# cooltools

cooltoolers

Apr 22, 2024

# OVERVIEW

The tools for your *.cool*s

Chromosome conformation capture technologies reveal the incredible complexity of genome folding. A growing number of labs and multiple consortia, including the 4D Nucleome, the International Nucleome Consortium, and ENCODE, are generating higher-resolution datasets to probe genome architecture across cell states, types, and organisms. Larger datasets increase the challenges at each step of computational analysis, from storage, to memory, to researchers' time. The recently-introduced cooler format readily handles storage of high-resolution datasets via a sparse data model.

**cooltools** leverages this format to enable flexible and reproducible analysis of high-resolution data. **cooltools** provides a suite of computational tools with a paired python API and command line access, which facilitates workflows either on high-performance computing clusters or via custom analysis notebooks. As part of the Open2C ecosystem, **cooltools** also provides detailed introductions to key concepts in Hi-C-data analysis with interactive notebook documentation.

If you use **cooltools** in your work, please cite **cooltools**: https://doi.org/10.1101/2022.10.31.514564.

CHAPTER

ONE

# INSTALLATION

## 1.1 Requirements

- Python 3.7+
- Scientific Python packages

## 1.2 Install using pip

Compile and install *cooltools* and its Python dependencies from PyPI using pip:

```
$ pip install cooltools
```

or install the latest version directly from github:

```
$ pip install https://github.com/open2c/cooltools/archive/refs/heads/master.zip
```

## 1.3 Install the development version

Finally, you can install the latest development version of *cooltools* from github. First, make a local clone of the github repository:

```
$ git clone https://github.com/open2c/cooltools
```

Then, you can compile and install *cooltools* in development mode, which installs the package without moving it to a system folder and thus allows immediate live-testing any changes in the python code.

```
$ cd cooltools
$ pip install -e ./
```

### 1.3.1 Visualization

Welcome to the cooltools visualization notebook!

Visualization is a crucial part of analyzing large-scale datasets. Before performing analyses of new Hi-C datasets, it is highly recommend to visualize the data. This notebook contains tips and tricks for visualization of coolers using cooltools.

Current topics:

- *Inspecting C-data stored in coolers*
- *Visualizing C-data with matplotlib*
- *Balancing: filtering bins, biases*
- *Coverage: cis/total profiles*
- Smoothing, interpolation, and adaptive coarsegraining

Future topics:

- higlass-python
- translocations, structural variants
- visualizing matrices for other organisms

[1]:

```
# import standard python libraries
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os
```

[2]:

```
# download test data
# this file is 145 Mb, and may take a few seconds to download
import cooltools
data_dir = './data/'
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir=data_dir)
print(cool_file)
```

```
./data/test.mcool
```

[3]:

```
#import python package for working with cooler files: https://github.com/open2c/cooler
import cooler
```

#### Inspecting C data

The file we just downloaded, test.mcool, contains Micro-C data from HFF cells for two chromosomes in a multi-resolution mcool format.

[4]:

```
# to print which resolutions are stored in the mcool, use list_coolers
cooler.fileops.list_coolers(f'{data_dir}/test.mcool')
```

[4]:

```
['/resolutions/1000',
 '/resolutions/10000',
 '/resolutions/100000',
 '/resolutions/1000000']
```

```
[5]: ### to load a cooler with a specific resolution use the following syntax:
     clr = cooler.Cooler(f'{data_dir}/test.mcool::resolutions/1000000')

     ### to print chromosomes and binsize for this cooler
     print(f'chromosomes: {clr.chromnames}, binsize: {clr.binsize}')

     ### to make a list of chromosome start/ends in bins:
     chromstarts = []
     for i in clr.chromnames:
         print(f'{i} : {clr.extent(i)}')
         chromstarts.append(clr.extent(i)[0])
```

```
chromosomes: ['chr2', 'chr17'], binsize: 1000000
chr2 : (0, 243)
chr17 : (243, 327)
```

Coolers store pairwise contact frequencies in sparse format, which can be fetched on demand as dense matrices. `clr.matrix` returns a matrix selector. The selector supports Python slice syntax `[]` and a `.fetch()` method. Slicing `clr.matrix()` with `[:]` fetches all bins in the cooler. Fetching can return either balanced, or corrected, contact frequences (`balance=True`), or raw counts prior to bias removal (`balance=False`).

In genome-wide C data for mammalian cells in interphase, the following features are typically observed:

- Higher contact frequencies within a chromosome as opposed to between chromosomes; this is consistent with observations of chromosome territories. See *below*.
- More frequent contacts between regions at shorter genomic separations. Characterizing this is explored in more detail in the contacts_vs_dist notebook.
- A plaid pattern of interactions, termed compartments. Characterizing this is explored in more detail in the compartments notebook.

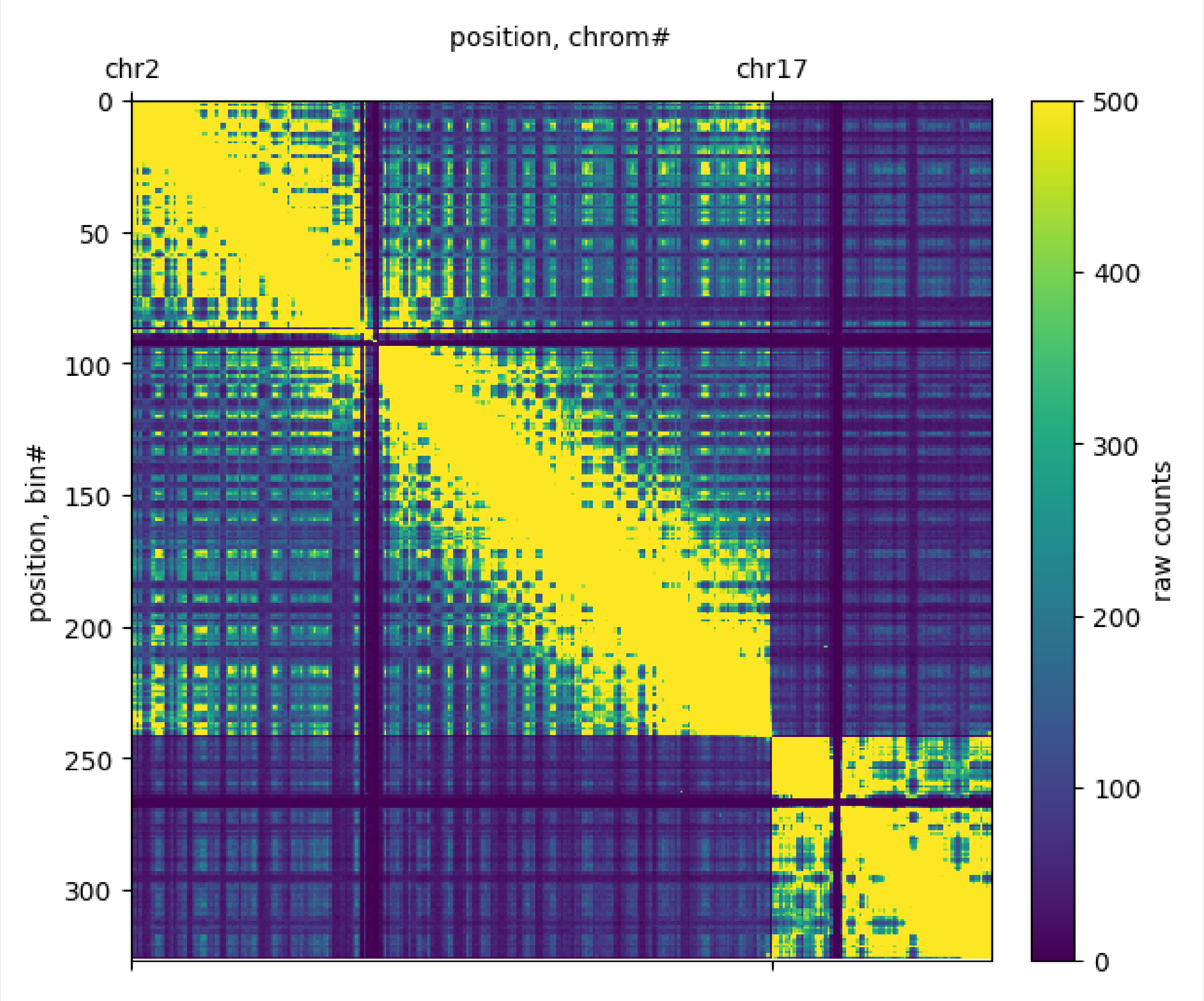
Each of these features are visible below.

