
chromosight Documentation

Release 1.6.3

Axel Cournac, Lyam Baudry, Cyril Matthey-Doret, Nadège Guielgu

Oct 05, 2022

Contents:

1	Tutorial	3
1.1	Tutorial	3
2	Demo	7
2.1	Example use of <code>chromosight</code> <code>quantify</code>	7
2.2	Plotting <code>chromosight</code> 's output	14
3	Reference API	19
3.1	<code>chromosight</code>	19
4	Indices and tables	41
	Python Module Index	43
	Index	45

Chromosight is a command line tool that uses computer vision to detect patterns on Hi-C chromosomal contact maps. It also exposes a python API for interoperability with other python packages. You can follow chromosight development on [Github](#).

1.1 Tutorial

The simplest way to run chromosight without any input data is to use:

```
chromosight test
```

Which will download a test dataset and run chromosight on it. This is useful to have a look at the output files.

1.1.1 Detection

`chromosight detect` takes input in the form of Hi-C matrices in cool format. This command allows to detect patterns on Hi-C maps, such as chromatin loops or domain (e.g. TADS) borders, and report their coordinates.

The following command line can be used to run loop detection (the default pattern):

```
chromosight detect -t12 sample1.cool results/sample1_loops
```

The program will run in parallel on 12 threads and write loop coordinates and their pattern matching scores in a file named `sample1_loops.tsv` inside the `results` folder. Those scores represent pearson correlation coefficients (i.e. between -1 and 1) between the loop kernel and the detected pattern. Similarly, to run domain borders detection, one can use:

```
chromosight detect --pattern borders -t12 sample1.cool results/sample1_borders
```

Which will write the coordinates and scores of borders in `results/sample1_borders.tsv`.

At this point, the `results` folder will also contain files `sample1_loops.json` and `sample1_borders.json`, which contain images of the matrix regions around each detected loop or border, respectively. These files are in JSON format, which can be natively loaded in most programming languages.

Chromosight has several command line options which can affect the output format or filter the results based on different criteria. All parameters have sane default values defined for each pattern, which are printed during the run, but these

can be overridden using command line options to optimize results if needed. The list of command line options can be shown using:

```
chromosight --help
```

1.1.2 Quantification

The `chromosight quantify` command can be used to assign a pattern matching score to a set of 2D coordinates for an input Hi-C matrix. It will accept coordinates in bed2d format (tab-separated text file with 6 columns without headers, where columns denote chrom1, start1, end1, chrom2, start2, end2), or the output coordinates file `chromosight detect`. This can be useful to score the same set of coordinates on multiple Hi-C libraries, for instance.

For example, to compute loop scores for the positions detected in `sample1.cool` for a second sample, one could use:

```
chromosight quantify results/sample1_loops.tsv sample2.cool results/sample2_loops
```

Similarly, for borders:

```
chromosight quantify --pattern=borders results/sample1_borders.tsv sample2.cool ↵  
↵results/sample2_borders
```

These commands will each generate two files in the `results` directory, named `sample2_loops.tsv` and `sample2_loops.json` for the first command, and `sample2_borders.tsv` and `sample2_borders.json` for the second. Those files have the same format as the output from `chromosight detect`.

`chromosight quantify` can also be useful to compute pattern scores at ChIP-seq peaks, genes, or other features of interest. [BEDtools](#) can be used to generate a 2D bed file from an input bed file.

Assuming we have a BED file of cohesin peaks, all 2-way combinations of peaks at distances between 20kb and 1Mb can be retrieved with the following command:

```
MINDIST=20000  
MAXDIST=1000000  
bedtools window -a cohesin_peaks.bed -b cohesin_peaks.bed -w $MAXDIST \  
| awk -vmdist=$MINDIST '$1 == $4 && ($5 - $2) >= mdist {print}' \  
| sort -k1,1 -k2,2n -k4,4 -k5,5n \  
> cohesin_combinations_20kb_1Mb.bed2d
```

To quantify a pattern present only on the diagonal (e.g. borders, hairpins), the following command can be used instead.

```
paste cohesin_peaks.bed cohesin_peaks.bed > cohesin_combinations_0.bed2d
```

1.1.3 Generating custom patterns

More advanced users with specific questions or problems may wish to create new patterns and configurations. Both `detect` and `quantify` will accept custom patterns through the `--kernel-config` option. In order to provide a custom pattern, the user needs 2 files:

- A JSON file containing default values for the different detection parameters.
- One or more text files containing the pattern kernel(s) (i.e. matrix) in the form of a dense numeric matrix.

A template configuration can be generated using `chromosight generate-config`. A preset on which the template will be based can be selected, `loops` being the default preset. For example, to generate a template config based on the borders pattern, the following command can be used:


```
chromosight generate-config --preset borders demo_pattern
```

This will generate a JSON file named `demo_pattern.json`, pre-filled with parameter values from the `borders` pattern. This JSON file will have the following contents:

```
{
  "name": "borders",
  "kernels": [
    "demo_pattern.1.txt",
    "demo_pattern.2.txt",
    "demo_pattern.3.txt"
  ],
  "max_dist": 1,
  "min_dist": 0,
  "max_iterations": 3,
  "max_perc_undetected": 30.0,
  "min_separation": 5000,
  "pearson": 0.3,
  "resolution": 5000
}
```

The user can edit the configuration parameters in a text editor. Notably, the `kernels` entry points to 3 files, `demo_pattern.[1-3].txt`, which have also been created by `chromosight generate-config`. Those 3 paths are relative to the config, which means the kernel files have to be in the same folder as the JSON config.

When given a config with multiple kernels, `chromosight detect` will scan the matrix once for each kernel and return the union of all detected coordinates for the different kernels. This is useful when a pattern is asymmetric and can be flipped in different orientations, for example.

Kernels matrices are text files and can be edited using external program, or alternatively, the user can use the `--click` option from `generate-config` in order to manually build the kernel by double-clicking on relevant regions in a Hi-C matrix.

Note: The `--click` option will consume lots of RAM as it visualises the entire Hi-C matrix and should be reserved for small or subsetted contact maps.

For example:

```
chromosight generate-config --click sample1.cool --win-size 15 demo_manual
```

This command will generate a config file based on the loops template (the default) and will display the contact map `sample1.cool`. Every time the user double-clicks on a pixel, a window of 15x15 pixels centered on that position is recorded. The operation can be repeated as many times as the user wishes, and when the window is closed, all windows are averaged, a slight gaussian blur is added to reduce the impact of random noise, and the resulting pileup is used as the kernel when writing the config files.

1.1.4 A note on borders and kernels

One constraint in `chromosight` is that kernels must have an odd number of rows/columns. This is because `chromosight` always reports the center pixel of each window when computing correlations. For patterns which do not have a central pixel, such as borders which are between two pixels, a choice has to be made when making the kernel. In the case of borders, the kernel is shifted so that the central pixel is always the pixel on the right of the border.

2.1 Example use of `chromosight quantify`

In this notebook, we demonstrate how `chromosight quantify` can be used to compare chromatin loops between *S. cerevisiae* cultures arrested in G1 phase vs metaphase. In this notebook, we re-analyse Hi-C data from [Garcia-Luis, J., Lazar-Stefanita, L., Gutierrez-Escribano, P. et al., 2019](#).

2.1.1 Input data:

Files used in this analysis are the output from `chromosight quantify`. Loop scores were computed on all 2-way combinations from a set of high confidence RAD21 binding sites separated by 10 to 50kb, on two Hi-C datasets at 2kb resolution: One with G1-arrested cells and the other with metaphase-arrested cells.

- `scer_w303_g1_2kb_SRR8769554.cool`: Hi-C matrix of cells stopped in G1 phase, at 2kb resolution. From [Dauban et al. 2020](#)
- `scer_w303_mitotic_2kb_merged.cool`: Hi-C matrix of metaphasic cells, at 2kb resolution. From [Garcia-Luis et al. 2019](#)
- `rad21.bed2d`: bed file containing all pairs of positions of RAD21 (cohesin) peaks in metaphasic *S. cerevisiae* separated by 10-50kb.

Note: see the end of this notebook for an explanation on how to generate a bed2d file from a ChIP-seq bed file.

2.1.2 Getting loop scores

Loop scores at all pairs of positions can be computed using `chromosight quantify`. However, to ensure scores are comparable, the number of contacts should be similar between matrices. When using cool files, cooler can be used for this operation:

```
$ cooler info input/scer_w303_mitotic_2kb_merged.cool | grep sum
"sum": 44048750

$ cooler info input/scer_w303_g1_2kb_SRR8769554.cool | grep sum
"sum": 5862820
```

The G1 matrix has around 5.8M contacts whereas the metaphase matrix has 44M. Fortunately, chromosight has a `--subsample` option, which can be used to bring both matrices to the same coverage before computing scores:

```
chromosight quantify --pattern loops \
  --subsample 5862820 \
  --win-fmt npy \
  scer_cohesin_peaks.bed2d \
  input/scer_w303_g1_2kb_SRR8769554.cool \
  quantify/rad21_g1

chromosight quantify --pattern loops \
  --subsample 5862820 \
  --win-fmt npy \
  input/scer_cohesin_peaks.bed2d \
  input/scer_w303_mitotic_2kb_merged.cool \
  quantify/rad21_metaphase
```

For each condition, `chromosight quantify` generates 2 files:

- A table containing the coordinates and pattern matching scores of all input coordinates.
- A numpy binary file containing a stack of images around the input coordinates. Those images are stored in the same order as the coordinates from the table.

```
quantify/
├── rad21_g1.npy
├── rad21_g1.tsv
├── rad21_metaphase.npy
└── rad21_metaphase.tsv
```

2.1.3 Analysing loop scores

We can now use python to load and compare results from chromosight quantify. Below are a series of analyses showing some examples of downstream processing that can be performed on chromosight results.

```
[203]: %config InlineBackend.figure_format = 'svg'
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import chromosight.kernels as ck
import scipy.stats as st
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

res = 2000
```

```
[204]: # Load images (vignettes) around RAD21 interactions coordinates
images_g = np.load('quantify/rad21_g1.npy')
```

(continues on next page)

