorithms.qnorm.**qnorm_counts_superdict**(*counts_superdict*, *primermap*, *tie='lowest'*, *regional=False*, *condition_on=None*, *reference=None*)

Convenience function for quantile normalizing a counts superdict data structure.

Parameters

- **counts_superdict** (*Dict[Dict[np.ndarray]]*) – The keys of the outer dict are replicate names, the keys of the inner dict are region names, the values are square symmetric arrays of counts for the specified replicate and region.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap describing the loci whose interaction counts are described in the `counts_superdict`.
- **tie** (*{'lowest', 'average'}*) – Pass `'lowest'` to set all tied entries to the value

of the lowest rank. Pass `'average'` to set all tied entries to the average value across the tied ranks.

- **regional** (*bool*) – Pass True to quantile normalize regions separately. Pass False to quantile normalize all regions together.
- **condition_on** (*Optional[str]*) – Pass a string key into the inner dicts of `primermap` to condition on that quantity. Current limitations: only works with `regional=True` and can only condition with exact equality (does not support conditioning on strata of a quantity). Pass None to not do conditional quantile normalization.
- **reference** (*Optional[str]*) – Pass a string key into the `counts_superdict` to indicate a replicate that should be used as a reference distribution to quantile normalize to.

Returns The keys of the outer dict are replicate names, the keys of the inner dict are region names, the values are square symmetric arrays of the quantile normalized counts for the specified replicate and region.

Return type Dict[Dict[np.ndarray]]

`lib5c.algorithms.qnorm.`**qnorm_fast**(*data*, *reference_index=None*)

Quantile normalizes a data set.

Simpler, faster implementation compared to `lib5c.algorithms.qnorm()`, but only supports `tie='lowest'` behavior and only takes an `np.ndarray` as input. This approach was developed and timed in this repsitory.

Parallelizable if `data` is a 2d np.ndarray; see `lib5c.algorithms.qnorm.qnorm_fast_parallel()`.

Parameters

- **data** (*np.ndarray*) – Two dimensional, with the columns representing the replicates to be qnormed. Quantile normalization will performed on the columns of `data`.
- **reference_index** (*int or str, optional*) – Pass a column index to serve as a reference distribution. Pass None to use the average of all distributions as the target distribution.

Returns The quantile normalized data.

Return type np.ndarray

Notes

This function is nan-safe. As long as each column of the input data contains the same number of nan's, nan's will only get averaged with other nan's, and they will get substituted back into their original positions. See the Examples section for an example of this.

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.qnorm import qnorm_fast
>>> qnorm_fast(np.array([[5,    4,    3],
...                      [2,    1,    4],
...                      [3,    4,    6],
...                      [4,    2,    8]]))
...
array([[5.66666667, 4.66666667, 2.        ],
```

(continues on next page)

(continued from previous page)

```
       [2.        , 2.        , 3.        ],
       [3.        , 4.66666667, 4.66666667],
       [4.66666667, 3.        , 5.66666667]])
>>> qnorm_fast(np.array([[     5, np.nan,      3],
...                      [     2,      1,      4],
...                      [np.nan,      4,      6],
...                      [     4,      2, np.nan]]))
...
array([[5.        ,        nan, 2.        ],
       [2.        , 2.        , 3.33333333],
       [       nan, 5.        , 5.        ],
       [3.33333333, 3.33333333,        nan]])
>>> qnorm_fast(np.array([[     5, np.nan,      3],
...                      [     2,      1,      4],
...                      [np.nan,      4,      6],
...                      [     4,      2, np.nan]]), reference_index=1)
...
array([[ 4., nan,  1.],
       [ 1.,  1.,  2.],
       [nan,  4.,  4.],
       [ 2.,  2., nan]])
```

lib5c.algorithms.qnorm.**qnorm_fast_parallel**(*data*, *reference_index=None*)

Quantile normalizes a data set.

Simpler, faster implementation compared to `lib5c.algorithms.qnorm()`, but only supports `tie='lowest'` behavior and only takes an `np.ndarray` as input. This approach was developed and timed in this repsitory.

Parallelizable if `data` is a 2d np.ndarray; see `lib5c.algorithms.qnorm.qnorm_fast_parallel()`.

Parameters

- **data** (*np.ndarray*) – Two dimensional, with the columns representing the replicates to be qnormed. Quantile normalization will performed on the columns of `data`.
- **reference_index** (*int or str, optional*) – Pass a column index to serve as a reference distribution. Pass None to use the average of all distributions as the target distribution.

Returns The quantile normalized data.

Return type np.ndarray

Notes

This function is nan-safe. As long as each column of the input data contains the same number of nan's, nan's will only get averaged with other nan's, and they will get substituted back into their original positions. See the Examples section for an example of this.

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.qnorm import qnorm_fast
>>> qnorm_fast(np.array([[5,    4,    3],
```

(continues on next page)

(continued from previous page)

```
...                       [2,    1,    4],
...                       [3,    4,    6],
...                       [4,    2,    8]]))
...
array([[5.66666667, 4.66666667, 2.        ],
       [2.        , 2.        , 3.        ],
       [3.        , 4.66666667, 4.66666667],
       [4.66666667, 3.        , 5.66666667]])
>>> qnorm_fast(np.array([[     5, np.nan,      3],
...                       [     2,      1,      4],
...                       [np.nan,      4,      6],
...                       [     4,      2, np.nan]]))
...
array([[5.        ,        nan, 2.        ],
       [2.        , 2.        , 3.33333333],
       [       nan, 5.        , 5.        ],
       [3.33333333, 3.33333333,        nan]])
>>> qnorm_fast(np.array([[     5, np.nan,      3],
...                       [     2,      1,      4],
...                       [np.nan,      4,      6],
...                       [     4,      2, np.nan]]), reference_index=1)
...
array([[ 4., nan,  1.],
       [ 1.,  1.,  2.],
       [nan,  4.,  4.],
       [ 2.,  2., nan]])
```

lib5c.algorithms.qnorm.**qnorm_parallel**(*data*, *tie='lowest'*, *reference_index=None*)

Quantile normalizes a data set.

Parallelizable if `data` is a 2d np.ndarray; see `lib5c.algorithms.qnorm.qnorm_parallel()`.

Parameters

- **data** (*2d numeric structure, or dict of 1d numeric structure*) – Anything that can be cast to array. Should be passed as row-major. Quantile normalization will performed on the columns of `data`.
- **tie** (*{'lowest', 'average'}, optional*) – Pass `'lowest'` to set all tied entries to the value of the lowest rank. Pass `'average'` to set all tied entries to the average value across the tied ranks.
- **reference_index** (*int or str, optional*) – If `data` is a row-major array, pass a column index to serve as a reference distribution. If `data` is a dict, pass a key of that dict that should serve as the reference distribution. Pass None to use the average of all distributions as the target distribution.

Returns The quantile normalized data. If `data` was passed as a dict, then a dict with the same keys is returned.

Return type 2d numpy array, or dict of 1d numpy array

Notes

This function is nan-safe. As long as each column of the input data contains the same number of nan's, nan's will only get averaged with other nan's, and they will get substituted back into their original positions. See the Examples section for an example of this.

Examples

```
>>> import numpy as np
>>> from lib5c.algorithms.qnorm import qnorm
>>> qnorm(np.array([[5,    4,    3],
...                 [2,    1,    4],
...                 [3,    4,    6],
...                 [4,    2,    8]]))
...
array([[5.66666667, 4.66666667, 2.        ],
       [2.        , 2.        , 3.        ],
       [3.        , 4.66666667, 4.66666667],
       [4.66666667, 3.        , 5.66666667]])
>>> qnorm(np.array([[    5, np.nan,      3],
...                 [    2,      1,      4],
...                 [np.nan,      4,      6],
...                 [    4,      2, np.nan]]))
...
array([[5.        ,        nan, 2.        ],
       [2.        , 2.        , 3.33333333],
       [       nan, 5.        , 5.        ],
       [3.33333333, 3.33333333,        nan]])
>>> qnorm(np.array([[    5, np.nan,      3],
...                 [    2,      1,      4],
...                 [np.nan,      4,      6],
...                 [    4,      2, np.nan]]), reference_index=1)
...
array([[ 4., nan,  1.],
       [ 1.,  1.,  2.],
       [nan,  4.,  4.],
       [ 2.,  2., nan]])
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]})
>>> list(sorted(res.items()))
[('A', array([5.66666667, 2.        , 3.        , 4.66666667])),
 ('B', array([4.66666667, 2.        , 4.66666667, 3.        ])),
 ('C', array([2.        , 3.        , 4.66666667, 5.66666667]))]
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]}, reference_index='C')
>>> list(sorted(res.items()))
[('A', array([8., 3., 4., 6.])),
 ('B', array([6., 3., 6., 4.])),
 ('C', array([3., 4., 6., 8.]))]
>>> res = qnorm({'A': [5, 2, 3, 4],
...              'B': [4, 1, 4, 2],
...              'C': [3, 4, 6, 8]}, reference_index='C', tie='average')
>>> list(sorted(res.items()))
[('A', array([8., 3., 4., 6.])),
 ('B', array([7., 3., 7., 4.])),
 ('C', array([3., 4., 6., 8.]))]
```

lib5c.algorithms.spline_normalization module

Module for fitting b-splines to 5C counts data as a method of bias correction.

class lib5c.algorithms.spline_normalization.**DiscreteBivariateEmpiricalSurface**(*xs*, *ys*, *zs*)

Bases: object

ev(*x*, *y*)

lib5c.algorithms.spline_normalization.**fit_spline**(*counts_list*, *primermap*, *bias_factor*, *knots=10*, *asymmetric=False*)

Fits a 2-D cubic b spline surface to the counts data as a function of the specified upstream and downstream bias factors.

Parameters

- **counts_list** (*List[Dict[str, np.ndarray]]*) – The counts data to fit the splines with.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap describing the loci. The bias_factor must be a key of the inner dict.
- **bias_factor** (*str*) – The bias factor to fit the model with.
- **knots** (*Optional[int]*) – The number of knots to use for the spline. If the bias factor is discrete, pass 0 to use an empirical discrete surface instead of a spline.
- **log** (*Optional[bool]*) – Pass true to fit the spline to logged data.
- **asymmetric** (*Optional[bool]*) – Pass True to iterate over only the upper triangular entries of the counts matrices. The default is False, which iterates over the whole counts matrices.

Returns List[Dict[str, np.ndarray]]] The first element of the tuple is the spline surface fit to the data. The second element contains the values of the spline surface evaluated at each point in the original counts dict. The third element contains the bias-corrected counts dicts.

Return type Tuple[LSQBivariateSpline, Dict[str, np.ndarray],

lib5c.algorithms.spline_normalization.**iterative_spline_normalization**(*counts_list*, *exp_list*, *primermap*, *bias_list*, *max_iter=100*, *eps=0.0001*, *knots=10*, *log=True*, *asymmetric=False*)

Convenience function for iteratively applying a set of spline normalization steps to a set of counts dicts.

Parameters

- **counts_list** (*List[Dict[str, np.ndarray]]*) – A list of observed counts dicts to normalize.
- **exp_list** (*List[Dict[str, np.ndarray]]*) – A list of expected counts dicts corresponding to the counts dicts in counts_list.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – Primermap or pixelmap describing the loci in this region.
- **bias_list** (*List[str]*) – A list of bias factors to remove from the counts. These strings must match metadata keys in primermap. That is to say, if bias_list is

`['length']` then we expect `primermap[region][i]['length']` to be a number representing the length of the `i` th fragment in the region specified by `region`. If multiple bias factors are specified, the algorithm will iteratively remove all of them from the data.

- **max_iter** (*int*) – The maximum number of iterations when iterating between bias factors.
- **eps** (*float*) – When the relative change in all models drops below this value convergence is declared.
- **knots** (*Union[int, List[int]]*) – Specifies the number of knots to put into the splines. Pass a single int to use the same number of knots in each model. Pass a list of ints of length equal to the length of `bias_list` to use `knots[i]` knots for the bias factor named `bias_list[i]`. If a bias factor is discrete, pass 0 for its knot number to use an empirical discrete surface instead of a spline.
- **log** (*bool*) – Pass True to fit the splines to log-scale data, reducing the effects of outliers.
- **asymmetric** (*bool*) – Pass True to construct models using only the upper-triangular elements of the counts matrices, which can lead to asymmetric models. By default, the algorithm iterates over all elements of the counts matrices, enforcing symmetry in the bias models but incurring some redundancy in the actual counts information.

Returns List[Dict[str, np.ndarray]], List[Dict[str, np.ndarray]]] The first element of the tuple is a dict mapping the bias factors specified in `bias_list` to BivariateSpline instances. The second element in the tuple is a dict mapping the bias factors specified in `bias_list` to counts dicts containing the evaluations of the spline fit to that bias factor at each point in the list of input counts dicts. The third element of the tuple is the normalized list of counts.

Return type Tuple[Dict[str, scipy.interpolate.BivariateSpline],

lib5c.algorithms.thresholding module

lib5c.algorithms.thresholding.**color_confusion**(*d*)
Extract the across-condition color confusion matrix.

Parameters **d** (Dataset) – Dataset processed by *two_way_thresholding()*.

Returns The 2x2 confusion matrix.

Return type np.ndarray

lib5c.algorithms.thresholding.**concordance_confusion**(*d*)
Extract the within-condition concordance confusion matrices.

Parameters **d** (Dataset) – Dataset processed by *two_way_thresholding()*.

Returns The keys are condition names, the values are the 2x2 confusion matrices.

Return type dict

lib5c.algorithms.thresholding.**count_clusters**(*d*)
Extract the final cluster counts.

Parameters **d** (Dataset) – Dataset processed by *two_way_thresholding()* called with *report_clusters=True*.

Returns The keys are the color names as strings, the values are integers representing the cluster counts.

Return type dict

lib5c.algorithms.thresholding.**filter_near_diagonal**(*df*, *distance=24000*, *drop=True*)
Drops rows from df where its 'distance' column is less than k.

Dropping occurs in-place.

Parameters

- **df** (*pd.DataFrame*) – Must have a 'distance' column.
- **distance** (*int*) – Threshold for distance (in bp).
- **drop** (*bool*) – Pass True to drop the filtered rows in-place. Pass False to return an index subset for the filtered rows instead.

lib5c.algorithms.thresholding.**kappa**(*d*)
Compute the Cohen's kappa values between the replicates of each condition.

Parameters **d** (Dataset) – Dataset processed by *two_way_thresholding()*.

Returns The keys are condition names, the values are the kappa values.

Return type dict

lib5c.algorithms.thresholding.**label_connected_components**(*colors*, *color*)
Labels the connected components of a specific loop color.

Parameters

- **colors** (*np.ndarray with string dtype*) – The matrix of colors.
- **color** (*str*) – The color to label.

Returns Same size and shape as colors, entries are ints which are the labels

Return type np.ndarray

Examples

```
>>> colors = np.array([['a', 'a', 'b', 'a'],
...                    ['a', 'a', 'b', 'b'],
...                    ['b', 'b', 'b', 'a'],
...                    ['a', 'b', 'a', 'a']])
>>> print(label_connected_components(colors, 'a'))
[[1 1 0 2]
 [1 1 0 0]
 [0 0 0 3]
 [2 0 3 3]]
```

lib5c.algorithms.thresholding.**size_filter**(*calls*, *threshold*)
Removes calls which are in connected components smaller than a threshold.

Parameters

- **calls** (*np.ndarray*) – Boolean matrix of calls.
- **threshold** (*int*) – Connected components smaller than this will be removed.

Returns The filtered calls.

Return type np.ndarray

Examples

```
>>> calls = np.array([[ True,  True, False,  True],
...                   [ True,  True, False, False],
...                   [False, False, False,  True],
...                   [ True, False,  True,  True]])
>>> size_filter(calls, 3)
array([[ True,  True, False, False],
       [ True,  True, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

lib5c.algorithms.thresholding.**two_way_thresholding**(*pvalues_superdict*, *primermap*, *conditions=None*, *significance_threshold=1e-15*, *bh_fdr=False*, *two_tail=False*, *concordant=False*, *distance_threshold=24000*, *size_threshold=3*, *background_threshold=0.6*, *report_clusters=True*)

All-in-one heavy-lifting function for thresholding.

Parameters

- **pvalues_superdict** (*dict of dict of np.ndarray*) – The p-values to threshold.
- **primermap** (*primermap*) – The primermap associated with the pvalues_superdict.
- **conditions** (*list of str, optional*) – The list of condition names. Pass None to skip condition comparisons.
- **significance_threshold** (*float*) – The p-value or q-value to threshold significance with.
- **bh_fdr** (*bool*) – Pass True to apply multiple testing correction (BH-FDR) before checking the `significance_threshold`.
- **two_tail** (*bool*) – If `bh_fdr=True`, pass True here to perform the BH-FDR on two-tailed p-values, but only report the significant right-tail events as loops. Note that two-tailed p-values are only accurate if p-values were called using a continuous distribution.
- **concordant** (*bool*) – Pass True to report only those interactions which are significant in all replicates in each condition. Pass False to combine evidence from all replicates within each condition instead.
- **distance_threshold** (*int*) – Interactions with interaction distance (in bp) shorter than this will not be called.
- **size_threshold** (*int*) – Interactions within connected components smaller than this will not be called.
- **background_threshold** (*float, optional*) – The p-value threshold to use to call a background loop class. Pass None to skip calling a background class.
- **report_clusters** (*bool*) – Pass True to perform a second pass of connected component counting at the very end, reporting the numbers of clusters in each color category to the returned Dataset.

Returns The results of the thresholding.

Return type *Dataset*

lib5c.algorithms.trimming module

Module for trimming low or "dead" 5C fragments away from 5C datasets.

lib5c.algorithms.trimming.**trim_counts**(*counts*, *indices*)
Removes specified rows and columns from the counts matrix.

Parameters

- **counts** (*np.ndarray*) – The square symmetric counts matrix to trim.
- **indices** (*Iterable[int]*) – The indices to wipe

Returns The trimmed counts matrix.

Return type np.ndarray

lib5c.algorithms.trimming.**trim_counts_superdict**(*counts_superdict*, *indices*)
Applies `trim_counts()` to each replicate in a `counts_superdict`.

Parameters

- **counts_superdict** (*Dict[str, np.ndarray]*) – The keys are replicate names, the values are the counts for that rep.
- **indices** (*Iterable[int]*) – The indices to trim.

Returns The keys are replicate names, the values are the trimmed counts for that rep.

Return type Dict[str, np.ndarray]

lib5c.algorithms.trimming.**trim_primers**(*primermap*, *counts_superdict*, *min_sum=100.0*, *min_frac=0.5*)
Trim a primermap using counts information from many replicates.

Parameters

- **primermap** (*List[Dict[str, Any]]*) – The primermap to trim. See `lib5c.parsers.primers.get_primermap()`.
- **counts_superdict** (*Dict[str, np.ndarray]*) – The keys are replicate names, the values are the counts for that rep.
- **min_sum** (*Optional[float]*) – Primers with a total cis sum lower than this value will be trimmed.
- **min_frac** (*Optional[float]*) – Primers with fewer than this fraction of nonzero interactions out of all their finite interactions will be trimmed.

Returns The first element is the trimmed primermap, the second is the set of indices of the original primermap which were removed.

Return type Tuple[List[Dict[str, Any]], Set[int]]

lib5c.algorithms.trimming.**wipe_counts**(*counts*, *indices*, *wipe_value=nan*)
Wipes specified rows and columns of the counts matrix with a specified value.

Parameters

- **counts** (*np.ndarray*) – The square symmetric counts matrix to wipe.
- **indices** (*Iterable[int]*) – The indices of the rows and columns to wipe.
- **wipe_value** (*Optional[float]*) – The value to wipe the selected indices with.

Returns The wiped counts matrix.

Return type np.ndarray

lib5c.algorithms.trimming.**wipe_counts_superdict**(*counts_superdict*, *indices*, *wipe_value=nan*)

Applies `wipe_counts()` to each replicate in a `counts_superdict`.

Parameters

- **counts_superdict** (*Dict[str, np.ndarray]*) – The keys are replicate names, the values are the counts for that rep.
- **indices** (*Iterable[int]*) – The indices to wipe
- **wipe_value** (*Optional[float]*) – The value to wipe the selected indices with.

Returns The keys are replicate names, the values are the wiped counts for that rep.

Return type Dict[str, np.ndarray]

Module contents

Subpackage containing algorithms for 5C data analysis.

Subpackage structure:

- *lib5c.algorithms.clustering* - algorithms for clustering together 5C interactions
- lib5c.algorithms.distributions - statistical modeling of 5C contact frequencies
- *lib5c.algorithms.filtering* - binning, smoothing, etc.
- *lib5c.algorithms.correlation* - pairwise correlations between replicates
- *lib5c.algorithms.determine_bins* - computing bins to tile 5C regions
- *lib5c.algorithms.donut_filters* - "donut" expected model correction
- *lib5c.algorithms.enrichment* - computing enrichments for genomic annotations between different 5C interaction classes
- *lib5c.algorithms.expected* - building expected models for 5C data
- *lib5c.algorithms.express* - "Express" normalization algorithms
- *lib5c.algorithms.knight_ruiz* - Knight-Ruiz matrix balancing
- *lib5c.algorithms.outliers* - finding and removing high spatial outliers in 5C contact matrices
- *lib5c.algorithms.qnorm* - quantile normalization standardize distributions between replicates
- *lib5c.algorithms.spline_normalization* - spline-based bias factor normalization
- *lib5c.algorithms.trimming* - removing "dead" or low-count primers from 5C datasets

lib5c.contrib package

Subpackages

lib5c.contrib.iced package

Submodules

lib5c.contrib.iced.balancing module

Module for interfacing with the external `iced` Python package, which provides access to the ICED matrix balancing algorithm.

`lib5c.contrib.iced.balancing.`**`iced_balance_matrix`**(*matrix*, *max_iter=3000*, *eps=0.0001*, *norm='l1'*, *imputation_size=0*)

Convenience function wrapping the `ICE_normalization` function from the external `iced` Python package, which balances a counts matrix using the ICE algorithm.

Parameters

- **`matrix`** (*`np.ndarray`*) – The counts matrix to balance.
- **`max_iter`** (*`int`*) – The maxiumum number of iterations to try.
- **`eps`** (*`float`*) – The relative size of error before declaring convergence.
- **`norm`** (*`{'l1', 'l2'}`*) – What norm to use as a distance measure.
- **`imputation_size`** (*`int`*) – Pass an int greater than 0 to replace NaN's in the matrix with a local median approximation. Pass 0 to skip imputation.

Returns The first element of the tuple is the balanced matrix. The second element is the bias vector.

Return type Tuple[np.ndarray, np.ndarray]

Module contents

lib5c.contrib.interlap package

Submodules

lib5c.contrib.interlap.util module

Module containing utilities for interfacing with InterLap objects from the external `interlap` Python package, which provides efficient binary interval search for finding overlapping genomic features.

`lib5c.contrib.interlap.util.`**`features_to_interlaps`**(*features*, *chroms=None*)

Converts feature dicts to InterLap objects.

Parameters

- **`features`** (*`dict of list of dict`*) – The keys of the outer dict should be chromosome names as strings. The values of the outer dict represent lists of features found on that chromosome. The inner dicts represent individual genomic features, with at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end'  : int
}
```

See `lib5c.parsers.load_features()` for more information.

- **chroms** (*list of str, optional*) – To create InterLap objects for only specified chromosomes, pass a list of their names. Pass None to create InterLap objects for all chromosomes.

Returns The keys are chromosome names as strings, the values are InterLap objects containing all features on the chromosome. The original feature dicts are saved in the data element of each interval in the InterLap.

Return type dict of InterLap

lib5c.contrib.interlap.util.**query_interlap**(*interlap*, *query_feature*)
Searches an InterLap object to find features that overlap a given query feature.

Parameters

- **interlap** (*InterLap*) – The InterLap object to search. Each interval in the InterLap object must have a data element, see `lib5c.contrib.interlap.util.features_to_interlaps()`.
- **query_feature** (*dict*) – Dict representing the genomic region in which to search for overlapping features. Must have at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end'  : int
}
```

Returns Each dict in the list represents a feature found in the InterLap object that overlaps the query feature.

Return type list of dict

Module contents

lib5c.contrib.luigi package

Submodules

lib5c.contrib.luigi.config module

This module provides a single string literal that is used to represent the default tree pipeline configuration file. It also provides a function to write this default configuration file to the disk.

lib5c.contrib.luigi.config.**drop_config_file**()
Drops the default config file in the current directory.

lib5c.contrib.luigi.pipeline module

Module implementing one particular strategy for wiring together the luigi Task subclasses defined in `lib5c.contrib.luigi.tasks` into a complete pipeline.

The pipeline is organized as a tree of Tasks, which matches perfectly with a tree of output directories. Each Task in the tree inherits from the mixin class TreeMixin and defines a `directory` string parameter. This parameter represents the output directory for that Task. Task classes can be reconstituted from directory strings via the `directory_to_task()` function.

The `directory_to_task()` function uses the `table` DictParameter of the TreeMixin, which maps user-selected short names for parameterized Tasks to Task class names as well as detailed parameters. An example of an entry in the `table` is:

```
"bin_amean_20_8": ["MakeBinned", {"window_function": "amean",
                                  "bin_width": 8000,
                                  "window_width": 20000}]
```

where the key, "bin_amean_20_8", is the user-selected short name for this particular parameterization of the MakeBinned Task class, and the value is a list of two elements. The first element is the Task class name as a string (in this case, MakeBinned, which extends `lib5c.contrib.luigi.tasks.BinTask` and mixes in TreeMixin). The second element is a dict containing the parameters to construct the Task with. With this entry in the table, when a folder named "bin_amean_20_8" occurs within the directory string, it will be interpreted as a MakeBinned Task with the parameters specified in this table entry.

The upstream Task that a particular Task depends on (i.e., its parent in the tree) can also be reconstituted by splitting off the last folder level in the directory string and calling `directory_to_task()` on what remains. This logic is implemented in `TreeMixin.preceding_task()` which allows any Task in the tree to know what tasks precede it in the pipeline.

TreeMixin also describes `rep` and `outfile_pattern` parameters. Together with `directory`, these parameters specify the exact output file of running a particular parameterized Task on one specific replicate, using the logic implemented in `TreeMixin.output()`.

The pipeline is orchestrated by an overall WrapperTask called PipelineTask which stores the `table` and passes it through to each TreeMixin Task. It also deduces the `all_reps` list (by peeking at the keys of `RawCounts.countsfiles` using the luigi config file) and passes it through to each TreeMixin Task as well. It stores a list of directory strings (representing leaf Tasks) in a `tasks` ListParameter. As a WrapperTask, it wraps all the leaf Tasks in `tasks` and all replicates in `all_reps` as appropriate. The leaf Tasks in turn use their `directory` strings to figure out what Tasks they depend on. In this way the entire tree of pipeline Tasks is created from just one PipelineTask.

class lib5c.contrib.luigi.pipeline.**DetermineBins**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.tasks.DetermineBinsTask*

Pipeline Task for DetermineBinsTask (the step which decides how to bin the 5C regions).

This Task is pre-wired to depend on the PrimeFile pipeline Task, and to write its output to an output folder called *bedfiles/*.

output()

Implementation of `output()`, pre-wired to write the output to the *bedfiles/* folder.

Returns The Target of this Task.

Return type luigi.Target

requires()

Implementation of `requires()`, pre-wired to depend on the PrimerFile pipeline Task.

Returns The Task that this Task depends on.

Return type luigi.Task

class lib5c.contrib.luigi.pipeline.**JointExpressInnerTask**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.JointInnerParallelMixin*, *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.ExpressTask*

Inner Task class for the MakeJointExpress JointTask.

class lib5c.contrib.luigi.pipeline.**JointInnerMixin**

Bases: `object`

Mixin class for inner Tasks wrapped by JointTask.

The inner Task of a JointTask depends on the preceding Task's output for all replicates.

This mixin provides a helper function `_match_input()` which subclasses can use to get a glob-based pattern that matches all the input files for the Task which precedes this Task. CmdTasks inheriting from this mixin only need to use this approach if they must describe all their input files using a single string (see JointExpressInnerTask for an example). CmdTasks that can simply list the exact input files they depend on can use something like:

```
[i.path for i in self.input()]
```

See QnormInnerTask for an example of this second approach.

A basic implementation of `requires()` is provided here and should work in most cases, but Task classes inheriting from JointInnerMixin must still define their own implementation of `output()`.

requires()
Basic implementation of `requires()` for inner Tasks of a JointTask.

This basic implementation assumes that the inner Task depends on the locus file and the preceding Task for each replicate in `all_reps`.

Subclasses may override this if they depend on more than just these inputs.

Returns The Tasks that this inner Task depends on.

Return type list of luigi.Task

class lib5c.contrib.luigi.pipeline.**JointInnerParallelMixin**
Bases: *lib5c.contrib.luigi.pipeline.JointInnerMixin*

Mixin class providing a simple implementation of `output()` for Task classes inheriting from JointInnerMixin.

output()
Simple implementation of `output()` for Task classes inheriting from JointInnerMixin.

This implemntation assumes that the output files are parallel to the input files (i.e., there is one for each replicate and it can be obtained by interpolating `rep` into the `outfile_pattern`).

Returns The Targets of this inner Task.

Return type list of luigi.Target

class lib5c.contrib.luigi.pipeline.**JointTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, luigi.task.WrapperTask

Mixin class for pipeline Tasks that operate on input from all replicates.

Tasks inheriting from JointTask become WrapperTasks, one of which can be created for each replicate, but each of which will depend on the same inner Task which does the actual work. In terms of the overall pipeline flow, this allows a piece of `directory` to map to a JointTask, which can be instantiated once for each replicate via the `rep` kwarg of the TreeMixin. All the JointTask instances will depend on a single inner Task inheriting from JointInnerMixin that actually does the work.

Tasks inheriting from JointTask must implement `get_inner_task_class()`, which should return a Task class which inherits from JointInnerMixin and actually does the work.

Since `get_inner_task_class()` just returns a Task class which must still be instantiated with the proper parameters, JointTask provides an overrideable hook, `get_inner_task_params()` to allow Task classes which inherit from JointTask to manually pass their parameters through to the inner Task. See `MakeQnorm.get_inner_task_params()` for an example.

The related helper function `get_inner_task_param_dict()` helps to simplify this process by automatically passing through key TreeMixin parameters like `table`, `directory`, `all_reps`, and the `@visualizable` visualization hook parameters.

get_inner_task_class()

get_inner_task_param_dict()
Constructs the complete dict of params for inner task instantiation.

Provides some important core defaults in the context of the tree pipeline, and injects whatever parameters are returned by `get_inner_task_params()`.

This is a helper function - subclasses should not override this function and should override `get_inner_task_params()` instead.

Returns The complete dict of params.

Return type dict

get_inner_task_params()
Hook to allow subclasses to supply extra parameters to their inner Tasks. Subclasses should override this function.

Returns Extra parameters to be supplied to the inner task upon construction.

Return type dict

get_rep_index()
Returns the index of the replicate this Task wraps the output for among all the replicates (in the order of `self.all_reps`).

Returns The index of the replicate this Task wraps the output for among all the replicates (in the order of `self.all_reps`).

Return type int

outfile_pattern = <luigi.parameter.Parameter object>

output()
Universal implementation of `output()` for JointTasks.

This implementation simply instantiates the inner Task and asks it for its outputs, returning the one that corresponds to the replicate of this JointTask. The assumption here is that the inner Task class's `output()` will be a list whose elements correspond to the replicates in `all_reps`.

Returns The Target of this JointTask.

Return type luigi.Target

rep = <luigi.parameter.Parameter object>

requires()
Universal implementation of `requires()` for JointTasks.

Simply put, the JointTask depends on its inner Task class, instantiated using the parameters obtained from `get_inner_task_params()` via `get_inner_task_param_dict()`.

Returns The Task instance of the inner Task that this WrapperTask depends on.

Return type luigi.Task

class lib5c.contrib.luigi.pipeline.**MakeBinned**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.BinTask*

Pipeline Task class for the binning step.

Unlike most countsfile-to-countsfile steps, the binning step needs to use two different locus Tasks as input: the primerfile and the binfile. Therefore, this class must provide a custom implementation of `requires()` to specify this.

bin_width = <luigi.parameter.IntParameter object>

requires()

Depends on both the binfile (represented by a DetermineBins instance) and the primerfile (represented by the PrimerFile instance) in addition to the preceding Task.

Returns The Tasks this Task depends on.

Return type tuple of luigi.Task

class lib5c.contrib.luigi.pipeline.**MakeCrossVariance**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.CrossVarianceTask*

Pipeline Task for the cross-replicate variance modeling step.

Even though this Task depends on multiple replicates, it is not implemented as a JointTask.

requires()

Depends on the preceding Task for the same replicate (assumed to be the expected counts) and the Task that precedes that Task (assumed to be the observed counts) for all replicates in this Task's condition.

This Task's condition is inferred to be the first condition in the comma-separated string parameter `conditions` that is a substring of `rep`. Other replicates match this condition if this condition is also a substring of their replicate names.

Returns The Tasks this Task depends on. The first Task is the locus info Task, the second is the expected Task for this replicate, and the remaining Tasks in the list are observed Tasks for all replicates in the same condition as this replicate.

Return type list of luigi.Task

class lib5c.contrib.luigi.pipeline.**MakeExpected**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.ExpectedTask*

Pipeline Task class for the expected modeling step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeExpress**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.ExpressTask*

Pipeline Task class for the express step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeIced**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.IcedTask*

Pipeline Task class for the ICED balancing step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeInteractionScores**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.InteractionScoreTask*

Pipeline Task class for InteractionScoreTask. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeJointExpress**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.JointTask*

Outer wrapper pipeline JointTask for the joint express step.

get_inner_task_class()
Points to JointExpressInnerTask, the inner Task for the joint express step.

Returns The inner Task class for this JointTask.

Return type luigi.Task

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.pipeline.**MakeKR**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.KnightRuizTask*

Pipeline Task class for the Knight-Ruiz balancing step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeLegacyPvaluesOne**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.LegacyPvaluesOneTask*

Pipeline Task for an old version of the p-value calling step. Deprecated.

requires()
Unlike the modern PvaluesTask which depends on obs, exp, and var, this old version only used the obs and the exp.

class lib5c.contrib.luigi.pipeline.**MakeLogged**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.LogTask*

Pipeline Task class for LogTask. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeObsMinusExp**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.SubtractTask*

Pipeline Task class for the obs-exp step (analogous to the obs/exp step but for data that have already been log-transformed).

requires()
Depends on both the preceding Task (assumed to be the expected counts) and the Task that precedes that Task (assumed to be the observed counts).

Returns The Tasks this Task depends on.

Return type tuple of luigi.Task

class lib5c.contrib.luigi.pipeline.**MakeObsOverExp**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.DivideTask*

Pipeline Task class for the obs/exp step.

requires()
Depends on both the preceding Task (assumed to be the expected counts) and the Task that precedes that Task (assumed to be the observed counts).

Returns The Tasks this Task depends on.

Return type tuple of luigi.Task

class lib5c.contrib.luigi.pipeline.**MakePvalues**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.PvalueTask*

Pipeline Task for the p-value calling step.

requires()
Depends on three Tasks: the preceding Task (assumed to be the variance counts), the Task that precedes that Task (assumed to be the expected counts) and the Task that precedes that Task (assumed to be the observed counts).

Returns The Tasks this Task depends on.

Return type tuple of luigi.Task

class lib5c.contrib.luigi.pipeline.**MakeQnorm**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.JointTask*

Outer wrapper pipeline JointTask for the qnorm step.

averaging = <luigi.parameter.BoolParameter object>

condition_on = <luigi.parameter.Parameter object>

get_inner_task_class()
Points to QnormInnerTask, the inner Task for the qnorm step.

Returns The inner Task class for this JointTask.

Return type luigi.Task

get_inner_task_params()
Passes through all the parameters for the qnorm step.

Returns The parameters for the qnorm step.

Return type dict

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

reference = <luigi.parameter.Parameter object>

regional = <luigi.parameter.BoolParameter object>

run()

class lib5c.contrib.luigi.pipeline.**MakeQvalues**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.QvaluesTask*

Pipeline Task class for the multiple testing correction step, which converts p-values to q-values. All functionality is handled by PerRepSimpleTreeMixin.

Note that the thresholding step performs its own multiple testing correction when parameterized with `bh_fdr=True`, so this step is never required.

class lib5c.contrib.luigi.pipeline.**MakeRaw**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, `luigi.task.Task`

Pipeline Task for performing the "raw" step of the pipeline.

This step doesn't actually do anything, so it just copies over the input countsfile (which is actually represented by a RawCounts Task) into the output directory tree. By having a separate step for this we guarantee that a) a raw countsfile can be found with a predictable name (in agreement with the replicate names which are set by the

keys of `RawCounts.countsfiles`) and in a predictable spot in the output directory structure, and b) the raw countsfile can be visualized using the same visualization hooks as any other step.

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.pipeline.**MakeRemoved**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.OutliersTask*

Pipeline Task class for the high outlier removal step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeSmoothed**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.SmoothTask*

Pipeline Task class for the smoothing step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeSpline**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.PerRepSimpleTreeMixin*, *lib5c.contrib.luigi.tasks.SplineTask*

Pipeline Task class for the explicit spline normalization step. All functionality is handled by PerRepSimpleTreeMixin.

class lib5c.contrib.luigi.pipeline.**MakeThreshold**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.JointInnerMixin*, *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.ThresholdTask*

Pipeline Task for the loop call thresholding step.

This Task is implemented as if it were the inner Task of a JointTask, but since there is only one ThresholdTask for all replicates, it does not need a corresponding WrapperTask to wrap itself across replicates.

It gets its implementation of `requires()` from JointInnerMixin, which correctly depends on the output of the preceding Task (assumed to be the p-values) across `all_reps`.

output()

Specifies the output file locations for the thresholding step.

These locations are controlled by the `outfile_pattern` (countsfile of final cluster assignments), `dataset_outfile` (table of complete results), and `kappa_confusion_outfile` (text file of summary information and concordance metrics).

Returns The Targets resulting from this Task.

Return type tuple of luigi.Target

class lib5c.contrib.luigi.pipeline.**MakeVariance**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.VarianceTask*

Pipeline Task for the variance modeling step.

requires()

Depends on both the preceding Task (assumed to be the expected counts) and the Task that precedes that Task (assumed to be the observed counts).

Returns The Tasks this Task depends on.

Return type tuple of luigi.Task

class lib5c.contrib.luigi.pipeline.**PerRepSimpleTreeMixin**
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*

Mixin class that adds the most common implementation of `requires()` to TreeMixin.

Most pipeline Tasks depend on two inputs: a primer or binfile, and the immediately preceding countsfile for the rep of the child Task.

Pipeline Tasks that depend on more than one countsfile (e.g., p-value calling), or all replicates (e.g., thresholding) cannot use this mixin, and instead must inherit from TreeMixin and define their own implementation of `requires()`.

requires()

class lib5c.contrib.luigi.pipeline.**PipelineTask**(**args*, ***kwargs*)
Bases: `luigi.task.WrapperTask`

Overall wrapper Task that orchestrates the entire pipeline.

Running this Task runs every leaf Task in the `tasks` ListParameter as well as all parent Tasks needed to get from the root (raw input countsfiles) to those leaves.

Tasks should be specified in the `tasks` ListParameter in the form of directory strings to the leaf Tasks (final step in a chain of Tasks).

Individual folders in the directory strings in `tasks` will be converted to properly parameterized Task instances via the `table` DictParameter, which should map folder names to lists of two items: the appropriate pipeline Task class name as a string, and a dict of parameters to instantiate that Task class with. See the module docstring for an example.

The leaf Tasks will automatically be parallelized across `all_reps` unless they are MakeThreshold (the Task class for which `rep` is always None).

requires()
Deduces `all_reps` and wraps all the leaf Tasks in `tasks` over all replicates if appropriate, passing though `table` and `all_reps`.

table = <luigi.parameter.DictParameter object>

tasks = <luigi.parameter.ListParameter object>

class lib5c.contrib.luigi.pipeline.**PrimerFile**(**args*, ***kwargs*)
Bases: `luigi.task.ExternalTask`

Pipeline Task for finding the input primerfile on the disk.

output()
Implementation of `output()`.

Returns A LocalTarget pointing to this Task's `primerfile` parameter, which should be the location of the input primerfile on the disk.

Return type luigi.Target

primerfile = <luigi.parameter.Parameter object>

class lib5c.contrib.luigi.pipeline.**QnormInnerTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.JointInnerParallelMixin*, *lib5c.contrib.luigi.pipeline.TreeMixin*, *lib5c.contrib.luigi.tasks.QnormTask*

Inner Task class for the MakeQnorm JointTask.

class lib5c.contrib.luigi.pipeline.**RawCounts**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.pipeline.TreeMixin*, `luigi.task.ExternalTask`

Pipeline Task for finding the raw input countsfiles on the disk.

This step is not resolved through the `table`, but instead uses its own DictParameter `countsfiles` which should map replicate names to the paths of the raw input countsfiles on the disk.

countsfiles = <luigi.parameter.DictParameter object>

outfile_pattern = <luigi.parameter.Parameter object>

output()
Looks up the location of the countsfile for this replicate using the `countsfiles` DictParameter and returns a LocalTarget pointing to it.

Returns The Target corresponding to the raw input countsfile represented by this Task.

Return type luigi.Target

rep = <luigi.parameter.Parameter object>

class lib5c.contrib.luigi.pipeline.**TreeMixin**
Bases: `object`

Core mixin class for pipeline Tasks. See the module docstring for more details.

If mixed with a `lib5c.contrib.luigi.tasks.CmdTask` subclass, the only luigi function that the derived class needs to implement is `requires()`.

all_reps = <luigi.parameter.ListParameter object>

directory = <luigi.parameter.Parameter object>

locus_info_task()
Returns the Task instance corresponding to the primerfile or binfile needed by this Task.

Returns The Task instance corresponding to the primerfile or binfile needed by this Task.

Return type luigi.Task

outfile_pattern = <luigi.parameter.Parameter object>

output()
Returns the luigi Target corresponding to the output file that is the direct result of running this Task.

Returns The Target corresponding to the output file that is the direct result of running this Task.

Return type luigi.Target

preceding_task(*rep=None*)
Returns the Task instance that precedes this Task.

Parameters **rep** (*str, optional*) – The replicate name to parameterize the parent Task with. Pass None if the Task is not a per-rep Task.

Returns The Task instance that precedes this Task.

Return type luigi.Task

rep = <luigi.parameter.Parameter object>

table = <luigi.parameter.DictParameter object>

lib5c.contrib.luigi.pipeline.**directory_to_task**(*directory*, *table*, *all_reps*, ***kwargs*)
Converts a directory to a TreeMixin Task class instance, using a provided table.

Parameters

- **directory** (*str*) – The directory identifying this task.

- **table** (*Dict[str, Tuple[str, dict[str, Any]]]*) – A map from directory parts to (Task class name, param dict) tuples.
- **all_reps** (*List[str]*) – A list of all the replicates.
- **kwargs** (*additional keyword arguments*) – Will be passed to the new Task instance. The most common kwarg is 'rep'.

Returns The specified Task instance.

Return type luigi.Task

lib5c.contrib.luigi.tasks module

Provides luigi Task subclasses that wrap the lib5c command line functions.

class lib5c.contrib.luigi.tasks.**BinTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.FilteringTask*

Task class for binning fragment-level countsfiles into binned countsfiles.

Wraps the `lib5c bin` command line command.

Input/output specification:

- `self.input()[0]`: the bin .bed file
- `self.input()[1]`: the primer .bed file
- `self.input()[2]`: the input fragment-level countsfile
- `self.output()`: the resulting countsfile of binned observed values

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**CmdTask**(**args*, ***kwargs*)
Bases: `luigi.task.Task`

Luigi Task parent class for Tasks whose `run()` behavior should be to execute a specific command on the command line.

Subclasses must implement `_construct_cmd_string()`, which should return a string corresponding to the command to be run on the command line.

If the `bsub` Python package is installed, the command will be executed using the bsub scheduling system, and the caller will wait for the job corresponding to the task to complete.

If the `bsub` Python package is not installed, the command will be simply executed via `subprocess`.

run()
Generic `run()` implementation for command line Tasks.

class lib5c.contrib.luigi.tasks.**CrossVarianceTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.VarianceTask*

Task class for computing variance estimates using the cross-replicate variance method.

Wraps the `lib5c variance` command line command called with `-s/--source cross_rep`.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file

- `self.input()[1]`: the input expected countsfile
- `self.input()[2:]`: the input observed countsfiles for each replicate
- `self.output()`: the resulting countsfile of variance estimates

This class defines a `conditions` Parameter which should be used to ensure that the input observed countsfiles passed in `self.input()[2:]` all belong to the same condition. This logic is not implemented here.

conditions = <luigi.parameter.Parameter object>

source = <luigi.parameter.Parameter object>

class lib5c.contrib.luigi.tasks.**DetermineBinsTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for determining bin locations.

Wraps the `lib5c determine-bins` command line command.

Input/output specification:

- `self.input()`: the input primer .bed file
- `self.output()`: the resulting bin .bed file

bin_width = <luigi.parameter.IntParameter object>

class lib5c.contrib.luigi.tasks.**DistributionTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

dist = <luigi.parameter.Parameter object>

log = <luigi.parameter.BoolParameter object>

mode = <luigi.parameter.Parameter object>

class lib5c.contrib.luigi.tasks.**DivideTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for dividing one countsfile by another.

Wraps the `lib5c divide` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the dividend (countsfile to divide)
- `self.input()[2]`: the divisor (countsfile to divide by)
- `self.output()`: the quotient (countsfile resulting from the division)

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**ExpectedTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for computing expected models.

Wraps the `lib5c expected` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file

- `self.input()[1]`: the input observed countsfile
- `self.output()`: the resulting countsfile of expected values

degree = <luigi.parameter.IntParameter object>

donut = <luigi.parameter.BoolParameter object>

donut_frac = <luigi.parameter.FloatParameter object>

exclude_near_diagonal = <luigi.parameter.BoolParameter object>

global_expected = <luigi.parameter.BoolParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

log_donut = <luigi.parameter.BoolParameter object>

log_transform = <luigi.parameter.Parameter object>

lowess = <luigi.parameter.BoolParameter object>

lowess_frac = <luigi.parameter.FloatParameter object>

max_with_lower_left = <luigi.parameter.BoolParameter object>

min_exp = <luigi.parameter.FloatParameter object>

monotonic = <luigi.parameter.BoolParameter object>

p = <luigi.parameter.IntParameter object>

plot_outfile = <luigi.parameter.Parameter object>

plot_outfile_hexbin = <luigi.parameter.BoolParameter object>

plot_outfile_kde = <luigi.parameter.BoolParameter object>

powerlaw = <luigi.parameter.BoolParameter object>

regression = <luigi.parameter.BoolParameter object>

run()

w = <luigi.parameter.IntParameter object>

class lib5c.contrib.luigi.tasks.**ExpressTask**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying Express bias correction to countsfiles.

Wraps the `lib5c express` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile
- `self.output()`: the resulting Express-normalized countsfile

bias = <luigi.parameter.BoolParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**FilteringTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Parent Task class for Tasks related to binning and smoothing.

inverse_weights = <luigi.parameter.BoolParameter object>

threshold = <luigi.parameter.FloatParameter object>

window_function = <luigi.parameter.Parameter object>

window_width = <luigi.parameter.IntParameter object>

wipe_unsmoothable_columns = <luigi.parameter.BoolParameter object>

class lib5c.contrib.luigi.tasks.**IcedTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying ICED bias correction to countsfiles.

Wraps the `lib5c iced` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile
- `self.output()`: the resulting ICED-normalized countsfile

bias = <luigi.parameter.BoolParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

imputation_size = <luigi.parameter.IntParameter object>

run()

class lib5c.contrib.luigi.tasks.**InteractionScoreTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for converting p-values to interaction scores.

Wraps the `lib5c interaction-score` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile of p-values
- `self.output()`: the resulting countsfile of interaction scores

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**KnightRuizTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying KR bias correction to countsfiles.

Wraps the `lib5c kr` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile
- `self.output()`: the resulting KR-normalized countsfile

bias = <luigi.parameter.BoolParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

imputation_size = <luigi.parameter.IntParameter object>

run()

class lib5c.contrib.luigi.tasks.**LegacyPvaluesOneTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.DistributionTask*

bias = <luigi.parameter.BoolParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**LegacyPvaluesTwoTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

bias = <luigi.parameter.BoolParameter object>

dist = <luigi.parameter.Parameter object>

distance_tolerance = <luigi.parameter.IntParameter object>

fractional_tolerance = <luigi.parameter.FloatParameter object>

grouping = <luigi.parameter.Parameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

log = <luigi.parameter.BoolParameter object>

mode = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**LegacyVisualizeFitTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.DistributionTask*, *lib5c.contrib.luigi.tasks.RegionalTaskMixin*

distance_scale = <luigi.parameter.IntParameter object>

expected_value = <luigi.parameter.FloatParameter object>

tolerance = <luigi.parameter.FloatParameter object>

class lib5c.contrib.luigi.tasks.**LegacyVisualizeVarianceTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.DistributionTask*, *lib5c.contrib.luigi.tasks.RegionalTaskMixin*

class lib5c.contrib.luigi.tasks.**LogTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for logging or unlogging a countsfile.

Wraps the `lib5c log` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile (to be logged)
- `self.output()`: the resulting countsfile (after logging)

log_base = <luigi.parameter.Parameter object>

pseudocount = <luigi.parameter.FloatParameter object>

unlog = <luigi.parameter.BoolParameter object>

class lib5c.contrib.luigi.tasks.**OutliersTask**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying high outlier removal to countsfiles.

Wraps the `lib5c outliers` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile
- `self.output()`: the resulting outlier-filtered countsfile

fold_threshold = <luigi.parameter.FloatParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

overwrite_value = <luigi.parameter.Parameter object>

run()

window_size = <luigi.parameter.IntParameter object>

class lib5c.contrib.luigi.tasks.**PvalueTask**(**args*, ***kwargs*)

Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for calling p-values.

Wraps the `lib5c pvalues` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input observed countsfile
- `self.input()[2]`: the input expected countsfile
- `self.input()[3]`: the input variance countsfile
- `self.output()`: the resulting countsfile of p-values

distribution = <luigi.parameter.Parameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

log = <luigi.parameter.BoolParameter object>

run()

vst = <luigi.parameter.BoolParameter object>

class lib5c.contrib.luigi.tasks.**QnormTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying quantile normalization to countsfiles.

Wraps the `lib5c qnorm` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1:]`: the input countsfiles
- `self.output()`: not specified explicitly, see below

Technically this class should specify a list of outputs, one for each input countsfile. In practice, this specification of outputs is left to whatever code strings together the pipeline. The `lib5c qnorm` command will produce output files on disk based on the `outfile_pattern` and the file names of the input countsfiles.

averaging = <luigi.parameter.BoolParameter object>

condition_on = <luigi.parameter.Parameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

outfile_pattern = <luigi.parameter.Parameter object>

reference = <luigi.parameter.Parameter object>

regional = <luigi.parameter.BoolParameter object>

run()

class lib5c.contrib.luigi.tasks.**QvaluesTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for converting p-values to q-values.

Wraps the `lib5c qvalues` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile of p-values
- `self.output()`: the resulting countsfile of q-values

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

method = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**RegionalTaskMixin**
Bases: `object`

Mixin class for Tasks that write a separate output file per region.

region = <luigi.parameter.Parameter object>

class lib5c.contrib.luigi.tasks.**SmoothTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.FilteringTask*

Task class for smoothing countsfiles.

Wraps the `lib5c smooth` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input observed countsfile
- `self.output()`: the resulting countsfile of smooth observed values

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**SplineTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for applying explicit spline bias correction to countsfiles.

Wraps the `lib5c spline` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input countsfile
- `self.output()`: the resulting spline-normalized countsfile

bias_factors = <luigi.parameter.ListParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

knots = <luigi.parameter.ListParameter object>

model_outfile = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**SubtractTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for subtracting one countsfile from another.

Wraps the `lib5c subtract` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the minuend (countsfile to subtract from)
- `self.input()[2]`: the subtrahend (countsfile to subtract)
- `self.output()`: the difference (countsfile resulting from the subtraction)

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

run()

class lib5c.contrib.luigi.tasks.**ThresholdTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for thresholding p-value countsfiles to call loops.

Wraps the `lib5c threshold` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1:]`: the input countsfiles of p-values
- `self.output()[0]`: the output countsfile of called loops
- `self.output()[1]`: the output text file summarizing the loop calls
- `self.output()[2]`: the output .csv file containing the complete analysis results

background_threshold = <luigi.parameter.FloatParameter object>

bh_fdr = <luigi.parameter.BoolParameter object>

concordant = <luigi.parameter.BoolParameter object>

conditions = <luigi.parameter.Parameter object>

dataset_outfile = <luigi.parameter.Parameter object>

distance_threshold = <luigi.parameter.IntParameter object>

heatmap = <luigi.parameter.BoolParameter object>

heatmap_outdir = <luigi.parameter.Parameter object>

kappa_confusion_outfile = <luigi.parameter.Parameter object>

run()

significance_threshold = <luigi.parameter.FloatParameter object>

size_threshold = <luigi.parameter.IntParameter object>

two_tail = <luigi.parameter.BoolParameter object>

class lib5c.contrib.luigi.tasks.**VarianceTask**(**args*, ***kwargs*)
Bases: *lib5c.contrib.luigi.tasks.CmdTask*

Task class for computing variance estimates.

Wraps the `lib5c variance` command line command.

Input/output specification:

- `self.input()[0]`: the primer or bin .bed file
- `self.input()[1]`: the input observed countsfile
- `self.input()[2]`: the input expected countsfile
- `self.output()`: the resulting countsfile of variance estimates

agg_fn = <luigi.parameter.Parameter object>

fitter = <luigi.parameter.Parameter object>

logx = <luigi.parameter.BoolParameter object>

logy = <luigi.parameter.BoolParameter object>

min_disp = <luigi.parameter.Parameter object>

min_dist = <luigi.parameter.IntParameter object>

min_obs = <luigi.parameter.FloatParameter object>

model = <luigi.parameter.Parameter object>

regional = <luigi.parameter.BoolParameter object>

source = <luigi.parameter.Parameter object>

x_unit = <luigi.parameter.Parameter object>

y_unit = <luigi.parameter.Parameter object>

lib5c.contrib.luigi.tasks.**add_visualization_hooks**(*f*, *pvalue=False*, *obs_over_exp=False*, *tetris=False*)

Decorator intended to wrap the `run()` method of luigi Task subclasses to automatically visualize the result of the Task class after it completes.

Parameters

- **f** (*function*) – The function to add visualization hooks to. Intended to be the `run()` method of luigi Task subclasses.
- **pvalue** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn using the p-value colorscale.
- **obs_over_exp** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn using the obs_over_exp colorscale.
- **tetris** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn as tetris heatmaps.

Returns The hooked function.

Return type function

lib5c.contrib.luigi.tasks.**get_all_lines**(*filename*)

Utility function for reading all lines from a file on disk.

Parameters **filename** (*str*) – The file to read from.

Returns The contents of the file.

Return type str

lib5c.contrib.luigi.tasks.**parallelize_reps**(*task_class*, *reps*, ***kwargs*)

Parallelizes any Task class whose constructor accepts a `rep` kwarg across a list of reps by creating a new WrapperTask.

Parameters

- **task_class** (*luigi.Task subclass*) – The Task to parallelize.
- **reps** (*list of str*) – List of reps to parallelize over.
- **kwargs** (*kwargs*) – Additional kwargs to pass through to the Task class.

Returns A WrapperTask which simply requires the original `task_class` to be run for every rep in `reps`.

Return type luigi.WrapperTask subclass

lib5c.contrib.luigi.tasks.**parallelize_reps_regions**(*task_class*, *reps*, *regions*, ***kwargs*)

Parallelizes any Task class whose constructor accepts `rep` and `region` kwargs across lists of reps and regions by creating a new WrapperTask.

Parameters

- **task_class** (*luigi.Task subclass*) – The Task to parallelize.
- **reps** (*list of str*) – List of reps to parallelize over.
- **regions** (*list of str*) – List of regions to parallelize over.
- **kwargs** (*kwargs*) – Additional kwargs to pass through to the Task class.

Returns A WrapperTask which simply requires the original `task_class` to be run for every rep in `reps` and every region in `regions`.

Return type luigi.WrapperTask subclass

`lib5c.contrib.luigi.tasks.`**visualizable**(*pvalue=False*, *obs_over_exp=False*, *tetris=False*)

Class decorator factory for luigi Task subclasses which allows the task to automatically visualize itself after completion by

1. adding `heatmap` and `heatmap_outdir` parameters to the Task and
2. decorating the Task's `run()` method with `add_visualization_hooks()`

Parameters

- **pvalue** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn using the p-value colorscale.
- **obs_over_exp** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn using the obs_over_exp colorscale.
- **tetris** (*bool*) – Pass True to denote that the visualized heatmaps should be drawn as tetris heatmaps.

Returns The class decorator.

Return type function

Module contents

lib5c.contrib.pybigwig package

Submodules

lib5c.contrib.pybigwig.bigwig module

Module for interfacing with the external `pyBigWig` Python package, which enables reading and searching genomic features from `.bigwig` files.

class `lib5c.contrib.pybigwig.bigwig.`**BigWig**(*filename*)

Bases: `object`

Wrapper class around pyBigWig, mostly to expose our own *query()* function.

bw

The underlying pyBigWig object.

Type pyBigWig object

query(*grange*, *stat='max'*, *num_bins=None*, *exact=True*)

Signature rework/wrapper around pyBigWig's *stats()* and *intervals()*.

Parameters

- **grange** (*Dict[str, Any]*) – The genomic range to query. Should have at least the following structure:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

- **num_bins** (*Optional[int]*) – Pass an integer to split *grange* into *num_bins* bins of equal width, and return a summary statistic for each bin. Pass None to return all bigwig features in *grange* without binning.
- **stat** (*str*) – The summary statistic to use if *num_bins* is not None.
- **exact** (*bool*) – Pass True to ignore bigwig zoom levels when computing summary statistics and return the exact answer instead.

Returns A list of bed features with 'value' keys representing the results of the query.

Return type List[Dict[str, Any]]

lib5c.contrib.pybigwig.bigwig.**bigwig_avail**()
Utility function for checking if pyBigWig is installed.

Returns True if pyBigWig is installed, otherwise False.

Return type bool

Module contents

lib5c.contrib.seaborn package

Module contents

Module contents

Subpackage for interfacing with non-standard third-party modules.

lib5c.core package

Submodules

lib5c.core.interactions module

class lib5c.core.interactions.**InteractionMatrix**(*matrix*, *locusmap=None*)
Bases: *lib5c.core.mixins.Picklable*, *lib5c.core.mixins.Annotatable*, *lib5c.core.mixins.Loggable*

Class representing pairwise architectural contact frequencies between genomic loci. At its heart, this is a square, symmetric matrix whose ij th entry corresponds to the interaction frequency between the i th genomic locus and the j th genomic locus. Optionally, some metadata for the genomic loci may be included.

matrix
The matrix of interaction frequencies.

Type square, symmetric numpy matrix

locusmap
Metadata for the genomic loci in the form of a LocusMap object. The size of the LocusMap passed should be equal to the size of `matrix`.

Type *LocusMap*, optional

Notes

Several accessor shortcuts are provided via this class's implementation of `__getitem__()`. Consult that function's docstring for more details.

Some elements of the `matrix` of an InteractionMatrix may be set to `np.nan` in cases where data is not available or where interactions are impossible. Impossible interactions are those involving fragments with the same directionality. If a LocusMap whose constituent Locus objects have `strand` keys in their `data` attribute is provided when an InteractionMatrix is created, the impossible interactions will be set to `np.nan` automatically.

delete(*index*, *inplace=True*)
Delete a Locus and all its interaction information from this InteractionMatrix.

Parameters

- **index** (*int*) – The index of the Locus to delete.
- **inplace** (*bool, optional*) – If True, the deletion is performed inplace, preserving the reference to the original InteractionMatrix, though the underlying `matrix` and `locusmap` attributes will be present as new objects. If False, the original InteractionMatrix object will be unaffected.

Returns The resulting InteractionMatrix. If the operation was performed in-place, this is just a reference to `this`.

Return type *InteractionMatrix*

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> counts = X + X.T
>>> im = InteractionMatrix(counts,
...                        locusmap=LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           strand='+', region='Sox2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           strand='-', region='Sox2'),
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
...           strand='-', region='Nestin'),
...     Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...           strand='+', region='Nestin')]))
...
>>> print(im)
```

(continues on next page)

(continued from previous page)

```
InteractionMatrix of size 4
[[nan  5. 10. nan]
 [ 5. nan nan 20.]
 [10. nan nan 25.]
 [nan 20. 25. nan]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> deleted_im = im.delete(2, inplace=False)  # non-in-place delete
>>> deleted_im.print_log()
InteractionMatrix created
deleted locus at index 2 with name 5C_326_Nestin_REV_9
>>> print(deleted_im)
InteractionMatrix of size 3
[[nan  5. nan]
 [ 5. nan 20.]
 [nan 20. nan]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> im.size()  # original object unaffected
4
>>> deleted_im['5C_329_Sox2_REV_4', '5C_326_Nestin_FOR_10']
20.0
>>> deleted_im = im.delete(2)  # in-place delete
>>> deleted_im.print_log()
InteractionMatrix created
deleted locus at index 2 with name 5C_326_Nestin_REV_9
>>> print(im)  # original object affected
InteractionMatrix of size 3
[[nan  5. nan]
 [ 5. nan 20.]
 [nan 20. nan]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
```

extract_region(*region*)

Extract a submatrix of this Interaction Matrix corresponding to a specified region.

Parameters **region** (*str*) – The name of the region to extract.

Returns The submatrix of this InteractionMatrix corresponding to the specified region. This is returned as a new, independent InteractionMatrix object (see Examples below).

Return type *InteractionMatrix*

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> X = np.arange(16, dtype=float).reshape((4, 4))
```

(continues on next page)

(continued from previous page)

```
>>> counts = X + X.T
>>> im = InteractionMatrix(counts,
...                        locusmap=LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           strand='+', region='Sox2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           strand='-', region='Sox2'),
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
...           strand='-', region='Nestin'),
...     Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...           strand='+', region='Nestin')]))
...
>>> im.matrix
matrix([[nan,  5., 10., nan],
        [ 5., nan, nan, 20.],
        [10., nan, nan, 25.],
        [nan, 20., 25., nan]])
>>> im.get_regions()
['Sox2', 'Nestin']
>>> sox2_im = im.extract_region('Sox2')
>>> sox2_im.print_log()
InteractionMatrix created
extracted region Sox2
>>> sox2_im.matrix
matrix([[nan,  5.],
        [ 5., nan]])
>>> nestin_im = im.extract_region('Nestin')
>>> nestin_im.matrix
matrix([[nan, 25.],
        [25., nan]])
```

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_primerfile('test/primers.bed')
>>> im.get_regions()
['Sox2', 'Nestin', 'Klf4', 'gene-desert', 'Nanog-V2', 'Olig1-Olig2',
'Oct4']
>>> im.size()
1551
>>> im['5C_329_Sox2_FOR_2', '5C_329_Sox2_REV_4'] = 1.0
>>> im['5C_329_Sox2_FOR_2', '5C_329_Sox2_REV_4']
1.0
>>> sox2_im = im.extract_region('Sox2')
>>> sox2_im.get_regions()
['Sox2']
>>> sox2_im.size()
265
>>> sox2_im['5C_329_Sox2_FOR_2', '5C_329_Sox2_REV_4']
1.0
>>> sox2_im['5C_329_Sox2_FOR_2', '5C_329_Sox2_REV_4'] = 2.0
>>> im['5C_329_Sox2_FOR_2', '5C_329_Sox2_REV_4']
1.0
```

extract_slice(*desired_slice*)

Gets a new InteractionMatrix object representing the interactions of a subset of the loci described in this InteractionMatrix as specified by a slice object.

Parameters **desired_slice** (*slice*) – The slice to use to subset this InteractionMatrix.

Returns The new InteractionMatrix.

Return type *InteractionMatrix*

Notes

Since LocusMap objects are sorted, slices with negative steps will be reversed before being applied to the `matrix` attribute.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im = InteractionMatrix(X + X.T)
>>> im.matrix
matrix([[ 0.,  5., 10., 15.],
        [ 5., 10., 15., 20.],
        [10., 15., 20., 25.],
        [15., 20., 25., 30.]])
>>> sliced_im = im[1:3]
>>> sliced_im.print_log()
InteractionMatrix created
sliced out slice(1, 3, None)
>>> sliced_im.matrix
matrix([[10., 15.],
        [15., 20.]])
>>> reverse_sliced_im = im[3:1:-1]
>>> reverse_sliced_im.matrix
matrix([[10., 15.],
        [15., 20.]])
```

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_primerfile('test/primers.bed')
>>> i = im['5C_329_Sox2_FOR_33']
>>> j = im['5C_329_Sox2_REV_89']
>>> im[i, i+1] = 1.0
>>> im[j, j-1] = 2.0
>>> im[i, j] = 3.0
>>> sliced_im = im[i:j+1]
>>> sliced_im[0, 1]
1.0
>>> sliced_im[-1, -2]
2.0
>>> sliced_im[0, -1]
3.0
>>> sliced_im.size() == 1 + j - i
True
>>> sliced_im.get_regions()
['Sox2']
>>> sliced_im.locusmap[0].get_name()
'5C_329_Sox2_FOR_33'
>>> sliced_im.locusmap[-1].get_name()
'5C_329_Sox2_REV_89'
```

flatten (*discard_nan=True*)

Flattens the interaction values in this InteractionMatrix into a flat, non-redundant array.

Parameters **discard_nan** (*bool, optional*) – If True, nan's will not be filtered out of the returned array.

Returns A flat, nonredundant array of the interaction values. The (i, j) th element of this InteractionMatrix's `matrix` attribute (for $i >= j$) ends up at the $(i \times (i+1)/2 + j)$ th index of the flattened array. If `discard_nan` was True, these indices will not necessarily match up and it will not be possible to unflatten the array.

Return type 1d numpy array

Notes

A more intuitive way to think about the ordering is to read down the columns of `matrix` from left to right, going to the next column whenever you reach the diagonal.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix(np.matrix([[   3.0, np.nan,    5.0],
...                                   [np.nan,    6.0, np.nan],
...                                   [   5.0, np.nan,    8.0]]))
...
>>> im.flatten()
array([3., 6., 5., 8.])
>>> im.flatten(discard_nan=False)
array([  3.,  nan,   6.,   5.,  nan,   8.])
```

flatten_cis (*discard_nan=True*)

Flattens only the cis interaction values in this InteractionMatrix into a flat, non-redundant array.

Parameters **discard_nan** (*bool, optional*) – If True, nan's will not be filtered out of the returned array.

Returns The result of `flatten()`-ing each region separately and concatenating the results.

Return type 1d numpy array

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> counts = X + X.T
>>> im = InteractionMatrix(counts,
...                        locusmap=LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           region='Sox2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           region='Sox2'),
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
```

(continues on next page)

(continued from previous page)

```
...            region='Nestin'),
...      Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...            region='Nestin')]))
...
>>> im[0, 1] = np.nan
>>> im[3, 3] = np.nan
>>> print(im)
InteractionMatrix of size 4
[[ 0. nan 10. 15.]
 [nan 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. nan]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> im.flatten_cis()
array([  0.,  10.,  20.,  25.])
>>> im.flatten_cis(discard_nan=False)
array([  0.,  nan,  10.,  20.,  25.,  nan])
```

classmethod from_binfile(*binfile*)

Factory method that creates an InteractionMatrix object whose `matrix` attribute is initialized with all zeros and whose genomic loci are described by data parsed from a BED file containing bin information.

Parameters **binfile** (*str*) – String reference to a BED file containing bin information to be parsed and used as metadata for the genomic loci.

Returns InteractionMatrix object.

Return type *InteractionMatrix*

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_binfile('test/bins_new.bed')
>>> im.print_log()
InteractionMatrix created
source binfile: test/bins_new.bed
>>> im.locusmap.size()
1807
>>> im.size()
1807
>>> im['gene-desert_BIN_047', 'Nestin_BIN_000']
0.0
```

classmethod from_countsfile(*countsfile*, *locusmap=None*, *primerfile=None*, *binfile=None*)

Factory method that creates an InteractionMatrix object from a .counts file.

Parameters

- **countsfile** (*str*) – String reference to the .counts file to parse.
- **locusmap** (LocusMap, *optional*) – Metadata for the genomic loci in the form of a LocusMap object.
- **primerfile** (*str, optional*) – String reference to a BED file containing primer information to be parsed and used as metadata for the genomic loci.

- **binfile** (*str, optional*) – String reference to a BED file containing bin information to be parsed and used as metadata for the genomic loci.

Returns InteractionMatrix object parsed from the .counts file.

Return type *InteractionMatrix*

Notes

At least one of `locusmap`, `primerfile`, or `binfile` must be passed. The `matrix` attribute of the returned InteractionMatrix will have the same size as whichever of these was passed.

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_countsfile(
...     'test/test_raw.counts', primerfile='test/primers.bed')
...
>>> im.print_log()
InteractionMatrix created
source countsfile: test/test_raw.counts
source primerfile: test/primers.bed
>>> im.locusmap.size()
1551
>>> im.size()
1551
>>> im['5C_325_Olig1-Olig2_FOR_38', '5C_331_gene-desert_REV_62']
1.0
>>> im['5C_325_Olig1-Olig2_REV_237', '5C_325_Olig1-Olig2_REV_230']
nan
```

classmethod from_list(*list_of_interaction_matrices*)

Factory method that creates an InteractionMatrix object via iterative addition of a list of InteractionMatrix objects.

Parameters **list_of_interaction_matrices** (*list of InteractionMatrix*) – The InteractionMatrix objects to concatenate.

Returns The resulting InteractionMatrix.

Return type *InteractionMatrix*

Notes

This function operates via naive iterate addition and so the following are basically equivalent:

```
summed_im = InteractionMatrix.from_list(list_of_im)
summed_im = sum(list_of_im, InteractionMatrix([]))
```

Neither implementation is particularly efficient. See the Examples section for details.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> im1 = InteractionMatrix(np.matrix([[1.0, 2.0], [2.0, 1.0]]))
>>> im2 = InteractionMatrix(np.matrix([[3.0, 4.0], [4.0, 3.0]]))
>>> summed_im = InteractionMatrix.from_list([im1, im2])
>>> summed_im.print_log()
InteractionMatrix created
created from addition
created from list
>>> summed_im.matrix
matrix([[1., 2., 0., 0.],
        [2., 1., 0., 0.],
        [0., 0., 3., 4.],
        [0., 0., 4., 3.]])
```

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> im1 = InteractionMatrix(np.matrix([[np.nan, 1.0], [1.0, np.nan]]),
...                         locusmap=LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           strand='+'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           strand='-')]))
...
>>> im2 = InteractionMatrix(np.matrix([[np.nan, 2.0], [2.0, np.nan]]),
...                         locusmap=LocusMap([
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
...           strand='-'),
...     Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...           strand='+')]))
...
>>> summed_im = InteractionMatrix.from_list([im1, im2])
>>> summed_im.matrix
matrix([[nan,  1.,  0., nan],
        [ 1., nan, nan,  0.],
        [ 0., nan, nan,  2.],
        [nan,  0.,  2., nan]])
```

classmethod from_locusmap(*locusmap*)

Factory method that creates an InteractionMatrix object whose `matrix` attribute is initialized with all zeros and whose genomic loci are described by a LocusMap object.

Parameters **locusmap** (LocusMap) – Metadata for the genomic loci in the form of a LocusMap object.

Returns InteractionMatrix object.

Return type *InteractionMatrix*

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import LocusMap, Locus
>>> locus_list = [Locus('chr3', 34109023, 34113109),
...               Locus('chr3', 34113147, 34116141)]
```

(continues on next page)

(continued from previous page)

```
...
>>> locus_map = LocusMap(locus_list)
>>> im = InteractionMatrix.from_locusmap(locus_map)
>>> im.print_log()
InteractionMatrix created
created from locusmap
>>> im.matrix
matrix([[0., 0.],
        [0., 0.]])
```

classmethod from_primerfile(*primerfile*)

Factory method that creates an InteractionMatrix object whose `matrix` attribute is initialized with all zeros and whose genomic loci are described by data parsed from a BED file containing primer information.

Parameters **primerfile** (*str*) – String reference to a BED file containing primer information to be parsed and used as metadata for the genomic loci.

Returns InteractionMatrix object.

Return type *InteractionMatrix*

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_primerfile('test/primers.bed')
>>> im.print_log()
InteractionMatrix created
source primerfile: test/primers.bed
>>> im.locusmap.size()
1551
>>> im.size()
1551
>>> im['5C_325_Olig1-Olig2_FOR_38', '5C_331_gene-desert_REV_62']
0.0
>>> im['5C_325_Olig1-Olig2_REV_237', '5C_325_Olig1-Olig2_REV_230']
nan
```

classmethod from_size(*size*)

Factory method that creates an InteractionMatrix object whose `matrix` attribute is initialized with all zeros and whose size is specified.

Parameters **size** (*int*) – Size of the `matrix` attribute of the desired InteractionMatrix object.

Returns InteractionMatrix object.

Return type *InteractionMatrix*

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_size(2)
>>> im.print_log()
InteractionMatrix created
created from size 2
>>> im.matrix
```

(continues on next page)

(continued from previous page)

```
matrix([[0., 0.],
        [0., 0.]])
```

get_regions()

Get the regions covered by this InteractionMatrix, if this can be deduced from its metadata. For this to work, the metadata in this InteractionMatrix's `locusmap` attribute must consist of Locus objects with `'name'` keys in their `data` attributes.

Returns The ordered list of region names as strings.

Return type list of str

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_primerfile('test/primers.bed')
>>> im.get_regions()
['Sox2', 'Nestin', 'Klf4', 'gene-desert', 'Nanog-V2', 'Olig1-Olig2',
'Oct4']
```

size()

Get the size of the InteractionMatrix. This value is equal to either dimension of the `matrix` attribute as well as the number of Locus objects in the associated LocusMap, if present.

Returns Size of this InteractionMatrix.

Return type long

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix.from_size(2)
>>> im.matrix
matrix([[0., 0.],
        [0., 0.]])
>>> im.size()
2
```

to_countsfile(*filename*, *omit_zeros=True*)

Write the interaction values in this InteractionMatrix to a countsfile.

Parameters

- **filename** (*str*) – String reference to file to write to.
- **omit_zeros** (*bool, optional*) – If True, lines will not be written to the outfile if the counts for that line are zero.

Examples

```
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> from lib5c.operators.standardization import Standardizer
>>> s = Standardizer()
```

(continues on next page)

(continued from previous page)

```
>>> lm = LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           region='Sox2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           region='Sox2'),
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
...           region='Nestin'),
...     Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...           region='Nestin')
... ])
...
>>> im1 = InteractionMatrix([[  0.,   5.,  10.,  15.],
...                          [  5.,  10.,  15.,  20.],
...                          [ 10.,  15.,  20.,  25.],
...                          [ 15.,  20.,  25.,  30.]], locusmap=lm)
...
>>> print(im1)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> im1.to_countsfile('test/core_test.counts')
>>> im2 = InteractionMatrix.from_countsfile('test/core_test.counts',
...                                         locusmap=lm)
...
>>> print(im2)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
```

unflatten(*values*)

Overwrite the `matrix` attribite of this InteractionMatrix object with values from a flat list, such as that created by `InteractionMatrix.flatten()`.

Parameters **values** (*1d iterable of float*) – A flat, nonredundant list of the interaction values. The (i, j) th element of this InteractionMatrix's `matrix` attribute (for $i >= j$) will be set to the $(i \times (i + 1)/2 + j)$ th value of the flattened list.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> im = InteractionMatrix(np.matrix([[   3.0, np.nan,    5.0],
...                                   [np.nan,    6.0, np.nan],
```

(continues on next page)

(continued from previous page)

```
...                                     [   5.0, np.nan,    8.0]]))
...
>>> values = im.flatten(discard_nan=False)
>>> values
array([  3.,  nan,   6.,   5.,  nan,   8.])
>>> values += 1
>>> values
array([  4.,  nan,   7.,   6.,  nan,   9.])
>>> im.unflatten(values)
>>> print(im)
InteractionMatrix of size 3
[[ 4. nan  6.]
 [nan  7. nan]
 [ 6. nan  9.]]
```

unflatten_cis(*values*)

Overwrite only the cis interaction values of the `matrix` attribite of this InteractionMatrix object with values from a flat list, such as that created by `InteractionMatrix.flatten_cis()`.

Parameters **values** (*1d list of float*) – A flat, nonredundant list of the cis interaction values. The order should match what would be expected from `flatten()`-ing each region of the InteractionMatrix seperately and concatenating the results.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> X = np.reshape(range(16), (4, 4)).astype(float)
>>> counts = X + X.T
>>> im = InteractionMatrix(counts,
...                        locusmap=LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2',
...           strand='+', region='Sox2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4',
...           strand='-', region='Sox2'),
...     Locus('chr3', 87282063, 87285636, name='5C_326_Nestin_REV_9',
...           strand='-', region='Nestin'),
...     Locus('chr3', 87285637, 87295935, name='5C_326_Nestin_FOR_10',
...           strand='+', region='Nestin')]))
...
>>> print(im)
InteractionMatrix of size 4
[[nan  5. 10. nan]
 [ 5. nan nan 20.]
 [10. nan nan 25.]
 [nan 20. 25. nan]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> values = im.flatten_cis(discard_nan=False)
>>> values
array([nan,  5., nan, nan, 25., nan])
>>> values += 1
```

(continues on next page)

(continued from previous page)

```
>>> values
array([nan,  6., nan, nan, 26., nan])
>>> im.unflatten_cis(values)
>>> print(im)
InteractionMatrix of size 4
[[nan  6. 10. nan]
 [ 6. nan nan 20.]
 [10. nan nan 26.]
 [nan 20. 26. nan]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
```

lib5c.core.loci module

class lib5c.core.loci.**Locus**(*chrom*, *start*, *end*, ***kwargs*)

Bases: *lib5c.core.mixins.Picklable*, *lib5c.core.mixins.Annotatable*

Basically anything with a chromosome, start, and end. Can also include arbitrary metadata.

chrom

The chromosome on which this locus resides (e.g., `'chr4'`).

Type str

start

The start coordinate for the zero-indexed, half-open interval occupied by the locus.

Type int

end

The end coordinate for the zero-indexed, half-open interval occupied by the locus.

Type int

data

Arbitrary additional data about the locus. This attribute is filled in with any kwargs passed to the constructor.

Type dict(str -> any)

Notes

Locus objects support comparison and ordering via the `total_ordering` decorator. See this class's implementations of `__eq__()` and `__lt__()` for more details.

Locus objects support the Annotatable mixin, which is the source of their `data` attribute and all its related functions.

as_dict()

Gets a dict representation of the Locus.

Returns

This dict is guaranteed to have at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

Other keys may be present if included in the `data` attribute.

Return type dict

Examples

```
>>> from lib5c.core.loci import Locus
>>> locus = Locus('chr3', 34109023, 34113109, strand='+')
>>> locus.as_dict() == \
...     {'chrom': 'chr3',
...      'start': 34109023,
...      'end': 34113109,
...      'strand': '+'}
True
```

get_name()

Gets the name of the locus, if present.

Returns The value of `data['name']` if it exists; None otherwise.

Return type str or None

Examples

```
>>> from lib5c.core.loci import Locus
>>> locus = Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2')
>>> locus.get_name()
'5C_329_Sox2_FOR_2'
```

get_region()

Gets the region of the locus, if present.

Returns The value of `data['region']` if it exists; None otherwise.

Return type str or None

Examples

```
>>> from lib5c.core.loci import Locus
>>> locus = Locus('chr3', 34109023, 34113109, region='Sox2')
>>> locus.get_region()
'Sox2'
```

get_strand()

Gets the strand of the locus, if present.

Returns The value of `data['strand']` if it exists; None otherwise.

Return type '+' or '-' or None

Examples

```
>>> from lib5c.core.loci import Locus
>>> locus = Locus('chr3', 34109023, 34113109, strand='+')
>>> locus.get_strand()
'+'
```

class lib5c.core.loci.**LocusMap**(*locus_list*)

Bases: *lib5c.core.mixins.Picklable*, *lib5c.core.mixins.Annotatable*, *lib5c.core.mixins.Loggable*

Representation of an organized group of Locus objects.

locus_list
Ordered list of unique Locus objects in the LocusMap.

Type list of Locus objects

regions
Ordered list of region names, as strings, present in the LocusMap. This list is filled in only when the Locus objects within the LocusMap have a `'region'` key in their `data` attribute.

Type list of str

name_dict
Dict mapping Locus names as strings to the Locus object with that name. This dict is filled in only when the Locus objects within the LocusMap have a `'name'` key in their `data` attribute.

Type dict(str -> Locus)

region_index_dict
Maps a region name and an index within the region to a Locus object within the specified region. This means that, for example,

```
locus_map.region_index_dict['Sox2'][3]
```

resolves to the Locus object that is the 4th Locus object of the Sox2 region in the LocusMap instance called `locus_map`.

Type dict(str -> list of Locus objects)

hash_to_index_dict
Maps a Locus object's hash to its index within `locus_list`.

Type dict(int -> int)

name_to_index_dict
Dict mapping Locus names as strings to their index within the LocusMap. This dict is filled in only when the Locus objects within the LocusMap have a `'name'` key in their `data` attribute.

Type dict(str -> int)

Notes

Locus objects in a LocusMap are ordered by the total ordering implemented by the Locus class. See that class's implementation of `__eq__()` and `__lt__()` for more details.

LocusMap objects support the Loggable and Annotatable mixins. See the Examples section for an example.

Examples

```
>>> from lib5c.core.loci import LocusMap
>>> locus_map = LocusMap.from_primerfile('test/primers.bed')
>>> locus_map.size()
1551
>>> locus_map.print_log()
LocusMap created
source primerfile: test/primers.bed
>>> locus_map.set_value('test key', 'test value')
>>> locus_map.get_value('test key')
'test value'
```

as_dict_of_list_of_dict()

Gets a primitive representation of the LocusMap, organized by region. Converts the `locus_list` attribute of this LocusMap from a list of Locus objects to dict whose keys are region names as strings and whose values are list of dicts representing the Locus objects in each region.

Returns The primitive representation of the LocusMap.

Return type dict(str -> list of dict)

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nest'),
...               Locus('chr3', 87285637, 87295935, region='Nest')]
...
>>> locus_map = LocusMap(locus_list)
>>> locus_map.as_dict_of_list_of_dict() == \
...     {'Sox2': [{'chrom': 'chr3', 'start': 34109023, 'end': 34113109,
...                'region': 'Sox2'},
...               {'chrom': 'chr3', 'start': 34113147, 'end': 34116141,
...                'region': 'Sox2'}],
...      'Nest': [{'chrom': 'chr3', 'start': 87282063, 'end': 87285636,
...                'region': 'Nest'},
...               {'chrom': 'chr3', 'start': 87285637, 'end': 87295935,
...                'region': 'Nest'}]}
True
```

as_list_of_dict()

Gets a primitive representation of the LocusMap. Converts the `locus_list` attribute of this LocusMap from a list of Locus objects to a list of dicts representing those Locus objects.

Returns The primitive representation of the LocusMap.

Return type list of dict

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...               Locus('chr3', 34113147, 34116141, name='Sox2_REV_4')]
```

(continues on next page)

(continued from previous page)

```
...
>>> locus_map = LocusMap(locus_list)
>>> locus_map.as_list_of_dict() == \
...     [{'chrom': 'chr3', 'start': 34109023, 'end': 34113109,
...       'name': 'Sox2_FOR_2'},
...      {'chrom': 'chr3', 'start': 34113147, 'end': 34116141,
...       'name': 'Sox2_REV_4'}]
True
```

by_index(*index*)

Get the Locus object contained in this LocusMap with a specified index.

Parameters **index** (*int*) – The index of the Locus to get.

Returns The Locus object with the specified index.

Return type *Locus*

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...               Locus('chr3', 34113147, 34116141, name='Sox2_REV_4')]
...
>>> locus_map = LocusMap(locus_list)
>>> print(locus_map.by_index(1))
Locus chr3:34113147-34116141
    name: Sox2_REV_4
```

by_name(*name*)

Get the Locus object contained in this LocusMap with a specified name.

Parameters **name** (*str*) – The name of the Locus to get.

Returns The Locus object with the specified name.

Return type *Locus*

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...               Locus('chr3', 34113147, 34116141, name='Sox2_REV_4')]
...
>>> locus_map = LocusMap(locus_list)
>>> print(locus_map.by_name('Sox2_REV_4'))
Locus chr3:34113147-34116141
    name: Sox2_REV_4
```

by_region_index(*region*, *index*)

Get the Locus object contained in this LocusMap with a specified index within a specified region. In other words, the `index` th Locus of the region with name `region`.

Parameters

- **region** (*str*) – The name of the region to look for the Locus in.

- **index** (*int*) – The index of the desired Locus within the specified region.

Returns The specified Locus.

Return type *Locus*

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin'),
...               Locus('chr3', 87285637, 87295935, region='Nestin')]
...
>>> locus_map = LocusMap(locus_list)
>>> print(locus_map.by_region_index('Nestin', 1))
Locus chr3:87285637-87295935
    region: Nestin
```

delete(*index*)

Creates a new LocusMap object that excludes the Locus at a specified index.