### Visualizing C data

#### Plotting raw counts

First, we plot raw counts with a linear colormap thresholded at 500 counts for the entire cooler. Note that the number of counts per cooler depends on the sequencing depth of the experiment, and a different threshold may be needed to see the same features.

```
[6]: f, ax = plt.subplots(
         figsize=(7,6))
     im = ax.matshow((clr.matrix(balance=False)[:]),vmax=500);
     plt.colorbar(im ,fraction=0.046, pad=0.04, label='raw counts')
     ax.set(xticks=chromstarts, xticklabels=clr.chromnames,
            xlabel='position, chrom#', ylabel='position, bin#')
     ax.xaxis.set_label_position('top')
```



### Plotting subregions

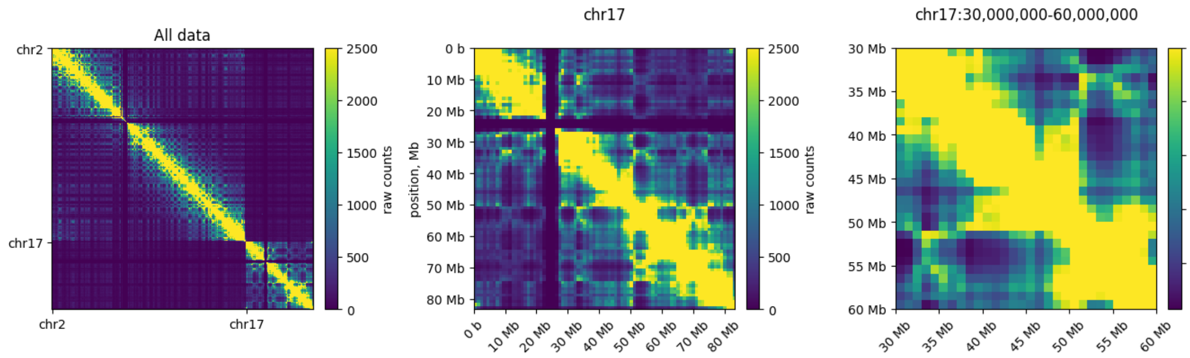
Below, we fetch and plot an individual chromosome (left) and a region of a chromosome (right) using `clr.fetch()`

```
[7]: # to plot ticks in terms of megabases we use the EngFormatter
# https://matplotlib.org/gallery/api/engineering_formatter.html
from matplotlib.ticker import EngFormatter
bp_formatter = EngFormatter('b')

def format_ticks(ax, x=True, y=True, rotate=True):
    if y:
        ax.yaxis.set_major_formatter(bp_formatter)
    if x:
        ax.xaxis.set_major_formatter(bp_formatter)
        ax.xaxis.tick_bottom()
    if rotate:
        ax.tick_params(axis='x',rotation=45)
```

(continues on next page)

(continued from previous page)

```
f, axs = plt.subplots(
    figsize=(14,4),
    ncols=3)

ax = axs[0]
im = ax.matshow(clr.matrix(balance=False)[:], vmax=2500);
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
ax.set_xticks(chromstarts)
ax.set_xticklabels(clr.chromnames)
ax.set_yticks(chromstarts)
ax.set_yticklabels(clr.chromnames)
ax.xaxis.tick_bottom()
ax.set_title('All data')

ax = axs[1]
im = ax.matshow(
    clr.matrix(balance=False).fetch('chr17'),
    vmax=2500,
    extent=(0,clr.chromsizes['chr17'], clr.chromsizes['chr17'], 0)
);
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
ax.set_title('chr17', y=1.08)
ax.set_ylabel('position, Mb')
format_ticks(ax)

ax = axs[2]
start, end = 30_000_000, 60_000_000
region = ('chr17', start, end)
im = ax.matshow(
    clr.matrix(balance=False).fetch(region),
    vmax=2500,
    extent=(start, end, end, start)
);
ax.set_title(f'chr17:{start:,}-{end:,}', y=1.08)
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
format_ticks(ax)
plt.tight_layout()
```



### Logarithmic color scale

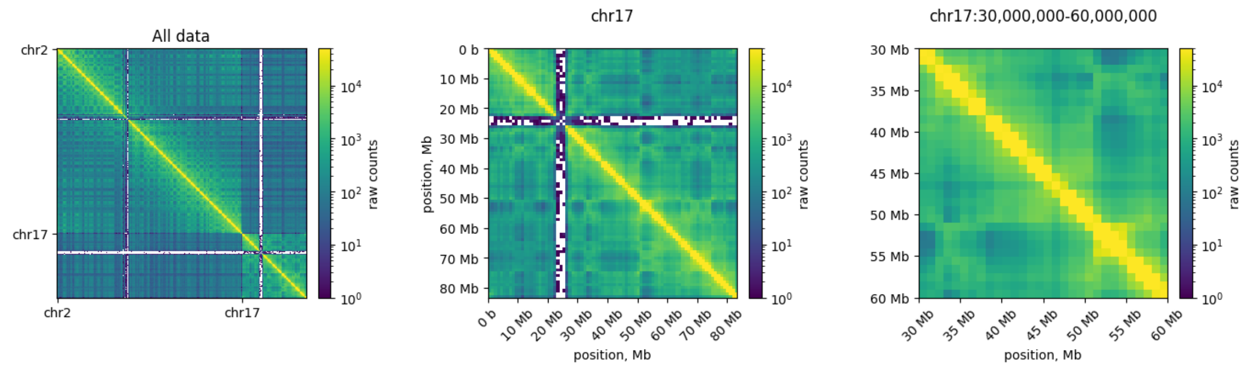
Since C data has a high dynamic range, we often plot the data in log-scale. This enables simultaneous visualization of features near and far from the diagonal in a consistent colorscale. Note that regions with no reported counts are evident as white stripes at both centromeres. This occurs because reads are not uniquely mapped to these highly-repetitive regions. These regions are masked before *matrix balancing*.