(continued from previous page)

```

images_m = np.load('quantify/rad21_metaphase.npy')

# Load lists of RAD21 interactions coordinates with their loop scores
# Compute loop size (i.e. anchor distance) for each RAD21 combination
get_sizes = lambda df: np.abs(df.start2 - df.start1)
loops_g = pd.read_csv('quantify/rad21_g1.tsv', sep='\t')
loops_g['loop_size'] = get_sizes(loops_g)
loops_m = pd.read_csv('quantify/rad21_metaphase.tsv', sep='\t')
loops_m['loop_size'] = get_sizes(loops_m)

# Merge data from both conditions into a single table
loops_g['condition'] = 'g1'
loops_m['condition'] = 'metaphase'
loops_df = pd.concat([loops_g, loops_m]).reset_index(drop=True)
images = np.concatenate([images_g, images_m])

# Remove NaN scores (e.g. in repeated regions or overlap the matrix edge)
nan_mask = ~np.isnan(loops_df['score'])
loops_df = loops_df.loc[nan_mask, :]
images = images[nan_mask, :, :]

# The loop kernel can be loaded using chromosight.kernels.loops
kernel = np.array(ck.loops['kernels'][0])
pileup_kw = {'vmin': -1, 'vmax': 1, 'cmap': 'seismic'}

```

2.1.4 Peeking at the input coordinates

Images around RAD21 sites 2-way combinations extracted by chromosight can be viewed using numpy and matplotlib. Note there are series off overlapping and slightly shifted images. This is because of adjacent RAD21 sites which are closer in the genome than the size of the vignettes.

```

[193]: # Decide how many rows and columns of images to show
r, c = 5, 15
valid_imgs = np.where(~loops_g.score.isnull() & ~loops_g.score.isnull())[0]
fig, axes = plt.subplots(r, c, figsize=(12, 4), subplot_kw={'xticks':[], 'yticks':[]})
# Show each image as a greyscale vignette
for i, ax in zip(valid_imgs, axes.flat):
    img = images_g[i, :, :] # Showing examples from the end of the image stack (M_
    ↪phase)
    ax.imshow(img, cmap=plt.cm.gray_r, interpolation='nearest')
plt.suptitle("Intersection between RAD21 sites, G1 phase")

```

```

[193]: Text(0.5, 0.98, 'Intersection between RAD21 sites, G1 phase')

```

```

[194]: fig, axes = plt.subplots(r, c, figsize=(12, 4), subplot_kw={'xticks':[], 'yticks':[]})
first_m = np.where(loops_df.condition == 'metaphase')[0][0]
# Show each image as a greyscale vignette
for i, ax in zip(valid_imgs, axes.flat):
    img = images_m[i, :, :] # Showing examples from the end of the image stack (M_
    ↪phase)
    ax.imshow(img, cmap=plt.cm.gray_r, interpolation='nearest')
plt.suptitle("Intersection between RAD21 sites, Metaphase")

```

```

[194]: Text(0.5, 0.98, 'Intersection between RAD21 sites, Metaphase')

```

2.1.5 Comparing the distribution of scores

The distribution of chromosight scores (i.e. correlation coefficients with the loop kernel) can be compared between the 2 conditions, revealing that metaphasic cells tend to have stronger loops.

```
[196]: sns.violinplot(data=loops_df, x='condition', y='score')
plt.ylabel('chromosight loop score')
plt.title('Comparison of loop scores between G1 and metaphasic cells')
plt.axhline(0, c='grey')

[196]: <matplotlib.lines.Line2D at 0x7f53f4892a50>
```

2.1.6 Using different metrics

Chromosight scores loops using their pearson correlation with a “loop kernel” (see below). However, one might want to use another metric than chromosight’s score to rank loops. One such metric commonly used in the litterature is the “corner score”, which uses the contrast between the center of the image (C) and the corner (R).

```
[197]: import matplotlib.patches as patches
fig, axes = plt.subplots(1, 2, sharex=True, sharey=True)
axes[0].imshow(np.log(kernel), **pileup_kw)
axes[0].set_title("Chromosight's loop kernel")
axes[1].imshow(np.zeros((17, 17)))
center_rect = patches.Rectangle(
    (8-2, 8-2), 4, 4, linewidth=1, edgecolor='r', facecolor='r'
)
corner_rect = patches.Rectangle(
    (17-5, 0), 4, 4, linewidth=1, edgecolor='g', facecolor='g'
)
axes[1].annotate('C', (8, 8), color='w', weight='bold', fontsize=14, ha='center', va=
    ↪ 'center')
axes[1].annotate('R', (14, 2), color='w', weight='bold', fontsize=14, ha='center', va=
    ↪ 'center')
axes[1].add_patch(center_rect)
axes[1].add_patch(corner_rect)
axes[1].set_title("Corner score: C - R")

[197]: Text(0.5, 1.0, 'Corner score: C - R')
```

The function defined below could be used to compute the corner score. It computes the difference between the average of contacts in the center and top right corner. Using the top right corner is better to avoid contacts enrichments for due to the diagonal. This is a pretty intuitive metric tailored based on expectations we have about loops. Here, we define center and corner radii as 10% of the image radius. For our 17x17 images, this means both regions will be $2+1 = 3 \times 3$ pixels.

```
[198]: def corner_score(image, prop_radius=0.1):
    """
    Compute a loop intensity score from a pileup

    Parameters
    -----
```

(continues on next page)

(continued from previous page)

```

image : numpy.array of floats
        2D array representing the window around a pattern.
prop_radius : float
        Proportion of image radius used when selecting
        center and corner contacts.

Returns
-----
float :
        Corner score, defined as mean(center) - mean(corner).
"""
n, m = image.shape
center = int(prop_radius * n)
half_h = n // 2
half_w = m // 2
le = half_h - center
ri = half_h + center + 1
hi = half_w - center
lo = half_w + center + 1
center_mean = np.nanmean(image[hi:lo, le:ri])
top_right_mean = np.nanmean(image[:hi, ri:])
return center_mean - top_right_mean

```

This homemade corner score correlates well with chromosight's pearson score:

```

[199]: import scipy.stats as st
loops_df['corner_score'] = [corner_score(m) for m in images]
comp_df = loops_df.loc[
    ~np.isnan(loops_df.corner_score) & ~np.isnan(loops_df.score), :
]
sns.regplot(data=comp_df, x='corner_score', y='score')
plt.title(
    r'Correlation between chromosight and corner score, $\rho$: '
    f'{np.round(st.pearsonr(comp_df.corner_score, comp_df.score)[0], 2)}')
[199]: Text(0.5, 1.0, 'Correlation between chromosight and corner score, $\rho$: 0.71')

```

By computing the pileup (average) of all patterns separately for G1 and M conditions, we can visually appreciate the stronger loop signal in metaphasic cells (M) compared to G1. Computing the chromosight and corner score directly on those pileups shows that the chromosight score makes it easier to discriminate the two conditions. The [-1,1] range is also convenient to interpret results. Note that the chromosight score below is just the pearson coefficient of the pileup with the loop kernel.

```

[200]: centroid_g1 = np.apply_along_axis(np.nanmean, 0, images[loops_df.condition == 'g1'])
centroid_m = np.apply_along_axis(np.nanmean, 0, images[loops_df.condition ==
    ↪ 'metaphase'])

fig, ax = plt.subplots(1, 2)
ax[0].imshow(np.log(centroid_g1), **pileup_kw)
ax[0].set_title(
    f'G1 corner score: {corner_score(centroid_g1):.2f}\n'
    f'G1 chromosight score: {np.round(st.pearsonr(centroid_g1.flat, kernel.flat)[0],
    ↪ 2)}')
)
ax[1].imshow(np.log(centroid_m), **pileup_kw)

```

(continues on next page)

(continued from previous page)

```
ax[1].set_title(
    f'M corner score: {corner_score(centroid_m):.2f}\n'
    f'M chromosight score: {st.pearsonr(centroid_m.flat, kernel.flat)[0]:.2f}'
)
plt.show()
```

Instead of summarizing the 2 conditions using only pileups, we can compare the ability of both score to separate the G1 and metaphasic cells based on the distribution of all patterns. Note that both scores are z-transformed to make their ranges comparable.

```
[201]: corner = comp_df.drop('score', axis=1).rename(columns={'corner_score': 'score'})
corner['metric'] = 'corner score'
corner['score'] = st.zscore(corner['score'])
chromo = comp_df.drop('corner_score', axis=1)
chromo['metric'] = 'chromosight'
chromo['score'] = st.zscore(chromo['score'])
comp_scores = pd.concat([corner, chromo]).reset_index(drop=True)
sns.violinplot(data=comp_scores, x='metric', y='score', split=True, hue='condition',
               inner='quartile')
plt.ylabel('metric z-score')
plt.title('Discriminative power: chromosight vs corner score')

[201]: Text(0.5, 1.0, 'Discriminative power: chromosight vs corner score')
```

2.1.7 Comparison of loop footprints

For vizualisation purposes, each window can be summarized to a 1D band representing the sum of columns or rows. Here, we compute both the average of rows and columns, and use the element-wise average of both 1D vectors. This gives a good approximation of a ‘loop footprint’ and is convenient for visualisation.

Each image is centered to its mean to homogenize the overall contact counts in windows. This avoid having globally darker or lighter images and emphasizes relative contrasts within the images.