Parameters **index** (*int*) – The index of the Locus to exclude.

Returns The new LocusMap.

Return type *LocusMap*

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [
...     Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...     Locus('chr3', 34113147, 34116141, name='Sox2_REV_4'),
...     Locus('chr3', 87282063, 87285636, name='Nestin_REV_9'),
...     Locus('chr3', 87285637, 87295935, name='Nestin_FOR_10')
... ]
...
>>> locus_map = LocusMap(locus_list)
>>> deleted_locus_map = locus_map.delete(2)
>>> deleted_locus_map.print_log()
LocusMap created
deleted locus at index 2 with name Nestin_REV_9
>>> for locus in deleted_locus_map:
...     print(locus)
Locus chr3:34109023-34113109
    name: Sox2_FOR_2
Locus chr3:34113147-34116141
    name: Sox2_REV_4
Locus chr3:87285637-87295935
    name: Nestin_FOR_10
```

extract_region(*region*)

Create a LocusMap representing the Locus objects in only one specified region of this LocusMap.

Parameters **region** (*str*) – The name of the region to extract.

Returns A new LocusMap restricted to the specified region.

Return type *LocusMap*

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin'),
...               Locus('chr3', 87285637, 87295935, region='Nestin')]
...
>>> locus_map = LocusMap(locus_list)
>>> extracted_locus_map = locus_map.extract_region('Sox2')
>>> extracted_locus_map.print_log()
LocusMap created
extracted region Sox2
>>> for locus in extracted_locus_map:
...     print(locus)
...
Locus chr3:34109023-34113109
    region: Sox2
Locus chr3:34113147-34116141
    region: Sox2
```

extract_slice(*desired_slice*)

Gets a new LocusMap object representing a subset of the Locus objects in this LocusMap specified by a slice.

Parameters **desired_slice** (*slice*) – The slice to use to subset this LocusMap.

Returns The new LocusMap.

Return type *LocusMap*

Notes

Since LocusMap objects are sorted, the returned LocusMap will always be sorted, regardless of the slice direction.

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [
...     Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...     Locus('chr3', 34113147, 34116141, name='Sox2_REV_4'),
...     Locus('chr3', 87282063, 87285636, name='Nestin_REV_9'),
...     Locus('chr3', 87285637, 87295935, name='Nestin_FOR_10')
... ]
...
>>> locus_map = LocusMap(locus_list)
>>> sliced_map = locus_map[1:3]
>>> sliced_map.print_log()
LocusMap created
sliced out slice(1, 3, None)
>>> for locus in sliced_map:
...     print(locus)
```

(continues on next page)

(continued from previous page)

```
...
Locus chr3:34113147-34116141
    name: Sox2_REV_4
Locus chr3:87282063-87285636
    name: Nestin_REV_9
```

classmethod from_binfile(*binfile*)

Factory method that creates a LocusMap object from a BED file containing bin information.

Parameters **binfile** (*str*) – String reference to the bin file.

Returns LocusMap object parsed from the bin file.

Return type *LocusMap*

Examples

```
>>> from lib5c.core.loci import LocusMap
>>> locus_map = LocusMap.from_binfile('test/bins_new.bed')
>>> locus_map.size()
1807
>>> locus_map.print_log()
LocusMap created
source binfile: test/bins_new.bed
```

classmethod from_list(*list_of_locusmaps*)

Factory method that creates a new LocusMap object from a list of existing LocusMap objects by concatenation.

Parameters **list_of_locusmaps** (*list of LocusMap*) – A list of LocusMap objects to be concatenated.

Returns The concatenated LocusMap.

Return type *LocusMap*

Notes

This function should be slightly more efficient than iterative addition. Therefore, it is preferred to use

```
summed_locus_map = LocusMap.from_list(list_of_locus_maps)
```

over

```
summed_locus_map = sum(list_of_locus_maps, LocusMap([]))
```

as evidenced by

```
> python -mtimeit `
  -s'from lib5c.core.loci import LocusMap' `
  -s'lm = LocusMap.from_primerfile(\"test/primers.bed\")' `
  -s'lm_list = [lm.extract_region(r) for r in lm.regions]' `
  's = LocusMap.from_list(lm_list)'
10 loops, best of 3: 48.2 msec per loop
```

versus

```
> python -mtimeit `
  -s'from lib5c.core.loci import LocusMap' `
  -s'lm = LocusMap.from_primerfile(\"test/primers.bed\")' `
  -s'lm_list = [lm.extract_region(r) for r in lm.regions]' `
  's = sum(lm_list, LocusMap([]))'
10 loops, best of 3: 174 msec per loop
```

Examples

```
>>> from lib5c.core.loci import LocusMap
>>> locus_map = LocusMap.from_primerfile('test/primers.bed')
>>> sox2_locus_map = locus_map.extract_region('Sox2')
>>> sox2_locus_map.size()
265
>>> sox2_locus_map.get_regions()
['Sox2']
>>> klf4_locus_map = locus_map.extract_region('Klf4')
>>> klf4_locus_map.size()
251
>>> klf4_locus_map.get_regions()
['Klf4']
>>> summed_locus_map = LocusMap.from_list([sox2_locus_map,
...                                        klf4_locus_map])
...
>>> summed_locus_map.print_log()
LocusMap created
created from list
>>> summed_locus_map.size()
516
>>> summed_locus_map.get_regions()
['Sox2', 'Klf4']
>>> builtin_sum_result = sum([sox2_locus_map, klf4_locus_map],
...                          LocusMap([]))
...
>>> builtin_sum_result.size()
516
>>> builtin_sum_result.get_regions()
['Sox2', 'Klf4']
```

classmethod from_list_of_dict(*list_of_dict*)

Factory method that creates a LocusMap object from a list of dicts that represent the Loci that the LocusMap should be composed of.

Parameters **list_of_dict** (*list of dict*) – A list of dicts, with each dict representing a Locus that should be created and put into the LocusMap.

Returns A LocusMap whose Locus objects are equivalent to the dicts passed in list_of_dict

Return type *LocusMap*

Examples

```
>>> from lib5c.core.loci import LocusMap
>>> list_of_dict = [{'chrom': 'chr3',
```

(continues on next page)

(continued from previous page)

```
...                   'start': 34109023,
...                   'end': 34113109,
...                   'name': 'Sox2_FOR_2'},
...                  {'chrom': 'chr3',
...                   'start': 34113147,
...                   'end': 34116141,
...                   'name': 'Sox2_REV_4'}]
...
>>> locus_map = LocusMap.from_list_of_dict(list_of_dict)
>>> locus_map.print_log()
LocusMap created
created from list of dict
>>> for locus in locus_map:
...     print(locus)
...
Locus chr3:34109023-34113109
    name: Sox2_FOR_2
Locus chr3:34113147-34116141
    name: Sox2_REV_4
>>> list_of_dict_dup = [{'chrom': 'chr3',
...                       'start': 34109023,
...                       'end': 34113109,
...                       'name': 'Sox2_FOR_2'},
...                      {'chrom': 'chr3',
...                       'start': 34113147,
...                       'end': 34116141,
...                       'name': 'Sox2_REV_4'},
...                      {'chrom': 'chr3',
...                       'start': 34113147,
...                       'end': 34116141,
...                       'name': 'duplicate Locus!'}]
...
>>> locus_map_dup = LocusMap.from_list_of_dict(list_of_dict_dup)
Traceback (most recent call last):
    ...
ValueError: Locus objects in LocusMap must be unique
```

classmethod from_primerfile(*primerfile*)

Factory method that creates a LocusMap object from a BED file containing primer information.

Parameters **primerfile** (*str*) – String reference to the primer file.

Returns LocusMap object parsed from the primer file.

Return type *LocusMap*

Examples

```
>>> from lib5c.core.loci import LocusMap
>>> locus_map = LocusMap.from_primerfile('test/primers.bed')
>>> locus_map.size()
1551
>>> locus_map.print_log()
LocusMap created
source primerfile: test/primers.bed
```

get_index(*name*)
Get the index of the Locus object in this LocusMap with a specified name.

Parameters **name** (*str*) – The name of the Locus to get the index for.

Returns The index of the Locus object with the specified name.

Return type int

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, name='Sox2_FOR_2'),
...               Locus('chr3', 34113147, 34116141, name='Sox2_REV_4')]
...
>>> locus_map = LocusMap(locus_list)
>>> locus_map.get_index('Sox2_REV_4')
1
```

get_index_by_hash(*hash_value*)
Get the Locus object contained in this LocusMap with a specified hash.

Parameters **hash_value** (*int*) – The hash of the Locus object to find.

Returns The index of the desired Locus object within `locus_list` if it exists within this LocusMap object, or None if it doesn't.

Return type int or None

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109),
...               Locus('chr3', 34113147, 34116141)]
...
>>> locus_map = LocusMap(locus_list)
>>> locus_hash = hash(Locus('chr3', 34113147, 34116141))
>>> locus_map.get_index_by_hash(locus_hash)
1
>>> locus_map.get_index_by_hash(123) is None
True
```

get_region_sizes()
Gets information about the number of Locus objects in each region.

Returns A dict mapping region names as strings to the number of Locus objects in that region.

Return type dict(str -> int)

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin')]
...
```

(continues on next page)

(continued from previous page)

```
>>> locus_map = LocusMap(locus_list)
>>> locus_map.get_region_sizes() == {'Sox2': 2, 'Nestin': 1}
True
```

get_regions()
Gets the regions spanned by the Locus objects in this LocusMap.

Returns The ordered list of region names.

Return type list of str

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin'),
...               Locus('chr3', 87285637, 87295935, region='Nestin')]
...
>>> locus_map = LocusMap(locus_list)
>>> locus_map.get_regions()
['Sox2', 'Nestin']
```

size()
Get the number of Locus objects in the LocusMap.

Returns The number of Locus objects in the LocusMap.

Return type int

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109),
...               Locus('chr3', 34113147, 34116141)]
...
>>> locus_map = LocusMap(locus_list)
>>> locus_map.size()
2
```

to_bedfile(*filename*, *fields=('name',)*)
Write this LocusMap to disk as a BED-formatted file.

Parameters

- **filename** (*str*) – String reference to the file to write to.
- **fields** (*list of str, optional*) – Specify additional columns in the BED file after the traditional chromosome, start, end. Columns should be specified in order as strings corresponding to keys in the `data` attributes on the Locus objects that make up this LocusMap.

Examples

```
>>> from lib5c.core.loci import Locus, LocusMap
>>> lm1 = LocusMap([
...     Locus('chr3', 34109023, 34113109, name='5C_329_Sox2_FOR_2'),
...     Locus('chr3', 34113147, 34116141, name='5C_329_Sox2_REV_4')
... ])
...
>>> lm1.to_bedfile('test/core_test_locusmap.bed')
>>> lm2 = LocusMap.from_primerfile('test/core_test_locusmap.bed')
>>> for locus in lm2:
...     print(locus)
Locus chr3:34109023-34113109
    name: 5C_329_Sox2_FOR_2
    number: 2
    orientation: 3'
    region: Sox2
    strand: +
Locus chr3:34113147-34116141
    name: 5C_329_Sox2_REV_4
    number: 4
    orientation: 5'
    region: Sox2
    strand: -
```

lib5c.core.mixins module

class lib5c.core.mixins.**Annotatable**

Bases: object

Mixin class for storing and accessing arbitrary annotation information on object instances.

data

Dict to store arbitrary annotation data. Typically, the keys will be strings.

Type dict

Notes

This mixin requires initialization. Subclasses must explicitly call

```
Annotatable.__init__(self)
```

somewhere in their constructor.

When a subclass's bound functions return a new instance, the following guidelines are recommended:

- addition/summing operations: create a new dict and update it
- other operations: copy a reference to `data` into the new instance

get_data()

Get this instance's `data` attribute.

Returns This instance's `data` attribute.

Return type dict

get_value(*key*)
Get the value of some key in data. This is equivalent to a get-or-None function for `data[key]`.

Parameters **key** (*str*) – The key to search for in this instance's `data`.

Returns The value of `data[key]`, or None if the key does not exist.

Return type any

Examples

```
>>> from lib5c.core.loci import Locus
>>> locus = Locus('chr3', 34109023, 34113109, num_genes=10)
>>> locus.get_value('num_genes')
10
>>> locus.get_value('num_ctcf_sites') is None
True
```

set_data(*data*)
Overwrite this instance's `data` attribute with a passed dict.

Parameters **data** (*dict*) – The dict to put in this instance's `data` attribute.

set_value(*key*, *value*)
Set the value for a specific key in this instance's `data` attribute.

Parameters

- **key** (*str*) – The key to set.
- **value** (*any*) – The value to store.

class lib5c.core.mixins.**Loggable**
Bases: `object`

Mixin class for supporting object event logging.

log
Each string in the list describes an event in this object's history.

Type list of str

Notes

This mixin requires initialization. Subclasses must explicitly call

```
Loggable.__init__(self)
```

somewhere in their constructor.

When a subclass's bound functions return a new instance, the following guidelines are recommended:

- addition/summing operations: use the empty log of the new instance
- other operations: copy a reference to `log` into the new instance

get_log()
Get this instance's log.

Returns This instance's log.

Return type list of str

log_event(*event*)
Add an event to the log.

Parameters **event** (*str*) – A string describing the event.

print_log()
Print this instance's log to the console.

class lib5c.core.mixins.**Picklable**
Bases: object

Mixin class for providing easy reading and writing of instances to disk via the pickle module.

classmethod from_pickle(*filename*)
Create a new instance of the class from a pickle file.

Parameters **filename** (*str*) – String reference to a pickle file to read from.

Returns Unpickled instance.

Return type cls

to_pickle(*filename*)
Write this instance to a pickle file.

Parameters **filename** (*str*) – String reference to the file to write this instance to.

Module contents

lib5c.operators package

Submodules

lib5c.operators.base module

class lib5c.operators.base.**InteractionMatrixOperator**
Bases: object

Abstract base class for objects that operate on single InteractionMatrix objects.

Subclasses should implement an `apply_inplace()` function that takes in an InteractionMatrix object and returns an InteractionMatrix object but is allowed to operate in-place. Additional parameters requried by the `apply_inplace()` function can be either passed to the function as `**kwargs` or stored in properties of the InteractionMatrixOperator subclass.

apply(*target*, ***kwargs*)
Apply this operator to a target InteractionMatrix, returning a copy.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to operate on.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type *InteractionMatrix*

apply_by_region(*target*, ***kwargs*)
Apply this operator independently to each region of a target InteractionMatrix.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to operate on.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type *InteractionMatrix*

Notes

To support logging, we used the following pattern:

1. Log on the target object to indicate regional application
2. Maintain a list of results of the application
3. Instantiate
4. Get the first result from the list and copy its log to the instance
5. Log on the result object to indicate end of regional application
6. Return the instance

If you see `'applying by region'` with no closing `'done applying by region'`, that indicates that you are looking at a target object for an apply-by-region operation that was not done in-place. Such a log line can be ignored.

If you see `'applying by region'` with a closing `'done applying by region'`, that indicates that you are looking at a result object for an apply-by-region operation that was not done in-place. The lines in the block show the log for only the first region, but each region was processed identically.

apply_inplace(*target*, ***kwargs*)
Apply this operator to a target InteractionMatrix in-place.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to operate on.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type *InteractionMatrix*

class lib5c.operators.base.**MultiInteractionMatrixOperator**
Bases: object

Abstract base class for objects that operate on multiple InteractionMatrix objects.

Subclasses should implement an `apply_inplace()` function that takes in an list of InteractionMatrix objects and returns a list of InteractionMatrix objects but is allowed to operate in-place. Additional parameters requried by the `apply_inplace()` function can be either passed to the function as `**kwargs` or stored in properties of the MultiInteractionMatrixOperator subclass.

apply(*targets*, ***kwargs*)
Apply this operator to a list of target InteractionMatrix objects.

Parameters

- **targets** (*list of InteractionMatrix*) – The list of InteractionMatrix objects to operate on simultaneously.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type list of InteractionMatrix

apply_by_region(*targets*, ***kwargs*)
Apply this operator independently to each region of the target InteractionMatrix objects.

Parameters

- **targets** (*list of InteractionMatrix*) – The list of InteractionMatrix objects to operate on simultaneously.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type list of InteractionMatrix

Notes

To support logging, we used the following pattern:

1. Log on all target objects to indicate regional application
2. Maintain a dict of list of results of the application
3. Instantiate the results
4. Get the first region of the first result from the dict of lists and copy its log to each instance
5. Log on all result objects to indicate end of regional application
6. Return the instances

If you see `'applying by region'` with no closing `'done applying by region'`, that indicates that you are looking at a target object for an apply-by-region operation that was not done in-place. Such a log line can be ignored.

If you see `'applying by region'` with a closing `'done applying by region'`, that indicates that you are looking at a result object for an apply-by-region operation that was not done in-place. The lines in the block show the log for only the first region, but each region was processed identically.

apply_inplace(*targets*, ***kwargs*)
Apply this operator to a target InteractionMatrix in-place.

Parameters

- **targets** (*list of InteractionMatrix*) – The list of InteractionMatrix objects to operate on simultaneously.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The result of applying the operation.

Return type list of InteractionMatrix

lib5c.operators.modeling module

class lib5c.operators.modeling.**EmpiricalPvalueOperator**
Bases: *lib5c.operators.base.InteractionMatrixOperator*

Operator for assigning empirical right-tail p-values to all interactions in an InteractionMatrix.

apply_inplace(*target*, ***kwargs*)
Transform the target InteractionMatrix, setting its interactions to their emprical right-tail p-values.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to transform.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The transformed InteractionMatrix.

Return type *InteractionMatrix*

Notes

This transformation uses the `kind='strict'` kwarg of `scipy.stats.percentileofscore()`, which means the resulting values represent the fraction of all the values that are greater than or equal to the value at that position.

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.modeling import EmpiricalPvalueOperator
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im = InteractionMatrix(X + X.T)
>>> print(im)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> epo = EmpiricalPvalueOperator()
>>> result = epo.apply(im)
>>> print(result)
InteractionMatrix of size 4
[[1.  0.9 0.8 0.6]
 [0.9 0.8 0.6 0.4]
 [0.8 0.6 0.4 0.2]
 [0.6 0.4 0.2 0.1]]
>>> result.print_log()
InteractionMatrix created
transformed to empirical p-values
```

lib5c.operators.qnorm module

class lib5c.operators.qnorm.**QuantileNormalizer**(*tie='lowest'*)

Bases: *lib5c.operators.base.MultiInteractionMatrixOperator*

Operator for quantile normalizing InteractionMatrix objects.

tie

How this QuantileNormalizer will resolve ties. If `'lowest'`, it will set all tied entries to the value of the lowest rank. If `'average'`, it will set all tied entries to the average value across the tied ranks.

Type {'lowest', 'average'}

Notes

This operator will first standardize the target InteractionMatrix objects, including propagation of nan's, if they have `locusmap` attributes defined. Otherwise, the target InteractionMatrix objects must be the same size.

apply_inplace(*targets*, ***kwargs*)
Quantile normalizes the target InteractionMatrix objects.

Parameters

- **targets** (*list of InteractionMatrix*) – The InteractionMatrix objects to quantile normalize. These must either have `locusmap` attributes or be the same size.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The standardized InteractionMatrix objects.

Return type list of InteractionMatrix

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.qnorm import QuantileNormalizer
>>> q = QuantileNormalizer()
>>> im1 = InteractionMatrix([[    5., np.nan,     3.],
...                          [np.nan,     2., np.nan],
...                          [    3., np.nan,     4.]])
...
>>> im2 = InteractionMatrix([[    4., np.nan,     4.],
...                          [np.nan,     1., np.nan],
...                          [    4., np.nan,     2.]])
...
>>> im3 = InteractionMatrix([[    3., np.nan,     6.],
...                          [np.nan,     4., np.nan],
...                          [    6., np.nan,     8.]])
...
>>> results = q.apply([im1, im2, im3])
>>> print(results[0])
InteractionMatrix of size 3
[[5.66666667        nan 3.        ]
 [       nan 2.                nan]
 [3.                nan 4.66666667]]
>>> print(results[1])
InteractionMatrix of size 3
[[4.66666667        nan 4.66666667]
 [       nan 2.                nan]
 [4.66666667        nan 3.        ]]
>>> print(results[2])
InteractionMatrix of size 3
[[2.                nan 4.66666667]
 [       nan 3.                nan]
 [4.66666667        nan 5.66666667]]
```

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> from lib5c.operators.qnorm import QuantileNormalizer
```

(continues on next page)

(continued from previous page)

```
>>> q = QuantileNormalizer()
>>> lm = LocusMap([
...     Locus('chr3', 34109023, 34113109),
...     Locus('chr3', 34113147, 34116141),
...     Locus('chr3', 87282063, 87285636),
...     Locus('chr3', 87285637, 87295935)
... ])
...
>>> im1 = InteractionMatrix([[  0.,   5.,  10.,  15.],
...                          [  5.,  10.,  15.,  20.],
...                          [ 10.,  15.,  20.,  25.],
...                          [ 15.,  20.,  25.,  30.]], locusmap=lm)
...
>>> im2 = InteractionMatrix([[    1., np.nan,    11.],
...                          [np.nan,    11.,    21.],
...                          [   11.,    21.,    16.]], locusmap=lm[:3])
...
>>> results = q.apply([im1, im2])
>>> print(results[0])
InteractionMatrix of size 3
[[ 0.5  nan 10.5]
 [ nan 10.5 15.5]
 [10.5 15.5 20.5]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87282063-87285636
>>> print(results[1])
InteractionMatrix of size 3
[[ 0.5  nan 10.5]
 [ nan 10.5 20.5]
 [10.5 20.5 15.5]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87282063-87285636
>>> results[0].print_log()
InteractionMatrix created
standardized with propagate_nan=True
deleted locus at index 3
qnormed with tie=lowest
```

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.qnorm import QuantileNormalizer
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin'),
...               Locus('chr3', 87285637, 87295935, region='Nestin')]
...
>>> locus_map = LocusMap(locus_list)
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im1 = InteractionMatrix(X + X.T, locusmap=locus_map)
>>> print(im1)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
```

(continues on next page)

(continued from previous page)

```
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> im2 = InteractionMatrix((X + X.T) + 1, locusmap=locus_map)
>>> print(im2)
InteractionMatrix of size 4
[[ 1.  6. 11. 16.]
 [ 6. 11. 16. 21.]
 [11. 16. 21. 26.]
 [16. 21. 26. 31.]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> q = QuantileNormalizer()
>>> results = q.apply_by_region([im1, im2])
>>> print(results[0])
InteractionMatrix of size 4
[[  0.5   5.5   0.    0. ]
 [  5.5  10.5   0.    0. ]
 [  0.    0.   20.5  25.5]
 [  0.    0.   25.5  30.5]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> print(results[1])
InteractionMatrix of size 4
[[  0.5   5.5   0.    0. ]
 [  5.5  10.5   0.    0. ]
 [  0.    0.   20.5  25.5]
 [  0.    0.   25.5  30.5]]
Associated LocusMap:
LocusMap comprising 4 loci
    Range: chr3:34109023-34113109 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
>>> results[0].print_log()
InteractionMatrix created
applying by region
extracted region Sox2
standardized with propagate_nan=True
qnormed with tie=lowest
done applying by region
```

lib5c.operators.standardization module

class lib5c.operators.standardization.**Standardizer**(*propagate_nan=True*)

Bases: *lib5c.operators.base.MultiInteractionMatrixOperator*

Operator for standardizing InteractionMatrix objects. This process reduces all InteractionMatrix objects passed to the lowest common denominator of loci. In other words, loci that are not present in every InteractionMatrix object will be discarded from all InteractionMatrix objects.

propagate_nan
If True, nan values will be propagated across InteractionMatrix objects.

Type bool

Notes

The InteractionMatrix objects supplied must have `locusmap` attributes.

apply_inplace(*targets*, ***kwargs*)
Apply the standardization operation to the target InteractionMatrix objects.

Parameters

- **targets** (*list of InteractionMatrix*) – The InteractionMatrix objects to standardize.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The standardized InteractionMatrix objects.

Return type list of InteractionMatrix

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.core.loci import Locus, LocusMap
>>> from lib5c.operators.standardization import Standardizer
>>> s = Standardizer()
>>> lm = LocusMap([
...     Locus('chr3', 34109023, 34113109),
...     Locus('chr3', 34113147, 34116141),
...     Locus('chr3', 87282063, 87285636),
...     Locus('chr3', 87285637, 87295935)
... ])
...
>>> im1 = InteractionMatrix([[  0.,   5.,  10.,  15.],
...                          [  5.,  10.,  15.,  20.],
...                          [ 10.,  15.,  20.,  25.],
...                          [ 15.,  20.,  25.,  30.]], locusmap=lm)
...
>>> im2 = InteractionMatrix([[   1., np.nan,   11.],
...                          [np.nan,   11.,   16.],
...                          [  11.,   16.,   21.]], locusmap=lm[:3])
...
>>> results = s.apply([im1, im2])
>>> print(results[0])
InteractionMatrix of size 3
[[ 0. nan 10.]
 [nan 10. 15.]
 [10. 15. 20.]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87282063-87285636
>>> print(results[1])
InteractionMatrix of size 3
[[ 1. nan 11.]
```

(continues on next page)

(continued from previous page)

```
 [nan 11. 16.]
 [11. 16. 21.]]
Associated LocusMap:
LocusMap comprising 3 loci
    Range: chr3:34109023-34113109 to chr3:87282063-87285636
>>> results[0].print_log()
InteractionMatrix created
standardized with propagate_nan=True
deleted locus at index 3
```

lib5c.operators.trimming module

class lib5c.operators.trimming.**InteractionTrimmer**(*value_threshold_lower=None*, *value_threshold_upper=None*, *locus_percentage_threshold_lower=None*, *locus_percentage_threshold_upper=None*, *global_percentage_threshold_lower=None*, *global_percentage_threshold_upper=None*, *locus_fold_threshold_lower=None*, *locus_fold_threshold_upper=None*, *global_fold_threshold_lower=None*, *global_fold_threshold_upper=None*)

Bases: *lib5c.operators.base.InteractionMatrixOperator*

Operator for removing specific interactions from an InteractionMatrix object according to specified criteria by setting their values to `np.nan`.

value_threshold_lower
If not None, interactions with values lower than this number will be removed.

Type float or None

value_threshold_upper
If not None, interactions with values higher than this number will be removed.

Type float or None

locus_percentage_threshold_lower
If not None, this percentage of interactions at each locus with the lowest values will be removed.

Type float or None

locus_percentage_threshold_upper
If not None, this percentage of interactions at each locus with the highest values will be removed.

Type float or None

global_percentage_threshold_lower
If not None, this percentage of interactions with the lowest values will be removed.

Type float or None

global_percentage_threshold_upper
If not None, this percentage of interactions with the highest values will be removed.

Type float or None

locus_fold_threshold_lower
If not None, interactions whose values are less than this many times the median value across either participating locus will be removed.

Type float or None

locus_fold_threshold_upper
If not None, interactions whose values are more than this many times the median value across either participating locus will be removed.

Type float or None

global_fold_threshold_lower
If not None, interactions whose values are less than this many times the median value across all interactions will be removed.

Type float or None

global_fold_threshold_upper
If not None, interactions whose values are more than this many times the median value across all interactions will be removed.

Type float or None

apply_inplace(*target*, ***kwargs*)
Apply the trimming operation to the target InteractionMatrix.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to trim.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The trimmed InteractionMatrix.

Return type *InteractionMatrix*

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.trimming import InteractionTrimmer
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im = InteractionMatrix(X + X.T)
>>> print(im)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(value_threshold_lower=5.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[nan nan 10. 15.]
[nan 10. 15. 20.]
[10. 15. 20. 25.]
[15. 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(value_threshold_upper=25.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
```

(continues on next page)

(continued from previous page)

```
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. nan]
 [15. 20. nan nan]]
>>> trimmer = InteractionTrimmer(locus_percentage_threshold_lower=25.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[nan nan nan nan]
 [nan 10. 15. 20.]
 [nan 15. 20. 25.]
 [nan 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(locus_percentage_threshold_upper=75.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[ 0.  5. 10. nan]
 [ 5. 10. 15. nan]
 [10. 15. 20. nan]
 [nan nan nan nan]]
>>> trimmer = InteractionTrimmer(global_percentage_threshold_lower=25.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[nan nan nan 15.]
 [nan nan 15. 20.]
 [nan 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(global_percentage_threshold_upper=75.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. nan]
 [10. 15. nan nan]
 [15. nan nan nan]]
>>> trimmer = InteractionTrimmer(locus_fold_threshold_lower=0.5)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[nan nan 10. 15.]
 [nan 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(locus_fold_threshold_upper=2.0)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[ 0.  5. 10. nan]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [nan 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(global_fold_threshold_lower=0.25)
>>> print(trimmer.apply(im))
InteractionMatrix of size 4
[[nan  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> trimmer = InteractionTrimmer(global_fold_threshold_upper=2.0)
>>> result = trimmer.apply(im)
>>> print(result)
InteractionMatrix of size 4
```

(continues on next page)

(continued from previous page)

```
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. nan]]
>>> result.print_log()
InteractionMatrix created
interactions trimmed with:
    global_fold_threshold_upper=2.0
```

class lib5c.operators.trimming.**LocusTrimmer**(*sum_threshold_upper=None*, *sum_threshold_lower=None*, *max_threshold=None*, *min_threshold=None*, *percentage_threshold_lower=None*, *percentage_threshold_upper=None*, *fold_threshold_upper=None*, *fold_threshold_lower=None*)

Bases: *lib5c.operators.base.InteractionMatrixOperator*

Operator for removing Loci from an InteractionMatrix object according to specified criteria.

sum_threshold_upper
If not None, Loci whose row sums are greater than this value will be removed.

Type float or None

sum_threshold_lower
If not None, Loci whose row sums are less than this value will be removed.

Type float or None

max_threshold
If not None, Loci containing at least one interaction above this value will be removed.

Type float or None

min_threshold
If not None, Loci containing at least one interaction below this value will be removed.

Type float or None

percentage_threshold_lower
If not None, this percentage of of the Loci with the lowest row sums will be removed.

Type float or None

percentage_threshold_upper
If not None, this percentage of of the Loci with the highest row sums will be removed.

Type float or None

fold_threshold_upper
If not None, Loci whose row sums are more than this many times the median row sum will be removed.

Type float or None

fold_threshold_lower
If not None, Loci whose row sums are less than this many times the median row sum will be removed.

Type float or None

apply_inplace(*target*, ***kwargs*)
Apply the trimming operation to the target InteractionMatrix.

Parameters

- **target** (InteractionMatrix) – The InteractionMatrix object to trim.
- **kwargs** (*other keyword arguments*) – To be utilized by subclasses.

Returns The trimmed InteractionMatrix.

Return type *InteractionMatrix*

Examples

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.trimming import LocusTrimmer
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im = InteractionMatrix(X + X.T)
>>> print(im)
InteractionMatrix of size 4
[[ 0.  5. 10. 15.]
 [ 5. 10. 15. 20.]
 [10. 15. 20. 25.]
 [15. 20. 25. 30.]]
>>> locus_trimmer = LocusTrimmer(sum_threshold_lower=35)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 3
[[10. 15. 20.]
 [15. 20. 25.]
 [20. 25. 30.]]
>>> locus_trimmer = LocusTrimmer(percentage_threshold_lower=50.0)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 2
[[20. 25.]
 [25. 30.]]
>>> locus_trimmer = LocusTrimmer(sum_threshold_upper=80.0)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 3
[[ 0.  5. 10.]
 [ 5. 10. 15.]
 [10. 15. 20.]]
>>> locus_trimmer = LocusTrimmer(percentage_threshold_upper=50.0)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 2
[[ 0.  5.]
 [ 5. 10.]]
>>> locus_trimmer = LocusTrimmer(min_threshold=0.0)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 3
[[10. 15. 20.]
 [15. 20. 25.]
 [20. 25. 30.]]
>>> locus_trimmer = LocusTrimmer(max_threshold=30.0)
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 3
[[ 0.  5. 10.]
 [ 5. 10. 15.]
 [10. 15. 20.]]
>>> locus_trimmer = LocusTrimmer(fold_threshold_lower=0.5)
```

(continues on next page)

(continued from previous page)

```
>>> print(locus_trimmer.apply(im))
InteractionMatrix of size 3
[[10. 15. 20.]
 [15. 20. 25.]
 [20. 25. 30.]]
>>> locus_trimmer = LocusTrimmer(fold_threshold_upper=1.5)
>>> result = locus_trimmer.apply(im)
>>> result.print_log()
InteractionMatrix created
loci trimmed with:
    fold_threshold_upper=1.5
deleted locus at index 3
>>> print(result)
InteractionMatrix of size 3
[[ 0.  5. 10.]
 [ 5. 10. 15.]
 [10. 15. 20.]]
>>> result = locus_trimmer.apply_inplace(im)
>>> im.print_log()
InteractionMatrix created
loci trimmed with:
    fold_threshold_upper=1.5
deleted locus at index 3
>>> print(im)
InteractionMatrix of size 3
[[ 0.  5. 10.]
 [ 5. 10. 15.]
 [10. 15. 20.]]
```

```
>>> import numpy as np
>>> from lib5c.core.interactions import InteractionMatrix
>>> from lib5c.operators.trimming import LocusTrimmer
>>> from lib5c.core.loci import Locus, LocusMap
>>> locus_list = [Locus('chr3', 34109023, 34113109, region='Sox2'),
...               Locus('chr3', 34113147, 34116141, region='Sox2'),
...               Locus('chr3', 87282063, 87285636, region='Nestin'),
...               Locus('chr3', 87285637, 87295935, region='Nestin')]
...
>>> locus_map = LocusMap(locus_list)
>>> X = np.arange(16, dtype=float).reshape((4, 4))
>>> im = InteractionMatrix(X + X.T, locusmap=locus_map)
>>> im.matrix
matrix([[ 0.,  5., 10., 15.],
        [ 5., 10., 15., 20.],
        [10., 15., 20., 25.],
        [15., 20., 25., 30.]])
>>> locus_trimmer = LocusTrimmer(percentage_threshold_lower=50.0)
>>> result = locus_trimmer.apply_by_region(im)
>>> print(result)
InteractionMatrix of size 2
[[10.  0.]
 [ 0. 30.]]
Associated LocusMap:
LocusMap comprising 2 loci
    Range: chr3:34113147-34116141 to chr3:87285637-87295935
    Regions: ['Sox2', 'Nestin']
```

(continues on next page)

(continued from previous page)

```
>>> result.print_log()
InteractionMatrix created
applying by region
extracted region Sox2
loci trimmed with:
    percentage_threshold_lower=50.0
deleted locus at index 0
done applying by region
```

Module contents

lib5c.parsers package

Submodules

lib5c.parsers.bed module

Module for parsing .bed files.

lib5c.parsers.bed.**load_features**(*bedfile*, *id_index=None*, *value_index=None*, *boundaries=None*, *strict=True*)

Loads the features from a .bed file into dicts and returns them.

Parameters

- **bedfile** (*str*) – String reference to location of .bed file to load features from.
- **id_index** (*int*) – If passed, indicates the column index of the id field.
- **value_index** (*int*) – If passed, indicates the column index of the value field.
- **boundaries** (*list of dicts*) – If passed, features will only be loaded if they intersect at least one of the features in this list. The features should be represented as dicts with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end'  : int
}
```

- **strict** (*boolean*) – If True, there must not be any incomplete lines in the bedfile.

Returns

The keys are chromosome names. The values are lists of features for that chromosome. The features are represented as dicts with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end'  : int,
    'id'   : str or None,
    'value': float or None
}
```

The 'id' and 'value' fields may be None if no feature ID's were provided in the BED file, but the keys will always be present in the returned dict.

Return type dict of lists of dicts

Notes

The parser will attempt to guess the column indices of the id and value fields based on the number of columns and the types of the column entries.

lib5c.parsers.bed.**main**()

lib5c.parsers.bias module

Module for parsing bias vector files.

lib5c.parsers.bias.**load_bias_vector**(*bias_file*, *pixelmap*)
Loads in bias vectors from a bias vector file.

Parameters

- **bias_file** (*str*) – String reference to the location of a .bias file to load bias vectors from.
- **pixelmap** (*Dict[str, List[Dict[str, Any]]]*) – A primermap or pixelmap specifying the information about the regions and loci whose bias factors are contained in the bias_file.

Returns The keys are region names as strings, the values are the one-dimensional bias vectors for that region.

Return type Dict[str, np.ndarray]

lib5c.parsers.config module

Module for wrappers around ConfigParser for parsing config files.

lib5c.parsers.config.**parse_config**(*configfile*, *name*)
Parses a section from a config file into a dict.

Parameters

- **configfile** (*str*) – The config file to parse.
- **name** (*str*) – The section name to parse.

Returns The data.

Return type dict

lib5c.parsers.counts module

Module for parsing .counts files.

lib5c.parsers.counts.**load_cis_trans_counts**(*countsfile*, *primermap*, *name_parser=<function default_primer_parser>*, *force_nan='always'*, *region_order=None*)

Loads the counts values from a primer-primer pair .counts file into a single square, symmetric array, and returns it.

Parameters

- **countsfile** (*str*) – String reference to location of .counts file to load counts from.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th primer in that region. Primers are represented as dicts with the following structure:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int
}
```

  See lib5c.parsers.primers.get_primermap().

- **name_parser** (*Optional[Callable[[str], Dict[str, Any]]]*) – Function that takes in the primer names in the countsfile and returns a dict containing key-value pairs containing information required to identify the primer. At a minimum, this dict must have the following structure:

```
{
    'region': str
}
```

  This information is necessary to deduce what region a given primer in the countsfile belongs to.

- **force_nan** (*Optional[str]*) – If 'always' is passed and if the primermap contains strand information, impossible ligations will be always set to nan. If 'implicit' is passed, impossible ligations will be set to nan when implied by the strand information in the primermap, but not when the ligations are explicitly present in the countsfile. If 'never' is passed, strand information will be ignored and impossible ligations will not be identified.
- **region_order** (*Optional[List[str]]*) – If passed, this list will be used to determine the order in which the regions will be concatenated in. If not passed, the regions will be concatenated in order of genomic coordinate.

Returns The square, symmetric array of counts.

Return type np.ndarray

lib5c.parsers.counts.**load_counts**(*countsfile*, *primermap*, *force_nan='always'*, *dtype=<class 'float'>*)

Loads the counts values from a primer-primer pair .counts file into square, symmetric arrays and returns them.

Parameters

- **countsfile** (*str*) – String reference to location of .counts file to load counts from.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th primer in that region. Primers are represented as dicts with the following structure:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int
}
```

See `lib5c.parsers.primers.load_primermap()`.

- **force_nan** (*Optional[str]*) – If 'always' is passed and if the primermap contains strand information, impossible ligations will be always set to nan. If 'implicit' is passed, impossible ligations will be set to nan when implied by the strand information in the primermap, but not when the ligations are explicitly present in the countsfile. If 'never' is passed, strand information will be ignored and impossible ligations will not be identified.
- **dtype** (*{int, float}*) – Sets the dtype for the matrix. If the value column contains strings this will be ignored and the dtype will be set to 'U25'.

Returns The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.

Return type Dict[str, np.ndarray]

`lib5c.parsers.counts.`**load_counts_by_name**(*countsfile*, *name_list=None*, *primermap=None*, *locusmap=None*, *force_nan='always'*, *region_order=None*)

Loads the counts values from any .counts file into a single square, symmetric array, and returns it.

Parameters

- **countsfile** (*str*) – String reference to location of .counts file to load counts from.
- **name_list** (*Optional[List[str]]*) – Ordered list of locus names as strings.
- **primermap** (*Optional[Dict[str, List[Dict[str, Any]]]]*) – The keys of the outer dict are region names. The values are lists, where the ith entry represents the ith primer in that region. Primers are represented as dicts with the following structure:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int
}
```

See lib5c.parsers.primers.get_primermap().

- **locusmap** (*Optional[LocusMap]*) – Locus information as a LocusMap object.
- **force_nan** (*Optional[str]*) – If 'always' is passed and if the primermap contains strand information, impossible ligations will be always set to nan. If 'implicit' is passed, impossible ligations will be set to nan when implied by the strand information in the primermap, but not when the ligations are explicitly present in the countsfile. If 'never' is passed, strand information will be ignored and impossible ligations will not be identified.
- **region_order** (*Optional[List[str]]*) – If passed, this list will be used to determine the order in which the regions will be concatenated in. If not passed, the regions will be concatenated in order of genomic coordinate. If name_list is passed, this kwarg is ignored.

Returns The square, symmetric array of counts.

Return type np.ndarray

lib5c.parsers.counts.**load_counts_legacy**(*countsfile*, *name_parser=<function default_bin_parser>*, *pixelmap=None*)

Loads the counts values from a binned .counts file into square, symmetric arrays and returns them.

Parameters

- **countsfile** (*str*) – String reference to location of .counts file to load counts from.
- **name_parser** (*Optional[Callable[[str], Dict[str, Any]]]*) – Function that takes in the bin name column of the countsfile and returns a dict containing key-value pairs containing information required to identify the bin. At a minimum, this dict must have the following structure:

```
{
    'region': str,
    'index': int
}
```

  This information is necessary to deduce what region a given bin in the countsfile belongs to. The index key is optional, but recommended. If present, its value should be the zero-based index of the bin within the region. If not present, the pixelmap will be searched to identify the bin index.

- **pixelmap** (*Optional[Dict[str, List[Dict[str, Any]]]]*) – The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th bin in that region. Bins are represented as dicts with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end'  : int,
    'name' : str
}
```

  See `lib5c.parsers.get_pixelmap()`. The pixelmap is used to identify the index of a bin within a region. If `name_parser` returns an index key, you can pass `None` here since the index will be determined from the bin name.

Returns The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.

Return type Dict[str, np.ndarray]

Notes

This function casts the counts values in the countsfile to floats, so it will work even if the countsfile actually contains pseudocounts or other non-integer values.

lib5c.parsers.counts.**main**()

lib5c.parsers.counts.**set_cis_trans_nans**(*counts*, *aggregated_primermap*)

Sets nan's in a complete cis and trans counts matrix for ligations considered impossible according to a primermap with strand information.

Parameters

- **counts** (*np.ndarray*) – Square, symmetric array storing the complete cis and trans counts, with the regions arranged as implied by the `aggregated_primermap`

- **aggregated_primermap** (*List[Dict[str, Any]]*) – The dicts in the lists represent primers, equal in number and order to the side length of the counts matrix, and have the following structure:

```
{
    'chrom'  : str,
    'start'  : int,
    'end'    : int,
    'strand' : '+' or '-'
}
```

See `lib5c.parsers.primers.get_primermap()` and `lib5c.util.primers.aggregate_primermap()`.

Notes

If the aggregated primermap passed has no strand information, this function will do nothing.

This function operates in-place.

`lib5c.parsers.counts.`**set_nans**(*counts*, *primermap*)

Sets nan's in counts dict for ligations considered impossible according to a primermap with strand information.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th primer in that region. Primers are represented as dicts with the following structure:

```
{
    'chrom'  : str,
    'start'  : int,
    'end'    : int,
    'strand' : '+' or '-'
}
```

See `lib5c.parsers.primers.get_primermap()`.

Notes

If the primermap passed has no strand information, this function will do nothing.

This function operates in-place.

lib5c.parsers.genes module

Module for parsing .bed files containing gene track information.

`lib5c.parsers.genes.`**load_gene_table**(*tablefile*)

Similar to `load_genes()`, but reads in a gzipped UCSC table file instead.

The main advantage of this approach is that genes parsed this way include human-readable gene symbols.

Parameters **tablefile** (*str*) – String reference to location of the gzipped table file to read.

Returns

The keys are chromosome names. The values are lists of genes for that chromosome. The genes are represented as dicts with the following structure:

```
{
    'start' : int,
    'end'   : int,
    'name'  : str,
    'id': str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

Blocks typically represent exons and are represented as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

Return type dict of lists of dicts

lib5c.parsers.genes.**load_genes**(*bedfile*)

Loads information for genes from a .bed file into dicts and returns them.