[8]:
```
# plot heatmaps at megabase resolution with 3 levels of zoom in log-scale with a␣
↪consistent colormap#
from matplotlib.colors import LogNorm

f, axs = plt.subplots(
    figsize=(14,4),
    ncols=3)
bp_formatter = EngFormatter('b')
norm = LogNorm(vmax=50_000)

ax = axs[0]
im = ax.matshow(
    clr.matrix(balance=False)[:],
    norm=norm,
)
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
ax.set_xticks(chromstarts)
ax.set_xticklabels(clr.chromnames)
ax.set_yticks(chromstarts)
ax.set_yticklabels(clr.chromnames)
ax.xaxis.tick_bottom()
ax.set_title('All data')

ax = axs[1]
im = ax.matshow(
    clr.matrix(balance=False).fetch('chr17'),
    norm=norm,
    extent=(0,clr.chromsizes['chr17'], clr.chromsizes['chr17'], 0)
);
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
ax.set_title('chr17', y=1.08)
ax.set(ylabel='position, Mb', xlabel='position, Mb')
format_ticks(ax)

ax = axs[2]
start, end = 30_000_000, 60_000_000
region = ('chr17', start, end)
im = ax.matshow(
    clr.matrix(balance=False).fetch(region),
    norm=norm,
    extent=(start, end, end, start)
);
ax.set_title(f'chr17:{start:,}-{end:,}', y=1.08)
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='raw counts');
ax.set(xlabel='position, Mb')
format_ticks(ax)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
```



## Colormaps

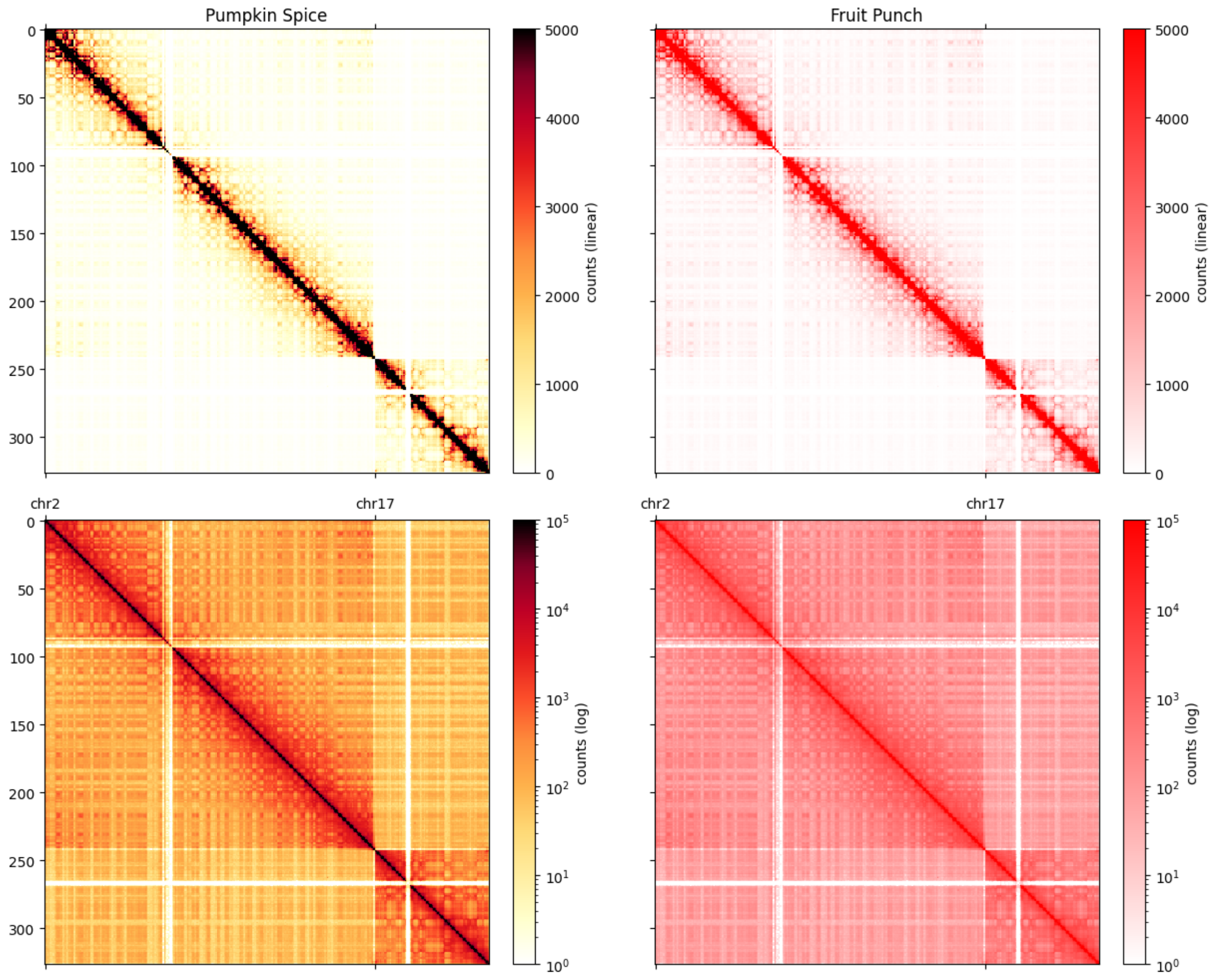
cooltools.lib.plotting registers a set of colormaps that are useful for visualizing C data. In particular, the fall colormap (inspired by colorbrewer) offers a high dynamic range, linear, option for visualizing Hi-C matrices. This often displays features more clearly than red colormaps.

```
[9]: ### plot the corrected data in fall heatmap and compare to the white-red colormap ###
### thanks for the alternative collormap naming to https://twitter.com/HiC_memes/status/
↪1286326919122825221/photo/1###
import cooltools.lib.plotting

vmax = 5000
norm = LogNorm(vmin=1, vmax=100_000)
fruitpunch = sns.blend_palette(['white', 'red'], as_cmap=True)

f, axs = plt.subplots(
    figsize=(13, 10),
    nrows=2,
    ncols=2,
    sharex=True, sharey=True)

ax = axs[0, 0]
ax.set_title('Pumpkin Spice')
im = ax.matshow(clr.matrix(balance=False)[:], vmax=vmax, cmap='fall');
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='counts (linear)');
plt.xticks(chromstarts,clr.chromnames);

ax = axs[0, 1]
ax.set_title('Fruit Punch')
im3 = ax.matshow(clr.matrix(balance=False)[:], vmax=vmax, cmap=fruitpunch);
plt.colorbar(im3, ax=ax, fraction=0.046, pad=0.04, label='counts (linear)');
plt.xticks(chromstarts,clr.chromnames);

ax = axs[1, 0]
im = ax.matshow(clr.matrix(balance=False)[:], norm=norm, cmap='fall');
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='counts (log)');
```

(continues on next page)

(continued from previous page)

```
plt.xticks(chromstarts,clr.chromnames);

ax = axs[1, 1]
im3 = ax.matshow(clr.matrix(balance=False)[:], norm=norm, cmap=fruitpunch);
plt.colorbar(im3, ax=ax, fraction=0.046, pad=0.04, label='counts (log)');
plt.xticks(chromstarts,clr.chromnames);

plt.tight_layout()
```