Bands are then sorted by loop size (i.e. distance between anchors) and plotted as a stack from shortest to longest distance interactions.

```
[ ]: # Center images by subtracting their mean
centered = images.copy()
for img in range(centered.shape[0]):
    centered[img] -= np.nanmean(centered[img])

# Summarise each image by taking the average of its row and col sums.
bands = (np.nansum(centered, axis=1) + np.nansum(centered, axis=2)) / 2

# Reorder bands by distance between anchors
sort_var = 'loop_size'
sorted_bands = bands[np.argsort(loops_df[sort_var]), :]
sorted_cond = loops_df.condition.iloc[np.argsort(loops_df[sort_var])]
sorted_centered = centered[np.argsort(loops_df[sort_var])]

# Define a subset to visualise (too many images so see them all at once)
#smallest_group = np.min(np.unique(sorted_cond, return_counts=True)[1])-1
#smallest_group = 500
```

(continues on next page)

(continued from previous page)

```
# Define saturation threshold for the colormaps
vmax_bands = np.percentile(bands, 99.9)
vmax_img = np.percentile(centered, 99)
```

```
[202]: fig, axes = plt.subplots(2, 2, figsize=(8, 10))
for i, cond in enumerate(['g1', 'metaphase']):
    axes[0, i].imshow(
        sorted_bands[sorted_cond == cond, :],
        cmap='afmhot_r',
        vmax=vmax_bands,
    )
    axes[0, i].set_title(cond)
    # Compute pileup by averaging all windows for each condition
    centroid = np.apply_along_axis(
        np.nanmean,
        0,
        images[loops_df.condition == cond],
    )
    axes[1, i].imshow(np.log(centroid), **pileup_kw)
    axes[0, i].set_aspect('auto')
    # The rest is just to improve figure aesthetics
    axes[0, i].set_xticks([])
    axes[1, i].set_yticks([])
    if i > 0:
        axes[0, i].set_yticks([])
    else:
        #axes[0, i].set_yticklabels([], ["10kb", "25kb", "50kb"])
        axes[0, i].set_yticks(
            [0, sorted_bands[sorted_cond == cond, :].shape[0]]
        )
        axes[0, i].set_yticklabels(
            ['10kb', '50kb'],
            minor=False,
            rotation=45
        )

    axes[1, i].set_xticks([0, centroid.shape[0] // 2, centroid.shape[0]])
    half_w = int((res * centroid.shape[0] // 2) / 1000)
    half_w_bp = int(half_w * res / 1000)
    axes[1, i].set_xticklabels([f"{-half_w_bp}kb", "0", f"{half_w_bp}kb"])
    #axes[1, i].set_title(f"corner score: {np.round(corner_score(centroid), 2)}")

axes[0, 0].set_ylabel('Distance between RAD21 sites')
axes[1, 0].set_ylabel('Loop pileups')
plt.suptitle(f'Loop bands for pairs of RAD21 sites')
#plt.savefig('figs/bands_pileup_prots.svg')
```

```
[202]: Text(0.5, 0.98, 'Loop bands for pairs of RAD21 sites')
```

2.1.8 Appendix: Generating a BED2D file

ChIP-seq peaks are often stored as BED files, containing genomic intervals where DNA-binding proteins are enriched. Such files can be used to generate a BED2D file for `chromosight quantify`. This is done by generating all possible 2-ways combinations of peaks that follow desired criteria. In the example below, we use `bedtools` and `awk` to

generate all intrachromosomal combinations where peaks are separated by more than 10kb and less than 50kb.

```
MINDIST=10000
MAXDIST=50000
bedtools window -a input/scer_cohesin_peaks.bed \
                -b input/scer_cohesin_peaks.bed \
                -w $MAXDIST \
    | awk -vmd=$MINDIST '$1 == $4 && ($5 - $2) >= md {print}' \
    | sort -k1,1 -k2,2n -k4,4 -k5,5n \
    > input/scer_cohesin_peaks.bed2d
```

2.2 Plotting chromosight's output

Chromosight generates tabular text files with loops coordinates and scores. This file can be loaded into your favorite scripting language for visualization. For the purpose of this demonstration, we show how to plot the contact maps with detected coordinates using python, pandas and cooler.

The data shown here was generated with the following commands:

```
chromosight detect data_test/example.cool -m8000 -M50000 -p0.35 detect/example_loops
chromosight detect data_test/example.cool --pattern borders detect/example_borders
chromosight detect data_test/example.cool --pattern hairpins detect/example_hairpins
```

Which will detect all loops of size 8-50kb in `example.cool` and filter those with a score above 0.35. The output files will be located in the `detect/` folder.

```
[35]: %config InlineBackend.figure_format = 'svg'
import re
import json
import numpy as np
import pandas as pd
import cooler
import matplotlib.pyplot as plt
import chromosight.utils.detection as cud

# Load detected patterns' tables
loops = pd.read_csv('detect/example_loops.tsv', sep='\t')
borders = pd.read_csv('detect/example_borders.tsv', sep='\t')
hairpins = pd.read_csv('detect/example_hairpins.tsv', sep='\t')

# Load Hi-C data in cool format
c = cooler.Cooler("../data_test/example.cool")
```

2.2.1 View the whole genome matrix

To plot the whole matrix with patterns, the matrix is extracted from the cool file and columns `bin1` and `bin2` are used. Those columns contain the genome-wide bin number of pattern coordinates, and matches the whole genome matrix. Plotting the whole genome is straightforward, but likely to take too much memory for larger genomes.

```
[36]: %matplotlib inline
# Plot the whole matrix
plt.figure(figsize=(10, 10))
mat = c.matrix(sparse=False, balance=True)[: ]
plt.imshow(mat ** 0.2, cmap='afmhot_r')
```

(continues on next page)

(continued from previous page)

```
plt.scatter(loops.bin2, loops.bin1, edgecolors='blue', facecolors='none', label='loops
↳')
plt.scatter(borders.bin2, borders.bin1, c='lightblue', label='borders')
plt.scatter(hairpins.bin2, hairpins.bin1, c='green', label='hairpins')
plt.legend()
plt.show()
```

2.2.2 View a matrix region

To reduce the amount of memory required, we can define a region of interest. The corresponding matrix region can be fetched from the cool file using cooler, and patterns falling within that region can be filtered using pandas. Since we want to overlay the patterns on top of the region matrix, the bin1 and bin2 columns should be adjusted to be relative to the region's start instead of the genome.

```
[ ]: def subset_region(df, region):
    """
    Given a pattern dataframe and UCSC region string, retrieve only patterns in that
    ↳region.
    """
    # Split the region string at each occurrence of - or : (yields 3 elements)
    chrom, start, end = re.split('[-:]', region)
    start, end = int(start), int(end)
    # Only keep patterns on the same chromosome as the region and
    # within the start-end interval
    subset = df.loc[
        (df.chrom1 == chrom) &
        (df.chrom2 == chrom) &
        (df.start1 >= start) &
        (df.start2 >= start) &
        (df.end1 < end) &
        (df.end2 < end), :
    ]
    return subset
```

```
[37]: # Select a region of interest
region = 'chr2:200000-300000'
mat = c.matrix(sparse=False, balance=True).fetch(region)

loops_sub = subset_region(loops, region)
borders_sub = subset_region(borders, region)
hairpins_sub = subset_region(hairpins, region)

# Make genome-based bin numbers relative to the region
for df in [loops_sub, borders_sub, hairpins_sub]:
    df.bin1 -= c.extent(region)[0]
    df.bin2 -= c.extent(region)[0]
```

```
[38]: %matplotlib inline
plt.figure(figsize=(7, 7))
plt.imshow(np.log10(mat), cmap='afmhot_r')
plt.scatter(loops_sub.bin2, loops_sub.bin1, edgecolors='blue', facecolors='none',
↳label='loops')
```

(continues on next page)

(continued from previous page)

```
plt.scatter(borders_sub.bin2, borders_sub.bin1, c='lightblue', label='borders')
plt.scatter(hairpins_sub.bin2, hairpins_sub.bin1, c='green', label='hairpins')
plt.legend()
plt.show()
```

2.2.3 Plot the distribution of scores

Scores of detected patterns are provided as Pearson correlation coefficient with the template and are stored in the 'score' column of the tabular output. Their distribution can be viewed with regular histogram functions. Since we use a threshold for detection (the `--pearson` option in the command line interface), the score lower end of the distribution will be truncated at this threshold.

Different patterns will have different score distributions and default thresholds.

```
[39]: %matplotlib inline
plt.figure(figsize=(8, 8))
fig, ax = plt.subplots(3, 1, sharex=True)
for i, (df, pat) in enumerate(zip([loops, borders, hairpins], ['loops', 'borders',
    ↪ 'hairpins'])):
    ax[i].hist(df.score, 20)
    ax[i].set_title(pat)
plt.tight_layout()

<Figure size 576x576 with 0 Axes>
```

2.2.4 Looking at detected patterns

Windows around detected patterns in the processed matrix are stored in the JSON / npy file when running chromosight's detect or quantify commands. These windows are in the same order as the coordinates in the output table.

```
[40]: # Load input json file into a dictionary
loop_wins = json.load(open('detect/example_loops.json', 'r'))
# Note that keys are string, as required by the JSON format,
# so we convert them to int() for convenience
loop_wins = {int(i): np.array(w) for i, w in loop_wins.items()}
# Make an empty 3D array of shape N_coords x height x width
wins = np.zeros((len(loop_wins.items()), *loop_wins[0].shape))
# Fill the 3D array with windows values
for i, w in loop_wins.items(): wins[i] = w
```

For example, we can plot the best 40 windows around detected loops ordered by score:

```
[41]: %matplotlib inline
plt.figure(figsize=(10, 10))

fig, ax = plt.subplots(8, 5, figsize=(8, 12))

for i, n in enumerate(np.argsort(loops.score)[39::-1]):
    m, s = np.nanmean(loop_wins[n]), np.nanstd(loop_wins[n])
    ax.flat[i].imshow((loop_wins[n] - m) / s, cmap='afmhot_r', vmax=4)
    ax.flat[i].set_title(f'{loops.score[n]:.2f}')
```

(continues on next page)

(continued from previous page)

```
ax.flat[i].set_xticks([])
ax.flat[i].set_yticks([])
plt.tight_layout()
```

```
<Figure size 720x720 with 0 Axes>
```

The pileup can also be re-computed from these windows using chromosight's helper function. This is useful to plot the pileup for a subset of the detected patterns, or just to generate the pileup plot with different aesthetics.

```
[42]: %matplotlib inline
plt.figure(figsize=(4, 4))
pileup = cud.pileup_patterns(wins)
plt.imshow(pileup, cmap='coolwarm', vmax=1.8, vmin=0)
```

```
[42]: <matplotlib.image.AxesImage at 0x7f571dcb6f50>
```


3.1 chromosight

3.1.1 chromosight package

Subpackages

chromosight.cli package

Submodules

chromosight.cli.chromosight module

Pattern exploration and detection

Explore and detect patterns (loops, borders, centromeres, etc.) in Hi-C contact maps with pattern matching.