Parameters **bedfile** (*str*) – String reference to location of .bed file to load genes from.

Returns

The keys are chromosome names. The values are lists of genes for that chromosome. The genes are represented as dicts with the following structure:

```
{
    'start' : int,
    'end'   : int,
    'name'  : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

Blocks typically represent exons and are represented as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

Return type dict of lists of dicts

lib5c.parsers.genes.**main**()

lib5c.parsers.hic module

Module for parsing Hi-C data from the Rao et al. 2014 paper.

lib5c.parsers.hic.**load_range_from_contact_matrix**(*contact_matrix_file*, *grange*, *region_name=''*, *norm_file=None*)

Parses a chunk of contact information out of a Hi-C contact matrix file.

The Hi-C contact matrix file format parsed by this function is the format used in the contact matrices uploaded to GEO for the Rao et al. 2014 paper. It is also the same format used by the Juicer tools dump command.

Parameters

- **contact_matrix_file** (*str*) – String reference to the Hi-C contact matrix file to parse.
- **grange** (*Dict[str, Any]*) – The genomic range to extract contact information for. This should be specified as a dict with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

- **region_name** (*Optional[str]*) – Name for this genomic region. If passed, it will be used to name the bins in the returned pixelmap.
- **norm_file** (*Optional[str]*) – String reference to a file containing a Hi-C bias vector corresponding to the `contact_matrix_file`. If passed, the data will be normalized using this vector.

Returns The first element of the tuple is the extract counts matrix for the requested genomic range. The second element of the tuple is a pixelmap generated for this region describing the specific bins that were extracted.

Return type Tuple[np.ndarray, List[Dict[str, Any]]]

lib5c.parsers.loops module

Module for parsing tables containing categorized loop information.

`lib5c.parsers.loops.`**load_loops**(*loopsfile*)

Reads categorized loop file into nested dict structure.

Parameters **loopsfile** (*str*) – String reference to the loop file to parse. Each line in the loop file corresponds to one loop. The lines are tab-separated. with four columns. These are, in order, the category the loop was categorized into (as a string), the region name the loop is in (as a string), the x-coordinate of the pixel which represents the loop (as an int), and the y-coordinate of the pixel which represent the loop (as an int).

Returns

The outer keys are loop categories as strings. The next level's keys are region names as strings. The innermost dicts represent loops. These inner loop dicts have the following structure:

```
{
    'x': int,
    'y': int
}
```

The ints represent the x and y coordinate, respectively, of the loop within the region.

Return type dict of dicts of dicts

lib5c.parsers.primer_names module

Module providing helper functions for working with primer naming conventions, necessary for parsing certain primer-files.

lib5c.parsers.primer_names.**dblalt_primer_parser**(*name*)
The double alternating primer name parser.

Parameters **name** (*str*) – The name of the primer found in the appropriate column of the primer bedfile.

Returns

This dict has the following structure:

```
{
    'region': str,
    'number': int,
    'orientation': "3'" or "5'",
    'name': str
}
```

These fields are parsed from the primer name.

Return type dict

Notes

You can write other name parsers to accommodate different primer naming conventions.

lib5c.parsers.primer_names.**default_bin_parser**(*name*)
The default bin name parser.

Parameters **name** (*str*) – The name of the bin found in the appropriate column of the bin bedfile.

Returns

This dict has the following structure:

```
{
    'region': str,
    'index': int
}
```

These fields are parsed from the bin name.

Return type dict

Notes

You can write other name parsers to accommodate different bin naming conventions.

lib5c.parsers.primer_names.**default_primer_parser**(*name*)
The default primer name parser.

Parameters **name** (*str*) – The name of the primer found in the appropriate column of the primer bedfile.

Returns

This dict has the following structure:

```
{
    'region': str,
    'number': int,
    'name': str
}
```

These fields are parsed from the primer name.

Return type dict

Notes

You can write other name parsers to accommodate different primer naming conventions.

lib5c.parsers.primer_names.**guess_primer_name_parser**(*name*)

Guesses the appropriate primer or bin name parser to use by looping through a list of possible parsers and testing if they work on a given primer name.

Parameters **name** (*str*) – The name of a primer to use for testing.

Returns The parser thought to be appropriate for this kind of primer name.

Return type function

lib5c.parsers.primers module

Module for parsing .bed files containing 5C primer and bin information.

lib5c.parsers.primers.**get_pixelmap_legacy**(*bedfile*, *name_parser=<function default_bin_parser>*)

Parameters

- **bedfile** (*str*) – String reference to a binned primer bedfile to use to generate the pixelmap.
- **name_parser** (*Optional[Callable[[str], Dict[str, Any]]]*) – Function that takes in the bin name column of the bedfile (the fourth column) and returns a dict containing key-value pairs to be added to the dict that represents that bin. At a minimum, this dict must have the following structure:

```
{
    'region': str
}
```

If the dict includes any keys that are already typically included in the bin dict, the values returned by this function will overwrite the usual values.

Returns

The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th bin in that region. Bins are represented as dicts with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end'  : int,
    'name' : str
}
```

Additional keys may be present if returned by `name_parser`.

Return type Dict[str, List[Dict[str, Any]]]

Notes

A pixelmap is a mapping from bins (specified by a region name and bin or primer index) to the genomic range covered by those bins.

`lib5c.parsers.primers.`**`load_primermap`**(*bedfile*, *name_parser=None*, *strand_index=5*, *region_index=None*, *column_names=None*)

Parameters

- **`bedfile`** (*`str`*) – String reference to a primer bedfile to use to generate the primermap.
- **`name_parser`** (*`Optional[Callable[[str], Dict[str, Any]]]`*) – Function that takes in the primer name column of the bedfile (the fourth column) and returns a dict containing key-value pairs to be added to the dict that represents that primer. At a minimum, this dict must have the following structure:

```
{
    'region': string
}
```

  If the dict includes any keys that are already typically included in the primer dict, the values returned by this function will overwrite the usual values. If None is passed, an appropriate name parser will be guessed based on the primer/bin names.
- **`strand_index`** (*`Optional[int]`*) – If an int is passed, the column with that index will be used to determine strand information for the primer. If `None` is passed, the algorithm will try to guess which column contains this information. If this fails, strand information will not be included in the primer dict. Acceptable strings to indicate primer strand are 'F'/'R', 'FOR'/'REV', and '+'/'-'. Primers on the + strand will be assumed to be oriented in the 3' direction, and primers on the - strand will be assumed to be oriented in the 5' direction, unless an 'orientation' key is provided in the dict returned by `name_parser`.
- **`region_index`** (*`Optional[int]`*) – If an int is passed, the column with that index will be used to determine the region the primer is in. This makes specifying `region_parser` optional and overrides the region it returns.
- **`column_names`** (*`Optional[List[str]]`*) – Pass a list of strings equal to the number of columns in the bedfile, describing the columns. The first four elements will be ignored. Special values include 'strand', which will set `strand_index`, and 'region', which will override `region_index`. All other values will end up as keys in the primer dicts. If this is not passed, this function will look for a header line in the primerfile, and if one is not found, a default header will be assumed.

Returns

The keys of the outer dict are region names. The values are lists, where the i th entry represents the i th primer in that region. Primers are represented as dicts with the following structure:

```
{
    'region'     : str
    'chrom'      : str,
    'start'      : int,
    'end'        : int,
    'name'       : str,
```

(continues on next page)

(continued from previous page)

```
    'strand'     : '+' or '-',
    'orientation': "3'" or "5'"
}
```

though strand and orientation may not be present, and additional keys may be present if returned by `name_parser`, passed in `column_names`, or if a header line is present.

Return type Dict[str, List[Dict[str, Any]]]

Notes

A primermap is a mapping from primers (specified by a region name and primer index) to the genomic range covered by those primers.

`lib5c.parsers.primers.`**`main`**`()`

lib5c.parsers.scaled module

Module for parsing .scaled files.

`lib5c.parsers.scaled.`**`load_scaled`**(*scaledfile*)
Loads the scaled values from a .scaled file into square, symmetric arrays and returns them.

Parameters **`scaledfile`** (*`str`*) – String reference to location of .scaled file to load counts from.

Returns The keys are the region names. The values are the arrays of scaled values for that region. These arrays are square and symmetric.

Return type dict of 2d arrays

`lib5c.parsers.scaled.`**`main`**`()`

lib5c.parsers.table module

Module for parsing table files, which function as a simple extension of .counts files to multiple replicates.

`lib5c.parsers.table.`**`load_table`**(*filename*, *primermap*, *sep='\t'*, *dtype=<class 'float'>*)
Loads a table into a counts_superdict structure.

Parameters

- **`filename`** (*`str`*) – The table file to load.
- **`primermap`** (*`primermap or pixelmap`*) – Defines the FFLJs or bin-bin pairs.
- **`sep`** (*`str`*) – The separator used in the table file.
- **`dtype`** (*`numpy-compatible dtype`*) – The dtype to use when constructing the arrays in the counts_superdict.

Returns The loaded counts_superdict.

Return type counts_superdict

lib5c.parsers.util module

Module containing utility functions for file parsing.

lib5c.parsers.util.**null_value**(*dtype*)
Utility method to get the appropriate null value given a numpy dtype.

Pandas has some logic for this, see http://pandas.pydata.org/pandas-docs/stable/missing_data.html

Parameters **dtype** (*np.dtype*) – The dtype to return a null value for.

Returns The default null value for this dtype.

Return type Any

lib5c.parsers.util.**parse_field**(*val*)
Utility function for parsing a value that could be an int, a float, or a string.

Parameters **val** (*str*) – The value to parse.

Returns The parsed value.

Return type Union[int, float, str]

Module contents

Subpackage for parsing various text files containing data important for 5C analysis.

lib5c.plotters package

Subpackages

lib5c.plotters.extendable package

Submodules

lib5c.plotters.extendable.base_extendable_heatmap module

Module for the BaseExtendableHeatmap class, which provides the basic functionality of the extendable heatmap plotter.

class lib5c.plotters.extendable.base_extendable_heatmap.**BaseExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)
Bases: *lib5c.plotters.extendable.extendable_figure.ExtendableFigure*

Minimal implementation of an ExtendableFigure organized around a contact frequency heatmap.

The heatmap is plotted using `plt.imshow()` using data from the `array` attribute, colored using the other attributes, and the resulting axis is accessible at `h['root']` where `h` is the ExtendableHeatmap instance.

New axes can be added to the margins of the heatmap by calling `add_margin_ax()`. The axis of the new axis that is parallel to the heatmap will have its limits set to match the `grange_x` or `grange_y` attributes. This allows plotting features on the margin axes using genomic coordinates instead of having to convert to pixel coordinates.

The root heatmap axis is still kept in units of pixels. To make drawing on this axis easier, this class provides `transform_feature()`, which will transform a genomic feature dict into heatmap pixel units.

array
Array of values to plot in the heatmap. Must be square.

Type np.ndarray

grange_x
The genomic range represented by the x-axis of this heatmap. The dict should have the form:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

Type dict

grange_y
The genomic range represented by the y-axis of this heatmap. If None, the heatmap is assumed to be symmetric.

Type dict, optional

colorscale
The (min, max) of the color range to plot the values in the array with.

Type tuple of float

colormap
The colormap to use when drawing the heatmap. Strings will be passed to *lib5c.plotters.colormaps.get_colormap()*. If *array* contains strings, pass a dict mapping those strings to colors.

Type str or matplotlib colormap or dict of colors

norm
Pass an instance of `matplotlib.colors.Normalize` to apply this normalization to the heatmap and colorbar.

Type matplotlib.colors.Normalize, optional

add_colorbar(*loc='right'*, *size='10%'*, *pad=0.1*, *new_ax_name='colorbar'*)
Adds a colorbar to the heatmap in a new axis.

Parameters

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new colorbar to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.

- **new_ax_name** (*str*) – A name for the new axis. The axis will be accessible as `h[new_ax_name]` where `h` is this ExtendableHeatmap instance.

Returns The newly added colorbar.

Return type pyplot colorbar

add_margin_ax(*loc='bottom'*, *size='10%'*, *pad=0.0*, *new_ax_name='new_h_axis'*, *axis_limits=(0, 1)*)

Adds a new axis to the margin of this ExtendableHeatmap.

Parameters

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the figure to add the new axis to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **new_ax_name** (*str*) – A name for the new axis. The axis will be accessible as `h[new_ax_name]` where `h` is this ExtendableHeatmap instance.

Returns The newly added axis.

Return type pyplot axis

transform_coord(*coord*, *axis='x'*)

Convenience function for transforming genomic coordinates to heatmap pixel coordinates along a specified axis.

Parameters

- **coord** (*int*) – The genomic coordinate in base pairs.
- **axis** (*{'x', 'y'}*) – The axis to convert the coordinate for.

Returns The `coord` expressed in heatmap pixel coordinates along the requested axis.

Return type float

transform_feature(*feature*, *axis='x'*)

Uses `transform_coord()` to transform an entire genomic feature dict to heatmap pixel coordinates.

Parameters

- **feature** (*dict*) – Represents a genomic feature. Should have 'chrom', 'start', and 'end' keys. The values for 'start' and 'end' should be in base pair units.
- **axis** (*{'x', 'y'}*) – The axis to convert the feature for.

Returns Will have keys 'chrom', 'start', and 'end', but the values for 'start' and 'end' will now be in units of heatmap pixels along the specified axis.

Return type dict

lib5c.plotters.extendable.bed_extendable_heatmap module

Module for the BedExtendableHeatmap class, which adds bed track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.bed_extendable_heatmap.**BedExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing bed track plotting functionality.

add_bed_track(*bed_tracks*, *loc='bottom'*, *size='3%'*, *pad=0.0*, *name='bed'*, *axis_limits=(0, 1)*, *intron_height=0.05*, *colors=None*, *track_label=None*)

Adds one bed track along either the x- or y-axis of the heatmap.

Parameters

- **bed_tracks** (*list of dict*) – Each dict should represent a bed feature and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'strand': '+' or '-'
}
```

  The 'strand' key is optional and is only used for color-coding bed features when `colors` is passed.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new bed track to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the bed track.
- **intron_height** (*float*) – The height to draw each bed feature with.
- **colors** (*dict, optional*) – Map from the value of the 'strand' key in the `bed_tracks` dicts (usually '+' or '-') to color name for bed features with that strand value (i.e., orientation). If not provided for a given strand or if the bed feature doesn't have a 'strand' key the color is black by default.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added bed track axis.

Return type pyplot axis

add_bed_tracks(*bed_tracks*, *size='3%'*, *pad=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *name=None*, *colors=None*, *track_label=None*)

Adds bed tracks for a single set of bed features to both the bottom and left side of the heatmap by calling

`add_bed_track()` twice.

Parameters

- **bed_tracks** (*list of dict*) – Each dict should represent a bed feature and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'strand': '+' or '-'
}
```

  The 'strand' key is optional and is only used for color-coding bed features when `colors` is passed.

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the bed track.
- **intron_height** (*float*) – The height to draw each bed feature with.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **colors** (*dict, optional*) – Map from the value of the 'strand' key in the `bed_tracks` dicts (usually '+' or '-') to color name for bed features with that strand value (i.e., orientation). If not provided for a given strand or if the bed feature doesn't have a 'strand' key the color is black by default.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added bed track axes.

Return type list of pyplot axes

lib5c.plotters.extendable.chipseq_extendable_heatmap module

Module for the ChipSeqExtendableHeatmap class, which adds ChIP-seq track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.chipseq_extendable_heatmap.**ChipSeqExtendableHeatmap**(*array*, *grange_x* *grange_y* *colorscale=l* *colormap='* *norm=No*

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing ChIP-seq track plotting functionality.

add_chipseq_track(*features*, *loc='bottom'*, *size='10%'*, *pad=0.05*, *axis_limits=None*, *linewidth=0.4*, *name='chipseq'*, *color='k'*, *track_label=None*)

Adds one ChIP-seq track along either the x- or y-axis of the heatmap.

Parameters

- **features** (*list of dict*) – Each feature should be a dict with at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end': int,
    'value': float
}
```

  Each feature will be drawn as a rectangle on the heatmap from 'start' to 'end' with height 'value'. If the 'value' key is missing, it will be assumed to be one for all features. To get data in this form from bigwig files, consult `lib5c.contrib.pybigwig.bigwig.BigWig.query()`.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new ChIP-seq track to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float, optional*) – Axis limits for the 'value' of the plotted features (heights of the rectangles) as a (min, max) tuple. Pass None to automatically scale the axis limits.
- **linewidth** (*float*) – The linewidth to use when drawing the rectangles. Pass smaller values for sharper features/peaks.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical or 'horizontal').
- **color** (*matplotlib color*) – The color to draw the rectangles with.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added ChIP-seq track axis.

Return type pyplot axis

add_chipseq_tracks(*features*, *size='10%'*, *pad=0.05*, *axis_limits=None*, *linewidth=0.4*, *name='chipseq'*, *color='k'*, *track_label=None*)

Adds ChIP-seq tracks for a single set of features to both the bottom and left side of the heatmap by calling `add_chipseq_track()` twice.

Parameters

- **features** (*list of dict*) – Each feature should be a dict with at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end': int,
```

(continues on next page)

(continued from previous page)

```
    'value': float
}
```

Each feature will be drawn as a rectangle on the heatmap from 'start' to 'end' with height 'value'. If the 'value' key is missing, it will be assumed to be one for all features. To get data in this form from bigwig files, consult `lib5c.contrib.pybigwig.bigwig.BigWig.query()`.

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float, optional*) – Axis limits for the 'value' of the plotted features (heights of the rectangles) as a (min, max) tuple. Pass None to automatically scale the axis limits.
- **linewidth** (*float*) – The linewidth to use when drawing the rectangles. Pass smaller values for sharper features/peaks.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **color** (*matplotlib color*) – The color to draw the rectangles with.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added ChIP-seq track axes.

Return type list of pyplot axes

lib5c.plotters.extendable.cluster_extendable_heatmap module

Module for the ClusterExtendableHeatmap class, which adds cluster outlining functionality for the extendable heatmap system.

class lib5c.plotters.extendable.cluster_extendable_heatmap.**ClusterExtendableHeatmap**(*array*, *grange_x* *grange_y* *colorscale=l* *colormap='* *norm=No*

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing cluster outlining functionality.

add_clusters(*cluster_array*, *colors=None*, *weight='100x'*, *outline_color=None*, *outline_weight='2x'*, *labels=None*, *fontsize=7*)

Adds clusters to the heatmap surface.

Parameters

- **cluster_array** (*np.ndarray*) – Array of cluster IDs. Should match size and shape of the underlying array this ExtendableHeatmap was constructed with.

- **colors** (*'random' or single color or list/dict of colors or None*) – Pass 'random' for random colors, pass a dict mapping cluster IDs to matplotlib colors to outline each cluster in the indicated color, pass None to skip outlining clusters.
- **weight** (*numeric or str*) – Pass a numeric to set the linewidth for the cluster outlines. Pass a string ending in "x" (such as "100x") to specify the line width as a multiple of the inverse of the number of pixels in the heatmap.
- **outline_color** (*matplotlib color or None*) – Pass a matplotlib color to outline the outlines (e.g. with neon green) to make them stand out more. Pass None to skip adding this extra outline.
- **outline_weight** (*numeric or str*) – ass a numeric to set the linewidth for the outlines of the cluster outlines. Pass a string ending in "x" (such as "2x") to specify the line width as a multiple of the outline linewidth.
- **labels** (*True, dict of str, or None*) – Pass True to simply label the clusters by their ID. Pass a mapping from cluster IDs to labels to label the clusters with a the labels. Pass None to skip outlining clusters.
- **fontsize** (*numeric*) – The font size to use for cluster labels.

label_cluster(*cluster*, *label*, *fontsize=7*)
Labels a cluster.

Parameters

- **cluster** (*list of {'x': int, 'y': int} dicts*) – The cluster to label.
- **label** (*str*) – The string to label the cluster with.
- **fontsize** (*numeric*) – The fontsize to use for the label.

outline_cluster(*cluster*, *color*, *linewidth=2*)
Outlines a single cluster in the specified color.

Parameters

- **cluster** (*list of {'x': int, 'y': int} dicts*) – The cluster to outline.
- **color** (*matplotlib color*) – The color to outline in.
- **linewidth** (*numeric*) – The linewidth to use.

lib5c.plotters.extendable.domain_extendable_heatmap module

Module for the DomainExtendableHeatmap class, which adds domain outlining functionality for the extendable heatmap system.

class lib5c.plotters.extendable.domain_extendable_heatmap.**DomainExtendableHeatmap**(*array*, *grange_x*, *grange_y=N* *colorscale=Nor* *colormap='obs* *norm=None*

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing ChIP-seq domain outlining functionality.

outline_domain(*domain*, *color='green'*, *linewidth=2*, *upper=True*)
Outlines a contact domain on the heatmap.

Parameters

- **domain** (*dict*) – A genomic feature dict describing the domain to be outlinerd. Should be a dict with at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

'start' and 'end' should be in units of base pairs (this function will handle the conversion to heatmap pixel units).

- **color** (*matplotlib color*) – The color to outline the domain with.
- **linewidth** (*float*) – The line width to use when outlining the domain. Pass a larger number for a thicker, more visible outline.
- **upper** (*bool*) – Pass True to draw the outline in the upper triangle of the heatmap. Pass False to draw it in the lower triangle.

outline_domains(*domains*, *color='green'*, *linewidth=2*, *upper=True*)
Outlines a set of contact domains on the heatmap by repeatedly calling `outline_domain()`.

Parameters

- **domains** (*list of dict*) – A list of domains, where each domain is represented as genomic feature dict with at least the following keys:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

'start' and 'end' should be in units of base pairs (this function will handle the conversion to heatmap pixel units).

- **color** (*matplotlib color*) – The color to outline the domains with.
- **linewidth** (*float*) – The line width to use when outlining the domains. Pass a larger number for a thicker, more visible outline.
- **upper** (*bool*) – Pass True to draw the outlines in the upper triangle of the heatmap. Pass False to draw them in the lower triangle.

lib5c.plotters.extendable.extendable_figure module

Module for the ExtendableFigure base class.

class lib5c.plotters.extendable.extendable_figure.**ExtendableFigure**
Bases: `object`

Base class for figures that can interactively and sequentially tack on new axes to themselves.

Uses a `divider` attribute obtained from `mpl_toolkits.axes_grid1.make_axes_locatable()` to add new axes to the figure. Clients can call `add_ax()` to add a new axis.

All the axes in the ExtendableFigure can be accessed by name using a dict- like interface: `f[name]` where `f` is the ExtendableFigure instance and `name` is the name of the axis. The ExtendableFigure starts out with one axis already present, called 'root'.

axes
The collection of named Axes represented by this object.

Type dict of matplotlib.axes.Axes

fig
The Figure instance this object represents.

Type matplotlib.figure.Figure

divider
This object serves as a coordinator for the allocation of new Axes to be appended to this ExtendableFigure.

Type mpl_toolkits.axes_grid1.axes_divider.AxesDivider

Examples

```
>>> import numpy as np
>>> from lib5c.plotters.extendable.extendable_figure import ExtendableFigure
>>> xs = np.arange(0, 10)
>>> f = ExtendableFigure()
>>> f['root'].imshow(np.arange(100).reshape((10,10)))
<matplotlib.image.AxesImage object at ...>
>>> f.add_ax('sin')
<matplotlib.axes._axes.Axes object at ...>
>>> f['sin'].plot(xs, np.sin(xs))
[<matplotlib.lines.Line2D object at ...>]
>>> f.add_colorbar('root')
>>> f.save('test/extendablefigure.png')
```

add_ax(*name*, *loc='bottom'*, *size='10%'*, *pad=0.1*)
Adds a new axis to this ExtendableFigure.

Parameters

- **name** (*str*) – A name for the new axis. The axis will be accessible as `f[new_ax_name]` where `f` is this ExtendableFigure instance.
- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the figure to add the new axis to.
- **size** (*str*) – The size of the new axis as a percentage of the main figure size. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.

Returns The newly created axis.

Return type pyplot axis

add_colorbar(*source_ax_name*, *loc='right'*, *size='10%'*, *pad=0.1*, *new_ax_name='colorbar'*)
Adds a colorbar to the heatmap in a new axis.

Parameters

- **source_ax_name** (*str*) – The name of the axis that this should be the colorbar for. This is where matplotlib will look to find color information for the new colorbar.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the figure to add the new colorbar to.
- **size** (*str*) – The size of the new axis as a percentage of the main figure size. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **new_ax_name** (*str*) – A name for the new axis. The axis will be accessible as `f[new_ax_name]` where `f` is this ExtendableFigure instance.

close()
Clears and closes the pyplot figure related to this ExtendableFigure.

save(*filename*)
Saves this ExtendableHeatmap to the disk as an image file.

Parameters **filename** (*str*) – The filename to save the image to.

lib5c.plotters.extendable.extendable_heatmap module

class lib5c.plotters.extendable.extendable_heatmap.**ExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.chipseq_extendable_heatmap.ChipSeqExtendableHeatmap*, *lib5c.plotters.extendable.domain_extendable_heatmap.DomainExtendableHeatmap*, *lib5c.plotters.extendable.gene_extendable_heatmap.GeneExtendableHeatmap*, *lib5c.plotters.extendable.legend_extendable_heatmap.LegendExtendableHeatmap*, *lib5c.plotters.extendable.ruler_extendable_heatmap.RulerExtendableHeatmap*, *lib5c.plotters.extendable.cluster_extendable_heatmap.ClusterExtendableHeatmap*, *lib5c.plotters.extendable.snp_extendable_heatmap.SNPExtendableHeatmap*, *lib5c.plotters.extendable.motif_extendable_heatmap.MotifExtendableHeatmap*, *lib5c.plotters.extendable.rectangle_extendable_heatmap.RectangleExtendableHeatmap*, *lib5c.plotters.extendable.bed_extendable_heatmap.BedExtendableHeatmap*, *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

Fully-extended ExtendableHeatmap class. Inherits from BaseExtendableHeatmap as well as all feature-providing classes.

Examples

```
>>> import numpy as np
>>> import matplotlib.patches as patches
>>> import matplotlib.colors as colors
>>> from lib5c.parsers.counts import load_counts
>>> from lib5c.parsers.primers import load_primermap
>>> from lib5c.parsers.bed import load_features
>>> from lib5c.parsers.table import load_table
>>> from lib5c.plotters.extendable import ExtendableHeatmap
```

(continues on next page)

(continued from previous page)

```
>>> primermap = load_primermap('test/bins.bed')
>>> counts = load_counts('test/test.counts', primermap)['Sox2']
>>> h = ExtendableHeatmap(
...     array=counts,
...     grange_x={'chrom': 'chr3', 'start': 34108879, 'end': 35104879},
...     colorscale=(-10, 10),
...     colormap='obs_over_exp',
...     norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03)
... )
>>> xs = np.arange(len(counts)) + 0.5
>>> h.add_rulers()
[<matplotlib.axes._axes.Axes object at ...>,
 <matplotlib.axes._axes.Axes object at ...>]
>>> h.add_refgene_stacks('mm9', pad_before=0.1,
...                      colors={'NM_011443': 'r', 'NR_015580': 'b'})
[[<matplotlib.axes._axes.Axes object at ...>,
  <matplotlib.axes._axes.Axes object at ...>],
 [<matplotlib.axes._axes.Axes object at ...>,
  <matplotlib.axes._axes.Axes object at ...>]]
>>> h.add_legend({'Sox2': 'red', 'Sox2 OT': 'blue'})
<matplotlib.legend.Legend object at ...>
>>> boundaries = [{'chrom': 'chr3', 'start': 34459302, 'end': 34576915}]
>>> h.add_chipseq_tracks(load_features('test/tracks/CTCF_ES.bed',
...                                    boundaries=boundaries)['chr3'],
...                      name='ES CTCF')
[<matplotlib.axes._axes.Axes object at ...,
 <matplotlib.axes._axes.Axes object at ...>]
>>> h.add_chipseq_tracks(load_features('test/tracks/CTCF_NPC.bed',
...                                    boundaries=boundaries)['chr3'],
...                      name='NPC CTCF', color='r')
[<matplotlib.axes._axes.Axes object at ...,
 <matplotlib.axes._axes.Axes object at ...>]
>>> h.add_ax('sin')
<matplotlib.axes._axes.Axes object at ...>
>>> h['sin'].plot(xs, np.sin(xs))
[<matplotlib.lines.Line2D object at ...>]
>>> xlim_of_h_sin_ax = h['sin'].set_xlim((0, len(counts)))
>>> xlim_of_h_sin_ax == (0, len(counts))
True
>>> h.add_colorbar()
<matplotlib.colorbar.Colorbar object at ...>
>>> text = h['horizontal_gene_track_1'].text(
...     34558297, 0.5, 'Sox2', va='center', ha='left', fontsize=5,
...     color='r')
>>> h['horizontal_gene_track_1'].add_patch(
...     patches.Rectangle([34541337, 0], 16960, 1, fill=False, ec='r')
... )
<matplotlib.patches.Rectangle object at ...>
>>> h.outline_domains(load_features('test/communities.bed')['chr3'])
>>> h.add_clusters(
...     load_table('test/colors.tsv', load_primermap('test/bins_new.bed'),
...                dtype='U25')['x']['Sox2'][0:41, 0:41],
...     colors='random')
>>> h.add_rectangles(['chr3:34541337-34568297_chr3:34651337-34678297'])
>>> h.add_rectangles(['chr3:34541337-34568297_chr3:34651337-34678297'],
...                  color='red', transpose=False)
>>> h.save('test/extendableheatmap.png')
```

lib5c.plotters.extendable.gene_extendable_heatmap module

Module for the GeneExtendableHeatmap class, which adds gene track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.gene_extendable_heatmap.**GeneExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing gene track plotting functionality.

To deal with the fact that genes may overlap (e.g., where a gene has multiple isoforms), this class uses the concept of "gene stacks". Each gene track in a gene stack represents a separate axis added to the ExtendableHeatmap. By packing a set of genes into separate "rows", functions like `add_gene_stack()` can plot each row in the stack as a separate gene track via `add_gene_track()`.

Most commonly, we will want to add reference gene tracks corresponding to a particular genome assembly. To make this easy, this class provides the `add_refgene_stack()` and `add_refgene_stacks()` functions.

add_gene_stack(*genes*, *loc='bottom'*, *size='3%'*, *pad_before=0.0*, *pad_within=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *padding=1000*, *colors=None*)

Adds one stack of gene tracks along either the x- or y-axis of the heatmap by packing one set of genes into rows and calling `add_gene_track()` once for every row.

Parameters

- **genes** (*list of dict*) – Each dict should represent a gene and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'id'    : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

  Each block represents an exon as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

  The 'name' and 'id' keys are optional and are only used when color- coding genes. See `lib5c.parsers.genes.load_genes()`.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new gene tracks to.

- **size** (*str*) – The size of each new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad_before** (*float*) – The padding to use between the existing parts of the figure and the newly added gene tracks.
- **pad_within** (*float*) – The padding to use between each newly added gene track.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of each new gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **padding** (*int*) – The padding to use when packing genes into rows, in units of base pairs. Genes that are within this many base pairs of each other will get packed into different rows.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The newly added gene track axes, one for each row of genes.

Return type list of pyplot axis

add_gene_stacks(*genes*, *size='3%'*, *pad_before=0.0*, *pad_within=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *padding=1000*, *colors=None*)

Adds a gene stack for a set of genes to both the bottom and left side of the heatmap by calling `add_gene_stack()` twice.

Parameters

- **genes** (*list of dict*) – Each dict should represent a gene and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'id'    : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

Each block represents an exon as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

The 'name' and 'id' keys are optional and are only used when color- coding genes. See `lib5c.parsers.genes.load_genes()`.

- **size** (*str*) – The size of each new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.

- **pad_before** (*float*) – The padding to use between the existing parts of the figure and the newly added gene tracks.
- **pad_within** (*float*) – The padding to use between each newly added gene track.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of each new gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **padding** (*int*) – The padding to use when packing genes into rows, in units of base pairs. Genes that are within this many base pairs of each other will get packed into different rows.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The first element of the outer list is a list of the newly added horizontal gene track axes, one for each row of genes. The second element is the same but for the newly added vertical gene track axes.

Return type list of lists of pyplot axis

add_gene_track(*genes*, *loc='bottom'*, *size='3%'*, *pad=0.0*, *new_ax_name='genes'*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *colors=None*)

Adds one gene track (for one row of genes) along either the x- or y-axis of the heatmap.

Parameters

- **genes** (*list of dict*) – Each dict should represent a gene and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'id'    : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

Each block represents an exon as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

The 'name' and 'id' keys are optional and are only used when color- coding genes. See `lib5c.parsers.genes.load_genes()`.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new gene track to.

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **new_ax_name** (*str*) – The name for the new axis. You can access the new axis later at `h[name]` where `h` is this ExtendableHeatmap instance.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The newly added gene track axis.

Return type pyplot axis

add_gene_tracks(*genes*, *size='3%'*, *pad=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *colors=None*)

Adds a gene track for a single row of genes to both the bottom and left side of the heatmap by calling `add_gene_track()` twice.

Parameters

- **genes** (*list of dict*) – Each dict should represent a gene and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'id'    : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

Each block represents an exon as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

The 'name' and 'id' keys are optional and are only used when color- coding genes. See `lib5c.parsers.genes.load_genes()`.

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.

- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The newly added gene track axes.

Return type list of pyplot axes

add_refgene_stack(*assembly*, *loc='bottom'*, *size='3%'*, *pad_before=0.0*, *pad_within=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *padding=1000*, *colors=None*)

Adds a gene stack to either the x- or y-axis of the heatmap by getting a set of reference genes for a specified genome assembly, and then passes that set of genes to `add_gene_stack()`.

Parameters

- **assembly** (*{'hg18', 'hg19', 'hg38', 'mm9', 'mm10'}*) – The genome assembly to load reference genes for.
- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new gene tracks to.
- **size** (*str*) – The size of each new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad_before** (*float*) – The padding to use between the existing parts of the figure and the newly added gene tracks.
- **pad_within** (*float*) – The padding to use between each newly added gene track.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of each new gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **padding** (*int*) – The padding to use when packing genes into rows, in units of base pairs. Genes that are within this many base pairs of each other will get packed into different rows.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The newly added gene track axes, one for each row of genes.

Return type list of pyplot axis

add_refgene_stacks(*assembly*, *size='3%'*, *pad_before=0.0*, *pad_within=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *padding=1000*, *colors=None*)

Adds a gene stack for a set of genes to both the bottom and left side of the heatmap by calling `add_refgene_stack()` twice.

Parameters

- **assembly** (*{'hg18', 'hg19', 'hg38', 'mm9', 'mm10'}*) – The genome assembly to load reference genes for.
- **size** (*str*) – The size of each new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad_before** (*float*) – The padding to use between the existing parts of the figure and the newly added gene tracks.
- **pad_within** (*float*) – The padding to use between each newly added gene track.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of each new gene track.
- **intron_height** (*float*) – Controls thickness of gene introns. Pass a larger number for thicker introns.
- **exon_height** (*float*) – Controls thickness of gene exons. Pass a larger number for thicker exons.
- **padding** (*int*) – The padding to use when packing genes into rows, in units of base pairs. Genes that are within this many base pairs of each other will get packed into different rows.
- **colors** (*dict, optional*) – Pass a dict mapping gene names or id's to matplotlib colors to color code those genes. Genes not in the dict will be colored black by default. Using gene names as keys should color all isoforms, while using gene id's as keys should color just the isoform matching the specified id.

Returns The first element of the outer list is a list of the newly added horizontal gene track axes, one for each row of genes. The second element is the same but for the newly added vertical gene track axes.

Return type list of lists of pyplot axis

lib5c.plotters.extendable.legend_extendable_heatmap module

Module for the LegendExtendableHeatmap class, which adds functionality for adding a legend to the extendable heatmap system.

class lib5c.plotters.extendable.legend_extendable_heatmap.**LegendExtendableHeatmap**(*array*, *grange_x*, *grange_y=N* *col-* *orscale=Nor* *col-* *ormap='obs* *norm=None*

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing functionality for adding a legend.

add_legend(*colors*, ***kwargs*)
Adds a legend to the ExtendableHeatmap.

Parameters

- **colors** (*dict*) – The entries to add to the legend. Keys should be string labels to use in the legend, and values should be matplotlib colors.
- **kwargs** (*kwargs*) – Will be passed through to `ax.legend()`.

Returns The newly created Legend.

Return type matplotlib.legend.Legend

lib5c.plotters.extendable.motif_extendable_heatmap module

Module for the MotifExtendableHeatmap class, which adds motif track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.motif_extendable_heatmap.**MotifExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing motif track plotting functionality.

add_motif_track(*motifs*, *loc='bottom'*, *size='3%'*, *pad=0.0*, *name='motif'*, *axis_limits=(0, 1)*, *intron_height=0.05*, *motif_linewidth=1*, *exon_height=0.5*, *colors=None*, *track_label=None*)
Adds one motif track along either the x- or y-axis of the heatmap.

Parameters

- **motifs** (*list of dict*) – Each dict should represent a motif instance and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'strand': '+' or '-'
}
```

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new motif track to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').

- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the motif track.
- **intron_height** (*float*) – Controls height of the rectangle spanning the length of the motif.
- **motif_linewidth** (*float*) – The linewidth to use when plotting.
- **exon_height** (*float*) – Controls the size of the arrowhead that indicates the motif orientation.
- **colors** (*dict, optional*) – Map from the value of the 'strand' key in the `motifs` dicts (usually '+' or '-') to color name for motifs with that strand value (i.e., orientation). If not provided for a given strand, color is black by default.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added motif track axis.

Return type pyplot axis

add_motif_tracks(*motifs*, *size='3%'*, *pad=0.0*, *axis_limits=(0, 1)*, *intron_height=0.05*, *exon_height=0.5*, *name='motif'*, *motif_linewidth=1*, *colors=None*, *track_label=None*)

Adds motif tracks for a single set of motifs to both the bottom and left side of the heatmap by calling `add_motif_track()` twice.

Parameters

- **motifs** (*list of dict*) – Each dict should represent a motif instance and could have the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'strand': '+' or '-'
}
```

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the motif track.
- **intron_height** (*float*) – Controls height of the rectangle spanning the length of the motif.
- **exon_height** (*float*) – Controls the size of the arrowhead that indicates the motif orientation.
- **name** (*str*) – Base name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **motif_linewidth** (*float*) – The linewidth to use when plotting.
- **colors** (*dict, optional*) – Map from the value of the 'strand' key in the `motifs` dicts (usually '+' or '-') to color name for motifs with that strand value (i.e., orientation). If not provided for a given strand, color is black by default.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added motif track axes.

Return type list of pyplot axes

lib5c.plotters.extendable.rectangle_extendable_heatmap module

Module for the RectangleExtendableHeatmap class, which adds rectangle outlining functionality to the extendable heatmap system.

class lib5c.plotters.extendable.rectangle_extendable_heatmap.**RectangleExtendableHeatmap**(*arr*
gra
gra
col-
ors
col-
orm
nor

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing rectangle outlining functionality.

Unlike cluster outlining, which outlines specific pixels of the heatmap, or domain outlining, which outlines the upper or lower triangles of diagonal- aligned squares corresponding to domains, rectangle outlining simply draws a rectangle on the heatmap whose sides are described by genomic ranges.

This can be used to outline cluster bounding boxes.

add_rectangle(*coords*, *color='#00FF00'*, *weight='100x'*, *transpose=True*)

Draw some rectangles (specified in genomic coordinates) on the heatmap.

Parameters

- **coords** (*tuple of dict or str*) – Genomic coordinates for the rectangle to be plotted. Can be a tuple:

```
({'chrom': str, 'start': int, 'end': int},
 {'chrom': str, 'start': int, 'end': int})
```

  where the first dict specifies the genomic coordinates of one side of the rectangle, and the second dict specifies the genomic coordinates of the other side. Can also be a string of the form `chrom:start-end_chrom:start-end`, where the _ separates genomic ranges which specify the coordinates of the two sides of the rectangle.

- **color** (*valid matplotlib color*) – The edge color to draw the rectangle with.
- **weight** (*int or str*) – The line width to use to draw the rectangle. Pass a string of the form `'100x'` to specify the line width as a multiple of the inverse of the size of this heatmap's array.
- **transpose** (*bool*) – Pass True to interpret the first feature in `coords` as the side of the rectangle parallel to the y-axis. Pass False to interpret it as the side parallel to the x-axis.

add_rectangles(*coords_list*, *color='#00FF00'*, *weight='100x'*, *transpose=True*)

Draw some rectangles (specified in genomic coordinates) on the heatmap.

Parameters

- **coords_list** (*list of tuple of dict or list of str*) – List of sets genomic coordinates (one set for each rectangle) to be plotted. The list can contain tuples:

```
({'chrom': str, 'start': int, 'end': int},
 {'chrom': str, 'start': int, 'end': int})
```

where the first dict specifies the genomic coordinates of one side of the rectangle, and the second dict specifies the genomic coordinates of the other side. The list can also contain strings of the form `chrom:start-end_chrom:start-end`, where the _ separates genomic ranges which specify the coordinates of the two sides of the rectangle.

- **color** (*valid matplotlib color*) – The edge color to draw the rectangles with.
- **weight** (*int or str*) – The line width to use to draw the rectangles. Pass a string of the form `'100x'` to specify the line width as a multiple of the inverse of the size of this heatmap's array.
- **transpose** (*bool*) – Pass True to interpret the first feature in each tuple or string as the side of the rectangle parallel to the y-axis. Pass False to interpret it as the side parallel to the x-axis.

lib5c.plotters.extendable.ruler_extendable_heatmap module

Module for the RulerExtendableHeatmap class, which adds ruler track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.ruler_extendable_heatmap.**RulerExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing ruler track plotting functionality.

add_ruler(*loc='bottom'*, *size='5%'*, *pad=0.0*, *new_ax_name='ruler'*, *axis_limits=(1, 0)*, *ruler_tick_height=0.3*, *ruler_text_baseline=0.5*, *no_ticks=False*, *fontsize=7*, *no_tick_precision=2*)

Adds one ruler track along either the x- or y-axis of the heatmap.