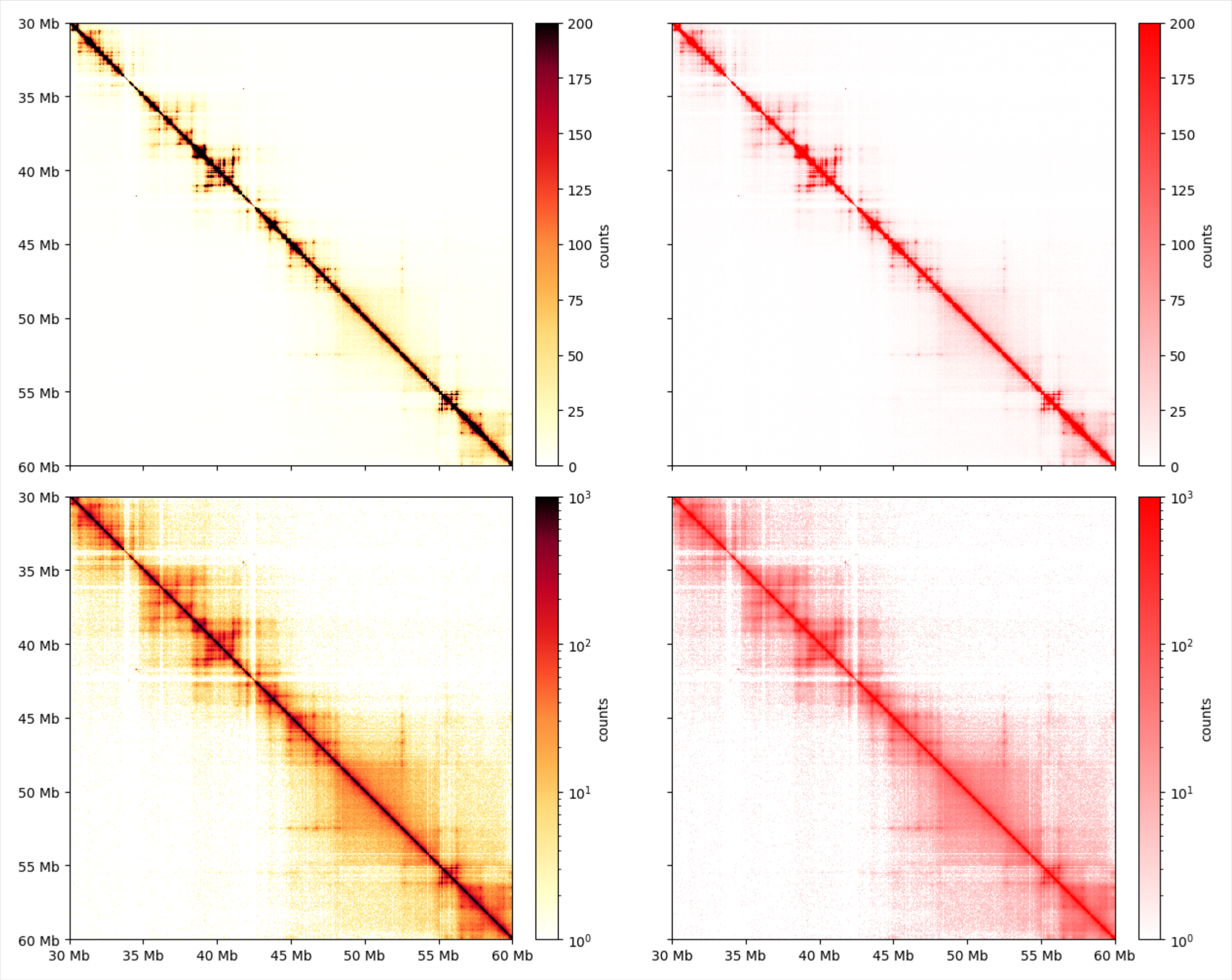
The utility of fall colormaps becomes more noticeable at higher resolutions and higher degrees of zoom.

```
[10]: ### plot the corrected data in fall heatmap ###
import cooltools.lib.plotting
clr_10kb = cooler.Cooler(f'{data_dir}/test.mcool::resolutions/10000')

region = 'chr17:30,000,000-35,000,000'
extents = (start, end, end, start)
norm = LogNorm(vmin=1, vmax=1000)

f, axs = plt.subplots(
```

(continues on next page)

(continued from previous page)

```
    figsize=(13, 10),
    nrows=2,
    ncols=2,
    sharex=True,
    sharey=True
)

ax = axs[0, 0]
im = ax.matshow(
    clr_10kb.matrix(balance=False).fetch(region),
    cmap='fall',
    vmax=200,
    extent=extents
);
plt.colorbar(im, ax=ax ,fraction=0.046, pad=0.04, label='counts');

ax = axs[0, 1]
im2 = ax.matshow(
    clr_10kb.matrix(balance=False).fetch(region),
    cmap=fruitpunch,
    vmax=200,
    extent=extents
);
plt.colorbar(im2, ax=ax, fraction=0.046, pad=0.04, label='counts');

ax = axs[1, 0]
im = ax.matshow(
    clr_10kb.matrix(balance=False).fetch(region),
    cmap='fall',
    norm=norm,
    extent=extents
);
plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04, label='counts');

ax = axs[1, 1]
im2 = ax.matshow(
    clr_10kb.matrix(balance=False).fetch(region),
    cmap=fruitpunch,
    norm=norm,
    extent=extents
);
plt.colorbar(im2, ax=ax, fraction=0.046, pad=0.04, label='counts');

for ax in axs.ravel():
    format_ticks(ax, rotate=False)
plt.tight_layout()
```



## Balancing

When `(balance=True)` is passed to cooler.matrix(), this applies correction weights calculated from matrix balancing. Matrix balancing (also called iterative correction and KR normalization) removes multiplicative biases, which constitute the majority of known biases, from C data. By default, the rows & columns of the matrix are normalized to sum to one (note that the colormap scale differs after balancing). Biases, also called weights for normalization, are stored in the `weight` column of the bin table given by `clr.bins()`.

[11]:
```
clr.bins()[:3]
```

[11]:
```
  chrom    start      end    weight
0  chr2        0  1000000  0.002441
1  chr2  1000000  2000000  0.002435
2  chr2  2000000  3000000  0.002728
```

Before balancing, cooler also applies filters to remove low-coverage bins (note that peri-centromeric bins are completely removed in the normalized data). Filtered bins are stored as `np.nan` in the weights.

Matrices appear visually smoother after removal of biases. Smoother matrices are expected for chromosomes, as adjacent regions along a chromosome are connected and should only slowly vary in their contact frequencies with other regions.