Usage:

```
chromosight detect [-kernel-config=FILE] [-pattern=loops] [-pearson=auto] [-win-size=auto] [-iterations=auto] [-win-fmt={json,npy}] [-norm={auto,raw,force}] [-subsample=no] [-inter] [-tsvd] [-smooth-trend] [-n-mads=5] [-min-dist=0] [-max-dist=auto] [-no-plotting] [-min-separation=auto] [-dump=DIR] [-threads=1] [-perc-zero=auto] [-perc-undetected=auto] <contact_map> <prefix>
```

```
chromosight generate-config [-preset loops] [-click contact_map] [-norm={auto,raw,norm}] [-win-size=auto] [-n-mads=5] [-chroms=CHROMS] [-inter] [-threads=1] <prefix>
```

```
chromosight quantify [-inter] [-pattern=loops] [-subsample=no] [-win-fmt=json] [-kernel-config=FILE] [-norm={auto,raw,norm}] [-threads=1] [-n-mads=5] [-win-size=auto] [-perc-undetected=auto] [-perc-zero=auto] [-no-plotting] [-tsvd] <bed2d> <contact_map> <prefix>
```

```
chromosight list-kernels [-long] [-mat] [-name=kernel_name] chromosight test
```

detect: performs pattern detection on a Hi-C contact map via template matching

generate-config: Generate pre-filled config files to use for detect and quantify. A config consists of a JSON file describing parameters for the analysis and path pointing to kernel matrices files. Those matrices files are tsv files with numeric values as kernel to use for convolution.

quantify: Given a list of pairs of positions and a contact map, computes the correlation coefficients between those positions and the kernel of the selected pattern.

list-kernels: Prints information about available kernels.

test: Download example data and run loop detection on it.

Arguments for detect:

contact_map The Hi-C contact map to detect patterns on, in cool format.

prefix Common path prefix used to generate output files. Extensions will be added for each file.

Arguments for quantify:

bed2d Tab-separated text files with columns chrom1, start1 end1, chrom2, start2, end2. Each line correspond to a pair of positions (i.e. a position in the matrix).

contact_map Path to the contact map, in cool format. **prefix** Common path prefix used to generate output files. Extensions will be added for each file.

Arguments for generate-config:

prefix Path prefix for config files. If prefix is a/b, files a/b.json and a/b.1.txt will be generated. If a given pattern has N kernel matrices, N txt files are created they will be named a/b.[1-N].txt.

-e, --preset=loops Generate a preset config for the given pattern. Preset configs available are “loops” and “borders”. [default: loops]

-c, --click=contact_map Show input contact map and uses double clicks from user to build the kernel. Warning: memory-heavy, reserve for small genomes or subsetted matrices.

-C, --chroms=CHROMS Comma-separated list of chromosome names. When used with **--click**, this will show each chromosome’s one-by-one sequentially instead of the whole genome. This is useful to reduce memory usage.

Arguments for list-kernels:

--name=kernel_name Only show information related to a particular kernel.[default: all]

--long Show default parameters in addition to kernel names.

--mat Prints an ascii representation of the kernel matrix.

Basic options:

-h, --help Display this help message.

--version Display the program’s current version.

--verbose Displays the logo.

-n, --norm={auto,raw,force} **Normalization / balancing behaviour.** **auto:** weights present in the cool file are used. **raw:** raw contact values are used. **force:** recompute weights and overwrite existing values. raw[default: auto]

-I, --inter Enable to consider interchromosomal contacts. Warning: Experimental feature with high memory consumption, only use with small matrices.

-m, --min-dist=auto Minimum distance from the diagonal (in base pairs). at which detection should operate. [default: auto]

- M, --max-dist=auto** Maximum distance from the diagonal (in base pairs) for detection. [default: auto]
- P, --pattern=loops** Which pattern to detect. This will use preset configurations for the given pattern. Possible values are: loops, loops_small, borders, hairpins and centromeres. [default: loops]
- p, --pearson=auto** Pearson correlation threshold when detecting patterns in the contact map. Lower values leads to potentially more detections, but more false positives. [default: auto]
- s, --subsample=INT** If greater than 1, subsample INT contacts from the matrix. If between 0 and 1, subsample a proportion of contacts instead. Useful when comparing matrices with different coverages. [default: no]
- t, --threads=1** Number of CPUs to use in parallel. [default: 1]
- u, --perc-undetected=auto** Maximum percentage of non-detectable pixels (nan) in windows allowed to report patterns. [default: auto]
- z, --perc-zero=auto** Maximum percentage of empty (0) pixels in windows allowed to report patterns. [default: auto]

Advanced options:

- d, --dump=DIR** Directory where to save matrix dumps during processing and detection. Each dump is saved as a compressed npz of a sparse matrix and can be loaded using `scipy.sparse.load_npz`.
- i, --iterations=auto** How many iterations to perform after the first template-based pass. [default: 1]
- k, --kernel-config=FILE** Optionally give a path to a custom JSON kernel config path. Use this to override pattern if you do not want to use one of the preset patterns.
- no-plotting** Disable generation of pileup plots.
- N, --n-mads=5** **Maximum number of median absolute deviations below** the median of the bin sums distribution allowed to consider detectable bins. [default: 5]
- S, --min-separation=auto** Minimum distance required between patterns, in basepairs. If two patterns are closer than this distance in both axes, the one with the lowest score is discarded. [default: auto]
- T, --smooth-trend** Use isotonic regression when detrending to reduce noise at long ranges. Do not enable this for circular genomes.
- V, --tsvd** Enable kernel factorisation via truncated svd. Accelerates detection, at the cost of slight inaccuracies. Singular matrices are truncated to retain 99.9% of the information in the kernel.
- w, --win-fmt={json, npy}** **File format used to store individual windows** around each pattern. Window order matches patterns inside the associated text file. Possible formats are json and npy. [default: json]
- W, --win-size=auto** Window size (width), in pixels, to use for the kernel when computing correlations. The pattern kernel will be resized to match this size. Linear interpolation is used to fill between pixels. If not specified, the default kernel size will be used instead. [default: auto]

`chromosight.cli.chromosight.capture_output` (*stderr_to=None*)

Capture the stderr of the test run.

`chromosight.cli.chromosight.cmd_detect` (*args*)

```
chromosight.cli.chromosight.cmd_generate_config(args)
chromosight.cli.chromosight.cmd_list_kernels(args)
chromosight.cli.chromosight.cmd_quantify(args)
chromosight.cli.chromosight.cmd_test(args)
chromosight.cli.chromosight.logo_version(logo, ver)
chromosight.cli.chromosight.main()
```

chromosight.cli.score module

Module contents

chromosight.kernels package

Chromosight’s kernel submodule contains each default kernel in the form of a dictionary with the kernel name. The items of the dictionaries are the key-value pairs from the kernel’s json file, with the kernel matrices pre-loaded under the “kernels” key. Here is the kernel submodule can be used to extract the first borders kernel:

```
import chromosight.kernels as ck
kernel = ck.borders['kernels'][0]
```

Module contents

Chromosight’s kernel submodule contains each default kernel in the form of a dictionary with the kernel name. The items of the dictionaries are the key-value pairs from the kernel’s json file, with the kernel matrices pre-loaded under the “kernels” key. Here is the kernel submodule can be used to extract the first borders kernel:

```
import chromosight.kernels as ck
kernel = ck.borders['kernels'][0]
```

A list of all available kernel names can also be accessed directly:

```
import chromosight.kernels as ck
names = ck.kernel_names
```

chromosight.utils package

Submodules

chromosight.utils.contacts_map module

Chromosight’s contact_map submodule contains classes to keep track of the different aspects of Hi-C data and automate standard pre-processing steps. The ContactMap class holds the whole genome informations and metadata (chromosome sizes, resolution, etc) without loading the actual contact map. Upon calling its “make_sub_matrices” method, it will generate a collection of ContactMap instances accessible via the sub_mats attribute. Each instance corresponds to an inter- or intra-chromosomal matrix and the Hi-C matrix of each chromosome is loaded and preprocessed upon instantiation.

```
class chromosight.utils.contacts_map.ContactMap(clr, extent, name="", detectable_bins=None, inter=False, max_dist=None, largest_kernel=0, dump=None, smooth=False, sample=None, use_norm=True)
```

Bases: `object`

Class to store and manipulate a simple Hi-C matrix, either intra or inter-chromosomal.

clr [`cooler.Cooler`] Reference to a cooler object containing Hi-C data.

extent [list of tuples of ints] List of two tuples containing the start and end bin numbers of both chromosomes from the submatrix.

matrix [`scipy.sparse.csr_matrix`] The contact map as a sparse matrix.

detectable_bins [tuple of arrays] List containing two arrays (rows and columns) of indices from bins considered detectable in the matrix.

inter [bool] True if the matrix represents contacts between two different chromosomes, False otherwise.

max_dist [int] Maximum distance (in bins) at which contact values should be analysed. Only valid for intra-chromosomal matrices.

dump [str] Base path where dump files will be generated. None means no dump.

name [str] Name of the submatrix (used for dumping).

smooth [bool] Whether isotonic regression should be used to smooth the signal for detrending. This will reduce noise at long ranges but assumes contacts can only decrease with distance from the diagonal. Do not use with circular chromosomes.

sample [int, float or None] Proportion of contacts to sample from the data if between 0 and 1. Number of contacts to keep if above 1. Keep all if None.

use_norm [bool] Whether to use the balanced matrix. If set to False, the raw contact counts are used.

create_mat ()

destroy_mat ()

Destroys contact map to clean up memory

detrend (**kwargs)

Executed at run time of the wrapped method. Executes the input function with its arguments, then dumps the matrix to target path. Note args[0] will always denote the instance of the wrapped method.

keep_distance

preprocess_inter_matrix (**kwargs)

Executed at run time of the wrapped method. Executes the input function with its arguments, then dumps the matrix to target path. Note args[0] will always denote the instance of the wrapped method.

preprocess_intra_matrix ()

remove_diags (**kwargs)

Executed at run time of the wrapped method. Executes the input function with its arguments, then dumps the matrix to target path. Note args[0] will always denote the instance of the wrapped method.

subsample (**kwargs)

Executed at run time of the wrapped method. Executes the input function with its arguments, then dumps the matrix to target path. Note args[0] will always denote the instance of the wrapped method.

```
class chromosight.utils.contacts_map.DumpMatrix(dump_name)
```

Bases: `object`

This class is used as a decorator to wrap ContactMap’s methods and generate dump files of the “matrix” attribute. The matrix should be a scipy.sparse matrix and will be saved in npy format. The full dump path will be:

```
inst.dump / os.path.basename(inst.name) + self.dump_name + “.npy”
```

Where inst is the ContactMap instance of the wrapped method and self is the DumpMatrix instance. If the inst has no dump attribute, no action is performed.