Parameters

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new ruler track to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **new_ax_name** (*str*) – The name for the newly created axis.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the ruler track.
- **ruler_tick_height** (*float*) – The height of the ruler ticks.

- **ruler_text_baseline** (*float*) – Controls how far away from the top of the new axis the text elements (chromosome name and region size) will be drawn.
- **no_ticks** (*bool*) – Pass True to skip plotting ruler ticks, and instead write the start and end coordinate of the plotted region in units of megabase pairs.
- **fontsize** (*float*) – The font size to use for adding labels to the ruler axis.
- **no_tick_precision** (*int*) – When `no_ticks` is True, round the start and end coordinates of the plotted region to this many digits after the decimal.

Returns The newly added ruler track axis.

Return type pyplot axis

add_rulers(*size='5%'*, *pad=0.0*, *axis_limits=(1*, *0)*, *ruler_tick_height=0.3*, *ruler_text_baseline=0.5*, *no_ticks=False*, *fontsize=7*, *no_tick_precision=2*)

Adds a ruler track to both the bottom and left side of the heatmap by calling `add_ruler()` twice.

Parameters

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float*) – Axis limits for the non-genomic axis of the ruler track.
- **ruler_tick_height** (*float*) – The height of the ruler ticks.
- **ruler_text_baseline** (*float*) – Controls how far away from the top of the new axis the text elements (chromosome name and region size) will be drawn.
- **no_ticks** (*bool*) – Pass True to skip plotting ruler ticks, and instead write the start and end coordinate of the plotted region in units of megabase pairs.
- **fontsize** (*float*) – The font size to use for adding labels to the ruler axis.
- **no_tick_precision** (*int*) – When `no_ticks` is True, round the start and end coordinates of the plotted region to this many digits after the decimal.

Returns The newly added gene ruler axes.

Return type list of pyplot axes

lib5c.plotters.extendable.snp_extendable_heatmap module

Module for the SNPExtendableHeatmap class, which adds SNP track plotting functionality for the extendable heatmap system.

class lib5c.plotters.extendable.snp_extendable_heatmap.**SNPExtendableHeatmap**(*array*, *grange_x*, *grange_y=None*, *colorscale=None*, *colormap='obs'*, *norm=None*)

Bases: *lib5c.plotters.extendable.base_extendable_heatmap.BaseExtendableHeatmap*

ExtendableHeatmap mixin class providing SNP track plotting functionality.

add_snp_track(*snps*, *loc='bottom'*, *size='3%'*, *pad=0.0*, *name='snp'*, *axis_limits=(0, 1)*, *snp_height=0.5*, *colors=None*, *track_label=None*)

Adds one SNP track along either the x- or y-axis of the heatmap.

Parameters

- **snps** (*list of dict*) – Each dict should represent a SNP and should have at least the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int
}
```

  If color-coding SNPs, include a 'name' or 'id' key.

- **loc** (*{'top', 'bottom', 'left', 'right'}*) – Which side of the heatmap to add the new SNP track to.
- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **name** (*str*) – The name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **axis_limits** (*tuple of float*) – Limits for the non-genomic axis of the SNP track.
- **snp_height** (*float*) – Height of SNP arrowheads in same units as `axis_limits`.
- **colors** (*dict, optional*) – Maps SNP ids or names to color names. Defaults to black if not passed or if a SNP's name or id are not found in the dict.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added SNP track axis.

Return type pyplot axis

add_snp_tracks(*snps*, *size='3%'*, *pad=0.0*, *axis_limits=(0, 1)*, *snp_height=0.5*, *name='snp'*, *colors=None*, *track_label=None*)

Adds SNP tracks for a single set of SNPs to both the bottom and left side of the heatmap by calling `add_snp_track()` twice.

Parameters

- **snps** (*list of dict*) – Each dict should represent a SNP and should have at least the following keys:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int
}
```

  If color-coding SNPs, include a 'name' or 'id' key.

- **size** (*str*) – The size of the new axis as a percentage of the main heatmap width. Should be passed as a string ending in '%'.
- **pad** (*float*) – The padding to use between the existing parts of the figure and the newly added axis.
- **axis_limits** (*tuple of float*) – Limits for the non-genomic axis of the SNP track.
- **snp_height** (*float*) – Height of SNP arrowheads in same units as `axis_limits`.
- **name** (*str*) – The name for the new axis. This name will be prefixed with the orientation of the track ('vertical' or 'horizontal').
- **colors** (*dict, optional*) – Maps SNP ids or names to color names. Defaults to black if not passed or if a SNP's name or id are not found in the dict.
- **track_label** (*str, optional*) – Pass a string to label the track.

Returns The newly added SNP track axes.

Return type list of pyplot axes

Module contents

Module describing the extendable heatmap plotting system.

The core idea behind this system is the ExtendableFigure API. Subclasses of ExtendableFigure inherit the `add_ax()` instance method, which tacks on additional matplotlib axes to an existing axis using a `divider` object obtained using the matplotlib function `mpl_toolkits.axes_grid1.make_axes_locatable()`. The new axes are stored in an `axes` dict instance variable on the ExtendableFigure and can subsequently be plotted on with any matplotlib-based function.

This means that an arbitrary number of additional data tracks can be added to the ExtendableFigure in a sequential, interactive manner.

The BaseExtendableHeatmap class extends ExtendableFigure and implements extra functionality specific for plotting contact frequency heatmaps. This includes a custom interface to `add_ax()` called `add_margin_ax()`, designed to make it easy to add tracks to the margins of a contact frequency heatmap.

In order to handle the wide diversity of tracks and overlays that may be added to a contact frequency heatmap, a wide variety of plug-in mixin classes are defined. These mixin classes each implement one specific feature in the form of a few related instance methods that internally call `add_margin_ax()`. For example, ChipSeqExtendableHeatmap provides a function `add_chipseq_track()` which uses `add_margin_ax()` to add a ChIP-seq track to the ExtendableHeatmap. The separation of different features into separate mixin classes makes the code easier to understand and also makes it easy to add new features.

The feature functions like `add_chipseq_track()` typically accept some data as well as parameters for creating the new axis. The functions typically create the new axis using `add_margin_ax()`, passing through the relevant parameters. The functions then plot the data to the newly created axis using matplotlib plotting functions. Finally, they return the new axis.

By convention, we often plot tracks on both the left and the bottom of the heatmap. To make this easier, most feature functions like `add_chipseq_track()` have associated helper functions like `add_chipseq_tracks()`, which simply calls `add_chipseq_track()` twice: once with `loc='bottom'` and once with `loc='left'`.

To add a new feature:

1. create a new mixin class extending BaseExtendableHeatmap that implements the functions for your feature,
2. place your new class in the `lib5c.plotters.extendable` module,

3. import your class in the top level module file `__init__.py` and add it to the `__all__` variable, and
4. add your class to the list of classes inherited by ExtendableHeatmap, making sure to place it before BaseExtendableHeatmap in the list.

In the future, this process may be automated.

Submodules

lib5c.plotters.asymmetric_colormap module

Module providing a function to create shifted colormaps.

lib5c.plotters.asymmetric_colormap.**shifted_colormap**(*cmap*, *start=0*, *midpoint=0.5*, *stop=1.0*, *name='shiftedcmap'*)

Function to offset the "center" of a colormap. Useful for data with a negative min and positive max and you want the middle of the colormap's dynamic range to be at zero

Parameters

- **cmap** (*matplotlib colormap*) – The matplotlib colormap to be altered
- **start** (*Optional[float]*) – Offset from lowest point in the colormap's range. Defaults to 0.0 (no lower ofset). Should be between 0.0 and `midpoint`.
- **midpoint** (*Optional[float]*) – The new center of the colormap. Defaults to 0.5 (no shift). Should be between 0.0 and 1.0. In general, this should be `1 - vmax/(vmax + abs(vmin))`. For example if your data range from -15.0 to +5.0 and you want the center of the colormap at 0.0, `midpoint` should be set to `1 - 5/(5 + 15))` or 0.75
- **stop** (*Optional[float]*) – Offset from highest point in the colormap's range. Defaults to 1.0 (no upper ofset). Should be between `midpoint` and 1.0.
- **name** (*Optional[str]*) – The name under which to register the new colormap.

lib5c.plotters.bias_heatmaps module

Module for plotting bias heatmaps.

lib5c.plotters.bias_heatmaps.**plot_bias_heatmap**(*obs_counts*, *exp_counts*, *primermap*, *factor*, *bins=None*, *n_bins=None*, *cmap=None*, *vmin=None*, *vmax=None*, *midpoint=None*, *log=True*, *region=None*, *agg=<function gmean>*, *asymmetric=False*, *print_variance=False*, *shuffle=0*, *zero_inflated=False*, *unique=False*, *despine=False*, *style='dark'*, *dpi=300*, ***kwargs*)

Plots a bias heatmap.

Parameters

- **obs_counts** (*Dict[str, np.ndarray]*) – The dict of observed counts.
- **exp_counts** (*Dict[str, np.ndarray]*) – The dict of expected counts.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – Primermap or pixelmap describing the loci in `obs_counts` and `exp_counts`.

- **factor** (*str*) – The bias factor to draw the bias heatmap for. This string must match a metadata key in `primermap`. That is to say, if `factor` is `'length'` then we expect `primermap[region][i]['length']` to be a number representing the length of the `i` th fragment in the region specified by `region`.
- **bins** (*Optional[Sequence[numeric]]*) – The endpoints of the bins to use to stratify the bias factor values. Either `bins` or `n_bins` must be specified.
- **n_bins** (*Optional[int]*) – The number of even-number bins to use to stratify the bias factor values. Either `bins` or `n_bins` must be specified.
- **cmap** (*Optional[matplotlib.colors.Colormap]*) – Pass a colormap to use for the heatmap. If this kwarg is not passed, the default 'bias' colormap is used.
- **vmin** (*Optional[float]*) – The minimum value to use for the heatmap. If this kwarg is not passed, the min of the data will be used.
- **vmax** (*Optional[float]*) – The maximum value to use for the heatmap. If this kwarg is not passed, the max of the data will be used.
- **midpoint** (*Optional[float]*) – The midpoint value to use for the colormap. If this kwarg is not passed, the colormap will be symmetric about its midpoint. This kwarg can be used to force the midpoint of the colormap to lie at a desired value, such as 0.
- **log** (*bool*) – Whether or not to show log-scale fold-enrichments in the heatmap.
- **region** (*Optional[str]*) – Pass a region name as a string to consider only the contacts in one particular region. If this kwarg is not passed, contacts for all regions in the input counts dicts will be used to generate the bias heatmap.
- **agg** (*Callable[[np.ndarray], float]*) – The aggregation function to use when summarizing the strata. This function should take in an array of floats and return a single summary value.
- **asymmetric** (*bool*) – Pass True to construct heatmaps using only the upper-triangular elements of the counts matrices, which can lead to asymmetric heatmaps. By default, the algorithm iterates over all elements of the counts matrices, enforcing symmetry in the bias models but incurring some redundancy in the actual counts information.
- **print_variance** (*bool*) – If True, the variance of the bias across the stratification grid will be printed in the plot title.
- **shuffle** (*int*) – Specify a number of random permutation null hypothesis simulations to perform.
- **zero_inflated** (*bool*) – Pass True here to treat the bias factor as "zero inflated", which will cause all the zero values to land in a dedicated "zero stratum" and allocate the remaining bins evenly among the positive data. This kwarg is ignored if the bins are passed explicitly.
- **unique** (*bool*) – Pass True to override *bins* and *n_bins* and simply put each unique value of the bias factor into its own stratum.
- **dpi** (*int*) – DPI to save figure at if auto-saving to a raster format.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The first element is the pyplot axis plotted on, which will always be present. The second element is the variance of the enrichment with respect to the bias factor grid. The third element is the percentile value for this variance obtained from simulations under the null hypothesis. The fourth element is the 95% RI for the null hypothesis. These last three will be nan if no simulations were performed.

Return type Tuple[pyplot axis, float]

Notes

The simulations were a cool idea, but in reality the 95% null hypothesis RI is incredibly small since it is very unlikely to see any enrichment at all if the obs and exp counts are forcibly de-correlated from the bias factors. Therefore the recommendation is to not simulate unless you're really sure you want it.

lib5c.plotters.boxplots module

Module for plotting locus boxplots that show the distribution of 5C counts at each locus across a region.

lib5c.plotters.boxplots.**plot_regional_locus_boxplot**(*regional_counts*, *color='darkgray'*, *median_color='firebrick'*, *median_linewidth=5.0*, *logged=True*, *sort=True*, *figsize=None*, *scaling_factor=0.05*, *dpi=300*, *despine=False*, ***kwargs*)

Plots a locus boxplot visualization showing the distribution of 5C counts at each locus across a region.

Parameters

- **regional_counts** (*np.ndarray*) – The matrix of counts for this region.
- **color** (*str*) – Color to fill the boxplots.
- **median_color** (*str*) – Color to mark the medians of the boxplots with.
- **median_linewidth** (*str*) – Linewidth to draw the medians of the boxplots with.
- **logged** (*bool*) – Pass True to use a log-scale counts axis.
- **sort** (*bool*) – Pass True to sort the boxplots from left to right in order of increasing median value.
- **figsize** (*Optional[Tuple[float, float]]*) – Pass a tuple of the form `(width, height)` to force the size of the figure. If this kwarg is not passed, the figure size will be determined automatically as `scaling_factor` times the number of loci in the region.
- **scaling_factor** (*float*) – If `figsize` is not passed, the figure size will be determined automatically as `scaling_factor` times the number of loci in the region.
- **dpi** (*int*) – DPI to save figure at if auto-saving to a raster format.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.clustering module

Module for visualizing architectural clusters.

lib5c.plotters.clustering.**compute_bounding_box**(*cluster_peaks*)

Computes a rectangular bounding box for a cluster.

Parameters **cluster_peaks** (*list of dicts of ints*) – Each dict in the list represents a peak in the cluster and has the form:

```
{
    'x': int,
    'y': int
}
```

where these ints represent the respective x- and y-coordinates of the peak within the region.

Returns

The returned dict represents the computed bounding box and has the form:

```
{
    'x_min': int,
    'x_max': int,
    'y_min': int,
    'y_max': int
}
```

where the ints describe the maximal and minimal indices of pixels along the x- and y-axes to be included within the bounding box, respectively.

Return type dict of ints

lib5c.plotters.clustering.**main**()

lib5c.plotters.clustering.**make_zoom_window**(*bounding_box*, *region_size*, *padding=2*, *invert=False*)

Computes a zoom window around a target bounding box.

Parameters

- **bounding_box** (*dict of ints*) – This dict represents the target bounding box and should have the form:

```
{
    'x_min': int,
    'x_max': int,
    'y_min': int,
    'y_max': int
}
```

where the ints describe the maximal and minimal indices of pixels along the x- and y-axes included within the bounding box, respectively.

- **region_size** (*int*) – The side length of the square region within which the target bounding box lies, in pixels.
- **padding** (*int*) – The number of pixels to include within the zoom window around the bounding box. If the zoom window is at the edge of the region, it may be impossible to pad by the specified number of pixels, in which case the zoom window will be shifted away from the edge of the region.
- **invert** (*bool*) – If True, the zoom window will be reflected across the diagonal.

Returns

This dict represents the computed zoom window and has the following structure:

```
{
    'x_start': int,
    'y_start': int,
```

(continues on next page)

(continued from previous page)

```
    'size': int
}
```

where x_start and y_start refer to the indices of the lowest-indexed pixel to be included in the zoom window along each axis, and size indicates the side length of the square zoom window in pixels.

Return type dict of int

lib5c.plotters.clustering.**plot_cluster**(*pixelmap*, *counts_superdict*, *cluster_peaks*, *cluster_region*, *colorscales='auto'*, *tracks=()*, *track_filename_generator=<function <lambda>>*, *conditions=()*, *track_scales='auto'*, *zoom_window='auto'*, *padding=2*, *invert=False*, *output_filename_generator=<function <lambda>>*, *heatmap_kwargs='default'*)

Plots a contact probability heatmaps centered on a cluster.

Parameters

- **pixelmap** (*pixelmap*) – The pixelmap to use for plotting the heatmap. See `lib5c.parsers.bed.get_pixelmap()`.
- **counts_superdict** (*dict of counts dicts*) – The contact probability data for each replicate to be plotted. The keys of the outer dict are replicate names as strings. The values of the outer dict are counts dicts whose keys are region names as strings and whose values are 2D square symmetric numpy arrays containing the counts for that region in that replicate. See `lib5c.util.counts_superdict`.
- **cluster_peaks** (*list of dicts of ints*) – Each dict in the list represents a peak in the cluster and has the form:

```
{
    'x': int,
    'y': int
}
```

where these ints represent the respective x- and y-coordinates of the peak within the region.

- **cluster_region** (*str*) – The name of the region in which the cluster lies.
- **colorscales** (*'auto' or dict of lists of numerics*) – If 'auto' is passed, the regional colorscales for the heatmaps are automatically computed using `lib5c.util.scales.compute_regional_obs_over_exp_scale()`. Alternatively, the colorscales may be specified as a dict whose keys are region names and whose values are lists of numerics of length two. The first element of this list will be the minumum value on the colorscale, while the second element will be the maximum value on the colorscale.
- **tracks** (*list of str*) – List of string identifiers for ChIP-seq tracks to include on the heatmaps. The identifiers will be used to find the appropriate BED files on the disk according to track_filename_generator.
- **track_filename_generator** (*function str -> str*) – When passed any string from tracks, this function should return a string reference to the BED file on the disk containing the BED features for that track.
- **conditions** (*list of str*) – List of string identifiers for distinct conditions in the dataset. If this list is not empty, it will be used to group replicates and tracks into distinct

groups that contain the identifiers in this list as substrings of their names. Replicates identified as belonging to a certain condition will be plotted with the tracks identified as belonging to that same condition. For example, consider a conditions list ['ES', 'NPC'], replicates 'NPC_Rep1' and 'ES_Rep1', and tracks 'ES_CTCF' and 'NPC_CTCF'. The 'ES_Rep1' heatmap will be drawn with the 'ES_CTCF' track, and the 'NPC_Rep1' heatmap will be drawn with the 'NPC_CTCF' track. If this list is empty, all tracks will be drawn with all heatmaps. Additionally, this kwarg is included in the call to `lib5c.util.scales.compute_track_scales()`, which computes the track scales if track_scales is 'auto' (see below).

- **track_scales** (*'auto' or dict of numerics*) – If 'auto' is passed, the track scales will be computed automatically using `lib5c.util.scales.compute_track_scales()`. The conditions kwarg will be included in this call. For example, consider a conditions list ['ES', 'NPC'], and tracks 'ES_CTCF' and 'NPC_CTCF'. Since the track names differ only in their condition identifier, these two tracks will be plotted using the same scale if track_scales is 'auto'. Alternatively, the track scales may be specified as a dict whose keys are the track names and whose values are numerics specifying the maximum value of the scale for that track. The minumum value of all tracks is assumed to be zero.
- **zoom_window** (*'auto' or dict of int*) – If 'auto' is passed, a zoom window for the cluster will be automatically computed. Alternatively, the zoom window may be specified as a dict with the following structure:

```
{
    'x_start': int,
    'y_start': int,
    'size': int
}
```

  where x_start and y_start refer to the indices of the lowest-indexed pixel to be included in the zoom window along each axis, and size indicates the side length of the square zoom window in pixels.
- **padding** (*int*) – If zoom_window is 'auto', this kwarg specifies how much padding to include around the cluster boundaries when computing the zoom window.
- **invert** (*bool*) – If zoom_window is 'auto', this kwarg specifies whether or not the automatically computed zoom window should be reflected across the diagonal, reversing the x- and y-axes on the resulting heatmaps.
- **output_filename_generator** (*function (str, str) -> str*) – This function should take in cluster_region and the name of a replicate and return the output filename to write the corresponding heatmap to.
- **heatmap_kwargs** (*'default' or dict*) – Passing a dict to this kwarg allows the specification of arbitrary kwargs to be included in the final call to `lib5c.plotters.heatmap.plot_heatmap()`. If 'default' is passed instead of a dict, a default dict of typical kwargs will be used.

lib5c.plotters.clustering.**plot_cluster_indices**(*pixelmap*, *clusters*, *output_filename_generator=<function <lambda>>*, *heatmap_kwargs='default'*)

Plots clusters for each region where each cluster is in a different color and each cluster is labeled with its index in the list of clusters for that region.

Parameters

- **pixelmap** (*pixelmap*) – The pixelmap to use for plotting the heatmap. See `lib5c.parsers.bed.get_pixelmap()`.
- **clusters** (*dict of lists of lists of dicts of ints*) – The keys of the outer dict are region names. The values (the outer lists) represent the list of clusters within each region. The inner lists represent clusters. Each dict in each inner list represents a peak in that cluster and has the form:

```
{
    'x': int,
    'y': int
}
```

  where these ints represent the respective x- and y-coordinates of the peak within the region. As an example, we should be able to index into clusters as follows:

```
clusters[region_name : str][index of cluster within region : int]
↪                   [index of peak within cluster : int] =
↪         {
        'x': int,
        'y': int
    }
```

- **output_filename_generator** (*function str -> str*) – This function should take in a region name and return the output filename to which the cluster index heatmap for that region should be written.
- **heatmap_kwargs** (*'default' or dict*) – Passing a dict to this kwarg allows the specification of arbitrary kwargs to be included in the final call to `lib5c.plotters.heatmap.plot_heatmap()`. If 'default' is passed instead of a dict, a default dict of typical kwargs will be used.

lib5c.plotters.colormaps module

Module for resolving string identifiers to matplotlib colormaps.

`lib5c.plotters.colormaps.`**get_colormap**(*name*, *reverse=False*, *set_bad=None*)

Get a colormap given its name.

Parameters

- **name** (*str*) – The name of the colormap. See the Notes for special values.
- **reverse** (*bool*) – Pass True to reverse the colormap.
- **set_bad** (*str, optional*) – Color to set as the `set_bad` color on the returned colormap. This is commonly used to represent NaN or undefined values.

Returns The requested colormap.

Return type matplotlib.colors.Colormap

Notes

If `name` matches a built-in matplotlib colormap, that colormap will be returned. If `name` matches one of the following special values, the corresponding specialized colomap will be returned:

- 'obs_over_exp': a colormap for visualizing fold changes in interaction frequencies

- 'is': a colormap for plotting interaction score heatmaps
- 'obs': a colormap for visualizing observed interaction frequencies
- 'bias': a colormap for plotting bias factor heatmaps

You can append '_bad_<color>' to any of these to set the set_bad color, for example: 'abs_obs_bad_green'

lib5c.plotters.colorscales module

Module containing useful colorscales with which to plot 5C heatmaps.

These colormaps are deprecated in favor of the colormap interface provided by `lib5c.plotters.colormaps.get_colormap()`.

lib5c.plotters.convergency module

Module for plotting bar charts illustrating motif convergency quantification results computed using the `lib5c.algorithms.convergency` module.

`lib5c.plotters.convergency.`**`plot_convergency`**(*convergency_results*, ***kwargs*)

Plots a convergency bar plot.

Parameters

- **`convergency_results`** (*dict*) – Return value of `lib5c.algorithms.convergency.compute_convergency()`.
- **`kwargs`** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.correlation module

Module for plotting pairwise correlation matrices.

`lib5c.plotters.correlation.`**`plot_correlation_matrix`**(*matrix*, *label_values=None*, *cluster=False*, *cbar=False*, *cmap='rocket_r'*, *colorscale=None*, *despine=False*, *style='dark'*, ***kwargs*)

Plots a pairwise corrrelation matrix as a heatmap.

Parameters

- **`matrix`** (*np.ndarray*) – The pairwise correlation matrix to visualize.
- **`label_values`** (*Optional[List[str]]*) – A list of strings labeling the columns of the matrix. If not passed, no labels will be included.
- **`cluster`** (*bool*) – Pass True to perform heirarchical clustering on the rows and columns of the matrix.
- **`cbar`** (*bool*) – Pass True to include a colorbar.
- **`cmap`** (*matplotlib colormap*) – Choose the colormap to use in the heatmap.
- **`colorscale`** (*Optional[Tuple[int]]*) – Pass a colorscale to use for the plot.

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

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.curve_fits module

Module for plotting curves fitted to x-y graphs.

lib5c.plotters.curve_fits.**plot_fit**(*x*, *y*, *fit*, *n_points=None*, *logx=True*, *logy=True*, *hexbin=True*, *colors=None*, *linewidth=4*, *xlim=None*, *ylim=None*, ***kwargs*)

Plots a fit over data.

Parameters

- **y** (*x,*) – The data points.
- **fit** (*function or np.ndarray or dict*) – If a function is passed, it should return an estimate of y as a function of x. If an `np.ndarray` is passed, it should be parallel to `x` and should contain the estimate of y for each x value. Pass a dict of functions or `np.ndarray` to plot multiple estimates.
- **n_points** (*int, optional*) – Pass an integer to subsample `x` with this many points when drawing the curve. Pass None to draw the curve using all values in `x`.
- **logy** (*logx,*) – Log the x- and/or y-axis.
- **hexbin** (*bool*) – Pass True to plot a hexbin plot instead of a scatterplot.
- **colors** (*dict*) – If `fit` is a dict, pass a dict mapping the keys of `fit` to valid matplotlib colors to force the colors of the curves.
- **linewidth** (*float*) – Line width to draw the fit with.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

Notes

If both xlim and ylim are passed as kwargs and hexbin=True, this function will attempt to set the extent of the hexbin plot using the xlim and ylim.

lib5c.plotters.distance_dependence module

Module for plotting distance dependence curves.

lib5c.plotters.distance_dependence.**plot_distance_dependence**(*counts_superdict*, *primermap*, *region=None*, *bins=None*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots distance dependence curves.

Parameters

- **counts_superdict** (*counts_superdict*) – The data to plot curves for.
- **primermap** (*primermap*) – The primermap associated with the counts_superdict.
- **region** (*str or None*) – Pass None to combine distance dependence information across all regions. Pass a region name to plot only that region's distance dependence curve.
- **bins** (*list of numeric*) – Bins to use to stratify interactions into distance groups. Should be in units of basepairs.
- **labels** (*dict or None*) – Pass a dict mapping the keys of counts_superdict to labels for plotting, or pass None to use the keys of counts_superdict as the labels.
- **levels** (*dict or None*) – Pass a dict mapping labels to levels to color-code the replicates by level. Pass None to give each replicate its own level.
- **colors** (*dict or None*) – Pass a dict mapping levels to matplotlib colors to decide what color to plot each level with. Pass None to automatically choose colors.
- **hue_order** (*list of str or None*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.distance_dependence.**plot_distance_dependence_parallel**(*counts_superdict*, *primermap*, *region=None*, *bins=None*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots distance dependence curves.

Parameters

- **counts_superdict** (*counts_superdict*) – The data to plot curves for.
- **primermap** (*primermap*) – The primermap associated with the counts_superdict.

- **region** (*str or None*) – Pass None to combine distance dependence information across all regions. Pass a region name to plot only that region's distance dependence curve.
- **bins** (*list of numeric*) – Bins to use to stratify interactions into distance groups. Should be in units of basepairs.
- **labels** (*dict or None*) – Pass a dict mapping the keys of counts_superdict to labels for plotting, or pass None to use the keys of counts_superdict as the labels.
- **levels** (*dict or None*) – Pass a dict mapping labels to levels to color-code the replicates by level. Pass None to give each replicate its own level.
- **colors** (*dict or None*) – Pass a dict mapping levels to matplotlib colors to decide what color to plot each level with. Pass None to automatically choose colors.
- **hue_order** (*list of str or None*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.distribution module

Module for plotting counts distributions.

lib5c.plotters.distribution.**plot_global_distributions**(*counts_superdict*, *logged=True*, *drop_zeros=False*, *shade=True*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots overlayed global distributions for many replicates from a counts superdict.

Parameters

- **counts_superdict** (*counts_superdict*) – The data to plot distributions of.
- **logged** (*bool*) – Pass True to log the data before plotting.
- **drop_zeros** (*bool*) – Pass True to drop zeros from the distributions.
- **shade** (*bool*) – Pass True to fill in the area under the distribution curves.
- **labels** (*dict or None*) – Pass a dict mapping the keys of counts_superdict to labels for plotting, or pass None to use the keys of counts_superdict as the labels.
- **levels** (*dict or None*) – Pass a dict mapping labels to levels to color-code the replicates by level. Pass None to give each replicate its own level.
- **colors** (*dict or None*) – Pass a dict mapping levels to matplotlib colors to decide what color to plot each level with. Pass None to automatically choose colors.
- **hue_order** (*list of str or None*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.distribution.**plot_regional_distribtions**(*regional_counts_superdict*, *logged=True*, *drop_zeros=False*, *shade=True*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots overlayed distributions for many replicates from a regional counts superdict.

Parameters

- **regional_counts_superdict** (*counts_superdict*) – The data to plot distributions of.
- **logged** (*bool*) – Pass True to log the data before plotting.
- **drop_zeros** (*bool*) – Pass True to drop zeros from the distributions.
- **shade** (*bool*) – Pass True to fill in the area under the distribution curves.
- **labels** (*dict or None*) – Pass a dict mapping the keys of counts_superdict to labels for plotting, or pass None to use the keys of counts_superdict as the labels.
- **levels** (*dict or None*) – Pass a dict mapping labels to levels to color-code the replicates by level. Pass None to give each replicate its own level.
- **colors** (*dict or None*) – Pass a dict mapping levels to matplotlib colors to decide what color to plot each level with. Pass None to automatically choose colors.
- **hue_order** (*list of str or None*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.distribution.**plot_regional_distribtions_parallel**(*regional_counts_superdict*, *logged=True*, *drop_zeros=False*, *shade=True*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots overlayed distributions for many replicates from a regional counts superdict.

Parameters

- **regional_counts_superdict** (*counts_superdict*) – The data to plot distributions of.
- **logged** (*bool*) – Pass True to log the data before plotting.
- **drop_zeros** (*bool*) – Pass True to drop zeros from the distributions.
- **shade** (*bool*) – Pass True to fill in the area under the distribution curves.
- **labels** (*dict or None*) – Pass a dict mapping the keys of counts_superdict to labels for plotting, or pass None to use the keys of counts_superdict as the labels.

- **levels** (*dict or None*) – Pass a dict mapping labels to levels to color-code the replicates by level. Pass None to give each replicate its own level.
- **colors** (*dict or None*) – Pass a dict mapping levels to matplotlib colors to decide what color to plot each level with. Pass None to automatically choose colors.
- **hue_order** (*list of str or None*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.enrichment module

Module for plotting visualizations of enrichments of annotations within categories of categorized loops.

lib5c.plotters.enrichment.**plot_annotation_vs_annotation_heatmap**(*annotationmaps*, *looping_classes*, *loop_type*, *axis_order=None*, *threshold=0*, *margin=1*, *vmin=-2.0*, *vmax=2.0*, *despine=False*, *style='dark'*, ***kwargs*)

Plot a heatmap of enrichments for a fixed loop category, varying the annotation on either side on the x- and y-axes.

Parameters

- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

  where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the

same size and shape as the indicated region, with string loop category names in the positions of categorized loops.

- **loop_type** (*str*) – The loop category to hold constant throughout the heatmap.
- **axis_order** (*list of str, optional*) – The annotations to include on each axis, in order. If None, falls back to `sorted(annotationmap.keys())`.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **vmin** (*float*) – The lowest fold change to show on the colorbar.
- **vmax** (*float*) – The highest fold change to show on the colorbar.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.enrichment.**plot_looptype_vs_annotation_heatmap**(*annotationmaps*, *looping_classes*, *constant_annotation*, *loop_type_order=None*, *annotation_order=None*, *threshold=0*, *margin=1*, *vmin=-2.0*, *vmax=2.0*, *despine=False*, *style='dark'*, ***kwargs*)

Plot a heatmap of enrichments for one fixed annotation, varying the loop category on the x-axis and the annotation on the other side on the y-axis.

Parameters

- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **constant_annotation** (*str*) – The annotation to hold constant throughout the heatmap.
- **loop_type_order** (*list of str*) – The loop categories to include on the x-axis, in order. If None, falls back to the sorted unique categories in `looping_classes`.
- **annotation_order** (*list of str, optional*) – The annotations to include on the y-axis, in order. If None, falls back to `sorted(annotationmap.keys())`.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **vmin** (*float*) – The lowest fold change to show on the colorbar.
- **vmax** (*float*) – The highest fold change to show on the colorbar.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

`lib5c.plotters.enrichment.`**plot_stack_bargraph**(*annotation_a*, *annotation_b*, *loop_types*, *labels*, *colors*, *annotationmaps*, *looping_classes*, *threshold=0*, *margin=1*, ***kwargs*)

Plots a bar graph with loop types arranged on the x-axis and the percentage of times `annotation_a` is interaction with `annotation_b` in all the loops of that loop type.

Parameters

- **annotation_a** (*str*) – First annotation you are intereted in.
- **annotation_b** (*str*) – Second annotation you are interested in.
- **loop_types** (*list of str*) – The order in which to arrange the loop types along the x-axis, from left to right. If you exclude a loop type from this list, it will be excluded from the heatmap.
- **labels** (*list of str*) – The labels you want to be assigned on the x-axis to each of the loop types. The label order should correspond to the order of `loop_types`.
- **colors** (*list of valid matplotlib colors*) – The colors to plot each bar with. The order should correspond to the order of `loop_types`.
- **annotationmaps** (*dict of annotationmap*) – A dict describing the annotations. In total, it should have the following structure:

```
{
    'annotation_a_name': {
        'region_1_name': list of int,
```

(continues on next page)

(continued from previous page)

```
        'region_2_name': list of int,
        ...
     },
    'annotation_b_name': {
        'region_1_name': list of int,
        'region_2_name': list of int,
        ...
    },
    ...
}
```

where `annotationmaps['annotation_a']['region_r']` should be a list of ints describing the number of `'annotation_a'``s present in each bin of ``'region_r'`.

- **looping_classes** (*dict of np.ndarray with str dtype*) – The keys should be region names as strings, the values should be square, symmetric arrays of the same size and shape as the indicated region, with string loop category names in the positions of categorized loops.
- **threshold** (*int*) – Bins are defined to contain an annotation if they are "hit" strictly more than `threshold` times by the annotation.
- **margin** (*int*) – A bin is defined to contain an annotation if any bin within `margin` bins is "hit" by the annotation. Corresponds to a "margin for error" in the intersection precision.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.expected module

Module for visualization of one-dimensional distance-dependent expected models.

Two convenience functions are exposed:

- `plot_bin_expected()`
- `plot_fragment_expected()`

which are bin- and fragment-level wrappers around `plot_log_log_expected()`. The other functions are utility functions.

These functions are all overloaded so that an arg called `distance_expected` can be replaced with a dict of named distance expecteds. This will result in an overlayed comparison of all the expected models in the dict.

The other functions in this module are private helper functions.

lib5c.plotters.expected.**plot_bin_expected**(*obs_matrix*, *distance_expected*, *hexbin=False*, *kde=False*, *color='r'*, *semilog=False*, *linewidth=4*, *title='1-D expected model'*, *ylabel='log counts'*, *xlabel='log distance'*, *legend=True*, ***kwargs*)

Convenience function for plotting a visualization of a one-dimensional distance-dependent expected model defined over bin-level data.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of real interaction data that the model will be compared to.
- **distance_expected** (*List[float]*) – The one-dimensional distance-dependence model. The `i` th element of the list should correspond to the expected value for interactions between loci separated by `i` bins. To compare multiple expected models to the same observed data, pass a dict or an OrderedDict whose keys are string names for the models.
- **hexbin** (*bool*) – Pass True to use a hexbin plot to represent the density of the real data.
- **kde** (*bool*) – Pass True to use a kernel density estimate to represent the density of the real data.
- **color** (*str*) – The color to draw the expected model line with. When comparing multiple models, this can be a dict or OrderedDict with the same keys as `distance_expected`.
- **semilog** (*bool*) – Pass True to leave the distance axis unlogged.
- **linewidth** (*float*) – Line width to draw the model with.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.expected.**plot_fragment_expected**(*obs_matrix*, *distance_expected*, *distance_matrix*, *hexbin=False*, *kde=False*, *color='r'*, *semilog=False*, *linewidth=4*, *title='1-D expected model'*, *ylabel='log counts'*, *xlabel='log distance'*, *legend=True*, ***kwargs*)

Convenience function for plotting a visualization of a one-dimensional distance-dependent expected model defined over fragment-level data.

Parameters

- **obs_matrix** (*np.ndarray*) – The matrix of real interaction data that the model will be compared to.
- **distance_expected** (*Dict[int, float]*) – A mapping from interaction distances in units of base pairs to the expected value at that distance. To compare multiple expected models to the same observed data, pass a dict or an OrderedDict whose keys are string names for the models.
- **distance_matrix** (*np.ndarray*) – The pairwise distance matrix for the fragments in this region.
- **hexbin** (*bool*) – Pass True to use a hexbin plot to represent the density of the real data.
- **kde** (*bool*) – Pass True to use a kernel density estimate to represent the density of the real data.
- **color** (*str*) – The color to draw the expected model line with. When comparing multiple models, this can be a dict or OrderedDict with the same keys as `distance_expected`.
- **semilog** (*bool*) – Pass True to leave the distance axis unlogged.
- **linewidth** (*float*) – Line width to draw the model with.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.expected.**plot_log_log_expected**(*obs_distances*, *obs_values*, *exp_distances*, *exp_values*, *hexbin=False*, *kde=False*, *color='r'*, *pseudocount=1*, *semilog=False*, *linewidth=4*, *title='1-D expected model'*, *ylabel='log counts'*, *xlabel='log distance'*, *legend=True*, ***kwargs*)

Plot a visualization of an expected model over real data.

Parameters

- **obs_distances** (*np.ndarray*) – Flat array of the distances of the `obs_values`.
- **obs_values** (*np.ndarray*) – Flat array of the real data values.
- **exp_distances** (*np.ndarray*) – Flat array of the distances of the `exp_values`.
- **exp_values** (*np.ndarray*) – Flat array of the expected data values predicted by the model. To compare multiple expected models to the same observed data, pass a dict or an OrderedDict whose keys are string names for the models.
- **title** (*str*) – Title to write on the plot.
- **ylabel** (*str*) – Label for the y-axis on the plot.
- **xlabel** (*str*) – Label for the x-axis on the plot.
- **hexbin** (*bool*) – Pass True to use a hexbin plot to represent the density of the real data.
- **kde** (*bool*) – Pass True to use a kernel density estimate to represent the density of the real data.
- **color** (*str*) – The color to draw the expected model line with. When comparing multiple models, this can be a dict or OrderedDict with the same keys as `distance_expected`.
- **pseudocount** (*int*) – Pseudocount to add to distances if called with `semilog=True`.
- **semilog** (*bool*) – Pass True to leave the distance axis unlogged.
- **linewidth** (*float*) – Line width to draw the model with.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.fits module

Module for plotting visualizations comparing fitted theoretical distributions to real data.

lib5c.plotters.fits.**plot_fit**(*data*, *frozen_dist*, *legend=True*, ***kwargs*)

Base function for plotting fits.

Parameters

- **data** (*np.ndarray*) – The real data to be compared to the theoretical distribution.
- **frozen_dist** (*scipy.stats.rv_frozen*) – The theoretical distribution to be compared to the real data.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.fits.**plot_group_fit**(*obs*, *exp*, *i*, *j*, *frozen_dist*, *local=False*, *p=5*, *w=15*, *group_fractional_tolerance=0.1*, *vst=False*, *log=False*, *legend=True*, ***kwargs*)

Convenience function to select a subset of some data and compare it to a frozen distribution via plot_fit().

Parameters

- **obs** (*np.ndarray*) – Regional matrix of the observed values.
- **exp** (*np.ndarray*) – Regional matrix of the expected values.
- **j** (*i,*) – Row and column indices, respectively, of the target point.
- **frozen_dist** (*scipy.stats.rv_frozen*) – The theoretical distribution to be compared to the real data.
- **local** (*bool*) – Pass True to compare the theoretical distribution to observed data points in a donut window around the target point. Pass False to compare the theoretical distribution to observed data points with similar expected values to the target point.
- **w** (*int*) – The outer radius of the donut window to use when local=True.
- **p** (*int*) – The inner radius of the donut window to use when local=True.
- **group_fractional_tolerance** (*float*) – The fractional tolerance in expected value used to select points with "similar" expected values when local=False.
- **vst** (*bool*) – Pass True if a VST-style step has been performed upstream and the expected values should be interpreted as already logged.
- **log** (*bool*) – Pass True to log the selected observed data points before plotting.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.heatmap module

Module providing `plot_heatmap()`, a wrapper function for the extendable heatmap system defined in the `lib5c.plotters.extendable` module.

lib5c.plotters.heatmap.**plot_heatmap**(*matrix*, *grange_x*, *grange_y=None*, *rulers=True*, *ruler_fontsize=7*, *genes=None*, *gene_colors=None*, *colorscale=None*, *colorbar=False*, *colormap='abs_obs'*, *motif_tracks=None*, *motif_track_colors=None*, *motif_track_labels=None*, *motif_linewidth=0.5*, *bed_tracks=None*, *bed_track_colors=None*, *bed_track_labels=None*, *chipseq_tracks=None*, *snp_tracks=None*, *snp_colors=None*, *snp_track_labels=None*, *chipseq_track_scales=None*, *chipseq_track_colors=None*, *chipseq_track_labels=None*, *domains=None*, *domain_color='g'*, *clusters=None*, *cluster_colors=None*, *dpi=800*, *despine=False*, *style=None*, ***kwargs*)

Wrapper function for creating ExtendableHeatmaps.

Parameters

- **matrix** (*np.ndarray*) – The matrix to plot in the heatmap.
- **grange_x** (*dict or list of dict*) – The genomic range represented by the x-axis of this heatmap. The dict should have the form:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

  Pass a list of dicts of this form (assumed to be sorted) to assume that the genomic range extends from the start of the first range to the end of the last range.