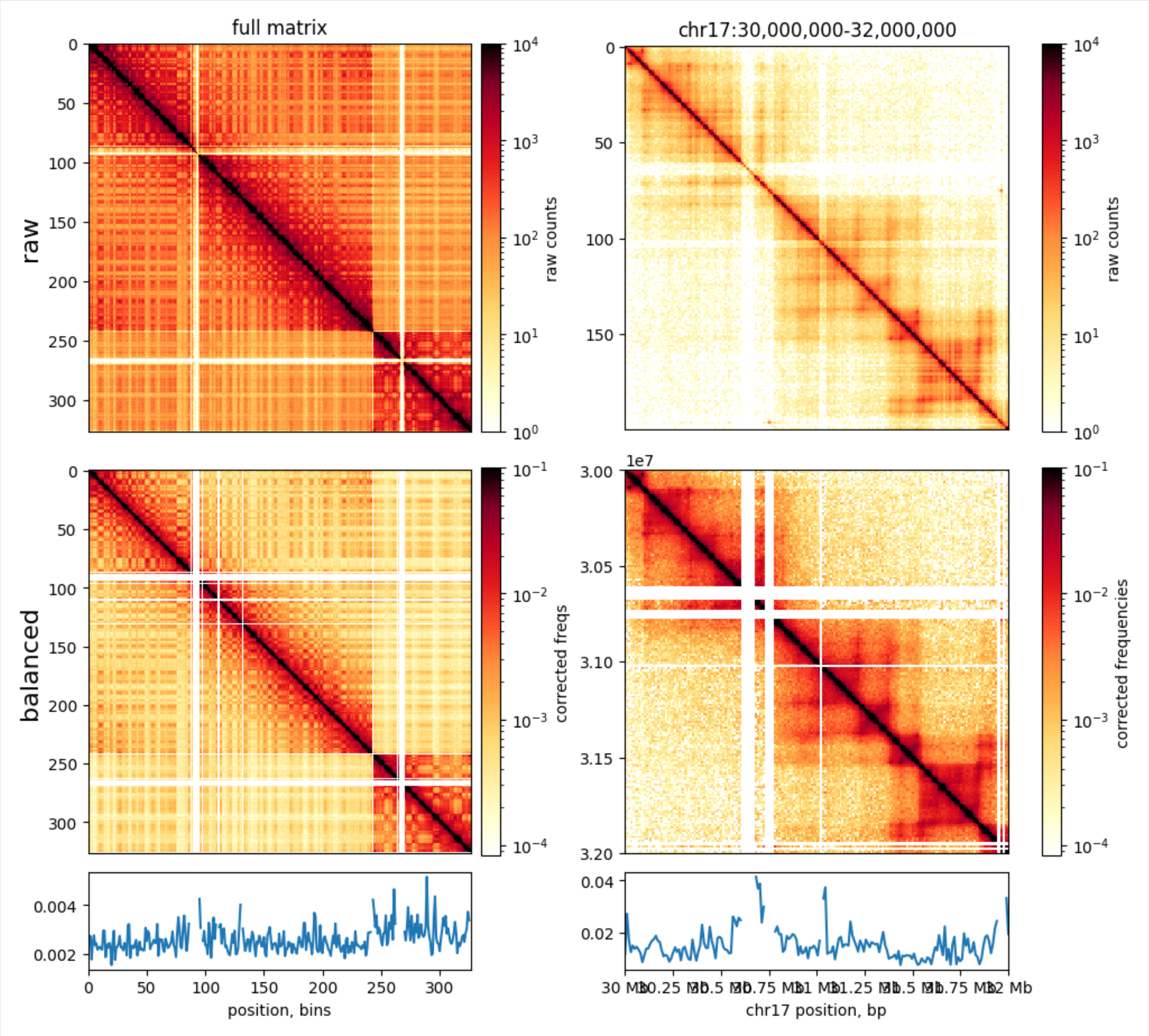
[12]:

```
### plot the raw and corrected data in logscale ###
from mpl_toolkits.axes_grid1 import make_axes_locatable

plt_width=4
f, axs = plt.subplots(
    figsize=( plt_width+plt_width+2, plt_width+plt_width+1),
    ncols=4,
    nrows=3,
    gridspec_kw={'height_ratios':[4,4,1],"wspace":0.01,'width_ratios':[1,.05,1,.05]},
    constrained_layout=True
)

norm = LogNorm(vmax=0.1)
norm_raw = LogNorm(vmin=1, vmax=10_000)

ax = axs[0,0]
im = ax.matshow(
    clr.matrix(balance=False)[:],
    norm=norm_raw,
    cmap='fall',
    aspect='auto'
);
ax.xaxis.set_visible(False)
ax.set_title('full matrix')
ax.set_ylabel('raw', fontsize=16)

cax = axs[0,1]
plt.colorbar(im, cax=cax, label='raw counts')

ax = axs[1,0]
im = ax.matshow(
    clr.matrix()[:],
    norm=norm,
    cmap='fall',
);
ax.xaxis.set_visible(False)
ax.set_ylabel('balanced', fontsize=16)

cax = axs[1,1]
plt.colorbar(im, cax=cax, label='corrected freqs')

ax1 = axs[2,0]
weights = clr.bins()[:]['weight'].values
ax1.plot(weights)
ax1.set_xlim([0, len(clr.bins()[:])])
ax1.set_xlabel('position, bins')

ax1 = axs[2,1]
ax1.set_visible(False)



start = 30_000_000
```

(continues on next page)

(continued from previous page)

```
end = 32_000_000
region = ('chr17', start, end)

ax = axs[0,2]
im = ax.matshow(
        clr_10kb.matrix(balance=False).fetch(region),
    norm=norm_raw,
    cmap='fall'
);
ax.set_title(f'chr17:{start:,}-{end:,}')
ax.xaxis.set_visible(False)

cax = axs[0,3]
plt.colorbar(im, cax=cax, label='raw counts');

ax = axs[1,2]
im = ax.matshow(
    clr_10kb.matrix().fetch(region),
    norm=norm,
    cmap='fall',
    extent=(start, end, end, start)
);
ax.xaxis.set_visible(False)

cax = axs[1,3]
plt.colorbar(im, cax=cax, label='corrected frequencies');

ax1 = axs[2,2]
weights = clr_10kb.bins().fetch(region)['weight'].values
ax1.plot(
    np.linspace(start, end, len(weights)),
    weights
)
format_ticks(ax1, y=False, rotate=False)
ax1.set_xlim(start, end);
ax1.set_xlabel('chr17 position, bp')

ax1 = axs[2,3]
ax1.set_visible(False)
```



### Coverage

Contact matrices often display varible tendency to make contacts within versus between chromosomes. This can be calculated in cooltools with `cooltools.coverage` and is often plotted as a ratio of (cis_coverage/total_coverage). Note that the total coverge is similar to, but distinct from, the iteratively calculated balancing weights (see above).