Parameters `dump_name` (*str, os.PathLike object or None*) – The basename of the file where to save the dump. If None, no action is performed.

```
class chromosight.utils.contacts_map.HicGenome (path, inter=False, kernel_config=None,
                                                dump=None, smooth=False, sam-
                                                ple=None)
```

Bases: `object`

Class used to manage relationships between whole genome and intra- or inter- chromosomal Hi-C sub matrices. Also handles reading and writing data.

clr [cooler.Cooler] Cooler object containing Hi-C data and related informations for the whole genome

sub_mats [pandas.DataFrame] Table containing intra- and optionally inter-chromosomal matrices.

detectable_bins [array of ints] Array containing indices of detectable rows and columns.

bins [pandas.DataFrame] Table containing bin related informations.

inter [bool] Whether interchromosomal matrices should be stored.

kernel_config [dict] Kernel configuration associated with the Hi-C genome

max_dist [int] Maximum scanning distance for convolution during pattern detection.

dump [str] Base path where dump files will be generated. None means no dump.

smooth [bool] Whether isotonic regression should be used to smooth the signal for detrending intrachromosomal sub matrices. This will reduce noise at long ranges but assumes contacts can only decrease with distance from the diagonal. Do not use with circular chromosomes.

sample [int, float or None] Proportion of contacts to sample from the data if between 0 and 1. Number of contacts to keep if above 1. Keep all if None.

bins_to_coords (*bin_idx*)

Converts a list of bin IDs to genomic coordinates based on the whole genome contact map.

Parameters `bin_idx` (*numpy.array of ints*) – A list of bin numbers corresponding to rows or columns of the whole genome matrix.

Returns A subset of the bins dataframe, with columns chrom, start, end where chrom is the chromosome name (str), and start and end are the genomic coordinates of the bin (int).

Return type pandas.DataFrame

compute_max_dist ()

Use the kernel config to compute max_dist

coords_to_bins (*coords*)

Converts genomic coordinates to a list of bin ids based on the whole genome contact map.

Parameters `coords` (*pandas.DataFrame*) – Table of genomic coordinates, with columns chrom, pos.

Returns Indices in the whole genome matrix contact map.

Return type numpy.array of ints

gather_sub_matrices()

Gathers all processed sub_matrices into a whole genome matrix

get_full_mat_pattern(chr1, chr2, patterns)

Converts bin indices of a list of patterns from an submatrix into their value in the original full-genome matrix.

Parameters

- **chr1** (*str*) – Name of the first chromosome of the sub matrix (rows).
- **chr2** (*str*) – Name of the second chromosome of the sub matrix (cols).
- **pattern** (*pandas.DataFrame*) – A dataframe of pattern coordinates. Each row is a pattern and columns should be bin1 and bin2, for row and column coordinates in the Hi-C matrix, respectively.

Returns full_patterns – A dataframe similar to the input, but with bins shifted to represent coordinates in the whole genome matrix.

Return type pandas.DataFrame

get_sub_mat_pattern(chr1, chr2, patterns)

Converts bin indices of a list of patterns from the whole genome matrix into their value in the desired intra- or inter-chromosomal sub-matrix.

Parameters

- **chr1** (*str*) – Name of the first chromosome of the sub matrix (rows).
- **chr2** (*str*) – Name of the second chromosome of the sub matrix (cols).
- **pattern** (*pandas.DataFrame*) – A dataframe of pattern coordinates. Each row is a pattern and columns should be bin1 and bin2, for row and column coordinates in the Hi-C matrix, respectively.

Returns full_patterns – A dataframe similar to the input, but with bins shifted to represent coordinates in the target sub-matrix.

Return type pandas.DataFrame

make_sub_matrices()

Generates a table of Hi-C sub matrices. Each sub matrix is either intra or interchromosomal. The table has 3 columns: chr1, chr2 and contact_map. The contact_map column contains instances of the ContactMap class.

Returns The table of sub matrices which will contain `n_chrom` rows if the `inter` attribute is set to false, or $(n_chrom^2) / 2 + n_chroms / 2$ if `inter` is True (that is, the upper triangle matrix).

Return type pandas.DataFrame

normalize(norm='auto', n_mads=5, threads=1)

If the instance's cooler is not balanced, finds detectable bins and applies ICE normalisation to the whole genome matrix. Newly computed biases are stored in the input file. If it is already balanced, detectable bins and weights will be extracted from the file.

Parameters

- **force_norm** (*str*) – Normalization behaviour. If 'auto', existing weights are reused and matrix is balanced only in the absence of weights. If 'raw', raw contact values are used. If 'force', weights are recomputed and the underlying cool file is overwritten.
- **n_mads** (*float*) – Maximum number of median absolute deviations (MADs) below the median of the distribution of logged bin sums to consider a bin detectable.

- **threads** (*int*) – Number of parallel threads to use for normalization.

chromosight.utils.detection module

Chromosight's detection submodule implements the bulk of chromosight convolution engine, as well as functions to perform the different steps of the detection algorithm (pre-processing, local maxima, post-processing...). The `pattern_detector` function orchestrate all those different steps.

`chromosight.utils.detection.filter_foci(foci_mat, min_size=2)`

Given an input matrix of labelled foci (continuous islands of equal nonzero values), filters out foci consisting of fewer pixels than `min_size`.

Parameters

- **foci_mat** (*scipy.sparse.coo_matrix*) – Input matrix of labelled foci. Pixels are numbered according to their respective foci. Pixels that are not assigned to a focus are 0.
- **min_size** (*int*) – Minimum number of pixels required to keep a focus. Pixels belonging to smaller foci will be set to 0.

Returns

- **num_filtered** (*int*) – Number of foci remaining after filtering.
- **filtered_mat** (*scipy.sparse.coo_matrix*) – Matrix of filtered foci.

`chromosight.utils.detection.label_foci(matrix)`

Given a sparse matrix of 1 and 0 values, find all foci of continuously neighbouring positive pixels and assign them a label according to their focus. Horizontal and vertical (4-way) adjacency is considered.

Parameters **matrix** (*scipy.sparse.coo_matrix of ints*) – The input matrix where to label foci. Should be filled with 1 and 0s.

Returns

- **num_foci** (*int*) – Number of individual foci identified.
- **foci_mat** (*scipy.sparse.coo_matrix:*) – The matrix with values replaced by their respective foci labels.

Example

```
>>> M.todense()
array([[1 0 0 0]
       [1 0 1 0]
       [1 0 1 1]
       [0 0 0 0]])
>>> num_foci, foci_mat = label_foci(M)
>>> num_foci
2
>>> foci_mat.todense()
array([[1 0 0 0]
       [1 0 2 0]
       [1 0 2 2]
       [0 0 0 0]])
```

`chromosight.utils.detection.normxcorr2` (*signal*, *kernel*, *max_dist=None*, *sym_upper=False*, *full=False*, *missing_mask=None*, *missing_tol=0.75*, *tsvd=None*, *pval=False*)

Computes the normalized cross-correlation of a dense or sparse signal and a dense kernel. The resulting matrix contains Pearson correlation coefficients.

Parameters

- **signal** (*scipy.sparse.csr_matrix* or *numpy.array*) – The input processed Hi-C matrix.
- **kernel** (*numpy.array*) – The pattern kernel to use for convolution.
- **max_dist** (*int*) – Maximum scan distance, in number of bins from the diagonal. If *None*, the whole matrix is convoluted. Otherwise, pixels further than this distance from the diagonal are set to 0 and ignored for performance. Only useful for intrachromosomal matrices.
- **sym_upper** (*False*) – Whether the matrix is symmetric and upper triangle. True for intrachromosomal matrices.
- **missing_mask** (*scipy.sparse.csr_matrix* of *bool* or *None*) – Matrix defining which pixels are missing (1) or not (0).
- **full** (*bool*) – If True, convolutions will be made in ‘full’ mode; the matrix is first padded with margins to allow scanning to the edges, and missing bins are also masked to exclude them when computing correlation scores. Computationally intensive
- **missing_mask** – Mask matrix denoting missing bins, where missing is denoted as True and valid as False. Can be *None* to ignore missing bin information. Only taken into account when *full=True*.
- **missing_tol** (*float*) – Proportion of missing values allowed in windows to keep the correlation coefficients.
- **tsvd** (*float* or *None*) – If a float between 0 and 1 is given, the input kernel is factorised using truncated SVD, keeping enough singular vectors to retain this proportion of information. Factorisation speeds up convolution at the cost of a loss of information. If the number of singular vectors required to retain the desired information is Disabled by default.
- **pval** (*bool*) – Whether to return a matrix of p-values.

Returns

- *scipy.sparse.csr_matrix* or *numpy.array* – The sparse matrix of correlation coefficients. Same type as the input signal.
- *scipy.sparse.csr_matrix* or *numpy.array* or *None* – A map of Benjamini-Hochberg corrected p-values (q-values). Same type as the input signal. If *pval=False*, this will be *None*.

`chromosight.utils.detection.pattern_detector` (*contact_map*, *kernel_config*, *kernel_matrix*, *coords=None*, *dump=None*, *full=False*, *tsvd=None*)

Detect patterns in a contact map by kernel matching, and extract windows around the detected patterns. If coordinates are provided, detection is skipped and windows are extracted around those coordinates.

Parameters

- **contact_map** (*ContactMap* object) – An object containing an inter- or intra-chromosomal Hi-C contact map and additional metadata.
- **kernel_config** (*dict*) – The kernel configuration, as documented in `chromosight.utils.io.load_kernel_config`

- **kernel_matrix** (*numpy.array*) – The kernel matrix to use for convolution as a 2D numpy array
- **coords** (*numpy.array of ints or None*) – A table with coordinates of patterns, with one pattern per row and 2 columns being the row and column number of the pattern in the input contact map. If this is provided, detection is skipped and quantification is performed on those coordinates.
- **dump** (*str or None*) – Folder in which dumps should be generated after each step of the detection process. If None, no dump is generated
- **tsvd** (*float or None*) – If a float between 0 and 1 is given, the input kernel is factorised using truncated SVD, keeping enough singular vectors to retain this proportion of information. Factorisation speeds up convolution at the cost of a loss of information. If the number of singular vectors required to retain the desired information is disabled by default.

Returns

- **filtered_chrom_patterns** (*pandas.DataFrame*) – A table of detected patterns with 4 columns: bin1, bin2, score, qvalue.
- **chrom_pattern_windows** (*numpy array*) – A 3D array containing the pile of windows around detected patterns.

`chromosight.utils.detection.pick_foci(mat_conv, pearson, min_size=2)`

Pick coordinates of local maxima in a sparse 2D convolution heatmap. A threshold computed based on the pearson argument is applied to the heatmap. All values below that threshold are set to 0. The coordinate of the maximum value in each focus (contiguous region of high values) is returned.