- **grange_y** (*dict, optional*) – The genomic range represented by the y-axis of this heatmap. If None, the heatmap is assumed to be symmetric.
- **rulers** (*bool*) – Pass True to include genomic coordinate rulers on the heatmap.
- **ruler_fontsize** (*int*) – Controls the fontsize for the ruler when `rulers` is True.
- **genes** (*str or list of dict*) – Pass None to skip plotting gene tracks. Pass one of 'mm9', 'mm10', 'hg18', 'hg19', or 'hg38' to add gene tracks for the selected reference genome. To plot a custom set of genes, pass a list of dicts of the form:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'strand': '+' or '-',
    'blocks': list of dicts
}
```

  Blocks typically represent exons and are represented as dicts with the following structure:

```
{
    'start': int,
    'end'  : int
}
```

- **gene_colors** (*dict, optional*) – Pass a dict mapping gene names or ID's as strings to valid matplotlib colors to plot specific genes in the specified colors.
- **colorscale** (*tuple of float, optional*) – Specify the range of the heatmap colorbar as a tuple of the form (min, max).
- **colorbar** (*bool*) – Pass True to include a colorbar.
- **colormap** (*str*) – Specify the colormap to use.
- **motif_tracks** (*list of str, optional*) – Pass file references to bed files to add to the heatmap as motif tracks.
- **motif_track_colors** (*dict, optional*) – Map from strand value (e.g., '+', '-') to color name for motifs with that strand value (i.e., orientation). If not provided for a given strand, color is 'k' by default.
- **motif_track_labels** (*list of str, optional*) – Parallel to `motif_tracks`, the ith string will be used to label the ith motif track.
- **motif_linewidth** (*float*) – Pass a linewidth to use when drawing motif instances.

- **bed_tracks** (*list of str, optional*) – Pass file references to bed files to add to the heatmap as bed tracks.
- **bed_track_colors** (*dict, optional*) – Map from strand value (e.g., '+', '-') to color name for bed features with that strand value (i.e., orientation). If not provided for a given strand, color is 'k' by default.
- **bed_track_labels** (Parallel to `bed_tracks`, the ith string will be used) – to label the ith bed track.
- **snp_tracks** (*list of str, optional*) – Pass file references to bed files to add to the heatmap as SNP tracks.
- **snp_colors** (*dict, optional*) – Map from SNP id's or names to colors. If not provided for a given SNP, color is 'k' by default.
- **snp_track_labels** (*list of str, optional*) – Parallel to `snp_tracks`, the ith string will be used to label the ith SNP track.
- **chipseq_tracks** (*list of str or list of lists of dicts, optional*) – Pass file references to bed, bedgraph, or bigwig files to add to the heatmap as chipseq/feature tracks. Alternatively, pass a list of feature lists where each feature is a dict with the form:

```
{
    'chrom': str,
    'start': int,
    'end': int,
    'value': float
}
```

  where the 'value' key is optional.

- **chipseq_track_scales** (*list of tuples of float, optional*) – Parallel to `chipseq_tracks`, the ith tuple should have the form (min, max) and should specify the axis limits of the ith track. Pass None to scale chipseq tracks automatically.
- **chipseq_track_colors** (*list of str, optional*) – Parallel to `chipseq_tracks`, the ith string should indicate the color of the ith chipseq track. Pass None to color all chipseq tracks black.
- **chipseq_track_labels** (*list of str, optional*) – Parallel to `chipseq_tracks`, the ith string will be used to label the ith chipseq track.
- **domains** (*list of dict, optional*) – Each dict should represent one domain and should have the form:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

- **domain_color** (*str*) – The color to use to outline the domains.
- **clusters** (*list of lists of dicts, optional*) – Each inner list should describe one cluster to be outlined. Each cluster is a list of dicts of the form:

```
{
    'x': int,
```

(continues on next page)

(continued from previous page)

```
    'y': int
}
```

where these integers represent indices of `matrix`.

- **`cluster_colors`** (*`list of str, optional`*) – Parallel to `clusters`, the ith string should indicate the color to use to outline the ith cluster. Pass None to outline all clusters in green.
- **`kwargs`** (*`kwargs`*) – Typical plotter kwargs.

Returns The resulting ExtendableHeatmap.

Return type lib5c.plotters.extendable.ExtendableHeatmap

lib5c.plotters.matrix module

Module for simple visualization of matrix data.

`lib5c.plotters.matrix.`**`plot_matrix`**(*matrix*, *log=False*, *pseudocount=0*, *cmap=None*, *dpi=800*, ***kwargs*)

Simple plotter function to quickly visualize a matrix.

Parameters

- **`matrix`** (*`np.ndarray or scipy.sparse.spmatrix`*) – The matrix to visualize.
- **`log`** (*`bool`*) – Pass True to log the matrix for visualization.
- **`pseudocount`** (*`int`*) – Pseudocount to add before logging. Ignored if `log` is False.
- **`cmap`** (*`matplotlib colormap, optional`*) – The colormap to use for visualizing the matrix entries. Default is 'viridis' with nan's shown in red.
- **`kwargs`** (*`kwargs`*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.pca module

Module for plotting visualizations of the results of principle component analyses.

`lib5c.plotters.pca.`**`plot_multi_pca`**(*proj*, *pcs=3*, *s=100*, *label_points=True*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Create a multi-component grid of PCA plots.

Parameters

- **`proj`** (*`np.ndarray`*) – The matrix of PCA-projected replicates.
- **`pcs`** (*`int`*) – How many principle components should be plotted.
- **`s`** (*`float`*) – The area of the points to plot on the scatterplot.
- **`label_points`** (*`bool`*) – Pass True to annotate each point with its label.
- **`labels`** (*`Optional[List[str]]`*) – String names identifying the replicates (the rows of `proj`). Pass None to simply label them with their row index within `proj`.

- **levels** (*Optional[Union[List[str], Dict[str, str]]]*) – The "level" for each replicate. Can be passed as a list of string (matching the order of the rows of `proj`), or a dict mapping the labels to levels. Each "level" gets one color and one entry in the legend. If None is passed each replicate gets its own level (`levels = labels`).
- **colors** (*Optional[Dict[str, str]]*) – Mapping from levels as strings to the color to use for that level. Pass None to use randomly assigned colors.
- **hue_order** (*Optional[List[str]]*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

Notes

If both `log` and `scaled` are True, the logarithm will be applied before scaling.

PCA will always mean-center the data.

lib5c.plotters.pca.**plot_pca**(*proj*, *pcs=(0, 1)*, *legend=True*, *s=100*, *label_points=True*, *labels=None*, *levels=None*, *colors=None*, *hue_order=None*, ***kwargs*)

Plots a PCA projection along two selected principal components.

Parameters

- **proj** (*np.ndarray*) – The matrix of PCA-projected replicates.
- **pcs** (*Tuple[int]*) – Which two (zero-indexed) principle components should be plotted.
- **legend** (*bool*) – Pass True to include a legend.
- **s** (*float*) – The area of the points to plot on the scatterplot.
- **label_points** (*bool*) – Pass True to annotate each point with its label.
- **labels** (*Optional[List[str]]*) – String names identifying the replicates (the rows of `proj`). Pass None to simply label them with their row index within `proj`.
- **levels** (*Optional[Union[List[str], Dict[str, str]]]*) – The "level" for each replicate. Can be passed as a list of string (matching the order of the rows of `proj`), or a dict mapping the labels to levels. Each "level" gets one color and one entry in the legend. If None is passed each replicate gets its own level (`levels = labels`).
- **colors** (*Optional[Dict[str, str]]*) – Mapping from levels as strings to the color to use for that level. Pass None to use randomly assigned colors.
- **hue_order** (*Optional[List[str]]*) – Pass a list of the level names to determine their order in the legend.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.queried_counts_heatmap module

Module for plotting contact frequency heatmaps for fragment-level 5C data.

lib5c.plotters.queried_counts_heatmap.**plot_queried_counts_heatmap**(*array*, *outfile*, *colorscale=(0, 0.98)*, *colorbar=False*, *cmap=None*)

Plot a fragment-level contact frequency heatmap of only the interactions actually queried by the 5C assay.

Parameters

- **array** (*np.ndarray*) – The non-symmetric, non-square matrix of queried counts for this region.
- **outfile** (*str*) – String reference to the file to write the heatmap to.
- **colorscale** (*Tuple[float]*) – The min and the max of the colormap, as a tuple of the form `(min, max)`.
- **colorbar** (*bool*) – Pass True to include a colorbar on the plot.
- **cmap** (*Optional[matplotlib.colors.Colormap]*) – The colormap to use for the heatmap. If this kwarg is not passed, `pyplot.cm.coolwarm` will be used as a default.

lib5c.plotters.scatter module

Module for plotting scatterplots of x-y graphs.

lib5c.plotters.scatter.**scatter**(*x*, *y*, *logx=True*, *logy=True*, *hexbin=True*, *xlim=None*, *ylim=None*, ***kwargs*)

Plots a scatterplot of data, either as a scatterplot or a hexbin plot.

Parameters

- **y** (*x,*) – The data points.
- **logy** (*logx,*) – Log the x- and/or y-axis.
- **hexbin** (*bool*) – Pass True to plot a hexbin plot instead of a scatterplot.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

Notes

If both xlim and ylim are passed as kwargs and hexbin=True, this function will attempt to set the extent of the hexbin plot using the xlim and ylim.

lib5c.plotters.splines module

Module for visualizing spline surfaces fit to 5C counts data.

lib5c.plotters.splines.**visualize_spline**(*counts_list*, *primermap*, *bias_factor*, *spline*, *grid_points=10*, *sample_rate=100*, *log=True*, *asymmetric=False*)

Open an interactive pyplot window showing a visualization of a spline surface, overlayed over representative 5C counts data.

Parameters

- **counts_list** (*List[Dict[str, np.ndarray]]*) – A list of counts dicts to use as data points to be compared to the spline surface.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap corresponding to the counts dicts in `counts_list`.
- **bias_factor** (*str*) – The bias factor being plotted. This string must match metadata keys in `primermap`. That is to say, if `bias_list` is `['length']` then we expect `primermap[region][i]['length']` to be a number representing the length of the `i` th fragment in the region specified by `region`.
- **spline** (*scipy.interpolate.BivariateSpline*) – The spline object to visualize.
- **grid_points** (*int*) – The number of grid points to use when constructing the wireframe of the surface represented by `spline`.
- **sample_rate** (*int*) – Only every `sample_rate` th real-data point will be included in the visualization to reduce computational load.
- **log** (*bool*) – Pass True to show counts on a log-scale axis.
- **asymmetric** (*bool*) – Pass True to iterate only over the upper-triangular elements of the counts matrices, which can lead to asymmetric visualizations. By default, the algorithm iterates over all elements of the counts matrices, enforcing symmetry in thevisualizations but incurring some redundancy in the actual counts information.

Notes

The spline will be displayed in an interactive window via `plt.show()`. If your default matplotlib backend is not interactive, this function will try to set your backend to TkAgg. If you prefer to use a different interactive backend, set the `MPLBACKEND` environment variable before invoking Python.

lib5c.plotters.variance module

Module for plotting mean-variance relationships.

lib5c.plotters.variance.**plot_mvr**(*exp*, *var*, *obs=None*, *num_groups=100*, *group_fractional_tolerance=0.1*, *exclude_offdiagonals=5*, *log=False*, *logx=False*, *logy=False*, *vst=False*, *scatter=False*, *hexbin=False*, *trim_limits=False*, *xlim=None*, *ylim=None*, ***kwargs*)

Plots a scatterplot of exp vs var.

Optionally, pass obs to instead scatterplot exp vs freshly re-estimated group variances, and overlay exp vs var as a smooth curve.

Parameters

- **exp** (*np.ndarray or dict of np.ndarray*) – Regional matrix of expected values. Pass a counts dict to combine all regions together.

- **var** (*np.ndarray or dict of np.ndarray*) – Regional matrix of variances. Pass a counts dict to combine all regions together.
- **obs** (*np.ndarray or dict of np.ndarray, optional*) – Regional matrix of observed values. Pass a counts dict to combine all regions together.
- **num_groups** (*int*) – The number of groups to re-esimtate group variances for.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **exclude_offdiagonals** (*int*) – Exclude this many off-diagonals from the variance re-estimation. Pass 0 to exclude only the exact diagonal. Pass -1 to exclude nothing.
- **log** (*bool*) – Pass True to log both exp and var.
- **logy** (*logx,*) – Pass True to draw the x- and/or y-axis on a log-scale.
- **vst** (*bool*) – Pass True to log only the exp (e.g., when var is already stabilized).
- **scatter** (*bool*) – Pass True to force plotting exp vs var as a scatterplot when obs is not passed. By default it will be a line plot.
- **hexbin** (*bool*) – Pass True when `scatter=True` to replace the scatterplot with a hexbin plot.
- **trim_limits** (*bool*) – If *obs* is passed, pass True to trim the x- and y-limits to the range of the group expected and variance values.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.variance.**plot_mvr_parallel**(*exp*, *var*, *obs=None*, *num_groups=100*, *group_fractional_tolerance=0.1*, *exclude_offdiagonals=5*, *log=False*, *logx=False*, *logy=False*, *vst=False*, *scatter=False*, *hexbin=False*, *trim_limits=False*, *xlim=None*, *ylim=None*, ***kwargs*)

Plots a scatterplot of exp vs var.

Optionally, pass obs to instead scatterplot exp vs freshly re-estimated group variances, and overlay exp vs var as a smooth curve.

Parameters

- **exp** (*np.ndarray or dict of np.ndarray*) – Regional matrix of expected values. Pass a counts dict to combine all regions together.
- **var** (*np.ndarray or dict of np.ndarray*) – Regional matrix of variances. Pass a counts dict to combine all regions together.
- **obs** (*np.ndarray or dict of np.ndarray, optional*) – Regional matrix of observed values. Pass a counts dict to combine all regions together.
- **num_groups** (*int*) – The number of groups to re-esimtate group variances for.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **exclude_offdiagonals** (*int*) – Exclude this many off-diagonals from the variance re-estimation. Pass 0 to exclude only the exact diagonal. Pass -1 to exclude nothing.
- **log** (*bool*) – Pass True to log both exp and var.

- **logy** (*logx,*) – Pass True to draw the x- and/or y-axis on a log-scale.
- **vst** (*bool*) – Pass True to log only the exp (e.g., when var is already stabilized).
- **scatter** (*bool*) – Pass True to force plotting exp vs var as a scatterplot when obs is not passed. By default it will be a line plot.
- **hexbin** (*bool*) – Pass True when `scatter=True` to replace the scatterplot with a hexbin plot.
- **trim_limits** (*bool*) – If *obs* is passed, pass True to trim the x- and y-limits to the range of the group expected and variance values.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.variance.**plot_overlay_mvr**(*exp*, *var*, *obs=None*, *num_groups=100*, *group_fractional_tolerance=0.1*, *exclude_offdiagonals=5*, *log=False*, *logx=False*, *logy=False*, *vst=False*, *scatter=False*, *scatter_colors=None*, *line_colors=None*, *legend='outside'*, ***kwargs*)

Plots a comparison of mean-variance relationships across regions.

Parameters

- **exp** (*dict of np.ndarray*) – Counts dict of expected values.
- **var** (*dict of np.ndarray*) – Counts dict of variance values.
- **obs** (*dict of np.ndarray, optional*) – Counts dict of observed values.
- **num_groups** (*int*) – The number of groups to re-esimtate group variances for.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **exclude_offdiagonals** (*int*) – Exclude this many off-diagonals from the variance re-estimation. Pass 0 to exclude only the exact diagonal. Pass -1 to exclude nothing.
- **log** (*bool*) – Pass True to log both exp and var.
- **logy** (*logx,*) – Pass True to draw the x- and/or y-axis on a log-scale.
- **vst** (*bool*) – Pass True to log only the exp (e.g., when var is already stabilized).
- **scatter** (*bool*) – Pass True to force plotting exp vs var as a scatterplot when obs is not passed. By default it will be a line plot.
- **line_colors** (*scatter_colors,*) – Mapping from region names to the color to use for that region. Pass None to use randomly assigned colors. Pass a single string to use the same color for all regions.
- **kwargs** (*kwargs*) – Typical plotter kwargs.

Returns The axis plotted on.

Return type pyplot axis

lib5c.plotters.variance.**prepare_exp_var_for_plotting**(*exp*, *var*, *obs=None*, *num_groups=100*, *group_fractional_tolerance=0.1*, *exclude_offdiagonals=5*, *log=False*, *vst=False*)

Prepares expected value and variance data for plotting.

Parameters

- **exp** (*np.ndarray or dict of np.ndarray*) – Regional matrix of expected values. Pass a counts dict to combine all regions together.
- **var** (*np.ndarray or dict of np.ndarray*) – Regional matrix of variances. Pass a counts dict to combine all regions together.
- **obs** (*np.ndarray or dict of np.ndarray, optional*) – Regional matrix of observed values. Pass a counts dict to combine all regions together.
- **num_groups** (*int*) – The number of groups to re-esimtate group variances for.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **exclude_offdiagonals** (*int*) – Exclude this many off-diagonals from the variance re-estimation. Pass 0 to exclude only the exact diagonal. Pass -1 to exclude nothing.
- **log** (*bool*) – Pass True to log both exp and var.
- **vst** (*bool*) – Pass True to log only the exp (e.g., when var is already stabilized).

Returns The first and second elements are parallel arrays of the exp, var pairs. The third element is a sort index into the exp, var pairs. The fourth and fifth elements are None if obs was not passed. If obs was passed, they are parallel arrays of raw obs, raw var pairs.

Return type tuple of np.ndarray

lib5c.plotters.variance.**prepare_exp_var_for_plotting_parallel**(*exp*, *var*, *obs=None*, *num_groups=100*, *group_fractional_tolerance=0.1*, *exclude_offdiagonals=5*, *log=False*, *vst=False*)

Prepares expected value and variance data for plotting.

Parameters

- **exp** (*np.ndarray or dict of np.ndarray*) – Regional matrix of expected values. Pass a counts dict to combine all regions together.
- **var** (*np.ndarray or dict of np.ndarray*) – Regional matrix of variances. Pass a counts dict to combine all regions together.
- **obs** (*np.ndarray or dict of np.ndarray, optional*) – Regional matrix of observed values. Pass a counts dict to combine all regions together.
- **num_groups** (*int*) – The number of groups to re-esimtate group variances for.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **exclude_offdiagonals** (*int*) – Exclude this many off-diagonals from the variance re-estimation. Pass 0 to exclude only the exact diagonal. Pass -1 to exclude nothing.

- **log** (*bool*) – Pass True to log both exp and var.
- **vst** (*bool*) – Pass True to log only the exp (e.g., when var is already stabilized).

Returns The first and second elements are parallel arrays of the exp, var pairs. The third element is a sort index into the exp, var pairs. The fourth and fifth elements are None if obs was not passed. If obs was passed, they are parallel arrays of raw obs, raw var pairs.

Return type tuple of np.ndarray

Module contents

Subpackage for visualizing and plotting data related to 5C analysis.

lib5c.structures package

Submodules

lib5c.structures.dataset module

Module for the Dataset class, which provides a wrapper around a pandas DataFrame allowing for representation of 5C data across replicates and stages of data processing both on disk and in memory.

class lib5c.structures.dataset.**Dataset**(*df*, *pixelmap=None*, *repinfo=None*)

Bases: object

Wrapper around a Pandas DataFrame.

df

Contains the core data in the Dataset. Columns should be either not hierarchical, or hierarchical with the lower level of the hierarchy matching the replicate names. The row index of this DataFrame must be '<upstream_fragment_name>_<downstream_fragment_name>'.

Type pd.DataFrame

pixelmap

A pixelmap to provide information about the fragments.

Type pixelmap, optional

repinfo

Its row index should be the replicate names, its columns can provide arbitrary information about each replicate, such as its condition, etc.

Type pd.DataFrame, optional

add_column_from_counts(*counts*, *name*)

Adds a new column to this Dataset's df.

The counts dict passed is assumed to match the pixelmap bound on this Dataset. If no pixelmap is bound, an ValueError will be raised.

Parameters

- **counts** (*dict of np.ndarray*) – Should contain the values that will make up the new column.
- **name** (*str*) – The name of the new column.

add_columns_from_counts_superdict(*counts_superdict*, *name*, *rep_order=None*)
Adds a new group of columns to the Dataset from a counts superdict structure.

Parameters

- **counts_superdict** (*dict of dict of np.ndarray*) – The outer keys are replicate names as strings, the inner keys are region names as strings, and the values are square, symmetric arrays of values for each replicate and region.
- **name** (*str*) – The name to use for the new group of columns.
- **rep_order** (*list of str, optional*) – Pass a list of replicate names to load the listed replicates in a specific order. Pass None to use the random order of the outer keys of counts_superdict.

apply_across_replicates(*fn*, *inputs*, *outputs*, ***kwargs*)
Applies a matrix-to-matrix function over the Dataset.

This is useful for functions that don't operate independently on each replicate of the Dataset.

The main advantage of this function is that it handles the unboxing of the replicates after a matrix-to-matrix function is applied. If you are looking to apply a matrix-to-vector function over the Dataset, you can do it with a one-liner, assigning the vector result(s) to the new column(s) immediately.

Parameters

- **fn** (*Callable*) – The function to apply. It should take in np.ndarrays as its inputs and return np.ndarrays with the same size and shape. If some inputs are specified as individual columns, they will be passed to fn as np.ndarrays shaped as column vectors.
- **inputs** (*list of (str or tuple of str)*) – The list of columns to pass as inputs to fn. Use a tuple of strings to access hierarchical columns. At least one input must refer to the top level of a hierarchical column, the first such column encountered will be used to determine the replicates to apply over. Non-hierarchical columns, or hierarchical columns fully specified by a tuple of strings will be passed to fn as column vectors.
- **outputs** (*list of str*) – Names of top-level output columns to be added to the Dataset. The second level will be automatically filled in with the replicate names.

apply_across_replicates_per_region(*fn*, *inputs*, *outputs*, *initial_values=0.0*, ***kwargs*)
Applies a matrix-to-matrix function over the Dataset in a per-region manner.

This is useful for functions that don't operate independently on each replicate of the Dataset, but which operate independently on each region of the Dataset.

The main advantage of this function is that it handles the unboxing of the replicates after a matrix-to-matrix function is applied. If you are looking to apply a matrix-to-vector function over the Dataset in a per-region manner, you can do it with apply_per_region(), feeding a hierarchical column as an input.

Parameters

- **fn** (*Callable*) – The function to apply. It should take in np.ndarrays as its inputs and return np.ndarrays with the same size and shape. If some inputs are specified as individual columns, they will be passed to fn as np.ndarrays shaped as column vectors.
- **inputs** (*list of (str or tuple of str)*) – The list of columns to pass as inputs to fn. Use a tuple of strings to access hierarchical columns. At least one input must refer to the top level of a hierarchical column, the first such column encountered will be used to determine the replicates to apply over. Non-hierarchical columns, or hierarchical columns fully specified by a tuple of strings will be passed to fn as column vectors.
- **outputs** (*list of str*) – Names of top-level output columns to be added to the Dataset. The second level will be automatically filled in with the replicate names.

- **initial_values** (*list of any*) – The values with which the new columns will be temporarily initialized. This should control the dtype of the new columns.

apply_per_region (*fn*, *inputs*, *outputs*, *initial_values=0.0*, ***kwargs*)
Apply a function over the Dataset on a per-region basis.

Parameters

- **fn** (*Callable*) – The function to apply. It should take in pd.Series's or pd.DataFrames as its args, in the same order as inputs, and it should return 1D vectors, in the same order as outputs.
- **inputs** (*list of (str or tuple of str)*) – The list of columns to pass as inputs to fn. Use a tuple of strings to access hierarchical columns. Omit the secound level of a hierarchical column to pass all replicates to fn as a single pd.DataFrame. A single string or tuple will be wrapped in a list automatically.
- **outputs** (*list of (str or tuple of str)*) – Names of output columns to be added to the Dataset. Use a tuple of strings to create hierarchical columns.
- **initial_values** (*list of any*) – The values with which the new columns will be temporarily initialized. This should control the dtype of the new columns.

apply_per_replicate (*fn*, *inputs*, *outputs*, ***kwargs*)
Applies a function over the Dataset on a per-replicate basis.

Parameters

- **fn** (*Callable*) – The function to apply. It should take in pd.Series's as its args, in the same order as inputs, and it should return 1D vectors, in the same order as outputs.
- **inputs** (*list of (str or tuple of str)*) – The list of columns to pass as inputs to fn. Use a tuple of strings to access hierarchical columns. At least one input must refer to the top level of a hierarchical column, the first such column encountered will be used to determine the replicates to apply over. Non-hierarchical columns, or hierarchical columns fully specified by a tuple of strings will be broadcast across all replicates.
- **outputs** (*list of str*) – Names of top-level output columns to be added to the Dataset. The second level will be automatically filled in with the replicate names.

apply_per_replicate_per_region (*fn*, *inputs*, *outputs*, *initial_values=0.0*, ***kwargs*)
Applies a function over the Dataset on a per-replicate, per-region basis.

Parameters

- **fn** (*Callable*) – The function to apply. It should take in pd.Series's as its args, in the same order as inputs, and it should return 1D vectors, in the same order as outputs.
- **inputs** (*list of (str or tuple of str)*) – The list of columns to pass as inputs to fn. Use a tuple of strings to access hierarchical columns. At least one input must refer to the top level of a hierarchical column, the first such column encountered will be used to determine the replicates to apply over. Non-hierarchical columns, or hierarchical columns fully specified by a tuple of strings will be broadcast across all replicates.
- **outputs** (*list of str*) – Names of top-level output columns to be added to the Dataset. The second level will be automatically filled in with the replicate names.
- **initial_values** (*list of any*) – The values with which the new columns will be temporarily initialized. This should control the dtype of the new columns.

counts (*name='counts'*, *rep=None*, *region=None*, *fill_value=None*, *dtype=None*)
Converts this Dataset to a regional_counts matrix, a counts dict, a counts_superdict, or a regional_counts_superdict.

Parameters

- **name** (*str*) – The top-level column name to extract.
- **rep** (*str, optional*) – If name corresponds to a hierarchical column, pass a rep name to extract only one rep (return type will be a counts dict). Pass None to return a counts_superdict with all reps. If name corresponds to a normal column, this kwarg will be ignored.
- **region** (*str, optional*) – Pass a region name as a string to extract data for only one region. If name corresponds to a hierarchical column and rep was not passed, the return type will be a regional_counts_superdict. Otherwise, the return type will be a regional_counts matrix. Pass None to extract data for all regions.
- **fill_value** (*any, optional*) – The fill value for the counts_superdict (for entries not present in the Dataset). Pass None to use np.nan.
- **dtype** (*dtype, optional*) – The dtype to use for the np.array's in the counts_superdict. Pass None to guess them from the Dataset. If the data being extracted is strings, 'U25' will be assumed.

Returns

- *regional_counts matrix, counts dict, counts_superdict, or*
- *regional_counts_superdict* – The data requested. See Parameters for explanation of return type. The general philosophy is that a counts_superdict will be returned, but any single-key levels will be squeezed.

dropna(*name='counts'*, *reps=None*)

Drops NA's from the underlying dataframe.

Parameters

- **name** (*str*) – The name of the column to decide to drop based on.
- **reps** (*list of str, optional*) – If name refers to a hierarchial column, pass a list of rep names to only drop based on these reps. Pass None to drop based on the presence of an NA in any rep. If name does not refer to a hierarchical column this kwarg is ignored.

classmethod from_counts_superdict(*counts_superdict*, *pixelmap*, *name='counts'*, *repinfo=None*, *rep_order=None*)

Creates a Datset from a counts_superdict and associated pixelmap.

Parameters

- **counts_superdict** (*counts_superdict*) – Contains the data that will be put into the Dataset.
- **pixelmap** (*pixelmap*) – Needed to establish the row index on the Dataset.
- **name** (*str*) – Top-level column name for the data.
- **repinfo** (*repinfo-style pd.Dataframe or list of str, optional*) – Repinfo to bind to the Dataset. Pass a list of condition names to automatically create a repinfo object.
- **rep_order** (*list of str, optional*) – Pass this to guarantee the order of the columns for the replicates. Pass None to accept a random order.

Returns The new Dataset.

Return type *Dataset*

classmethod from_table_file(*table_file*, *name='counts'*, *sep=None*, *pixelmap=None*, *repinfo=None*)

Creates a Dataset from a table file.

The first column of the table file should be a FFLJ ID.

The remaining columns should be count values for each replicate.

The first row should specify the replicate names for each column.

Parameters

- **table_file** (*str*) – The table file to read counts from.
- **name** (*str*) – Top-level column name for the data.
- **sep** (*str*) – The separator to use when parsing the table file.' ' for tsv tables, ',' for csv tables. Pass None to guess this from the filename.
- **pixelmap** (*pixelmap, optional*) – A pixelmap to bind to the Dataset.
- **repinfo** (*repinfo-style pd.Dataframe, optional*) – Repinfo to bind to the Dataset.

Returns The new Dataset.

Return type *Dataset*

classmethod load(*filename*, *sep=None*)

Loads a Dataset from disk.

Parameters

- **filename** (*str*) – The .csv or .tsv file to load the Dataset from. If a pixelmap or repinfo file is found next to this file, these files will also be loaded into the Dataset.
- **sep** (*str, optional*) – The separator to use when parsing the .csv/.tsv. Pass None to deduce this automatically from the file extension.

Returns The loaded Dataset.

Return type *Dataset*

save(*filename*, *sep=None*)

Writes this Dataset to disk as a .csv/.tsv, and optionally writes the pixelmap and/or repinfo files to disk right next to it if either or both of these data structures exist in the Dataset.

Parameters

- **filename** (*str*) – The filename to write to.
- **sep** (*str, optional*) – The separator to use when writing the file. If `filename` ends with .csv or .tsv and `sep` is None, the separator will be determined automatically by the extension, but you can pass a value here to override it.

select(*name='counts'*, *rep=None*, *region=None*)

Get a subset of this Dataset's DataFrame corresponding to a desired column, replicate, and/or region.

Parameters

- **name** (*str*) – The column name of a hierarchical or non-hierarchical column.
- **rep** (*str, optional*) – If `name` refers to a hierarchical column, you must specify which replicate you want to select data from by passing its name here.
- **region** (*str, optional*) – To select data from only one region, pass its name here. Pass None to select data from all regions.

Module contents

lib5c.tools package

Submodules

lib5c.tools.bias_heatmap module

lib5c.tools.bias_heatmap.**add_bias_heatmap_tool**(*parser*)

lib5c.tools.bias_heatmap.**bias_heatmap_tool**(*parser*, *args*)

lib5c.tools.bin module

lib5c.tools.bin.**add_bin_tool**(*parser*)

lib5c.tools.bin.**bin_tool**(*parser*, *args*)

lib5c.tools.boxplot module

lib5c.tools.boxplot.**add_boxplot_tool**(*parser*)

lib5c.tools.boxplot.**boxplot_tool**(*parser*, *args*)

lib5c.tools.colorscale module

lib5c.tools.colorscale.**add_colorscale_tool**(*parser*)

lib5c.tools.colorscale.**colorscale_tool**(*parser*, *args*)

lib5c.tools.convergency module

lib5c.tools.convergency.**add_convergency_tool**(*parser*)

lib5c.tools.convergency.**convergency_tool**(*parser*, *args*)

lib5c.tools.correlation module

lib5c.tools.correlation.**add_correlation_tool**(*parser*)

lib5c.tools.correlation.**correlation_tool**(*parser*, *args*)

lib5c.tools.dd_curve module

lib5c.tools.dd_curve.**add_dd_curve_tool**(*parser*)

lib5c.tools.dd_curve.**dd_curve_tool**(*parser*, *args*)

lib5c.tools.determine_bins module

lib5c.tools.determine_bins.**add_determine_bins_tool**(*parser*)

lib5c.tools.determine_bins.**determine_bins_tool**(*parser*, *args*)

lib5c.tools.distribution module

lib5c.tools.distribution.**add_distribution_tool**(*parser*)

lib5c.tools.distribution.**distribtion_tool**(*parser*, *args*)

lib5c.tools.divide module

lib5c.tools.divide.**add_divide_tool**(*parser*)

lib5c.tools.divide.**divide_tool**(*parser*, *args*)

lib5c.tools.enrichment module

lib5c.tools.enrichment.**add_enrichment_tool**(*parser*)

lib5c.tools.enrichment.**enrichment_tool**(*parser*, *args*)

lib5c.tools.expected module

lib5c.tools.expected.**add_expected_tool**(*parser*)

lib5c.tools.expected.**expected_tool**(*parser*, *args*)

lib5c.tools.express module

lib5c.tools.express.**add_express_tool**(*parser*)

lib5c.tools.express.**express_tool**(*parser*, *args*)

lib5c.tools.heatmap module

lib5c.tools.heatmap.**add_heatmap_tool**(*parser*)

lib5c.tools.heatmap.**heatmap_tool**(*parser*, *args*)

lib5c.tools.helpers module

lib5c.tools.helpers.**infer_level_mapping**(*rep_names*, *triggers*)

Infers a mapping from replicate names to level names (i.e., classes or conditions) using a simple trigger substring approach.

A replicate is assigned to a level if the level's trigger substring is a substring of the replicate name.

Parameters

- **rep_names** (*list of str*) – The replicate names to assign levels to.
- **triggers** (*dict or list of str*) – Pass a dict mapping trigger substrings to level names, or pass a list of level names to use the level names as their own trigger substrings.

Returns A mapping from rep_names to level names.

Return type dict

lib5c.tools.helpers.**infer_replicate_names**(*infiles*, *as_dict=False*, *pattern=None*)

Infers replicate names given a list of filenames.

Parameters

- **infiles** (*list of str*) – The filenames to consider.
- **as_dict** (*bool*) – Pass True to make this function return a dict mapping the the infiles to their inferred replicate names.
- **pattern** (*str, optional*) – If the infiles are glob-based matches to a patten containing one wildcard, pass that pattern to use it to reconstruct the replicate names.

Returns If as_dict is False (the default), this is just the list of inferred rep names, in the same order as infiles. If as_dict is True, this is a dict mapping the original infiles to their inferred replicate names.

Return type list of str or dict

lib5c.tools.helpers.**resolve_expected_models**(*expected_model_string*, *observed_counts*, *primermap*, *level=None*)

Convenience helper for resolving expected models.

Parameters

- **expected_model_string** (*str*) – If None, we expect to estimate fresh expected models from `observed_counts`. If a path to a specific countsfile, we expect that it contains the expected model to be used for all the observed counts. If a glob-expandable path, we expect that each file matching the pattern is to be used for one of the observed counts (assuming they too are in glob order).
- **observed_counts** (*list of dict of np.ndarray*) – Each element in the list is one replicate, represented as a counts dict of observed values.
- **primermap** (*primermap*) – The primermap to use for parsing files, etc.
- **level** (*{'bin', 'fragment'}*) – The level to use if a fresh expected modeul must be estimated.

Returns The resolved expected models.

Return type list of dict of np.ndarray

lib5c.tools.helpers.**resolve_level**(*primermap*, *level='auto'*)

Resolves the level of some input data.

Parameters

- **primermap** (*primermap*) – Primermap to try to resolve the level of.
- **level** (*str*) – If you already know the level, you can pass it as a string here.

Returns The resolved level.

Return type str

Notes

This function operates on a "three in a row" heuristic: if the first three bins in the primemap are all of the same size, then we guess that it's bin level data.

lib5c.tools.helpers.**resolve_parallel**(*parser*, *args*, *subcommand=''*, *key_arg='infile'*, *root_command='lib5c'*)

Parallelizes as a command via bsub if it is available.

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser used to parse the args for the root command.
- **args** (*argparse.Namespace*) – The args parsed by the parser.
- **subcommand** (*str*) – The particular subcommand of the root command being invoked.
- **key_arg** (*str*) – The argument to parallelize over.
- **root_command** (*str*) – The string used to invoke the root command.

lib5c.tools.helpers.**resolve_primerfile**(*infile*, *primerfile=None*)

Searches for a primerfile next to in infile.

Parameters

- **infile** (*str or list of str*) – The infile(s) to look next to.
- **primerfile** (*str, optional*) – If you already know where the primerfile is pass it here to skip the search.

Returns The primerfile.

Return type str

lib5c.tools.helpers.**split_self_regionally**(*regions*, *script='lib5c'*, *hang=False*)

Allows a command line script that accepts a –region flag to split itself into a separate command run for each region.

Parameters

- **regions** (*list of str*) – The regions to split into.
- **script** (*str*) – The name of the script to invoke.
- **hang** (*bool*) – Pass True to cause the original executing process to hang until all the bsub jobs spawned by this function complete. This does nothing if bsub is not available.

lib5c.tools.hic_extract module

lib5c.tools.hic_extract.**add_hic_extract_tool**(*parser*)

lib5c.tools.hic_extract.**hic_extract_tool**(*parser*, *args*)

lib5c.tools.hic_extract.**parse_range_string**(*range_string*)

lib5c.tools.iced module

lib5c.tools.iced.**add_iced_tool**(*parser*)

lib5c.tools.iced.**iced_tool**(*parser*, *args*)

lib5c.tools.interaction_score module

lib5c.tools.interaction_score.**add_interaction_score_tool**(*parser*)

lib5c.tools.interaction_score.**interaction_score_tool**(*parser*, *args*)

lib5c.tools.kr module

lib5c.tools.kr.**add_kr_tool**(*parser*)

lib5c.tools.kr.**kr_tool**(*parser*, *args*)

lib5c.tools.lib5c_toolbox module

class lib5c.tools.lib5c_toolbox.**CustomHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: argparse.HelpFormatter

format_help()

lib5c.tools.lib5c_toolbox.**lib5c_toolbox**(*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

lib5c.tools.log module

lib5c.tools.log.**add_log_tool**(*parser*)

lib5c.tools.log.**log_tool**(*parser*, *args*)

lib5c.tools.outliers module

lib5c.tools.outliers.**add_outliers_tool**(*parser*)

lib5c.tools.outliers.**outliers_tool**(*parser*, *args*)

lib5c.tools.parents module

lib5c.tools.pca module

lib5c.tools.pca.**add_pca_tool**(*parser*)

lib5c.tools.pca.**pca_tool**(*parser*, *args*)

lib5c.tools.pipeline module

lib5c.tools.pipeline.**add_pipeline_tool**(*parser*)

lib5c.tools.pipeline.**pipeline_tool**(*parser*, *args*)

lib5c.tools.pvalue_histogram module

lib5c.tools.pvalue_histogram.**add_pvalue_histogram_tool**(*parser*)

lib5c.tools.pvalue_histogram.**pvalue_histogram_tool**(*parser*, *args*)

lib5c.tools.pvalues module

lib5c.tools.pvalues.**add_pvalues_tool**(*parser*)

lib5c.tools.pvalues.**pvalues_tool**(*parser*, *args*)

lib5c.tools.qnorm module

lib5c.tools.qnorm.**add_qnorm_tool**(*parser*)

lib5c.tools.qnorm.**qnorm_tool**(*parser*, *args*)

lib5c.tools.qvalues module

lib5c.tools.qvalues.**add_qvalues_tool**(*parser*)

lib5c.tools.qvalues.**qvalues_tool**(*parser*, *args*)

lib5c.tools.remove module

lib5c.tools.remove.**add_remove_tool**(*parser*)

lib5c.tools.remove.**remove_tool**(*parser*, *args*)

lib5c.tools.smooth module

lib5c.tools.smooth.**add_smooth_tool**(*parser*)

lib5c.tools.smooth.**smooth_tool**(*parser*, *args*)

lib5c.tools.spline module

lib5c.tools.spline.**add_spline_tool**(*parser*)

lib5c.tools.spline.**spline_tool**(*parser*, *args*)

lib5c.tools.subtract module

lib5c.tools.subtract.**add_subtract_tool**(*parser*)

lib5c.tools.subtract.**subtract_tool**(*parser*, *args*)

lib5c.tools.threshold module

lib5c.tools.threshold.**add_threshold_tool**(*parser*)

lib5c.tools.threshold.**threshold_tool**(*parser*, *args*)

lib5c.tools.trim module

lib5c.tools.trim.**add_trim_tool**(*parser*)

lib5c.tools.trim.**trim_tool**(*parser*, *args*)

lib5c.tools.variance module

lib5c.tools.variance.**add_variance_tool**(*parser*)

lib5c.tools.variance.**variance_tool**(*parser*, *args*)

lib5c.tools.visualize_fits module

lib5c.tools.visualize_fits.**add_visualize_fits_tool**(*parser*)

lib5c.tools.visualize_fits.**visualize_fits_tool**(*parser*, *args*)

lib5c.tools.visualize_splines module

lib5c.tools.visualize_splines.**add_visualize_splines_tool**(*parser*)

lib5c.tools.visualize_splines.**visualize_splines_tool**(*parser*, *args*)

lib5c.tools.visualize_variance module

lib5c.tools.visualize_variance.**add_visualize_variance_tool**(*parser*)

lib5c.tools.visualize_variance.**visualize_variance_tool**(*parser*, *args*)

Module contents

lib5c.util package

Submodules

lib5c.util.annotationmap module

Module containing utilities for constructing annotatiomaps.

Annotatiomaps record the number of BED features of a certain type present at a given linear bin as specified by a pixelmap.

lib5c.util.annotationmap.**main**()

lib5c.util.annotationmap.**make_annotationmaps**(*pixelmap*, *directory='./annotations'*, *add_wildcard=True*)

Gets a dict of annotationmaps, one for every BED file in a specified directory.

Parameters

- **pixelmap** (*pixelmap*) – The pixelmap to use to generate the annotationmap. See lib5c.parsers.bed.get_pixelmap().
- **directory** (*str*) – The directory to look in for BED files describing the annotations.
- **add_wildcard** (*bool*) – Pass True to add a 'wildcard' annotation that has 100 hits in every bin. Useful for doing "unsided" enrichments later.

Returns The keys of the outer dict are annotation names as parsed from the names of the BED files in directory. The values are annotationmaps. See lib5c.util.annotationmap.get_single_annotatiomap().

Return type dict of dict of lists

lib5c.util.annotationmap.**make_single_annotationmap**(*annotation*, *pixelmap*)

Generates an annotationmap given an annotation and a pixelmap.

Parameters

- **annotation** (*dict of lists of dicts*) – The keys are chromosome names. The values are lists of features for that chromosome. The features are represented as dicts with the following structure:

```
{
    'chrom': str
    'start': int,
    'end'  : int,
}
```

  See lib5c.parsers.bed.load_features().

- **pixelmap** (*pixelmap*) – The pixelmap to use to generate the annotationmap. See lib5c.parsers.bed.get_pixelmap().

Returns The keys of the dictionary are region names. The values are lists, where the i th entry represents the number of intersections between the annotation and the i th bin of that region.

Return type dict of lists

lib5c.util.ast_eval module

Module for safely evaluating arithmetic expressions in strings.

http://stackoverflow.com/a/9558001

lib5c.util.ast_eval.**eval_**(*node*, *variables*)

Inner function for safely evaluating a particular node of an abstract syntax tree. Called recursively to evaluate a string as part of eval_expr().

Parameters

- **node** (*ast.AST*) – The node of the abstract syntax tree to evaluate.
- **variables** (*dict*) – Dictionary of named variables to use during evaluation.

Returns The result of evaluating the node.

Return type numeric

lib5c.util.ast_eval.**eval_expr**(*expr*, *variables=None*)

Safely evaluates a simple mathematical expression passed as a string.

Parameters

- **expr** (*str*) – The expression to evaluate.
- **variables** (*dict, optional*) – Dict mapping variable names to values. These variables can then be used as substrings of `expr`. Pass None to evaluate the expression without using any named variables.

Returns The result of evaluating the expression.

Return type numeric

Examples

```
>>> eval_expr('2^6')
4
>>> eval_expr('2**6')
64
>>> eval_expr('1 + 2*3**(4^5) / (6 + -7)')
-5.0
```

lib5c.util.bed module

Module containing utilities for manipulating BED files and BED features.

BED features are commonly represented as dicts with the following structure:

```
{
    'chrom': str
    'start': int,
    'end'  : int,
}
```

but may also contain additional fields.

lib5c.util.bed.**check_intersect**(*a*, *b*)

Checks to see if two features intersect.

Parameters **b** (*a,*) – The two features to check for intersection.

Returns True if the features intersect, False otherwise.

Return type bool

Notes

Features are represented as dicts with the following structure:

```
{
    'chrom': str
    'start': int,
    'end'  : int,
}
```

See `lib5c.parsers.bed.load_features()`.

`lib5c.util.bed.`**`count_intersections`**(*query_feature*, *feature_set*)
Counts the number of times a query feature is hit by a set of other features.

Parameters

- **`query_feature`** (*`Dict[str, Any]`*) – The feature to count intersections for.
- **`feature_set`** (*`List[Dict[str, Any]]`*) – The set of features to intersect with the query feature.

Returns The number of intersections

Return type int

Notes

Features are represented as dicts with the following structure:

```
{
    'chrom': str
    'start': int,
    'end'  : int,
}
```

See `lib5c.parsers.bed.load_features()`.

`lib5c.util.bed.`**`flatten_features`**(*features*)
Flattens a features dict and returns a flat list of features.

Typically, BED features are kept in dicts organized by chromosome. For example, this is the data structure returned by `lib5c.parsers.bed.load_features()`. When a flat list is desired, this function can be used to flatten the dictionary into a simple list.

Parameters **`features`** (*`Dict[str, List[Dict[str, Any]]]`*) – The keys are chromosome names. The values are lists of features for that chromosome. The features are represented as dicts with at least the following keys:

```
{
    'start': int,
    'end'  : int
}
```

Returns

These dicts, which represent the same features as those contained in the original dict, have the following keys:

```
{
    'chrom': str,
    'start': int,
    'end'  : int
}
```

as well as any additional keys that were present in the inner dicts of the features dict passed to this function.

Return type List[Dict[str, Any]]

Notes

If the dicts that describe the features already contain a 'chrom' key, that key's value will get overwritten during the flattening.

lib5c.util.bed.**get_mid_to_mid_distance**(*fragment_a*, *fragment_b*)
Gets the mid-to-mid distance between two fragments.

Parameters **fragment_b** (*fragment_a,*) – The fragments to find the distance between. The fragments must be represented as dicts with at least the following keys:

```
{
    'start': int,
    'end': int
}
```

Returns The mid-to-mid distance

Return type float

lib5c.util.bed.**get_midpoint**(*fragment*, *force_int=False*)
Gets the midpoint of a fragment.

Parameters

- **fragment** (*Dict[str, Any]*) – The fragment to find the midpoint of. The fragment must be represented as a dict with at least the following keys:

```
{
    'start': int,
    'end': int
}
```

- **force_int** (*bool*) – Return an int rounded towards zero instead of a float.

Returns The midpoint of the fragment, rounded towards zero if force_int is True.

Return type float

Examples

```
>>> fragment = {'start': 50, 'end': 100}
>>> get_midpoint(fragment)
75.0
```

lib5c.util.bed.**main**()

lib5c.util.bed.**parse_feature_from_string**(*grange_string*)
Parses BED feature from a string specifying the genomic range.

Parameters **grange_string** (*str*) – The genomic range to parse, specified as a string of the form <chrom>:<start>-<end>. The interval is interpreted as a BED interval (0-based index, half-open interval).

Returns The BED feature dict, which has keys 'chrom', 'start', and 'end'.

Return type dict

lib5c.util.bedgraph module

Module containing utilities for manipulating bedgraph files.

lib5c.util.bedgraph.**in_boundaries**(*peak*, *chrom*, *boundaries*)

Checks to see if a given feature on a given chromosome is within the boundaries.

Parameters

- **peak** (*dict*) – The feature to check. This dict should have the following structure:

```
{
    'start': int,
    'end': int
}
```

  See lib5c.parsers.bed.load_features().

- **chrom** (*str*) – The chromosome this feature is on.
- **boundaries** (*dict of dicts*) – Information about the region boundaries. The outer keys are region names as strings. The values are dicts describing the boundaries of that region with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end': int
}
```

Returns True if the feature is in one of the regions, False otherwise.