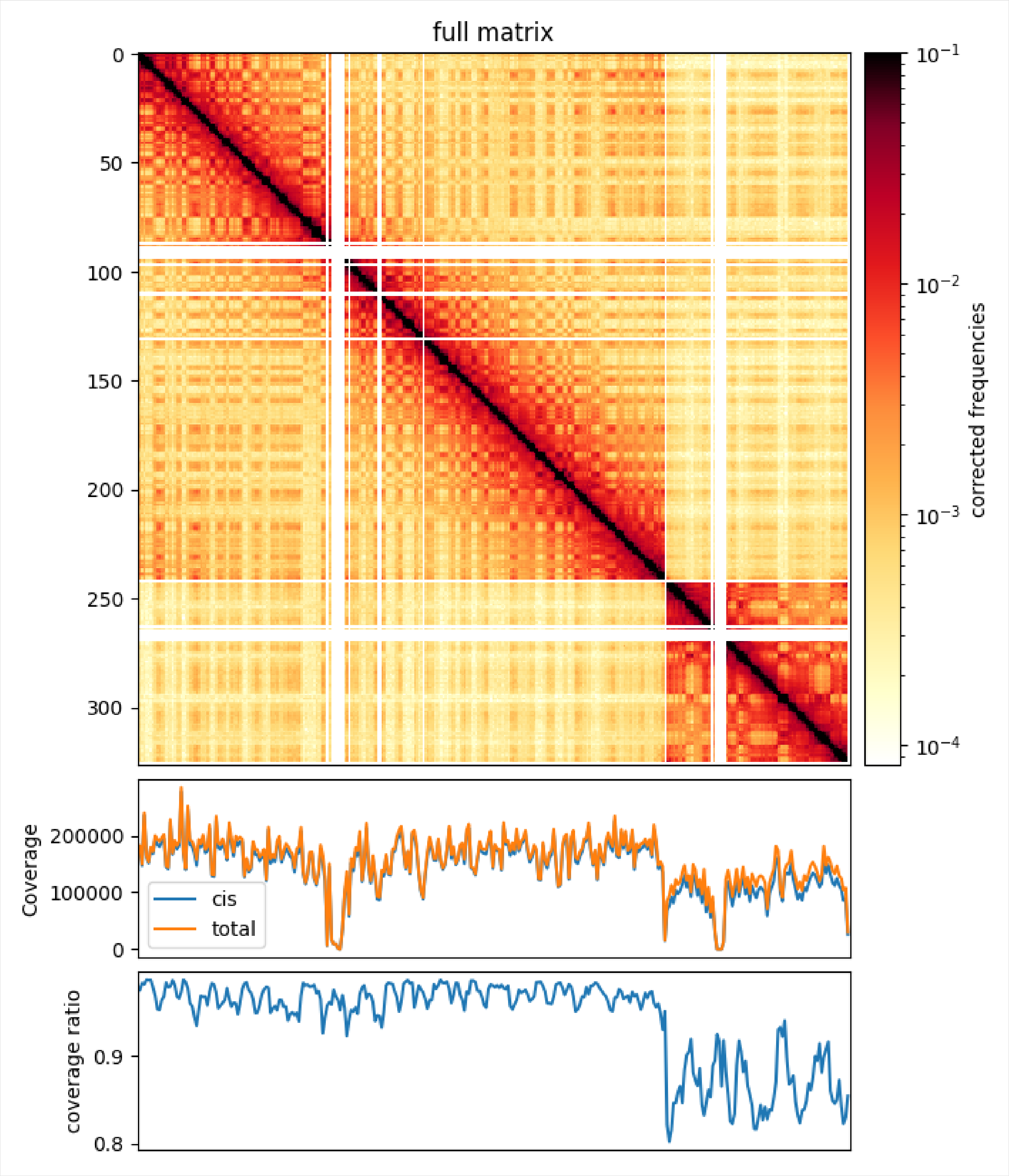
[13]:
```
cis_coverage, tot_coverage = cooltools.coverage(clr)

f, ax = plt.subplots(
    figsize=(15, 10),
)

norm = LogNorm(vmax=0.1)

im = ax.matshow(
```

(continues on next page)

(continued from previous page)

```
    clr.matrix()[:],
    norm=norm,
    cmap='fall'
);
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.1)
plt.colorbar(im, cax=cax, label='corrected frequencies');
ax.set_title('full matrix')
ax.xaxis.set_visible(False)

ax1 = divider.append_axes("bottom", size="25%", pad=0.1, sharex=ax)
weights = clr.bins()[:]['weight'].values
ax1.plot( cis_coverage, label='cis')
ax1.plot( tot_coverage, label='total')
ax1.set_xlim([0, len(clr.bins()[:])])
ax1.set_ylabel('Coverage')
ax1.legend()
ax1.set_xticks([])

ax2 = divider.append_axes("bottom", size="25%", pad=0.1, sharex=ax)
ax2.plot( cis_coverage/ tot_coverage)
ax2.set_xlim([0, len(clr.bins()[:])])
ax2.set_ylabel('coverage ratio')
```

```
[13]: Text(0, 0.5, 'coverage ratio')
```


full matrix
0
50
100
150
200
250
300
10^−1
10^−2
10^−3
10^−4
corrected frequencies
Coverage
200000
100000
0
cis
total
coverage ratio
0.9
0.8

### Smoothing & Interpolation

When working with C data at high resolution, it is often useful to smooth matrices. cooltools provides a method, `adaptive_coarsegrain`, which adaptively smoothing corrected matrices based on the number of counts in raw matrices. For visualization it is also often useful to interpolate over filtered out bins.

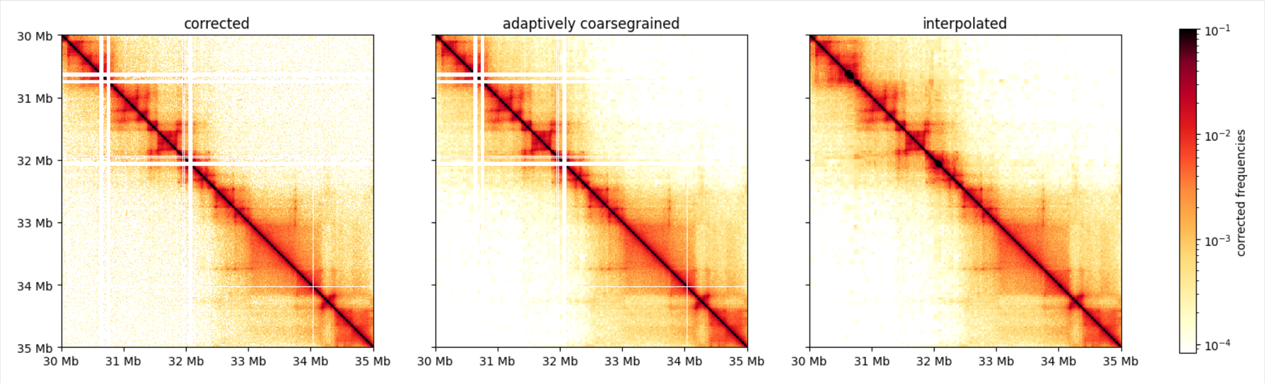
[14]:

```
from cooltools.lib.numutils import adaptive_coarsegrain, interp_nan

clr_10kb = cooler.Cooler(f'{data_dir}/test.mcool::resolutions/10000')
start = 30_000_000
end = 35_000_000
region = ('chr17', start, end)
extents = (start, end, end, start)

cg = adaptive_coarsegrain(clr_10kb.matrix(balance=True).fetch(region),
                              clr_10kb.matrix(balance=False).fetch(region),
                              cutoff=3, max_levels=8)

cgi = interp_nan(cg)

f, axs = plt.subplots(
    figsize=(18,5),
    nrows=1,
    ncols=3,
    sharex=True, sharey=True)

ax = axs[0]
im = ax.matshow(clr_10kb.matrix(balance=True).fetch(region), cmap='fall', norm=norm,␣
↪extent=extents)
ax.set_title('corrected')

ax = axs[1]
im2 = ax.matshow(cg, cmap='fall', norm=norm, extent=extents)
ax.set_title(f'adaptively coarsegrained')

ax = axs[2]
im3 = ax.matshow(cgi, cmap='fall', norm=norm, extent=extents)
ax.set_title(f'interpolated')

for ax in axs:
    format_ticks(ax, rotate=False)

plt.colorbar(im3, ax=axs, fraction=0.046, label='corrected frequencies')
```

```
/home1/rahmanin/.conda/envs/open2c/lib/python3.9/site-packages/cooltools/lib/numutils.py:
↪1376: RuntimeWarning: invalid value encountered in divide
  val_cur = ar_cur / armask_cur
```