Parameters

- **mat_conv** (*scipy.sparse.coo_matrix of floats*) – A 2D sparse convolution heatmap.
- **pearson** (*float*) – Minimum correlation coefficient required to consider a pixel as candidate. Increasing this value reduces the amount of false positive patterns.
- **min_size** (*int*) – Minimum number of pixels required to keep a focus. Pixels belonging to smaller foci will be set to 0.

Returns

- **foci_coords** (*numpy.array of ints*) – 2D array of coordinates for identified patterns corresponding to foci maxima. None is no pattern was detected.
- **labelled_mat** (*scipy.sparse.coo_matrix*) – The map of detected foci. Pixels which were assigned to a focus are given an integer as their focus ID. Pixels not assigned to a focus are set to 0.

`chromosight.utils.detection.pileup_patterns(pattern_windows)`

Generate a pileup (arithmetic mean) from a stack of pattern windows.

Parameters **pattern_windows** (*numpy.array of floats*) – 3D numpy array of detected windows. Shape is (N, H, W) where N is the number of windows, H the height, and W the width of each window.

Returns 2D numpy array containing the pileup (arithmetic mean) of input windows.

Return type numpy.array of floats

`chromosight.utils.detection.remove_neighbours(patterns, win_size=8)`

Identify patterns that are too close from each other to exclude them. The pattern with the highest score are kept in priority.

Parameters

- **patterns** (*numpy.array of float*) – 2D Array of patterns, with 3 columns: bin1, bin2 and score.
- **win_size** (*int*) – The maximum number of pixels at which patterns are considered overlapping.

Returns Boolean array indicating which patterns are valid (True values) and which are overlapping neighbours (False values)

Return type *numpy.array of bool*

`chromosight.utils.detection.validate_patterns` (*coords, matrix, conv_mat, detectable_bins, kernel_matrix, drop=True, zero_tol=0.3, missing_tol=0.75*)

Given a list of pattern coordinates and a contact map, drop or flag patterns in low detectability regions or too close to matrix boundaries. Also returns the surrounding window of Hi-C contacts around each detected pattern.

Parameters

- **coords** (*numpy.array of ints*) – Coordinates of all detected patterns in the sub matrix. One pattern per row, the first column is the matrix row, second column is the matrix col.
- **matrix** (*scipy.sparse.csr_matrix*) – Hi-C contact map of the sub matrix.
- **conv_mat** (*scipy.sparse.csr_matrix*) – Convolution product of the kernel with the Hi-C sub matrix.
- **detectable_bins** (*list of numpy.array*) – List of two 1D numpy arrays of ints representing ids of detectable rows and columns, respectively.
- **kernel_matrix** (*numpy.array of float*) – The kernel that was used for pattern detection on the Hi-C matrix.
- **zero_tol** (*float*) – Proportion of zero pixels allowed in a pattern window to consider it valid.
- **missing_tol** (*float*) – Proportion of missing pixels allowed in a pattern window to consider it valid.
- **drop** (*bool*) – Whether to discard pattern coordinates and windows from patterns which fall outside the matrix or do not pass validation. If those are kept, they will be given a score of `np.nan` and their windows will be filled with `np.nan`.

Returns

- **filtered_coords** (*pandas.DataFrame*) – Table of coordinates that passed the filters. The dataframe has 3 columns: bin1 (rows), bin2 (col) and score (the correlation coefficient).
- **filtered_windows** (*numpy.array*) – 3D numpy array of signal windows around detected patterns. Each window spans axes 1 and 2, and they are stacked along axis 0.

`chromosight.utils.detection.xcorr2` (*signal, kernel, threshold=0.0001, tsvd=None*)

Cross correlate a dense or sparse 2D signal with a dense 2D kernel.

Parameters

- **signal** (*scipy.sparse.csr_matrix or numpy.array of floats*) – A 2-dimensional numpy array $M \times N$ acting as the detrended Hi-C map.
- **kernel** (*numpy.array of floats*) – A 2-dimensional numpy array $M_k \times N_k$ acting as the pattern template.

- **threshold** (*float*) – Convolution score below which pixels will be set back to zero to save on time and memory.
- **tsvd** (*float or None*) – If a float between 0 and 1 is given, the input kernel is factorised using truncated SVD, keeping enough singular vectors to retain this proportion of information. Factorisation speeds up convolution at the cost of a loss of information. If the number of singular vectors required to retain the desired information is Disabled by default.
- -----
- **out** (*scipy.sparse.csr_matrix or numpy.array*) – Convolution product of signal by kernel. Same type as the input signal.

chromosight.utils.io module

Chromosight's io submodule contains input/output related functions to load contact matrices in cool format, and save output patterns coordinates and windows.

`chromosight.utils.io.check_prefix_dir(prefix)`

Checks for existence of the parent directory of an output prefix

`chromosight.utils.io.download_file(url, file, length=16384)`

`chromosight.utils.io.load_bed2d(path)`

Loads only the first 6 columns of a 2D BED file. Will sniff for header and generate a default header only if none is present. Compatible with output of chromosight detect.

Parameters `path` (*str*) – The path to the 2D BED file.

Returns The content of the 2D BED file as a dataframe with 6 columns. Header will be: chrom1, start1, end1, chrom2, start2, end2.

Return type `pandas.DataFrame`

`chromosight.utils.io.load_cool(cool_path)`

Reads a cool file into memory and parses it into a COO sparse matrix and an array with the starting bin of each chromosome.

Parameters `cool` (*str*) – Path to the input .cool file.

Returns

- **mat** (*scipy.sparse.coo_matrix*) – Output sparse matrix in coordinate format
- **chroms** (*pandas.DataFrame*) – Table of chromosome information. Each row contains the name, length, first and last bin of a chromosome.
- **bins** (*pandas.DataFrame*) – Table of genomic bins information. Each row contains the chromosome, genomic start and end coordinates of a matrix bin.
- **bin_size** (*int*) – Matrix resolution. Corresponds to the number of base pairs per matrix bin.

`chromosight.utils.io.load_kernel_config(kernel, custom=False)`

Load a kernel configuration from input JSON file.

All parameters associated with the kernel along its kernel matrices are loaded into a dictionary.

A kernel config file is a JSON file with the following structure:

```
{ "name": str, "kernels": [
```

```

    str, ...
], "max_dist": int, "min_dist": int, "max_iterations": int, "max_perc_zero": float,
"max_perc_undetected": float, "pearson": float "resolution": int
}

```

The kernels field should contain a list of path to kernel matrices to be loaded. These path should be relative to the config file. When loaded, the kernel field will contain the target matrices as 2D numpy arrays.

The kernel matrices files themselves are raw tsv files containing a dense matrix of numeric value as read by the `numpy.loadtxt` function.

Other fields are:

- **name** : Name of the pattern
- **max_dist** : maximum distance in basepairs to scan from the diagonal
- **max_iterations**: maximum number of scanning iterations to perform
- **max_perc_zero**: Maximum percentage of empty (0) pixels to include a pattern
- **max_perc_zero**: Maximum percentage of missing (nan) pixels to include a pattern
- **pearson**: Increasing this value reduces false positive patterns.
- **resolution**: Basepair resolution for the kernel matrix.

Parameters

- **kernel** (*str*) – The name of the built-in pattern configuration to load if custom is False. Otherwise, the path to the custom JSON configuration file to load.
- **custom** (*bool*) – Determines if a custom JSON configuration file must be loaded, or if a preset configuration is used.

Returns **kernel_config** – A dictionary containing a key: value pair for each parameter as well as list of kernel matrices under key 'kernels'.

Return type `dict`

`chromosight.utils.io.progress(count, total, status=)`
Basic progress bar in terminal.

Parameters

- **count** (*float*) – Current task id.
- **total** (*float*) – Maximum task id.
- **status** (*str*) – Info to write on the side of the bar.

`chromosight.utils.io.save_windows(windows, output_prefix, fmt='json')`

Write windows surrounding detected patterns to a npy or json file. The file contains a 3D array where windows are piled on axis 0, matrix rows are on axis 1 and columns on axis 2.

Parameters

- **windows** (*numpy.array of floats*) – 3D numpy array with axes (windows, rows, columns).
- **output_prefix** (*str*) – Output path where the file will be saved, an extension will be added based on the value of "format".

- **format** (*str*) – Format in which to save windows. Can be either npy for numpy’s binary format, or json for a general purpose text format.

`chromosight.utils.io.write_patterns(coords, output_prefix, dec=10)`

Writes coordinates to a text file.

Parameters

- **coords** (*pandas.DataFrame*) – Pandas dataframe containing the coordinates and score of one detected pattern per row.
- **pattern_name** (*str*) – Name of the pattern. Will be the basename of the output file.
- **output_dir** (*str*) – Output path where the file will be saved.
- **dec** (*int*) – Number of decimals to keep in correlation scores and p-values.

chromosight.utils.plotting module

Chromosight’s plotting submodule contains utilities to visualize the pileup of detected patterns or the input matrix. It also implements an interactive map recording the coordinates of double clicks.

`chromosight.utils.plotting.click_finder(mat, half_w=8, xlab=None, ylab=None)`

Given an input Hi-C matrix, show an interactive window and record coordinates where the user double-clicks. When the interactive window is closed, the stack of windows around recorded coordinates is returned.

Parameters

- **mat** (*scipy.sparse.csr_matrix*) – The input Hi-C matrix to display interactively.
- **half_w** (*int*) – Half width of the windows to return. The resulting windows
- **xlab** (*str*) – Horizontal label to display below the matrix.
- **ylab** (*str*) – Vertical label to display next to the matrix.

Returns 3D stack of images around coordinates recorded interactively. The shape of the stack is (N, w, w) where N is the number of coordinates and w is 2*half_w.

Return type `numpy.array`

`chromosight.utils.plotting.pileup_plot(pileup_pattern, output_prefix, name='pileup_patterns')`

Wrapper around `matplotlib.pyplot.imshow` to visualize the pileup of patterns detected by chromosight

`chromosight.utils.plotting.plot_whole_matrix(clr, patterns, out=None, region=None, region2=None, log_transform=False)`

Visualise the input matrix with a set of patterns overlaid on top. Can optionally restrict the visualisation to a region.

Parameters

- **mat** (*scipy.sparse.csr_matrix*) – The whole genome Hi-C matrix to be visualized.
- **patterns** (*pandas.DataFrame*) – The set of patterns to be plotted on top of the matrix. One pattern per row, 3 columns: bin1, bin2 and score of types int, int and float, respectively.
- **region** (*str*) – The genomic range, in UCSC format, corresponding to rows to be plotted in the matrix. If not given, the whole matrix is used. If only region is given, but not region2, the matrix is subsetting on rows and columns to show a region on the diagonal.