Return type bool

lib5c.util.bedgraph.**main**()

lib5c.util.bedgraph.**reduce_bedgraph**(*bedfile*, *pixelmap*)

Reduces a bedgraph file by excluding peaks that fall outside of the regions described by a pixelmap.

Parameters

- **bedfile** (*str*) – String reference to the bedgraph file to reduce.
- **pixelmap** (*dict of lists of dicts*) – The keys of the outer dict are region names. The values are lists, where the `i` th entry represents the `i` th bin in that region. Bins are represented as dicts with at least the following structure:

```
{
    'chrom': string,
    'start': int,
    'end'  : int
}
```

  See `lib5c.parsers.primers.get_pixelmap()`.

Returns

The keys are chromosome names. The values are lists of peaks for that chromosome. The peaks are represented as dicts with at least the following keys:

```
{
    'start': int,
    'end'  : int
}
```

Return type dict of lists of dicts

lib5c.util.config module

lib5c.util.config.**parse_config**(*configfile*)
Parses a configfile in to a ConfigParser object.

Parameters **configfile** (*str*) – The configfile to parse.

Returns The parsed ConfigParser instance.

Return type ConfigParser

lib5c.util.counts module

Module containing utilities for manipulating 5C counts.

lib5c.util.counts.**abs_diff_counts**(*a*, *b*)
Computes the absolute value of the difference between two counts matrices in parallel across regions.

Parameters **b** (*a,*) – The two counts dicts to take the absolute difference between.

Returns The absolute value of the difference between two counts dicts.

Return type Dict[str, np.ndarray]

lib5c.util.counts.**apply_nonredundant**(*func*, *counts*, *primermap=None*)
Applies a function to some counts over the non-redundant elements of the matrix or matrices.

Parameters

- **func** (*callable*) – The function to apply.
- **counts** (*np.ndarray or dict of np.ndarray*) – The counts to apply the function to.
- **primermap** (*primermap, optional*) – If counts is a dict, pass a primermap to reconstruct the resulting counts dict.

Returns The result of the operation.

Return type np.ndarray or dict of np.ndarray

lib5c.util.counts.**apply_nonredundant_parallel**(*func*, *counts*, *primermap=None*)
Applies a function to some counts over the non-redundant elements of the matrix or matrices.

Parameters

- **func** (*callable*) – The function to apply.
- **counts** (*np.ndarray or dict of np.ndarray*) – The counts to apply the function to.
- **primermap** (*primermap, optional*) – If counts is a dict, pass a primermap to reconstruct the resulting counts dict.

Returns The result of the operation.

Return type np.ndarray or dict of np.ndarray

lib5c.util.counts.**calculate_pvalues**(*counts*, *distribution=<scipy.stats._continuous_distns.norm_gen object>*, *percentile_threshold=None*)

Applies lib5c.util.counts.calculate_regional_pvalues() to each region in a counts dict independently and returns the results as a parallel counts dict containing the p-value information.

Parameters

- **counts** (*dict of 2d numpy arrays*) – The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.
- **distribution** (*subclass of scipy.stats.rv_continuous*) – The distribution to use to model the data.
- **percentile_threshold** (*float between 0 and 100 or None*) – If passed, the `distribution` kwarg is ignored and p-value modeling is skipped. Instead, the returned data struture will contain dummy p-values, which will be 0.0 whenever the peak passes the percentile threshold, and 1.0 otherwise. This percentile threshold is applied independently for each region.

Returns The first element of the tuple is a counts dict containing p-values for each region. The keys are the region names. The values are the arrays of p-values for that region. These arrays are square and symmetric. The second element of the tuple is a dict containing information about the values of the parameters used when modeling each region's counts. The keys of this dict are region names, and its values are tuples of floats describing these parameters for that region with the following structure. The number and order of the floats will match the return value of `rv_continuous.fit()` for the particular distribution specified by the `distribution` kwarg.

Return type (dict of 2d numpy arrays, dict of tuples of floats) tuple

Notes

If you only need the p-values and don't care about the stats, you can also just do a dict comprehension as shown here:

```
{
    region: calculate_regional_pvalues(counts[region])[2]
    for region in counts.keys()
}
```

lib5c.util.counts.**calculate_regional_pvalues**(*regional_counts*, *distribution=<scipy.stats._continuous_distns.norm_gen object>*, *params=None*, *percentile_threshold=None*)

Models the distribution of counts within a region as a normal distribution with mean mu and standard deviation sigma, then returns an array of p-values for each pairwise interaction assuming that normal distribution.

Parameters

- **regional_counts** (*2d numpy array*) – The square, symmetric array of counts for one region.
- **distribution** (*subclass of scipy.stats.rv_continuous*) – The distribution to use to model the data.
- **params** (*tuple of floats or None*) – The parameters to plug into the distribution specified by the `distribution` kwarg when modeling the data. If None is passed, the parameters will be automatically calculated using `rv_continuous.fit()`. The number

and order of the floats should match the return value of `rv_continuous.fit()` for the particular distribution specified by the `distribution` kwarg.

- **percentile_threshold** (*float between 0 and 100 or None*) – If passed, the distribution and params kwargs are ignored and p-value modeling is skipped. Instead, the returned data struture will contain dummy p-values, which will be 0.0 whenever the peak passes the percentile threshold, and 1.0 otherwise.

Returns The tuple of floats contains the values of the parameters used to model the distribution. If `percentile_threshold was passed`, this tuple will contain only one float, which will be the value used for thresholding. The 2d numpy array is the p-value for each count.

Return type (tuple of floats or None, 2d numpy array) tuple

lib5c.util.counts.**convert_pvalues_to_interaction_scores**(*pvalues*)

Calculates interaction scores from p-values.

Parameters **pvalues** (*np.ndarray*) – An array of p-values for a single region

Returns An array of interaction scores for a single region

Return type np.ndarray

lib5c.util.counts.**distance_filter**(*matrix*, *k=5*)

Wipes the first *k* off-diagonals of *matrix* with *np.nan*.

Parameters

- **matrix** (*np.ndarray*) – The matrix to distance filter.
- **k** (*int*) – The number of off-diagonals to wipe.

Returns The wiped matrix.

Return type np.ndarray

lib5c.util.counts.**divide_regional_counts**(**list_of_regional_counts*)

Perform element-wise serial division on a list of regional counts matrices.

Parallelizable; see `lib5c.util.counts.parallel_divide_counts()`.

Propagates nan's. Emits nan's when dividing by zero.

Parameters **list_of_regional_counts** (*List[np.ndarray]*) – The list of regional counts matrices to divide.

Returns The quotient.

Return type np.ndarray

lib5c.util.counts.**extract_queried_counts**(*regional_counts*, *regional_primermap*)

Starting from a square, symmetric counts matrix containing primer-level contact information, return a non-square, non-symmetric matrix where the 5'-oriented fragments sit in the rows of the matrix while the 3'-oriented fragments sit in the columns. This restricts the input matrix to only the pairwise contacts that were actually queried by the 5C assay.

Parameters

- **regional_counts** (*np.ndarray*) – The classic square, symmetric counts matrix for this region.
- **regional_primermap** (*List[Dict[str, Any]]*) – The primermap describing the fragments in this region. It must contain a 'orientation' metadata key so that `regional_primermap[i]['orientation']` is `"5'"` when the fragment was targeted by a 5'-oriented primer and `"3'"` otherwise.

Returns The np.ndarray is the queried counts matrix, as described above. The two lists of dicts are lists of the primers corresponding to the rows and columns, respectively, of the queried counts matrix.

Return type np.ndarray, list of dict, list of dict

lib5c.util.counts.**flatten_and_filter_counts**(*counts*, *min_filters=None*, *max_filters=None*)

Flattens and filters multiple counts dicts (typically containing different types or stages of data) in parallel, applying customizable filters.

Parameters

- **counts** (*dict of dict of np.ndarray or dict of np.ndarray*) – Outer keys are always names of the types of count dicts. Inner keys are optional and represent region names. If this level of the dict is omitted this function will flatten all the regional counts matrices. Values are always square symmetric counts matrices.
- **max_filters** (*min_filters,*) – Map outer keys of `counts` to minimum or maximum values for that type of counts.

Returns The dict's values are the parallel flattened and filtered count vectors. It has an extra 'dist' key for interaction distance in bin units. The array is a boolean index into the original flattened counts shape representing which positions have been filtered. The list is the order the regions were flattened in, or None if `counts` had only one level of keys.

Return type dict of np.ndarray, np.ndarray, list of str

Notes

To separately flatten each region, you can do:

flat, idx, _ = parallelize_regions(flatten_and_filter_counts)({r: {t: counts[t][r] for t in types} for r in regions})

where `flat` will be a dict of dict of flattened vectors (outer keys are regions, inner keys are types) and `idx` will be a dict of boolean indices (keys are regions).

lib5c.util.counts.**flatten_counts**(*counts*, *discard_nan=False*)

Flattens each region in a counts dictionary into a flat, nonredundant list.

Parameters

- **counts** (*dict of 2d numpy arrays*) – The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.
- **discard_nan** (*bool*) – If True, nan's will not be present in the returned lists.

Returns The keys are the region names. The values are flat, nonredundant lists of counts for that region. The `(i, j)` th element of the counts for a region (for `i >= j`) ends up at the `(i*(i+1)/2 + j)` th index of the flattened list for that region. If discard_nan is True, then the nan elements will be missing and this specific indexing will not be preserved.

Return type dict of lists of floats

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import flatten_counts
>>> counts = {'a': np.array([[1, 2], [2, 3.]]),
...           'b': np.array([[np.nan, 4], [4, 5.]])}
>>> flat_counts = flatten_counts(counts)
>>> list(sorted(flat_counts.keys()))
['a', 'b']
>>> flat_counts['a']
array([1., 2., 3.])
>>> flat_counts['b']
array([nan,  4.,  5.])
>>> flat_counts = flatten_counts(counts, discard_nan=True)
>>> flat_counts['a']
array([1., 2., 3.])
>>> flat_counts['b']
array([4.,  5.])
```

lib5c.util.counts.**flatten_counts_to_list**(*counts*, *region_order=None*, *discard_nan=False*)

Flattens counts for all regions into a single, flat, nonredudant list.

Parameters

- **counts** (*dict of 2d numpy arrays*) – The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.
- **region_order** (*list of str*) – List of string reference to region names in the order the regions should be concatenated when constructing the flat list. If None, the regions will be concatenated in arbitrary order.
- **discard_nan** (*bool*) – If True, nan's will not be present in the returned list.

Returns The concatenated flattened regional counts. For information on the order in which flattened regional counts are created in, see `lib5c.util.counts.flatten_regional_counts()`. If discard_nan is True, then the nan elements will be missing and this specific indexing will not be preserved.

Return type 1d numpy array

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import flatten_counts_to_list
>>> counts = {'a': np.array([[1, 2], [2, 3.]]),
...           'b': np.array([[np.nan, 4], [4, 5.]])}
>>> flatten_counts_to_list(counts, region_order=['a', 'b'])
array([  1.,   2.,   3.,  nan,   4.,   5.])
>>> flatten_counts_to_list(counts, region_order=['b', 'a'])
array([nan,  4.,  5.,  1.,  2.,  3.])
>>> flatten_counts_to_list(counts, region_order=['a', 'b'],
...                        discard_nan=True)
array([1., 2., 3., 4., 5.])
```

lib5c.util.counts.**flatten_obs_and_exp**(*obs*, *exp*, *discard_nan=True*, *log=False*)

Convenience function for flattening observed and expected counts together.

Parameters

- **exp** (*obs,*) – Regional matrices of observed and expected values, respectively. Pass counts dicts to redirect the call to `flatten_obs_and_exp_counts()`.
- **discard_nan** (*bool*) – Pass True to discard nan's from the returned vectors.
- **log** (*bool*) – Pass True to log the returned vectors.

Returns The flattened vectors of obsereved and expected values, respectively.

Return type np.ndarray, np.ndarray

lib5c.util.counts.**flatten_obs_and_exp_counts**(*obs_counts*, *exp_counts*, *discard_nan=True*, *log=False*)

Convenience function for flattening observed and expected counts together.

Parameters

- **exp_counts** (*obs_counts,*) – Counts dicts of observed and expected values, respectively.
- **discard_nan** (*bool*) – Pass True to discard nan's from the returned vectors.
- **log** (*bool*) – Pass True to log the returned vectors.

Returns The flattened vectors of obsereved and expected values, respectively.

Return type np.ndarray, np.ndarray

lib5c.util.counts.**flatten_regional_counts**(*regional_counts*, *discard_nan=False*)

Flattens the counts for a single region into a flat, nonredundant list.

Parameters

- **regional_counts** (*2d numpy array*) – The square, symmetric array of counts for one region.
- **discard_nan** (*bool*) – If True, nan's will not be present in the returned list.

Returns A flat, nonredundant lists of counts. The `(i, j)` th element of the `regional_counts` array (for `i >= j`) ends up at the `(i*(i+1)/2 + j)` th index of the flattened array. If `discard_nan` was True, these indices will not necessarily match up and it will not be possible to unflatten the array.

Return type 1d numpy array

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import flatten_regional_counts
>>> a = np.array([[ 1, 4, -7], [4, 5, np.nan], [-7, np.nan, 9.]])
>>> a
array([[ 1.,  4., -7.],
       [ 4.,  5., nan],
       [-7., nan,  9.]])
>>> flatten_regional_counts(a)
array([  1.,   4.,   5.,  -7.,  nan,   9.])
>>> flatten_regional_counts(a, discard_nan=True)
array([ 1.,  4.,  5., -7.,  9.])
```

lib5c.util.counts.**flip_pvalues**(*regional_counts*)

To some approximation, convert counts matrices containing left-tail p-values to right-tail p-values or vice-versa.

Parameters **regional_counts** (*np.ndarray*) – The counts matrix containing p-values to flip.

Returns The flipped p-values.

Return type np.ndarray

lib5c.util.counts.**fold_pvalues**(*regional_counts*)

Folds one-tail p-values into two-tail p-values using `convert_to_two_tail()`. Only valid for p-values called using continuous distributions.

Parameters **regional_counts** (*np.ndarray*) – An array of one-tail p-values for a single region.

Returns An array of the corresponding two-tail p-values.

Return type np.ndarray

lib5c.util.counts.**impute_values**(*regional_counts*, *size=5*)

Impute missing (nan) values in a counts matrix using a local median estimate.

Parameters

- **regional_counts** (*np.ndarray*) – The counts matrix to imupte.
- **size** (*int*) – The size of the window used to compute the local median. Should be an odd integer.

Returns The counts matrix with missing values filled in with the local median estimates.

Return type np.ndarray

lib5c.util.counts.**log_regional_counts**(*regional_counts*, *pseudocount=1.0*, *base='e'*)

Logs a regional counts matrix.

Parallelizable; see `lib5c.util.counts.parallel_log_counts()`.

Emits nan when logging a negative number, and -inf when logging zero.

Parameters

- **regional_counts** (*np.ndarray*) – The counts matrix to log.
- **pseudocount** (*float*) – Psuedocount to add before logging.
- **base** (*str or float*) – The base to use when logging. Acceptable string values are 'e', '2', or '10'.

Returns The logged counts matrix.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import log_regional_counts
>>> a = np.exp(np.array([[1, 2], [2, 4.]]))
>>> log_regional_counts(a, pseudocount=0)
array([[1., 2.],
       [2., 4.]])
>>> a -= 1 #  the default pseudocount will add this back before logging
>>> a[0, 0] = -2  # what happens to negative values?
>>> log_regional_counts(a)
```

(continues on next page)

(continued from previous page)

```
array([[nan,  2.],
       [ 2.,  4.]])
>>> b = np.power(42, np.array([[1, 2], [2, 4.]]))
>>> log_regional_counts(b, base=42, pseudocount=0)
array([[1., 2.],
       [2., 4.]])
```

lib5c.util.counts.**norm_counts**(*counts*, *order=1*)

Attempt at defining a "norm" for counts dicts by simply summing a matrix p-norm over the regions.

Parameters

- **counts** (*Dict[str, np.ndarray]*) – The counts dict to compute a norm for.
- **order** (*int*) – The order of the matrix norm, as described by `numpy.linalg.norm`.

Returns The norm of the counts dict.

Return type float

lib5c.util.counts.**parallel_divide_counts**(**list_of_regional_counts*)

Perform element-wise serial division on a list of regional counts matrices.

Parallelizable; see `lib5c.util.counts.parallel_divide_counts()`.

Propagates nan's. Emits nan's when dividing by zero.

Parameters **list_of_regional_counts** (*List[np.ndarray]*) – The list of regional counts matrices to divide.

Returns The quotient.

Return type np.ndarray

lib5c.util.counts.**parallel_log_counts**(*regional_counts*, *pseudocount=1.0*, *base='e'*)

Logs a regional counts matrix.

Parallelizable; see `lib5c.util.counts.parallel_log_counts()`.

Emits nan when logging a negative number, and -inf when logging zero.

Parameters

- **regional_counts** (*np.ndarray*) – The counts matrix to log.
- **pseudocount** (*float*) – Psuedocount to add before logging.
- **base** (*str or float*) – The base to use when logging. Acceptable string values are 'e', '2', or '10'.

Returns The logged counts matrix.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import log_regional_counts
>>> a = np.exp(np.array([[1, 2], [2, 4.]]))
>>> log_regional_counts(a, pseudocount=0)
array([[1., 2.],
       [2., 4.]])
```

(continues on next page)

(continued from previous page)

```
>>> a -= 1 #  the default pseudocount will add this back before logging
>>> a[0, 0] = -2  # what happens to negative values?
>>> log_regional_counts(a)
array([[nan,  2.],
       [ 2.,  4.]])
>>> b = np.power(42, np.array([[1, 2], [2, 4.]]))
>>> log_regional_counts(b, base=42, pseudocount=0)
array([[1., 2.],
       [2., 4.]])
```

lib5c.util.counts.**parallel_subtract_counts**(**list_of_regional_counts*)

Perform element-wise serial subtraction on a list of regional counts matrices.

Parallelizable; see `lib5c.util.counts.parallel_subtract_counts()`.

Propagates nan's.

Parameters **list_of_regional_counts** (*List[np.ndarray]*) – The list of regional counts matrices to divide.

Returns The quotient.

Return type np.ndarray

lib5c.util.counts.**parallel_unlog_counts**(*regional_counts*, *pseudocount=1.0*, *base='e'*)

Unlogs a regional counts matrix.

Parallelizable; see `lib5c.util.counts.parallel_unlog_counts()`.

Emits nan's when the input counts are nan.

Parameters

- **regional_counts** (*np.ndarray*) – The counts matrix to unlog.
- **pseudocount** (*float*) – Psuedocount to subtract after unlogging.
- **base** (*str or float*) – The base to use when unlogging. Acceptable string values are 'e', '2', or '10'.

Returns The unlogged counts matrix.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import log_regional_counts, unlog_regional_counts
>>> a = np.array([[1, 2], [2, 4.]])
>>> log_regional_counts(unlog_regional_counts(a))
array([[1., 2.],
       [2., 4.]])
>>> log_regional_counts(unlog_regional_counts(a, base=42), base=42)
array([[1., 2.],
       [2., 4.]])
```

lib5c.util.counts.**propagate_nans**(*regional_counts_a*, *regional_counts_b*)

Propagate nan values between two matrices.

Parameters **regional_counts_b** (*regional_counts_a,*) – The matrices to propagate nan's between. These should have the same shape.

Returns The nan-propagated versions of the input matrices, in the order they were passed.

Return type Tuple[np.ndarray, np.ndarray]

lib5c.util.counts.**queried_counts_to_pvalues**(*queried_counts*)

Convert a queried counts matrix to a matrix of equivalent right-tail p-values using the emprical CDF.

Parameters **queried_counts** (*np.ndarray*) – The matrix of queried counts for this region. See `lib5c.util.counts.extract_queried_counts()`.

Returns The empirical p-value queried counts matrix for this region.

Return type np.ndarray

lib5c.util.counts.**regional_counts_to_pvalues**(*regional_counts*)

Convert a counts matrix to a matrix of equivalent right-tail p-values using the emprical CDF.

Parameters **regional_counts** (*np.ndarray*) – The counts matrix for this region.

Returns The empirical p-value counts matrix for this region.

Return type np.ndarray

lib5c.util.counts.**subtract_regional_counts**(**list_of_regional_counts*)

Perform element-wise serial subtraction on a list of regional counts matrices.

Parallelizable; see `lib5c.util.counts.parallel_subtract_counts()`.

Propagates nan's.

Parameters **list_of_regional_counts** (*List[np.ndarray]*) – The list of regional counts matrices to divide.

Returns The quotient.

Return type np.ndarray

lib5c.util.counts.**unflatten_counts**(*flat_counts*)

Apply `unflatten_regional_counts()` in parallel to a dict of flat regional counts to get back the original counts dict.

Parameters **flat_counts** (*Dict[str, List[float]]*) – The keys are region names as strings. The values are flat, nonredundant lists of counts for that region. The `(i*(i+1)/2 + j)` th element of each list will end up at both the `(i, j)` th and the `(j, i)` th element of the returned array for that region.

Returns The keys are region names as strings. The values are square, symmetric array representations of the counts for that region. The `(i*(i+1)/2 + j)` th element of flat_regional_counts will end up at both the `(i, j)` th and the `(j, i)` th element of this array.

Return type Dict[str, np.ndarray]

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import unflatten_counts
>>> flattened_counts = {'a': np.array([1, 2, 3.]),
...                     'b': np.array([np.nan, 4, 5.])}
>>> counts = unflatten_counts(flattened_counts)
>>> list(sorted(counts.keys()))
['a', 'b']
>>> counts['a']
```

(continues on next page)

(continued from previous page)

```
array([[1., 2.],
       [2., 3.]])
>>> counts['b']
array([[nan,  4.],
       [ 4.,  5.]])
```

lib5c.util.counts.**unflatten_counts_from_list**(*flattened_counts_array*, *region_order*, *pixelmap*)

Unflattens a single list of flattened counts from many regions into a standard counts dict structure.

Parameters

- **flattened_counts_array** (*1d numpy array*) – The list of flattened counts to be unflattened. See `lib5c.util.counts.flatten_counts_to_list()`.
- **region_order** (*list of str*) – The list of region names in the order that the regions were concatenated in when making the `flattened_counts_list` See `lib5c.util.counts.flatten_counts_to_list()`.
- **pixelmap** (*dict of list of dict*) – A pixelmap or primermap. This will be used to determine the size of each region. See `lib5c.parsers.primers.get_pixelmap()` or `lib5c.parsers.primers.get_primermap()`.

Returns The keys are the region names. The values are the arrays of counts values for that region. These arrays are square and symmetric.

Return type dict of 2d numpy arrays

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import unflatten_counts_from_list
>>> flat_counts = np.array([1, 2, 3., np.nan, 4, 5.])
>>> pixelmap = {'a': [{}, {}], 'b': [{}, {}]}
>>> counts = unflatten_counts_from_list(flat_counts, ['a', 'b'], pixelmap)
>>> list(sorted(counts.keys()))
['a', 'b']
>>> counts['a']
array([[1., 2.],
       [2., 3.]])
>>> counts['b']
array([[nan,  4.],
       [ 4.,  5.]])
```

lib5c.util.counts.**unflatten_regional_counts**(*flat_regional_counts*)

Turn a list of flattened counts back into a square symmetric array.

Parameters **flat_regional_counts** (*1d numpy array*) – A flat, nonredundant array of counts. The `(i*(i+1)/2 + j)` th element of this list will end up at both the `(i, j)` th and the `(j, i)` th element of the returned array.

Returns A square, symmetric array representation of the counts. The `(i*(i+1)/2 + j)` th element of `flat_regional_counts` will end up at both the `(i, j)` th and the `(j, i)` th element of this array.

Return type 2d numpy array

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import unflatten_regional_counts
>>> b = np.array([ 1, 4, 5, -7, np.nan, 9.])
>>> b
array([  1.,   4.,   5.,  -7.,  nan,   9.])
>>> unflatten_regional_counts(b)
array([[ 1.,  4., -7.],
       [ 4.,  5., nan],
       [-7., nan,  9.]])
```

lib5c.util.counts.**unlog_regional_counts**(*regional_counts*, *pseudocount=1.0*, *base='e'*)

Unlogs a regional counts matrix.

Parallelizable; see `lib5c.util.counts.parallel_unlog_counts()`.

Emits nan's when the input counts are nan.

Parameters

- **regional_counts** (*np.ndarray*) – The counts matrix to unlog.
- **pseudocount** (*float*) – Psuedocount to subtract after unlogging.
- **base** (*str or float*) – The base to use when unlogging. Acceptable string values are 'e', '2', or '10'.

Returns The unlogged counts matrix.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> from lib5c.util.counts import log_regional_counts, unlog_regional_counts
>>> a = np.array([[1, 2], [2, 4.]])
>>> log_regional_counts(unlog_regional_counts(a))
array([[1., 2.],
       [2., 4.]])
>>> log_regional_counts(unlog_regional_counts(a, base=42), base=42)
array([[1., 2.],
       [2., 4.]])
```

lib5c.util.counts_superdict module

Module for creating and performing operations on `counts_superdict` data structures. `counts_superdict` data structures areconceptually just collections of named counts dicts. Typically, the members of the collection correspond to replicates or conditions in a 5C experiment. `counts_superdict` data structures are implemented as dicts of dicts of 2D numpy arrays as shown in this example:

```
counts_superdict[replicate_name][region_name] = 2D numpy array
```

Here the innermost values are the square symmetric 2D numpy arrays representing the counts for the specified replicate and region. In other words, they are a dict whose keys are replicate names as strings and whose values are standard counts data structures. See `lib5c.parsers.counts.load_counts()` or `lib5c.parsers.counts.load_primer_counts()`.

lib5c.util.counts_superdict.**counts_superdict_to_matrix**(*counts_superdict*, *rep_order=None*, *region_order=None*, *discard_nan=False*)

Convert a counts_superdict structure to a matrix.

Parameters

- **counts_superdict** (*Dict[Dict[np.ndarray]]*) – The keys of the outer dict are replicate names, the keys of the inner dict are region names, the values are square symmetric arrays of counts for the specified replicate and region.
- **rep_order** (*Optional[List[str]]*) – The order in which the replicates should be arranged. Pass None to use the order of `counts_superdict.keys()`.
- **region_order** (*Optional[List[str]]*) – The order in which the regions should be concatenated. Pass None to use the order of the keys.
- **discard_nan** (*bool*) – If True, positions containing `nan` values will be removed.

Returns Each row is a replicate, each column is a position. The order of the columns is described in `lib5c.util.counts.flatten_counts_to_list()`, honoring the passed region order.

Return type np.ndarray

Examples

```
>>> import numpy as np
>>> counts_superdict = {'Rep1': {'A': np.array([[1, 2], [2, 3]]),
...                              'B': np.array([[4, 8], [8, 10]])},
...                     'Rep2': {'A': np.array([[3, 5], [5, 6]]),
...                              'B': np.array([[7, 9], [9, np.nan]])}}
>>> rep_order = ['Rep1', 'Rep2']
>>> region_order = ['A', 'B']
>>> counts_superdict_to_matrix(counts_superdict, rep_order, region_order)
array([[  1.,   2.,   3.,   4.,   8.,  10.],
       [  3.,   5.,   6.,   7.,   9.,  nan]])
>>> counts_superdict_to_matrix(counts_superdict, rep_order, region_order,
...                            discard_nan=True)
array([[1., 2., 3., 4., 8.],
       [3., 5., 6., 7., 9.]])
```

lib5c.util.counts_superdict.**make_atlas**(*counts_superdict*, *pvalues_superdict=None*, *distribution=<scipy.stats._continuous_distns.norm_gen object>*, *percentile_threshold=None*)

Computes the maximum counts and minimum p-values for each region of a 5C dataset across multiple replicates or conditions.

Parameters

- **counts_superdict** (*dict of counts dicts*) – The counts for all replicates or conditions to be used to generate the atlas. The keys are the replicate or condition names, the values are standard counts dicts containing the counts for that replicate. See `lib5c.parsers.counts.load_counts()` or `lib5c.parsers.counts.load_primer_counts()`.
- **pvalues_superdict** (*dict of counts dicts or None*) – The counts for all replicates or conditions to be used to generate the atlas. The keys are the replicate or con-

dition names, the values are standard counts dicts containing the p-values for that replcicate. If None is passed, the p-values will be automatically computed using `lib5c.util.counts.calculate_regional_pvalues()`.

- **distribution** (*subclass of scipy.stats.rv_continuous*) – If pvalue_superdict is None, this kwarg specifies the distribution to use when modeling the counts.
- **percentile_threshold** (*float between 0 and 100 or None*) – If passed, all p-value modeling kwargs are ignored and all p-value modeling steps are skipped. Instead, the returned atlas peaks will contain a dummy p-value, which will be 0.0 whenever the peak passes the percentile threshold, and 1.0 otherwise.

Returns The first element in the tuple is the counts dict contatining the maximum counts observed across all replicates at each primer or bin combination in each region. The second element in the tuple is the parallel counts dict containing the corresponding minimum p-values.

Return type (counts dict, counts dict) tuple

`lib5c.util.counts_superdict.`**make_atlas_peaks**(*counts_superdict*, *pvalues_superdict=None*, *max_counts=None*, *min_pvalues=None*, *distribution=<scipy.stats._continuous_distns.norm_gen object>*, *percentile_threshold=None*)

Reshapes count and p-value information across multiple replicates into a peaks data structure compatible with the `lib5c.algorithms.clustering` subpackage.

Parameters

- **counts_superdict** (*dict of counts dicts*) – The counts for all replicates or conditions to be used to generate the atlas. The keys are the replicate or condition names, the values are standard counts dicts containing the counts for that replicate. See `lib5c.parsers.counts.load_counts()` or `lib5c.parsers.counts.load_primer_counts()`.
- **pvalues_superdict** (*dict of counts dicts or None*) – The p-values for all replicates or conditions to be used to generate the atlas. The keys are the replicate or condition names, the values are standard counts dicts containing the p-values for that replicate. See `lib5c.parsers.counts.load_counts()` or `lib5c.parsers.counts.load_primer_counts()`. If None is passed, p-values will be automatically computed for each replicate for each region by modeling each region within each replicate independently with the distribution specified by the distribution kwarg. See `lib5c.util.counts.calculate_regional_pvalues()`.
- **max_counts** (*counts dict or None*) – A standard counts dict containing the maximum count value observed for every interaction in each region across all replicates. If None is passed, this will be computed automatically based on the `counts_superdict`. See `lib5c.util.counts.make_atlas()`.
- **min_pvalues** (*counts dict or None*) – A standard counts dict containing the minimum p-value observed for every interaction in each region across all replicates. If None is passed, this will be computed automatically based on the `pvalues_superdict`. See `lib5c.util.counts.make_atlas()`.
- **distribution** (*subclass of scipy.stats.rv_continuous*) – If pvalue_superdict is None, this kwarg specifies the distribution to use when modeling the counts.
- **percentile_threshold** (*float between 0 and 100 or None*) – If passed, all p-value modeling kwargs are ignored and all p-value modeling steps are skipped. Instead,

the returned atlas peaks will contain a dummy p-value, which will be 0.0 whenever the peak passes the percentile threshold, and 1.0 otherwise.

Returns

The returned peaks data structure has the following structure:

```
atlas_peaks[region_name] = {
                                'x': int,
                                'y': int,
                                'value': float,
                                'pvalue': float,
                                'replicates' : {
                                    rep_name: {
                                        'value': float,
                                        'pvalue': float
                                    }
                                }
                            }
```

Here `region_name` is a string referring to the region name. The 'x' and 'y' keys describe the x- and y-coordinate of the peak, respectively. The 'value' and 'pvalue' keys describe the max count and minimum p-value observed at those coordinates within the specified region across all replicates. The 'replicates' key's value is a dict containing more information about the values and p-values observed at the specified coordinates in each of the replicates in the original counts_superdict and pvalues_superdict.

Return type complex dict structure

Notes

Only the lower-triangular and diagonal elements (those for which the x-coordinate is greater than or equal to the y-coordinate) of each region's counts are included in the returned data structure to prevent duplication of data.

lib5c.util.counts_superdict.**make_pvalues_superdict**(*counts_superdict*, *distribution=<scipy.stats._continuous_distns.norm_gen object>*, *percentile_threshold=None*)

Makes a counts_superdict-like data structure containing automatically computed p-values for each replicate for each region. The counts within each region are modeled independently for each replicate using the distribution specified by the distribution kwarg. See `lib5c.util.counts.calculate_regional_pvalues()`.

Parameters

- **counts_superdict** (*dict of counts dicts*) – The counts for all replicates or conditions for which p-values should be computed. The keys are the replicate or condition names, the values are standard counts dicts containing the counts for that replicate. See `lib5c.parsers.counts.load_counts()` or `lib5c.parsers.counts.load_primer_counts()`.
- **distribution** (*subclass of scipy.stats.rv_continuous*) – The distribution to use when modeling the counts.
- **percentile_threshold** (*float between 0 and 100 or None*) – If passed, the distribution kwarg is ignored and p-value modeling is skipped. Instead, the returned data struture will contain dummy p-values, which will be 0.0 whenever the peak passes the percentile threshold, and 1.0 otherwise. This percentile threshold is applied independently for each region and each replicate.

Returns A parallel data structure containing the corresponding p-values.

Return type dict of counts dicts

lib5c.util.counts_superdict.**matrix_to_counts_superdict**(*matrix*, *rep_order*, *region_order*, *pixelmap*)

Converts a matrix back into a counts_superdict structure.

Parameters

- **matrix** (*np.ndarray*) – The matrix to convert to a counts superdict. The rows are replicates, the columns are bin-bin pairs or FFLJs.
- **rep_order** (*list of str*) – The replicate names to use in the counts_superdict, in the order of the columns of `matrix`.
- **region_order** (*list of str*) – The order in which the regions were concatenated down the rows of `matrix`.
- **pixelmap** (*pixelmap*) – Needed to precompute the size of each region when preparing the counts_superdict.

Returns The counts_superdict representation of the input matrix.

Return type counts_superdict

Examples

```
>>> import numpy as np
>>> matrix = np.array([[1, 2, 3, 4, 8, 10],
...                    [3, 5, 6, 7, 9, np.nan]])
>>> rep_order = ['Rep1', 'Rep2']
>>> region_order = ['A', 'B']
>>> pixelmap = {'A': [{}, {}], 'B': [{}, {}]}
>>> counts_superdict = matrix_to_counts_superdict(
...     matrix, rep_order, region_order, pixelmap)
>>> list(sorted(counts_superdict.keys()))
['Rep1', 'Rep2']
>>> list(sorted(counts_superdict['Rep1'].keys()))
['A', 'B']
>>> list(sorted(counts_superdict['Rep2'].keys()))
['A', 'B']
>>> counts_superdict['Rep1']['A']
array([[1., 2.],
       [2., 3.]])
>>> counts_superdict['Rep1']['B']
array([[ 4.,  8.],
       [ 8., 10.]])
>>> counts_superdict['Rep2']['A']
array([[3., 5.],
       [5., 6.]])
>>> counts_superdict['Rep2']['B']
array([[ 7.,  9.],
       [ 9., nan]])
```

lib5c.util.demo_data module

lib5c.util.demo_data.**download_gzipped_file**(*url*, *target*, *chunk_size=16384*)
Downloads and unzips a gzipped remote file.

Parameters

- **url** (*str*) – The URL of the remote file.
- **target** (*str*) – The complete path to write the unzipped file to.

lib5c.util.demo_data.**edit_demo_config**(*config_file='luigi.cfg'*)
Given the path to a default pipeline config file on disk, this function edits that default config file in place to prepare it for the pipeline tutorial.

Parameters **config_file** (*str*) – The path to a default pipeline config file.

lib5c.util.demo_data.**ensure_demo_data**(*dest_dir='input'*)
Downloads the demo data if it doesn't exist yet.

The primerfile will be edited to remove regions other than Sox2 and Klf4.

Parameters **dest_dir** (*str*) – The directory to download the demo data to. Will be created if it doesn't exist.

lib5c.util.demo_data.**main**()

lib5c.util.dictionaries module

Module containing utility functions for working with dicts.

lib5c.util.dictionaries.**reduced_get**(*key*, *dicts*, *default=None*)
Reduced version of `dict.get()`.

Parameters

- **key** (*any*) – The key to search for in dicts.
- **dicts** (*iterable of dicts*) – The dicts to search for key. Dicts further right have higher priority.
- **default** (*any, optional*) – The default value if the key is not found in any of the dicts.

Returns The value of the key in the furthest-right dict, or default if none of the dicts had the key.

Return type any

lib5c.util.distributions module

Module containing utility functions for parametrizing statistical distributions.

lib5c.util.distributions.**call_pvalues**(*obs*, *exp*, *var*, *dist_gen*, *log=False*)
Call right-tail p-values for obs against a theoretical distribution whose family is specified by dist_gen an whose first two moments are specified by exp and var, respectively.

Parameters

- **obs** (*float*) – The observed value to call a p-value for.
- **var** (*exp,*) – The first two moments of the null distribution.

- **dist_gen** (*scipy.stats.rv_generic or str*) – The null distribution to parameterize. If a string is passed this will be replaced with *getattr(scipy.stats, dist_gen)*.
- **log** (*bool*) – Pass True to attempt to convert exp and var to log-scale.

Returns The right-tail p-value.

Return type float

Notes

This function is array-safe.

lib5c.util.distributions.**convert_parameters**(*mu*, *sigma_2*, *dist_gen*, *log=False*)

Obtain correct scipy.stats parameterizations for selected one- and two- parameter distributions given a desired mean and variance.

Parameters

- **mu** (*float*) – The mean of the desired distribution.
- **sigma_2** (*float*) – The variance of the desired distribution.
- **dist_gen** (*scipy.stats.rv_generic*) – The target distribution.
- **log** (*bool*) – Pass True to attempt to convert exp and var to log-scale.

Returns The appropriate scipy.stats parameters.

Return type tuple of float

lib5c.util.distributions.**freeze_distribution**(*dist_gen*, *mu*, *sigma_2*, *log=False*)

Create a frozen distribution of a given type, given a mean and variance.

Parameters

- **dist_gen** (*scipy.stats.rv_generic*) – The distribution to use.
- **mu** (*float*) – The desired mean.
- **sigma_2** (*float*) – The desired variance.
- **log** (*bool*) – Pass True to attempt to convert mu and sigma_2 to log-scale.