[14]:

```
<matplotlib.colorbar.Colorbar at 0x7f24dcaca730>
```



```
[15]: ### future:
      # - yeast maps
      # - translocations
      # - higlass
```

---

This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.2 Contacts vs distance

Welcome to the cooltools expected & contacts-vs-distance notebook!

In Hi-C maps, contact frequency decreases very strongly with **genomic separation** (also referred to as **genomic distance**). In the Hi-C field, this decay is often interchangeably referred to as the:

- **expected** because one "expects" a certain average contact frequency at a given genomic separation
- **scaling** which is borrowed from the polymer physics literature
- **P(s) curve** contact *probability*, *P*, as a function of genomic *separation*, *s*.

The rate of decay of contacts with genomic separation reflects the polymeric nature of chromosomes and can tell us about the global folding patterns of the genome.

This decay has been observed to vary through the cell cycle, across cell types, and after degredation of structural maintenance of chromosomes complexes (SMCs) in both interphase and mitosis.

The goals of this notebook are to:

- calculate the P(s) of a given cooler
- plot the P(s) curve
- smooth the P(s) curve with logarithmic binning
- plot the derivative of P(s)
- plot the P(s) between two different genomic regions
- plot the matrix of average contact frequencies between different chromosomes

```
[1]: %load_ext autoreload
     %autoreload 2
```

[2]:

```
# import core packages
import warnings
warnings.filterwarnings("ignore")
from itertools import combinations
import os

# import semi-core packages
import matplotlib.pyplot as plt
from matplotlib import colors
%matplotlib inline
plt.style.use('seaborn-v0_8-poster')
import numpy as np
import pandas as pd
from multiprocessing import Pool

# import open2c libraries
import bioframe

import cooler
import cooltools

from packaging import version
if version.parse(cooltools.__version__) < version.parse('0.5.2'):
    raise AssertionError("tutorial relies on cooltools version 0.5.2 or higher,"+
                         "please check your cooltools version and update to the latest")

# count cpus
num_cpus = os.getenv('SLURM_CPUS_PER_TASK')
if not num_cpus:
    num_cpus = os.cpu_count()
num_cpus = int(num_cpus)
```

[3]:

```
# download test data
# this file is 145 Mb, and may take a few seconds to download
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir='./data/')
print(cool_file)
```

```
./data/test.mcool
```

[4]:

```
# Load a Hi-C map at a 1kb resolution from a cooler file.
resolution = 1000 # note this might be slightly slow on a laptop
                  # and could be lowered to 10kb for increased speed
clr = cooler.Cooler('data/test.mcool::/resolutions/'+str(resolution))
```

In addition to data stored in a cooler, the analyses below make use of where chromosomal arms start and stop to calculate contact frequency versus distance curves within arms. For commonly-used genomes, bioframe can be used to fetch these annotations directly from UCSC. For less commonly-used genomes, a table of arms, or chromosomes can be loaded in directly with pandas, e.g.

```
chromsizes = pd.read_csv('chrom.sizes', sep='\t')
```

Regions for calculating expected should be provided as a viewFrame, i.e. a dataframe with four columns, chrom, start, stop, name, where entries in the name column are unique and the intervals are non-overlapping. If the chromsizes table does not have a name column, it can be created with `bioframe.core.construction.`

add_ucsc_name_column(bioframe.make_viewframe(chromsizes)).

[5]:

```
# Use bioframe to fetch the genomic features from the UCSC.
hg38_chromsizes = bioframe.fetch_chromsizes('hg38')
hg38_cens = bioframe.fetch_centromeres('hg38')
# create a view with chromosome arms using chromosome sizes and definition of centromeres
hg38_arms = bioframe.make_chromarms(hg38_chromsizes,  hg38_cens)

# select only those chromosomes available in cooler
hg38_arms = hg38_arms[hg38_arms.chrom.isin(clr.chromnames)].reset_index(drop=True)
hg38_arms
```

[5]:

```
   chrom     start        end     name
0   chr2         0   93900000   chr2_p
1   chr2  93900000  242193529   chr2_q
2  chr17         0   25100000  chr17_p
3  chr17  25100000   83257441  chr17_q
```

### Calculate the P(s) curve

To calculate the average contact frequency as a function of genomic separation, we use the fact that each diagonal of a Hi-C map records contacts between loci separated by the same genomic distance. For example, the 3rd diagonal of our matrix contains contacts between loci separated by 3-4kb (note that diagonals are 0-indexed). Thus, we calculate the average contact frequency, *P(s)*, at a given genomic distance, *s*, as the average value of all pixels of the corresponding diagonal. This operation is performed by `cooltools.expected_cis`.

Note that we calculate the *P(s)* separately for each chromosomal **arm**, by providing `hg38_arms` as a `view_df`. This way we will ignore contacts accross the centromere, which is generally a good idea, since such contacts have a slightly different decay versus genomic separation.

[6]:

```
# cvd == contacts-vs-distance
cvd = cooltools.expected_cis(
    clr=clr,
    view_df=hg38_arms,
    smooth=False,
    aggregate_smoothed=False,
    nproc=num_cpus #if you do not have multiple cores available, set to 1
)
```

```
INFO:root:creating a Pool of 10 workers
```

This function calculates average contact frequency for raw and normalized interactions ( `count.avg` and `balanced.avg`) for each diagonal and each regions in the `hg38_arms` of a Hi-C map. It aslo keeps the sum of raw and normalized interaction counts (`count.sum` and `balanced.sum`) as well as the number of valid (i.e. non-masked) pixels at each diagonal, `n_valid`.