- **region2** (*str*) – The genomic range, in UCSC format, of columns to be plotted in the matrix. Region must also be provided, or this will be ignored.
- **log_transform** (*bool*) – Whether to log transform the matrix.

`chromosight.utils.plotting.print_ascii_mat` (*mat*, *adjust=True*, *colored=False*,
print_str=True)

Given a 2D numpy array of float, print it in ASCII art.

Parameters

- **mat** (*np.array of floats*) – Matrix to visualize.
- **adjust** (*bool*) – Whether to adjust the drawing size to termina width.
- **colored** (*bool*) – Whether to use colors.
- **print_str** (*bool*) – If true, the ASCII art is printed to stdout, otherwise it is stored in a string and returned.

Returns An empty string is returned if `print_str` is `True`, otherwise the ASCII art is returned as a string.

Return type `str`

chromosight.utils.preprocessing module

Chromosight's preprocessing submodule implements a number of functions to operate on Hi-C contact maps before detection. These functions can be used to improve the signal or filter unneeded signal. There are also functions to edit (zoom, crop, factorize) kernel matrices.

`chromosight.utils.preprocessing.check_missing_mask` (*signal*, *mask*)

Ensure all elements defined as missing by the mask are set to zero in the signal. If this is not the case, raises an error.

Parameters

- **signal** (*numpy.ndarray of floats or scipy.sparse.csr_matrix of floats*) – The signal to be checked.
- **mask** (*numpy.ndarray of bools or scipy.sparse.csr_matrix of bools*) – The mask defining missing values as `True` and valid values as `False`.

`chromosight.utils.preprocessing.crop_kernel` (*kernel*, *target_size*)

Crop a kernel matrix to target size horizontally and vertically. If the target size is even, the target size is adjusted to the next integer up.

Parameters

- **kernel** (*numpy.ndarray of floats*) – Image to crop.
- **target_size** (*tuple of ints*) – Tuple defining the target shape of the kernel, takes the form (rows, cols) where rows and cols are odd numbers.

Returns `cropped` – New image no larger than target dimensions

Return type `numpy.ndarray of floats`

`chromosight.utils.preprocessing.detrrend` (*matrix*, *detectable_bins=None*, *max_dist=None*,
smooth=False, *fun=<function nanmean>*,
max_val=10)

Detrends a Hi-C matrix by the distance law. The input matrix should have been normalised beforehand.

Parameters

- **matrix** (*scipy.sparse.csr_matrix*) – The normalised intrachromosomal Hi-C matrix to detrend.
- **detectable_bins** (*tuple*) – Tuple containing a list of detectable rows and a list of columns on which to perform detrending. Poorly interacting indices have been excluded.
- **max_dist** (*int*) – Maximum number of bins from the diagonal at which to compute trend.
- **smooth** (*bool*) – Whether to use isotonic regression to smooth the trend.
- **fun** (*callable*) – Function to use on each diagonal to compute the trend.
- **max_val** (*float or None*) – Maximum value in the detrended matrix. Set to None to disable

Returns The detrended Hi-C matrix.

Return type `numpy.ndarray`

`chromosight.utils.preprocessing.diag_trim(mat, n)`

Trim an upper triangle sparse matrix so that only the first n diagonals are kept.

Parameters

- **mat** (*scipy.sparse.csr_matrix or numpy.ndarray*) – The sparse matrix to be trimmed
- **n** (*int*) – The number of diagonals from the center to keep (0-based).

Returns The diagonally trimmed upper triangle matrix with only the first n diagonal.

Return type `scipy.sparse.dia_matrix` or `numpy.ndarray`

`chromosight.utils.preprocessing.distance_law(matrix, detectable_bins=None, max_dist=None, smooth=True, fun=<function nanmean>)`

Computes genomic distance law by averaging over each diagonal in the upper triangle matrix. If a list of detectable bins is provided, pixels in missing bins will be excluded from the averages. A maximum distance can be specified to define how many diagonals should be computed.

Parameters

- **matrix** (*scipy.sparse.csr_matrix*) – the input matrix to compute distance law from.
- **detectable_bins** (*numpy.ndarray of ints*) – An array of detectable bins indices to consider when computing distance law.
- **max_dist** (*int*) – Maximum distance from diagonal, in number of bins in which to compute distance law
- **smooth** (*bool*) – Whether to use isotonic regression to smooth the distance law.
- **fun** (*callable*) – A function to apply on each diagonal. Defaults to mean.

Returns `dist` – the output genomic distance law.

Return type `np.ndarray`

Example

```

>>> m = np.ones((3,3))
>>> m += np.array([1,2,3])
>>> m
array([[2., 3., 4.],
       [2., 3., 4.],
       [2., 3., 4.]])
>>> distance_law(csr_matrix(m))
array([3. , 3.5, 4. ])

```

`chromosight.utils.preprocessing.erase_missing`(*signal*, *valid_rows*, *valid_cols*, *sym_upper=True*)

Given a sparse matrix, set all pixels in missing (invalid) bins to 0.

Parameters

- **signal** (*scipy.sparse.csr_matrix of floats*) – Input signal on which to erase values.
- **valid_rows** (*numpy.ndarray of ints*) – Indices of rows considered valid (not missing).
- **valid_cols** (*numpy.ndarray of ints*) – Indices of columns considered valid (not missing).
- **sym_upper** (*bool*) – Define if the input signal is upper symmetric.

Returns The input signal with all values in missing bins set to 0

Return type `scipy.sparse.csr_matrix`

`chromosight.utils.preprocessing.factorise_kernel`(*kernel*, *prop_info=0.999*)

Performs truncated SVD on an input kernel, returning the singular vectors necessary to retain a given proportion of information contained in the kernel.

Parameters

- **kernel** (*numpy.ndarray of floats*) – The input 2D kernel to factorise.
- **prop_info** (*float*) – Proportion of information to retain.

Returns A tuple containing the truncated left and right singular matrices, where each singular vector has been multiplied by the square root of their respective singular values.

Return type tuple of `numpy.ndarrays` of floats

`chromosight.utils.preprocessing.frame_missing_mask`(*mask*, *kernel_shape*, *sym_upper=False*, *max_dist=None*)

Adds a frame around input mask, given a kernel. The goal of this frame is define margins around the matrix where the kernel will not perform convolution (denoted by 1). If the matrix is upper symmetric, a margin of half the kernel's width is added below the diagonal and a maximum distance from the diagonal above which margins need not be drawn can be considered. Otherwise Margins are simply added on all 4 sides of the matrix.

signal	kernel	
<pre> _ </pre>	<pre> _ </pre>	<pre>==> ##### # # # # # # # # ##### -----</pre>

Parameters

- **mask** (*scipy.sparse.csr_matrix of bool*) – The mask around which to add margins.
- **kernels_shape** (*tuple of ints*) – The number of rows and kernel in the input kernel. Margins will be half these values.
- **sym_upper** (*bool*) – Whether the signal is a symmetric upper triangle matrix. If so, values on a margin below the diagonal will be masked.
- **max_dist** (*int or None*) – Number of diagonals to keep

Returns **framed_mask** – The input mask with a padding of True around the edges. If sym_upper is True, a padding is also added below the diagonal.

Return type *scipy.sparse.csr_matrix of bool*

`chromosight.utils.preprocessing.get_detectable_bins(mat, n_mads=3, inter=False)`

Returns lists of detectable indices after excluding low interacting bins based on the proportion of zero pixel values in the matrix bins.

Parameters

- **mat** (*scipy.sparse.coo_matrix*) – A Hi-C matrix in the form of a 2D numpy array or coo matrix
- **n_mads** (*int*) – Number of median absolute deviation below the median required to consider bins non-detectable.
- **inter** (*bool*) – Whether the matrix is interchromosomal. Default is to consider the matrix is intrachromosomal (i.e. upper symmetric).

Returns tuple of 2 1D arrays containing indices of low interacting rows and columns, respectively.

Return type *numpy ndarray*

`chromosight.utils.preprocessing.make_missing_mask(shape, valid_rows, valid_cols, max_dist=None, sym_upper=False)`

Given lists of valid rows and columns, generate a sparse matrix mask with missing pixels denoted as 1 and valid pixels as 0. If a max_dist is provided, upper symmetric matrices will only be flagged up to max_dist pixels from the diagonal.

Parameters

- **shape** (*tuple of ints*) – Shape of the mask to generate.
- **valid_rows** (*numpy.ndarray of ints*) – Array with the indices of valid rows that should be set to 0 in the mask.
- **valid_cols** (*numpy.ndarray of ints*) – Array with the indices of valid rows that should be set to 0 in the mask.
- **max_dist** (*int or None*) – The maximum diagonal distance at which masking should take place.
- **sym_upper** (*bool*) – Whether the matrix is symmetric upper. If so, max_dist is ignored

Returns The mask containing False values where pixels are valid and True valid where pixels are missing

Return type *scipy.sparse.csr_matrix of bool*

`chromosight.utils.preprocessing.resize_kernel` (*kernel*, *kernel_res=None*, *signal_res=None*, *factor=None*, *min_size=7*, *quiet=False*)

Resize a kernel matrix based on the resolution at which it was defined and the signal resolution. E.g. if a kernel matrix was generated for 10kb and the input signal is 20kb, kernel size will be divided by two. If the kernel is enlarged, pixels are interpolated with a spline of degree 1. Alternatively, a resize factor can be provided. In the example above, the factor would be 0.5.

Parameters

- **kernel** (*numpy.ndarray*) – Kernel matrix.
- **kernel_res** (*int*) – Resolution for which the kernel was designed. Mutually exclusive with factor.
- **signal_res** (*int*) – Resolution of the signal matrix in basepair per matrix bin. Mutually exclusive with factor.
- **factor** (*float*) – Resize factor. Can be provided as an alternative to `kernel_res` and `signal_res`. Values above 1 will enlarge the kernel, values below 1 will shrink it.
- **min_size** (*int*) – Lower bound, in number of rows/column allowed when resizing the kernel.
- **quiet** (*bool*) – Suppress warnings if resize factor was adjusted.

Returns `resized_kernel` – The resized input kernel.