Returns A frozen distribution of the specified type with specified mean and variance.

Return type scipy.stats.rv_frozen

Notes

This function does not perform any fitting, because it assumes that the first two moments directly and uniquely identify the desired distribution.

Examples

```
>>> frozen_dist = freeze_distribution(stats.poisson, 4.0, 4.0)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
poisson distribution with mean 4.00 and variance 4.00
```

```
>>> frozen_dist = freeze_distribution(stats.nbinom, 4.0, 3.0)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
nbinom distribution with mean 4.00 and variance 4.00
```

```
>>> frozen_dist = freeze_distribution(stats.nbinom, 3.0, 4.0)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
nbinom distribution with mean 3.00 and variance 4.00
```

```
>>> frozen_dist = freeze_distribution(stats.norm, 4.0, 3.0)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
norm distribution with mean 4.00 and variance 3.00
```

```
>>> frozen_dist = freeze_distribution(stats.logistic, 4.0, 3.0)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
logistic distribution with mean 4.00 and variance 3.00
>>> mu = 2  # mean of a normal random variable X
>>> sigma_2 = 16  # variance of X
>>> scale = np.exp(mu)  # parameter conversion for scipy.stats.lognorm
>>> s = np.sqrt(sigma_2)  # ditto
>>> y = stats.lognorm(s=s, scale=scale)  # a lognormal RV: exp(X) = Y
>>> m, v = y.stats(moments='mv')  # mean and variance of Y
>>> m, v
(array(22026.4657948...), array(4.31123106e+15))
>>> frozen_dist = freeze_distribution(stats.logistic, m, v, log=True)
>>> print('%s distribution with mean %.2f and variance %.2f'
...       % ((frozen_dist.dist.name,) + frozen_dist.stats(moments='mv')))
logistic distribution with mean 2.00 and variance 16.00
```

lib5c.util.distributions.**log_parameters**(*m*, *v*)

Attempts to guess appropriate log-scale mean and variance parameters given non-log scale estimators of these quantities, under the assumptions of a lognormal model.

Based on https://en.wikipedia.org/wiki/Log-normal_distribution#Notation

Parameters

- **m** (*float*) – The non-log scale mean.
- **v** (*float*) – The non-log scale variance.

Returns The first float is the log-scale mean, the second is the log-scale variance.

Return type float, float

Notes

This function is array-safe.

Examples

```
>>> mu = 2  # mean of a normal random variable X
>>> sigma_2 = 16  # variance of X
>>> scale = np.exp(mu)  # parameter conversion for scipy.stats.lognorm
>>> s = np.sqrt(sigma_2)  # ditto
>>> y = stats.lognorm(s=s, scale=scale)  # a lognormal RV: exp(X) = Y
>>> m, v = y.stats(moments='mv')  # mean and variance of Y
>>> m, v
(array(22026.4657948...), array(4.31123106e+15))
>>> log_parameters(m, v)  # recover moments of X from moments of Y
(2.0, 16.0)
```

lib5c.util.donut module

Module containing utility functions related to creating and using donut-shaped windows in the style of Rao et al. (Cell 2014).

lib5c.util.donut.**donut_footprint**(*p*, *w*)

Constructs a donut footprint matrix.

Parameters

- **p** (*int*) – The inner radius of the donut.
- **w** (*int*) – The outer radius of the donut.

Returns Square matrix containing 1s in the positions selected by the footprint and 0s in the positions ignored by the footprint.

Return type np.ndarray with int dtype

Notes

This function is duplicated in lib5c.algorithms.donut_filters.donut_filt() to preserve easy synchronization of that inherited module.

Examples

```
>>> donut_footprint(1, 3)
array([[1, 1, 1, 0, 1, 1, 1],
       [1, 1, 1, 0, 1, 1, 1],
       [1, 1, 0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 1, 1],
       [1, 1, 1, 0, 1, 1, 1],
       [1, 1, 1, 0, 1, 1, 1]])
```

lib5c.util.donut.**lower_left_footprint**(*p*, *w*)

Creates a lower left donut filter footprint.

Parameters

- **p** (*int*) – Inner radius of donut.
- **w** (*int*) – Outer radius of donut.

Returns The desired footprint. Square array with shape *(2*w+1, 2*w+1)*.

Return type np.ndarray

lib5c.util.donut.**make_donut_selector**(*i*, *j*, *p*, *w*, *n*)

Creates a boolean selector array with a donut shape around a particular coordinate to be used to index into a square matrix of known size.

Parameters

- **i** (*int*) – The row index of the point around which the donut should be constructed.
- **j** (*int*) – The column index of the point around which the donut should be constructed.
- **p** (*int*) – The inner radius of the donut.
- **w** (*int*) – The outer radius of the donut.
- **n** (*int*) – The size of the target matrix.

Returns The donut selector. It will have boolean dtype and shape (n, n).

Return type np.ndarray

Examples

```
>>> make_donut_selector(5, 4, 1, 3, 10).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 0, 1, 1, 1, 0, 0],
       [0, 1, 1, 0, 0, 0, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 0, 0, 1, 1, 0, 0],
       [0, 1, 1, 1, 0, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> make_donut_selector(5, 1, 1, 3, 10).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> make_donut_selector(5, 8, 1, 3, 10).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> make_donut_selector(1, 5, 1, 3, 10).astype(int)
array([[0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
       [0, 0, 1, 1, 1, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> make_donut_selector(8, 5, 1, 3, 10).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 1, 1, 0]])
```

lib5c.util.donut.**pad_and_crop**(*x*, *margin*, *limit*)

Utility function to assist in padding and cropping footprints.

Parameters

- **x** (*int*) – The coordinate of the point in question.
- **margin** (*int*) – The size of a buffer around x which should not be included in the padding. In the context of padding and cropping footprints, this is the outer radius of the footprint. Whatever part of the margin extends beyond the limit must be cropped away.
- **limit** (*int*) – The coordinate of the edge (extreme allowed index) of the desired final matrix.

Returns The first int is the number of pixels to pad by, the second is the number of pixels that must be cropped away after padding.

Return type int, int

Examples

```
>>> pad_and_crop(0, 3, 0)
(0, 3)
>>> pad_and_crop(5, 3, 5)
(0, 3)
>>> pad_and_crop(3, 3, 0)
(0, 0)
>>> pad_and_crop(3, 2, 5)
(0, 0)
>>> pad_and_crop(5, 3, 0)
(2, 0)
>>> pad_and_crop(3, 3, 10)
(4, 0)
```

lib5c.util.grouping module

Module for constructing groups of points with particular properties.

lib5c.util.grouping.**group_obs_by_exp**(*obs*, *exp*, *num_groups=100*, *group_fractional_tolerance=0.1*, *log=True*, *min_group_count=2*, *exclude_offdiagonals=5*)

Groups observed points according to their expected values.

Parameters

- **obs** (*np.ndarray or dict of np.ndarray*) – Vector, matrix, or counts dict of observed values.
- **exp** (*np.ndarray or dict of np.ndarray*) – Vector, matrix, or counts dict of expected values.
- **num_groups** (*int*) – The number of groups to make.
- **group_fractional_tolerance** (*float*) – The width of each group, specified as a fractional tolerance in the expected value.
- **log** (*bool*) – Pass True to space the groups out logarithmically.
- **min_group_count** (*int*) – Discard groups that have fewer than this many values in them.
- **exclude_offdiagonals** (*int*) – If *obs* and *exp* are not already vectors, discard this many off-diagonals from their square matrices before flattening. Pass 0 to exclude only the exact diagonal, and pass -1 to exclude nothing.

Returns The first array contains the expected values chosen as the centers of the groups. The list contains each group as an array of observed values.

Return type np.ndarray, list of np.ndarray

lib5c.util.lowess module

Module for performing lowess fitting. Consists mostly of a convenience wrapper around `statsmodels.nonparametric.smoothers_lowess.lowess()`.

lib5c.util.lowess.**constant_fit**(*x*, *y*, *logx=False*, *logy=False*, *agg='median'*)

Same signature as `lowess_fit()` and `group_fit()`, but instead of fitting `y` against `x`, simply applies an aggregating function to `y`.

Parameters

- **x** (*Any*) – Ignored, present only for signature parity with other fitters.
- **y** (*np.ndarray*) – The y values to fit.
- **logx** (*Any*) – Ignored, present only for signature parity with other fitters.
- **logy** (*bool*) – Pass True to perform the fit on the scale of `log(y)`.
- **agg** (*{'median', 'mean', 'lowess'}*) – The function to use to aggregate y-values.

Returns This function takes in `x` values, ignores them completely, and simply returns the constant estimated `y` value on the original `y` scale (regardless of what is passed for `logy`).

Return type function

lib5c.util.lowess.**group_fit**(*x*, *y*, *logx=False*, *logy=False*, *agg='median'*, *left_boundary=None*, *right_boundary=None*, *n_windows=100*, *window_width=0.2*)

Simpler alternative to lowess fitting using a sliding window mean.

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.
- **agg** (*{'median', 'mean', 'lowess'}*) – The function to use to aggregate within groups.
- **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.
- **n_windows** (*int*) – The number of windows to use (spaced uniformly across the range of `x`).
- **window_width** (*float*) – The width of each window, defined as a fraction of its x-value.

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

lib5c.util.lowess.**lowess_agg**(*y*, *it=3*)

Performs an aggregation operation equivalent to lowess. Should behave like an outlier-resistant mean.

Parameters

- **y** (*np.ndarray*) – The values to aggregate.
- **it** (*int*) – The number of residual-based reweightings to perform.

Returns The lowess-implemented outlier-resistant mean.

Return type float

lib5c.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`.

lib5c.util.lru_cache module

Module containing a backport of Python 3.3's least recently used cache decorator.

The original module was downloaded from http://code.activestate.com/recipes/578078/, then modified to support hashing of counts dicts and annotationmaps.

`lib5c.util.lru_cache.`**`lru_cache`**(*maxsize=100*, *typed=False*)

Least-recently-used cache decorator.

If *maxsize* is set to None, the LRU features are disabled and the cache can grow without bound.

If *typed* is True, arguments of different types will be cached separately. For example, f(3.0) and f(3) will be treated as distinct calls with distinct results.

Arguments to the cached function must be hashable.

View the cache statistics named tuple (hits, misses, maxsize, currsize) with f.cache_info(). Clear the cache and statistics with f.cache_clear(). Access the underlying function with f.__wrapped__.

See: http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

lib5c.util.mathematics module

Module for mathematical utility functions.

`lib5c.util.mathematics.`**`gmean`**(*array*, *pseudocount=1*, *axis=None*)

Compute the geometric mean of an input array.

Parameters

- **array** (*np.ndarray*) – The array to take the geometric mean over.
- **pseudocount** (*float*) – The pseudocount to add to the elements of `array` before logging.
- **axis** (*int, optional*) – The axis to compute the mean over.

Returns The geometric mean.

Return type float

lib5c.util.mathematics.**symmetrize**(*array*, *source='lower'*)

Symmetrizes a square array using its lower triangular entries.

Parameters

- **array** (*np.ndarray*) – Array to symmetrize. Must be square.
- **source** (*{'lower', 'upper'}*) – What triangle of the matrix to symmetrize with.

Returns A symmetrized copy of `array`.

Return type np.ndarray

lib5c.util.mathematics.**zero_nans**(*array*)

Zeros all the nan's in an array. Useful for cases where functions like *np.nansum()* are not available (e.g., *scipy.ndimage.convolve()*).

Parameters **array** (*np.ndarray*) – The array to zero nan's in.

Returns Copy of *array* with all nan's set to zero.

Return type np.ndarray

lib5c.util.metrics module

Module containing utility functions used for computing scoring metrics.

lib5c.util.metrics.**cohens_kappa**(*y1*, *y2*)

Computes Cohen's kappa score for the agreement between two classifications.

Implementation taken from *sklearn.metrics.cohen_kappa_score()*.

Parameters **y2** (*y1,*) – The two classifications.

Returns The kappa.

Return type float

lib5c.util.optimization module

Module containing utility functions for curve optimization and root finding.

lib5c.util.optimization.**array_newton**(*func*, *x0*, *fprime=None*, *args=()*, *tol=1.48e-08*, *maxiter=50*, *fprime2=None*, *failure_idx_flag=None*)

This function is deprecated, with `scipy>=1.2.0`, you can call `scipy.optimize.newton()` instead.

Finds roots of a scalar function `func` given a vector of initial guesses `x0` and parallel vectors of additional arguments to `func` (passed in `args`) in a vectorized fashion. Similar to calling `scipy.optimize.newton()` in a for loop, but more performant and more legible. Bootlegged from https://github.com/scipy/scipy/pull/8357 preceeding its official release.

Parameters

- **func** (*function*) – The scalar function to minimize. Should be vectorized (when a vector of independent inputs is passed it should return a vector of independent outputs). Signature should be `func(x, *args)` where `x0` contains initial guesses for `x` and `args` represents the additional arguments.
- **x0** (*np.ndarray*) – Initial guesses.
- **fprime** (*function, optional*) – The derivative of `func`. If not passed, this will be estimated with the secant method.
- **args** (*tuple*) – Extra arguments to be passed to `func`.
- **tol** (*float*) – The allowable error of the zero value.
- **maxiter** (*int*) – Maximal number of iterations.
- **fprime2** (*function, optional*) – The second derivative of `func`. If passed, Halley's method will be used. If not passed, the normal Newton-Raphson method or the secant method is used.
- **failure_idx_flag** (*bool, optional*) – Pass True to return two extra boolean arrays specifying which optimizations failed or encountered zero derivatives, respectively.

Returns

- **root** (*np.ndarray*) – The identified zeros of `func`.
- **failures** (*np.ndarray of bool, optional*) – Only returned if `failure_idx_flag` is True. Indicates which elements failed to converge.
- **zero_der** (*np.ndarray of bool, optional*) – Only returned if `failure_idx_flag` is True. Indicates which elements had a zero derivative.

lib5c.util.optimization.**quadratic_log_log_fit**(*x*, *y*)

Fit a pure-quadratic function `y = a * x**2` using a loss function in log-log space.

Parameters

- **x** (*np.ndarray*) – Flat vector of `x` values to fit.
- **y** (*np.ndarray*) – Flat vector of `y` values to fit.

Returns The fitted function.

Return type np.poly1d

lib5c.util.parallelization module

Module providing utilities for parallelization of operations on 5C data.

The most important thing exposed in this module is the `@parallelize_regions` decorator, which automatically overloads any function to accept regional dicts of any of its arguments and process the regions in parallel via the `multiprocess` package.

The other functions in this module are either example functions to show how it works (`test_function_one`, etc.) or private helper functions.

lib5c.util.parallelization.**main**()

lib5c.util.parallelization.**test_function_four**(*x*, *y*)

lib5c.util.parallelization.**test_function_one**(*count*)

lib5c.util.parallelization.**test_function_three**(*x*, *y*)

lib5c.util.parallelization.**test_function_two**(*count*, *multiplier=4*)

lib5c.util.plotting module

Module containing utilities related to plotting.

lib5c.util.plotting.**adjust_plot**(*ax=None*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel=None*, *xticks=None*, *yticks=None*, *title=None*, *despine=True*, *legend=None*)

Multipurpose plot adjustment method.

Parameters

- **ax** (*pyplot axis*) – The axis to operate on.
- **xlim** (*tuple of numeric*) – Pass a tuple of the form (min, max) to set the x-limits of the plot.
- **ylim** (*tuple of numeric*) – Pass a tuple of the form (min, max) to set the y-limits of the plot.
- **xlabel** (*str*) – Label for the x-axis.
- **ylabel** (*str*) – Label for the y-axis.
- **xticks** (*int, list of int, or tuple of list of ints*) – Pass an int to use this as the spacing for the xticks. Pass a `(positions, labels)` tuple to call `plt.xticks(positions,labels)`. Pass anything else to pass it directly to `plt.xticks()`.
- **yticks** (*int, list of int, or tuple of list of ints*) – Pass an int to use this as the spacing for the yticks. Pass a `(positions, labels)` tuple to call `plt.yticks(positions,labels)`. Pass anything else to pass it directly to `plt.yticks()`.
- **title** (*str*) – Title for the plot.
- **despine** (*bool*) – Pass True to despine the plot.
- **legend** (*str or bool or None*) – Pass False to remove the legend, pass True to add a default legend, pass 'outside' to move the legend outside the plot area, pass None to leave the legend alone.

lib5c.util.plotting.**compute_hexbin_extent**(*xlim*, *ylim*, *logx=False*, *logy=False*)

Helper function for computing the `extent` kwarg of `plt.hexbin()`.

Parameters

- **ylim** (*xlim,*) – Tuple of *(x_min, x_max)* and *(y_min, y_max)*, respectively. If either is None, no attempt will be made to set the extent.
- **logy** (*logx,*) – Whether or not `plt.hexbin()` is being called with `xscale='log'` and/or `yscale='log'`, respectively.

Returns The extent if it could be computed or None otherwise.

Return type list or None

lib5c.util.pretty_decorator module

Module providing the `@pretty_decorator` meta-decorator.

lib5c.util.pretty_decorator.**pretty_decorator**(*dec*)

Decorator to turn any existing decorator into a "pretty" decorator.

A "pretty" decorator retains the signature information of the original function, improving the readability of `help(func)` and auto-generated documentation.

This functionality is completely cosmetic.

Requires the optional dependency `decorator` - if importing this fails, the decorator will not be modified.

Parameters **dec** (*function*) – The decorator to prettify.

Returns The pretty decorator.

Return type function

Examples

```
>>> from lib5c.util.pretty_decorator import pretty_decorator
>>> @pretty_decorator
... def dec1(func):
...     def wrapped_func(*args, **kwargs):
...         print("you've been decorated")
...         return func(*args, **kwargs)
...     return wrapped_func
>>> @dec1
... def my_func(a, b):
...     "Adds two numbers"
...     return a + b
>>> my_func(1, 2)
you've been decorated
3
>>> help(my_func)
Help on function my_func in module lib5c.util.pretty_decorator:

my_func(a, b)
    Adds two numbers
```

lib5c.util.primers module

Module containing utilities for manipulating 5C primer information.

lib5c.util.primers.**aggregate_primermap**(*primermap*, *region_order=None*)

Aggregates a primermap into a single list.

Parameters

- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – Primermap to aggregate. See `lib5c.parsers.primers.get_primermap()`.
- **region_order** (*Optional[List[str]]*) – Order in which regions should be concatenated. If None, the regions will be concatenated in order of increasing genomic coordinate. See `lib5c.util.primers.determine_region_order()`.

Returns The dicts represent primers in the same format as the inner dicts of the passed primermap; however, they exist as a single flat list instead of within an outer dict structure. The regions are arranged within this list in contiguous blocks, arranged in the order specified by the region_order kwarg.

Return type List[Dict[str, Any]]

Notes

This function returns a list of references to the original primermap, under the assumption that primer dicts are rarely modified. To avoid this, pass a copy of the primermap instead of the original primermap.

lib5c.util.primers.**determine_region_order**(*primermap*)
Orders regions in a primermap by genomic coordinate.

Parameters **primermap** (*Dict[str, List[Dict[str, Any]]]*) – Primermap containing information about the regions to be ordered. See `lib5c.parsers.primers.get_primermap()`.

Returns List of ordered region names.

Return type List[str]

lib5c.util.primers.**guess_bin_step**(*regional_pixelmap*)
Guesses the bin step from a regional pixelmap.

Parameters **regional_pixelmap** (*List[Dict[str, Any]]*) – Ordered list of bins for a single region.

Returns The guessed bin step for this pixelmap.

Return type int

lib5c.util.primers.**natural_sort_key**(*s*)
Function to enable natural sorting of alphanumeric strings.

Parameters **s** (*str*) – String being sorted.

Returns This list is an alternative represenation of the input string that will sort in natural order.

Return type List[Union[int, str]]

Notes

Function written by SO user http://stackoverflow.com/users/15055/claudiu and provided in answer http://stackoverflow.com/a/16090640.

lib5c.util.sampling module

Module providing utilities related to sampling.

lib5c.util.sampling.**uniform_range_coverage_sample**(*data*, *n_points*, *log_space=False*)
Performs a deterministic "sampling" step that draws the requested number of evenly-spaced samples from the data.

Even-spacing can be defined in terms of the native space of the data, or in terms of log space.

The marginal distribution of the drawn samples should be approximately uniform - it will not match the empirical distribution of `data`.

Parameters

- **data** (*np.ndarray*) – The data to sample from.
- **n_points** (*int*) – The number of points to sample

- **log_space** (*bool*) – Pass True to space the points evenly in log space, otherwise the points will be spaced evenly in the native space of `data`.

lib5c.util.scales module

Module for determining various scales for visualization purposes.

`lib5c.util.scales.`**compute_regional_obs_over_exp_scale**(*counts_superdict*, *region*)

Computes a typical scale for visualizing observed over expected counts for a specified region.

Parameters

- **counts_superdict** (*dict of dicts of 2d numpy arrays*) – The keys of the outer dict are replicate names. The values are dicts corresponding to counts dicts (see `lib5c.parsers.counts.load_counts()`), whose keys are region names and whose values are the arrays of counts values for that region. These arrays are square and symmetric.
- **region** (*str*) – The region for which the scale should be computed.

Returns The first element of this list is the minimum value of the computed scale. The second element of this list is the maximum value of the computed scale.

Return type list of float

Notes

The returned scale is computed as the mean of the observed over expected counts for the selected region across all replicates, plus and minus two and a half times the mean of the standard deviations for the selected region across all replicates for the maximum and the minimum, respectively.

`lib5c.util.scales.`**compute_regional_obs_scale**(*counts_superdict*, *region*, *top_percentile=98*)

Computes a typical scale for visualizing observed for a specified region.

Parameters

- **counts_superdict** (*dict of dicts of 2d numpy arrays*) – The keys of the outer dict are replicate names. The values are dicts corresponding to counts dicts (see `lib5c.parsers.counts.load_counts()`), whose keys are region names and whose values are the arrays of counts values for that region. These arrays are square and symmetric.
- **region** (*str*) – The region for which the scale should be computed.
- **top_percentile** (*int*) – The upper percentile to use when determinig the max of the scale.

Returns The first element of this list is the minimum value of the computed scale. The second element of this list is the maximum value of the computed scale.

Return type list of float

Notes

The returned scale is computed as ranging from 0 to the average of the 98th percentiles across the replicates.

lib5c.util.scales.**compute_track_scales**(*tracks*, *pixelmap*, *conditions=()*, *filename_generator=<function <lambda>>*)

Computes regional zero-to-max scales for visualizing ChIP-seq tracks.

Parameters

- **tracks** (*list of str*) – List of string identifiers for the tracks to compute scales for. The identifiers will be used to find the appropriate BED files on the disk according to filename_generator.
- **pixelmap** (*pixelmap*) – The pixelmap to use when identifying the region names and boundaries. See lib5c.parsers.bed.get_pixelmap().
- **conditions** (*list of str*) – List of string identifiers for the conditions. If this kwarg is passed, the tracks will be grouped by condition, and tracks that differ only in the condition identifier will be assigned the same scale. If this list is empty (as it is by default), no such grouping of tracks is performed.
- **filename_generator** (*function str -> str*) – When passed any string from tracks, this function should return a string reference to the BED file on the disk containing the BED features for that track.

Returns The keys of the outer dict are the elements of tracks. The values are dicts whose keys are region names and whose values are the maximum values within that region for that track.

Return type dict of dict of numeric

Notes

Since this function computes a zero-to-max scale, the minimum of the scale is always implicitly taken to be zero.

When conditions is not empty, the tracks are grouped by removing all instances of all condition identifiers from the track identifiers and then comparing them for equality. When the sequence of characters defined by a condition identifier appears elsewhere in the track name in a location not intended to identify the condition of the track, this may lead to failure to correctly group tracks by condition.

lib5c.util.scales.**main**()

lib5c.util.slicing module

Module containing utility functions related to matrix and primermap slicing.

lib5c.util.slicing.**convert_grange_to_slice**(*grange*, *regional_primermap*)

Finds the slice of a regional primermap/pixelmap that covers a given grange.

Parameters

- **grange** (*dict*) – The genomic range the slice should cover. Should have keys 'chrom', 'start', 'end'.
- **regional_primermap** (*list of dict*) – The primermap whose indices should make up the slice.

Returns The slice.

Return type slice

Examples

```
>>> regional_primermap = [{'chrom': 'chr1', 'start': 100, 'end': 200},
...                       {'chrom': 'chr1', 'start': 300, 'end': 400},
...                       {'chrom': 'chr1', 'start': 500, 'end': 600}]
>>> grange = {'chrom': 'chr1', 'start': 350, 'end': 550}
>>> convert_grange_to_slice(grange, regional_primermap)
slice(1, 3, None)
>>> grange = {'chrom': 'chr1', 'start': 300, 'end': 600}
>>> convert_grange_to_slice(grange, regional_primermap)
slice(1, 3, None)
```

lib5c.util.slicing.**convert_slice_to_grange**(*s*, *regional_pixelmap*)

Converts an index-based slice into a dict describing the genomic range covered by the slice.

Parameters

- **s** (*slice*) – The slice.
- **regional_pixelmap** (*list of dict*) – The pixelmap or primermap being indexed into by the slice.

Returns The genomic range covered by the slice, as a dict with 'chrom', 'start', and 'end' keys.

Return type dict

Examples

```
>>> regional_pixelmap = [{'chrom': 'chr1', 'start': 100, 'end': 200},
...                      {'chrom': 'chr1', 'start': 300, 'end': 400},
...                      {'chrom': 'chr1', 'start': 500, 'end': 600}]
>>> (convert_slice_to_grange(slice(1, 3), regional_pixelmap) ==
...  {'chrom': 'chr1', 'start': 300, 'end': 600})
True
```

lib5c.util.slicing.**slice_matrix_by_grange**(*matrix*, *regional_primermap_x*, *grange_x*, *grange_y=None*, *regional_primermap_y=None*)

Convenience function for slicing matrices.

Parameters

- **matrix** (*np.ndarray*) – The matrix to slice.
- **regional_primermap_x** (*list of dict*) – The primermap describing the column indices of `matrix`.
- **grange_x** (*dict*) – The genomic range to slice the x-axis of the matrix with (columns).
- **grange_y** (*dict, optional*) – The genomic range to slice the y-axis of the matrix with (rows). Pass None to assume that the two ranges are equal.
- **regional_primermap_y** (*list of dict, optional*) – The primermap describing the row indices of `matrix`. If `matrix` is symmetric, pass None.

Returns The np.ndarray is the sliced matrix. The two dicts are the x- and y-axis ranges actually represented by the matrix slice.

Return type np.ndarray, dict, dict

lib5c.util.sorting module

Module providing utility functions related to sorting data.

lib5c.util.sorting.**rankdata_plus**(*a*, *method='average'*)

Assign ranks to data, dealing with ties appropriately.

Slight modification of `scipy.stats.rankdata()` that returns the sorter in addition to the ranks; this allows the sort information to be re-used without having to sort again.

Ranks begin at 1. The *method* argument controls how ranks are assigned to equal values. See[1] for further discussion of ranking methods.

Parameters

- **a** (*array_like*) – The array of values to be ranked. The array is first flattened.
- **method** (*str, optional*) – The method used to assign ranks to tied elements. The options are 'average', 'min', 'max', 'dense' and 'ordinal'.

  'average': The average of the ranks that would have been assigned to all the tied values is assigned to each value.

  'min': The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)

  'max': The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.

  'dense': Like 'min', but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.

  'ordinal': All values are given a distinct rank, corresponding to the order that the values occur in *a*.

  The default is 'average'.

Returns **ranks, sorter** – Ranks is an array of length equal to the size of *a*, containing rank scores. Sorter is the argsort result from the initial sorting.

Return type ndarray, ndarray

References

Examples

```
>>> from lib5c.util.sorting import rankdata_plus
>>> np.array_equal(rankdata_plus([0, 2, 3, 2])[1],  # the sorter
...                np.array([0, 1, 3, 2]))
True
>>> np.array_equal(rankdata_plus([0, 2, 3, 2])[0],  # the ranks
...                np.array([1., 2.5, 4., 2.5]))
True
>>> np.array_equal(rankdata_plus([0, 2, 3, 2], 'min')[0],  # the ranks
...                np.array([1, 2, 4, 2]))
True
>>> np.array_equal(rankdata_plus([0, 2, 3, 2], 'max')[0],  # the ranks
...                np.array([1, 3, 4, 3]))
```

(continues on next page)

[1] "Ranking", http://en.wikipedia.org/wiki/Ranking

(continued from previous page)

```
True
>>> np.array_equal(rankdata_plus([0, 2, 3, 2], 'dense')[0],  # the ranks
...                np.array([1, 2, 3, 2]))
True
>>> np.array_equal(rankdata_plus([0, 2, 3, 2], 'ordinal')[0],  # the ranks
...                np.array([1, 2, 4, 3]))
True
```

lib5c.util.statistics module

Module containing utility functions related to statistical transformation, correction, or combination of p-values.

lib5c.util.statistics.**adjust_pvalues**(*pvalue_matrix*, *method='fdr_bh'*)

Performs multiple testing correction on a matrix of p-values.

Parameters

- **pvalue_matrix** (*np.ndarray*) – The matrix of p-values, representing many tests that were performed.
- **method** (*str*) – Method used for testing and adjustment of pvalues. Can be either the full name or initial letters. Available methods are

```
`bonferroni` : one-step correction
`sidak` : one-step correction
`holm-sidak` : step down method using Sidak adjustments
`holm` : step-down method using Bonferroni adjustments
`simes-hochberg` : step-up method  (independent)
`hommel` : closed method based on Simes tests (non-negative)
`fdr_bh` : Benjamini/Hochberg  (non-negative)
`fdr_by` : Benjamini/Yekutieli (negative)
`fdr_tsbh` : two stage fdr correction (non-negative)
`fdr_tsbky` : two stage fdr correction (non-negative)
```

Returns The matrix of adjusted p-values. Same size and shape as pvalue_matrix.

Return type np.ndarray

lib5c.util.statistics.**convert_to_two_tail**(*pvalue*)

Converts a one-tail p-value to a two-tail p-value.

Only valid for continuous distributions (otherwise I think the PMF evaluation at the obs value needs to be taken into account when reversing the test).

Array-safe, nan-safe, position-independent.

Parameters **pvalue** (*float*) – The one-tail p-values to be folded into two-tail pvalues.

Returns The two-tail p-values.

Return type float

lib5c.util.statistics.**stouffer**(*pvalue_matrix*, *axis=None*)

Combines a matrix of p-values along a specified axis using Stouffer's Z-transform.

Parameters

- **pvalue_matrix** (*np.ndarray*) – Matrix of p-values.

- **axis** (*int, optional*) – The axis to combine p-values along. Pass None to combine all of them.

Returns Array of combined p-values.

Return type np.ndarray

Notes

Element-by-element, this should be equivalent to:

```
scipy.stats.combine_pvalues(pvalue_matrix, method='stouffer')[1]
```

except that this function is vectorized.

Examples

```
>>> from scipy.stats import combine_pvalues
>>> stouffer([0.1, 0.1]) == combine_pvalues([0.1, 0.1],
...                                         method='stouffer')[1]
True
>>> stouffer(np.array([[0.1, 0.1],
...                    [0.2, 0.2]]), axis=1)
array([0.03496316, 0.11697758])
```

lib5c.util.stratification module

Module containing utility functions for stratifying data.

lib5c.util.stratification.**conservative_qcut**(*array*, *num_quantiles*, *add_zero=True*, *pad_right_endpoint=False*)

Similar to pd.qcut(), but designed for stratifying quantities with zero-inflation. All zeros get put into the first stratum, and then the rest of the data is qcut.

Parameters

- **array** (*np.ndarray*) – The data to stratify.
- **num_quantiles** (*int*) – How many strata to generate.
- **add_zero** (*bool*) – Pass True to include the zeros in the final stratification in their own bin (increasing the number of bins returned by 1). Pass False to exclude zeros from the stratification.
- **pad_right_endpoint** (*False*) – If the right endpoint of your last bin is interpreted as open, pass True here to extend this right endpoint by a small number so that the highest value is not excluded from the stratification.

Returns The binning scheme, as a list of n+1 bin endpoints. This is the format expected by pd.cut() or plt.hist().

Return type np.ndarray

lib5c.util.system module

Module containing utility functions to assist in system-specific compatibility.

lib5c.util.system.**check_outdir**(*outfile*)
Makes sure the directories needed to write an output file exist.

Parameters **outfile** (*str*) – Path to a hypothetical output file to be written to the disk.

lib5c.util.system.**shell_quote**(*string*)
Quote a string with the correct quotes (double quotes on Windows and single quotes otherwise) to prevent the shell from eating the quotes.

Useful for when assembling command line strings with arguments that must be quoted.

Parameters **string** (*str*) – The string to quote.

Returns The appropriately-quoted version of the input string.

Return type str

lib5c.util.system.**splitall**(*path*)
Recursively splits a path into a list of all its parts.

https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ ch04s16.html

Parameters **path** (*str*) – A path to split.

Returns The parts of the path.

Return type List[str]

lib5c.util.table module

Utility functions related to table files.

lib5c.util.table.**make_fflj_id_map**(*primermap*)
Creates a map from unique FFLJ or bin-bin pair IDs to (region, row, column) coordinate tuples.

Parameters **primermap** (*primermap or pixelmap*) – Defines the fragments or bins that will get IDs assigned.

Returns The keys are FFLJ or bin-bin pair IDs as strings. The values are (region, row, column) coordinate tuples.

Return type Dict[str, Tuple]

Module contents

Subpackage containing miscellaneous utilities for 5C data analysis.

lib5c.writers package

Submodules

lib5c.writers.bedgraph module

Module for writing bedgraph files.

lib5c.writers.bedgraph.**main**()

lib5c.writers.bedgraph.**write_bedgraph**(*peaks*, *outfile*, *name=''*, *desc=''*)
Writes a set of peaks to a bedgraph file.

Parameters

- **peaks** (*dict of lists of dicts*) – The peaks that should be written. The keys are chromosome names. The values are lists of features for that chromosome. The features are represented as dicts with at least the following keys:

```
{
    'chrom': str
    'start': int,
    'end'  : int,
    'value': number
}
```

- **outfile** (*str*) – String reference to the file to write to.
- **name** (*str*) – String to write in the name field of the header line.
- **desc** (*str*) – String to write in the description field of the header line.

lib5c.writers.bias module

Module for writing bias vector files.

lib5c.writers.bias.**write_cis_bias_vector**(*bias*, *primermap*, *outfile*, *region_order=None*)
Writes a dict of bias vectors to a file.

Parameters

- **bias** (*Dict[str, np.ndarray]*) – The keys are region names as strings, the values are the one-dimensional bias vector for that region.
- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap or pixelmap that describes the loci whose bias factors are contained in the bias vectors.
- **outfile** (*str*) – A string reference to a file to write the bias vector to.
- **region_order** (*Optional[List[str]]*) – Pass a list of region names as strings to force the regions to be written in that order. If this kwarg is not passed, the regions will be written in the order of `primermap.keys()`.

lib5c.writers.config module

Module for wrappers around ConfigParser for writing config files.

lib5c.writers.config.**write_config**(*outfile*, *name*, *data*)
Writes key-value data to a simple config file.

Parameters

- **outfile** (*str*) – File to write to.
- **name** (*str*) – Section name for the config.
- **data** (*dict*) – The data to write in the config.

lib5c.writers.correlation module

Module for writing pairwise correlation matrices to text files.

lib5c.writers.correlation.**write_correlation_table**(*correlation_matrix*, *outfile*, *labels=None*, *sep='\t'*)

Write a pairwise correlation matrix to a text file.

Parameters

- **correlation_matrix** (*np.ndarray*) – The pairwise correlation matrix to write.
- **outfile** (*str*) – String reference to the file to write to.
- **labels** (*Optional[List[str]]*) – Pass a list of strings equal to the number of rows/columns in the `correlation_matrix` to label the rows and columns in the output file with these labels in the headers.
- **sep** (*str*) – The separator to use to separate columns in the written text file.

lib5c.writers.counts module

Module for writing .counts files.

lib5c.writers.counts.**main**()

lib5c.writers.counts.**write_cis_trans_counts**(*counts*, *outfile*, *primermap*, *omit_zeros=True*)

Writes a counts file including both the cis and trans counts.

Parameters

- **counts** (*2d numpy array*) – The square, symmetric array of counts to be written. This must be a single matrix that contains the cis contacts for all regions as well as the trans contacts between them. The rows of the matrix must correspond to genomic loci in order of increasing genomic coordinate.
- **outfile** (*str*) – String reference to the file to write counts to.
- **primermap** (*dict of lists of dicts*) – The keys of the outer dict are region names. The values are lists, where the `i` th entry represents the `i` th primer or bin in that region. Primers or bins are represented as dicts with the following structure:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'orientation': "3'" or "5'"
}
```

  The strand key is optional and only makes sense when writing primer-primer interaction counts. If present, impossible primer-primer combinations will be omitted from the output. See `lib5c.parsers.primers.get_primermap()` or `lib5c.parsers.primers.get_pixelmap()`.

- **omit_zeros** (*bool*) – If True, lines will not be written to the outfile if the counts for that line are zero.

lib5c.writers.counts.**write_counts**(*counts*, *outfile*, *primermap*, *skip_zeros=False*)

Writes a standard counts file.

Parameters

- **counts** (*dict of 2d arrays*) – The counts to be written. The keys are the region names. The values are the arrays of counts values for that region. These arrays should be square and symmetric.
- **outfile** (*str*) – String reference to the file to write counts to.
- **primermap** (*dict of lists of dicts*) – The keys of the outer dict are region names. The values are lists, where the `i` th entry represents the `i` th primer or bin in that region. Primers or bins are represented as dicts with the following structure:

```
{
    'chrom' : str,
    'start' : int,
    'end'   : int,
    'name'  : str,
    'orientation': "3'" or "5'"
}
```

  The strand key is optional and only makes sense when writing primer-primer interaction counts. If present, impossible primer-primer combinations will be omitted from the output. See `lib5c.parsers.primers.get_primermap()` or `lib5c.parsers.primers.get_pixelmap()`.
- **skip_zeros** (*bool*) – Pass True to omit writing output lines for bin-bin pairs with zero, nan, or empty string value.

lib5c.writers.hic module

Module for writing contact matrices to text files formatted in the style of Rao et al.'s GEO submission.

`lib5c.writers.hic.`**write_rao_matrix**(*matrix*, *resolution*, *outfile*)

Function for writing matrices to text files formatted in the style of Rao et al.'s GEO submission.

See https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE63525 for more details.

Parameters

- **matrix** (*np.ndarray*) – The matrix to write to file.
- **resolution** (*int*) – The resolution of the matrix, in base pairs.
- **outfile** (*str*) – String reference to the file to write to.

Notes

To write matrices for all chromosomes in one call, try:

write_rao_matrix(counts_dict, 40000, {chrom: 'outdir/%s.matrix' % chrom for chrom in counts_dict})

Examples

```
>>> import numpy as np
>>> from lib5c.writers.hic import write_rao_matrix
>>> from lib5c.parsers.hic import load_range_from_contact_matrix
```

(continues on next page)

(continued from previous page)

```
>>> matrix = np.arange(16).reshape((4, 4))
>>> matrix += matrix.T
>>> write_rao_matrix(matrix, 10000, 'test/rao_matrix_written.matrix')
>>> grange = {'chrom': 'chr1', 'start': 0, 'end': 40000}
>>> parsed_matrix, _ = load_range_from_contact_matrix(
...     'test/rao_matrix_written.matrix', grange)
>>> np.all(matrix == parsed_matrix)
True
```

lib5c.writers.pca module

Module for writing the results of principle component analyses to disk.

lib5c.writers.pca.**write_pca**(*outfile*, *proj*, *rep_names=None*, *pve=None*)
Write PCA projections to a csv file.

Parameters

- **outfile** (*str*) – String reference to the file to write to.
- **proj** (*np.ndarray*) – The PCA projections to plot
- **rep_names** (*Optional[List[str]]*) – The replicate names in the order of the rows of `proj`, to be used to write the header line.
- **pve** (*Optional[List[float]]*) – The percent variance explained for each component, to be included in the footer.

lib5c.writers.primers module

Module for writing primermaps and pixelmaps to bedfiles.

lib5c.writers.primers.**write_pixelmap_legacy**(*pixelmap*, *outfile*)
Write a pixelmap to a bin bedfile.

Parameters

- **pixelmap** (*Dict[str, List[Dict[str, Any]]]*) – The pixelmap to write. See `lib5c.parsers.primers.get_pixelmap()`.
- **outfile** (*str*) – String reference to the file to write to.

lib5c.writers.primers.**write_primermap**(*primermap*, *outfile*, *extra_column_names=None*)
Write a primermap to a primer bedfile.

Parameters

- **primermap** (*Dict[str, List[Dict[str, Any]]]*) – The primermap to write. See `lib5c.parsers.primers.load_primermap()`.
- **outfile** (*str*) – String reference to the file to write to.
- **extra_column_names** (*Optional[List[str]]*) – Names of additional columns to include in the bedfile.

lib5c.writers.table module

Module for writing table files.

lib5c.writers.table.**write_table**(*filename*, *counts_superdict*, *primermap*, *sep='\t'*)
Writes a counts_superdict structure as a single table file.

Parameters

- **filename** (*str*) – The filename to write to.
- **counts_superdict** (*counts_superdict*) – The counts_superdict to write.
- **primermap** (*primermap or pixelmap*) – Defines the FFLJs or bin-bin pairs.
- **sep** (*str*) – The separator to use when writing the table file.

lib5c.writers.wustl module

Module for writing WashU Epigenome Browser-style interaction data files.

lib5c.writers.wustl.**main**()

lib5c.writers.wustl.**write_wustl**(*counts*, *outfile*, *pixelmap*)
Writes a WashU Epigenome Browser-style interaction data file.

Parameters

- **counts** (*dict of 2d arrays*) – The counts to be written. The keys are the region names. The values are the arrays of counts values for that region. These arrays should be square and symmetric.
- **outfile** (*str*) – String reference to the file to write counts to.
- **pixelmap** (*dict of lists of dicts*) – The keys of the outer dict are region names. The values are lists, where the `i` th entry represents the `i` th bin in that region. Bins are represented as dicts with the following structure:

```
{
    'chrom': str,
    'start': int,
    'end'  : int,
    'name' : str
}
```

See `lib5c.parsers.get_pixelmap()`.

Module contents

Subpackage containing functions for writing output files to disk.

6.1.2 Module contents

A library for 5C data analysis.

Subpackage structure:

- *lib5c.algorithms* - main algorithms for analysis

- *lib5c.contrib* - integrations with third-party packages
- *lib5c.parsers* - file parsing
- *lib5c.plotters* - data visualization
- *lib5c.tools* - command line interface for `lib5c`
- *lib5c.util* - various utility functions
- *lib5c.writers* - file writing

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

W

X

Y

Z