Return type `numpy.ndarray`

`chromosight.utils.preprocessing.set_mat_diag` (*mat*, *diag=0*, *val=0*)

Set the *n*th diagonal of a symmetric 2D numpy array to a fixed value. Operates in place.

Parameters

- **mat** (*numpy.ndarray*) – Symmetric 2D array of floats.
- **diag** (*int*) – 0-based index of the diagonal to modify. Use negative values for the lower half.
- **val** (*float*) – Value to use for filling the diagonal

`chromosight.utils.preprocessing.subsample_contacts` (*M*, *n_contacts*)

Bootstrap sampling of contacts in a sparse Hi-C map.

Parameters

- **M** (*scipy.sparse.coo_matrix*) – The input Hi-C contact map in sparse format.
- **n_contacts** (*int*) – The number of contacts to sample.

Returns A new matrix with a fraction of the original contacts.

Return type `scipy.sparse.coo_matrix`

`chromosight.utils.preprocessing.sum_mat_bins` (*mat*)

Compute the sum of matrices bins (i.e. rows or columns) using only the upper triangle, assuming symmetrical matrices.

Parameters **mat** (*scipy.sparse.coo_matrix*) – Contact map in sparse format, either in upper triangle or full matrix.

Returns 1D array of bin sums.

Return type `numpy.ndarray`

`chromosight.utils.preprocessing.valid_to_missing(valid, size)`

Given an array of valid indices, return the corresponding array of missing indices.

Parameters

- **valid** (*numpy.ndarray of ints*) – The valid indices.
- **size** (*int*) – The size of the matrix (maximum possible index + 1).

Returns **missing** – The missing indices.

Return type `numpy.ndarray of ints`

`chromosight.utils.preprocessing.zero_pad_sparse(mat, margin_h, margin_v, fmt='coo')`

Adds margin of zeros around an input sparse matrix.

Parameters

- **mat** (*scipy.sparse.csr_matrix*) – The matrix to be padded.
- **margin_h** (*int*) – The width of the horizontal margin to add on the left and right of the matrix.
- **margin_v** (*int*) – The width of the vertical margin to add on the top and bottom of the matrix.
- **fmt** (*string*) – The desired scipy sparse format of the output matrix

Returns The padded matrix of dimensions $(m + 2 * \text{margin_h}, n + 2 * \text{margin_v})$.

Return type `scipy.sparse.csr_matrix`

Examples

```
>>> m = sp.csr_matrix(np.array([[1, 2], [10, 20]]))
>>> zero_pad_sparse(m, 2, 1).toarray()
array([[ 0,  0,  0,  0,  0,  0],
       [ 0,  0,  1,  2,  0,  0],
       [ 0,  0, 10, 20,  0,  0],
       [ 0,  0,  0,  0,  0,  0]])
```

`chromosight.utils.preprocessing.ztransform(matrix)`

Z transformation for Hi-C matrices.

Parameters **matrix** (*scipy.sparse.coo_matrix*) – A Hi-C matrix in sparse format.

Returns The detrended Hi-C matrix

Return type `scipy.sparse.coo_matrix`

chromosight.utils.stats module

Chromosight's stats submodule contains helper function to compute statistical estimators from distributions

`chromosight.utils.stats.corr_to_pval(corr, n, rho0=0)`

Given a list of Pearson correlation coefficient, convert them to two-sided log10 p-values. The p-values are computed via the fisher transformation described on: <https://w.wiki/Ksu>

Parameters

- **corr** (*numpy.array*) – The Pearson correlation coefficients.

- **n** (*int* or *numpy.array*) – The number of observations used to compute correlation coefficients. Can be given as an array of the same size as *corr* to give the number of sample in each coefficient.
- **rho0** (*float*) – The correlation value under *h0*. We test if *corr* is significantly different from *rho0*.

Returns The array of log10-transformed two-sided p-values, same size as *corr*.

Return type *numpy.array*

`chromosight.utils.stats.fdr_correction` (*pvals*)

Applies false discovery rate correction via the Benjamini-Hochberg procedure to adjust input p-values for multiple testing. .

Parameters **pvals** (*numpy.array of floats*) – Array of uncorrected p-values.

Returns **fdr** – Array of corrected p-values (q-values).

Return type *numpy.array of floats*

Module contents

Submodules

chromosight.version module

Module contents

CHAPTER 4

Indices and tables

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

C

- `chromosight`, [39](#)
- `chromosight.cli`, [22](#)
- `chromosight.cli.chromosight`, [19](#)
- `chromosight.kernels`, [22](#)
- `chromosight.utils`, [39](#)
- `chromosight.utils.contacts_map`, [22](#)
- `chromosight.utils.detection`, [26](#)
- `chromosight.utils.io`, [30](#)
- `chromosight.utils.plotting`, [32](#)
- `chromosight.utils.preprocessing`, [33](#)
- `chromosight.utils.stats`, [38](#)
- `chromosight.version`, [39](#)

B

`bins_to_coords()` (chromosight.utils.contacts_map.HicGenome method), 24

C

`capture_output()` (in module chromosight.cli.chromosight), 21
`check_missing_mask()` (in module chromosight.utils.preprocessing), 33
`check_prefix_dir()` (in module chromosight.utils.io), 30
`chromosight` (module), 39
`chromosight.cli` (module), 22
`chromosight.cli.chromosight` (module), 19
`chromosight.kernels` (module), 22
`chromosight.utils` (module), 39
`chromosight.utils.contacts_map` (module), 22
`chromosight.utils.detection` (module), 26
`chromosight.utils.io` (module), 30
`chromosight.utils.plotting` (module), 32
`chromosight.utils.preprocessing` (module), 33
`chromosight.utils.stats` (module), 38
`chromosight.version` (module), 39
`click_finder()` (in module chromosight.utils.plotting), 32
`cmd_detect()` (in module chromosight.cli.chromosight), 21
`cmd_generate_config()` (in module chromosight.cli.chromosight), 21
`cmd_list_kernels()` (in module chromosight.cli.chromosight), 22
`cmd_quantify()` (in module chromosight.cli.chromosight), 22
`cmd_test()` (in module chromosight.cli.chromosight), 22
`compute_max_dist()` (chromosight.utils.contacts_map.HicGenome method), 24

`mosight.utils.contacts_map.HicGenome` method), 24
`ContactMap` (class in chromosight.utils.contacts_map), 22
`coords_to_bins()` (chromosight.utils.contacts_map.HicGenome method), 24
`corr_to_pval()` (in module chromosight.utils.stats), 38
`create_mat()` (chromosight.utils.contacts_map.ContactMap method), 23
`crop_kernel()` (in module chromosight.utils.preprocessing), 33

D

`destroy_mat()` (chromosight.utils.contacts_map.ContactMap method), 23
`detrend()` (chromosight.utils.contacts_map.ContactMap method), 23
`detrend()` (in module chromosight.utils.preprocessing), 33
`diag_trim()` (in module chromosight.utils.preprocessing), 34
`distance_law()` (in module chromosight.utils.preprocessing), 34
`download_file()` (in module chromosight.utils.io), 30
`DumpMatrix` (class in chromosight.utils.contacts_map), 23

E

`erase_missing()` (in module chromosight.utils.preprocessing), 35

F

`factorise_kernel()` (in module chromosight.utils.preprocessing), 35

`fdr_correction()` (in module *chromosight.utils.stats*), 39
`filter_foci()` (in module *chromosight.utils.detection*), 26
`frame_missing_mask()` (in module *chromosight.utils.preprocessing*), 35

G

`gather_sub_matrices()` (*chromosight.utils.contacts_map.HicGenome* method), 24
`get_detectable_bins()` (in module *chromosight.utils.preprocessing*), 36
`get_full_mat_pattern()` (*chromosight.utils.contacts_map.HicGenome* method), 25
`get_sub_mat_pattern()` (*chromosight.utils.contacts_map.HicGenome* method), 25

H

HicGenome (class in *chromosight.utils.contacts_map*), 24

K

`keep_distance` (*chromosight.utils.contacts_map.ContactMap* attribute), 23

L

`label_foci()` (in module *chromosight.utils.detection*), 26
`load_bed2d()` (in module *chromosight.utils.io*), 30
`load_cool()` (in module *chromosight.utils.io*), 30
`load_kernel_config()` (in module *chromosight.utils.io*), 30
`logo_version()` (in module *chromosight.cli.chromosight*), 22

M

`main()` (in module *chromosight.cli.chromosight*), 22
`make_missing_mask()` (in module *chromosight.utils.preprocessing*), 36
`make_sub_matrices()` (*chromosight.utils.contacts_map.HicGenome* method), 25

N

`normalize()` (*chromosight.utils.contacts_map.HicGenome* method), 25
`normxcorr2()` (in module *chromosight.utils.detection*), 26

P

`pattern_detector()` (in module *chromosight.utils.detection*), 27
`pick_foci()` (in module *chromosight.utils.detection*), 28
`pileup_patterns()` (in module *chromosight.utils.detection*), 28
`pileup_plot()` (in module *chromosight.utils.plotting*), 32
`plot_whole_matrix()` (in module *chromosight.utils.plotting*), 32
`preprocess_inter_matrix()` (*chromosight.utils.contacts_map.ContactMap* method), 23
`preprocess_intra_matrix()` (*chromosight.utils.contacts_map.ContactMap* method), 23
`print_ascii_mat()` (in module *chromosight.utils.plotting*), 33
`progress()` (in module *chromosight.utils.io*), 31

R

`remove_diags()` (*chromosight.utils.contacts_map.ContactMap* method), 23
`remove_neighbours()` (in module *chromosight.utils.detection*), 28
`resize_kernel()` (in module *chromosight.utils.preprocessing*), 36

S

`save_windows()` (in module *chromosight.utils.io*), 31
`set_mat_diag()` (in module *chromosight.utils.preprocessing*), 37
`subsample()` (*chromosight.utils.contacts_map.ContactMap* method), 23
`subsample_contacts()` (in module *chromosight.utils.preprocessing*), 37
`sum_mat_bins()` (in module *chromosight.utils.preprocessing*), 37

V

`valid_to_missing()` (in module *chromosight.utils.preprocessing*), 37
`validate_patterns()` (in module *chromosight.utils.detection*), 29

W

`write_patterns()` (in module *chromosight.utils.io*), 32

X

`xcorr2()` (in module *chromosight.utils.detection*), 29

Z

`zero_pad_sparse()` (in module *chromosight.utils.preprocessing*), [38](#)
`ztransform()` (in module *chromosight.utils.preprocessing*), [38](#)