[7]:

```
display(cvd.head(4))
display(cvd.tail(4))
```

```
  region1 region2  dist  dist_bp  contact_frequency  n_total  n_valid  \
0  chr2_p  chr2_p     0        0                NaN    93900    86055
1  chr2_p  chr2_p     1     1000                NaN    93899    85282
2  chr2_p  chr2_p     2     2000           0.098270    93898    84918
3  chr2_p  chr2_p     3     3000           0.042805    93897    84649
```

(continues on next page)

(continued from previous page)

```
    count.sum  balanced.sum   count.avg  balanced.avg
0         NaN           NaN         NaN           NaN
1         NaN           NaN         NaN           NaN
2  10842540.0   8344.916674  115.471469      0.098270
3   4733321.0   3623.417357   50.409715      0.042805
```

```
        region1  region2   dist   dist_bp  contact_frequency  n_total  \
325448  chr17_q  chr17_q  58154  58154000                NaN        4
325449  chr17_q  chr17_q  58155  58155000                NaN        3
325450  chr17_q  chr17_q  58156  58156000                NaN        2
325451  chr17_q  chr17_q  58157  58157000                NaN        1

        n_valid  count.sum  balanced.sum  count.avg  balanced.avg
325448        0        0.0           0.0        0.0           NaN
325449        0        0.0           0.0        0.0           NaN
325450        0        0.0           0.0        0.0           NaN
325451        0        0.0           0.0        0.0           NaN
```

Note that the data from the first couple of diagonals are masked. This is done intentionally, since interactions at these diagonals (very short-ranged) are contaminated by non-informative Hi-C byproducts - dangling ends and self-circles.

### Plot the P(s) curve

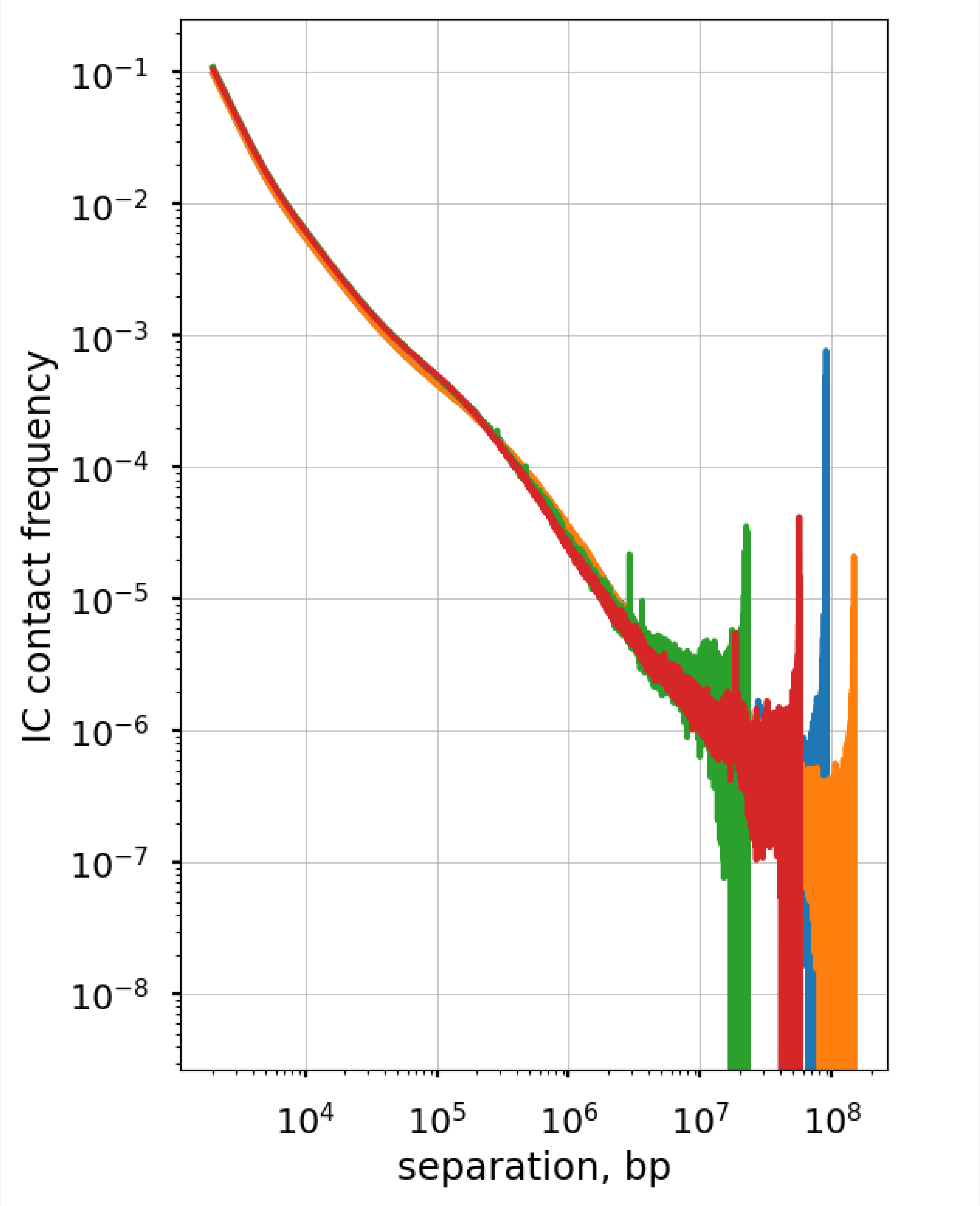
Time to plot *P(s)* !

The first challenge is that Hi-C has a very wide dynamic range. Hi-C probes genomic separations ranging from 100s to 100,000,000s of basepairs and contact frequencies also tend to span many orders of magnitude.

Plotting such data in the linear scale would reveal only a part of the whole picture. Instead, we typically switch to double logarithmic (aka log-log) plots, where the x and y coordinates vary by orders of magnitude.

With the flags used above, `expected_cis()` does not smooth or aggregate across regions. This can lead to noisy P(s) curves for each region:

[8]:
```
f, ax = plt.subplots(1,1)

for region in hg38_arms['name']:
    ax.loglog(
        cvd['dist_bp'].loc[cvd['region1']==region],
        cvd['contact_frequency'].loc[cvd['region1']==region],
    )
    ax.set(
        xlabel='separation, bp',
        ylabel='IC contact frequency')
    ax.set_aspect(1.0)
    ax.grid(lw=0.5)
```



The non-smoothed curves plotted above form characteristic "fans" at longer separations. This happens for two reasons: (a) we plot values of **each** diagonal separately and thus each decade of s contains 10x more points, and (b) due to the polymer nature of chromosomes, contact frequency at large genomic separations are lower and thus more affected by sequencing depth.

This issue is more that just cosmetic, as this noise would prevent us from doing finer analyses of *P(s)* and propagate into data derived from *P(s)*. However, there is a simple solution: we can smooth *P(s)* over multiple diagonals. This works because *P(s)* changes very gradually with *s*, so that consecutive diagonals have similar values. Furthermore, we can make each subsequent smoothing window wider than the previous one, so that each order of magnitude of genomic separation contains the same number of windows. Such aggregation is a little tricky to perform, so `cooltools.expected` implements this operation.

### Smoothing & aggregating P(s) curves

Instead of the flags above, we can pass flags to `expected_cis()` that return smoothed and aggregated columns for futher analysis (which are on by default).

Note that the plots below use smooth_sigma=0.1, which is relatively conservative, and this parameter can be lowered (with discretion) for sufficiently high-resolution datasets.

```
[9]: cvd_smooth_agg = cooltools.expected_cis(
         clr=clr,
         view_df=hg38_arms,
         smooth=True,
         aggregate_smoothed=True,
         smooth_sigma=0.1,
         nproc=num_cpus
     )
```

```
INFO:root:creating a Pool of 10 workers
```

```
[10]: display(cvd_smooth_agg.head(4))
```

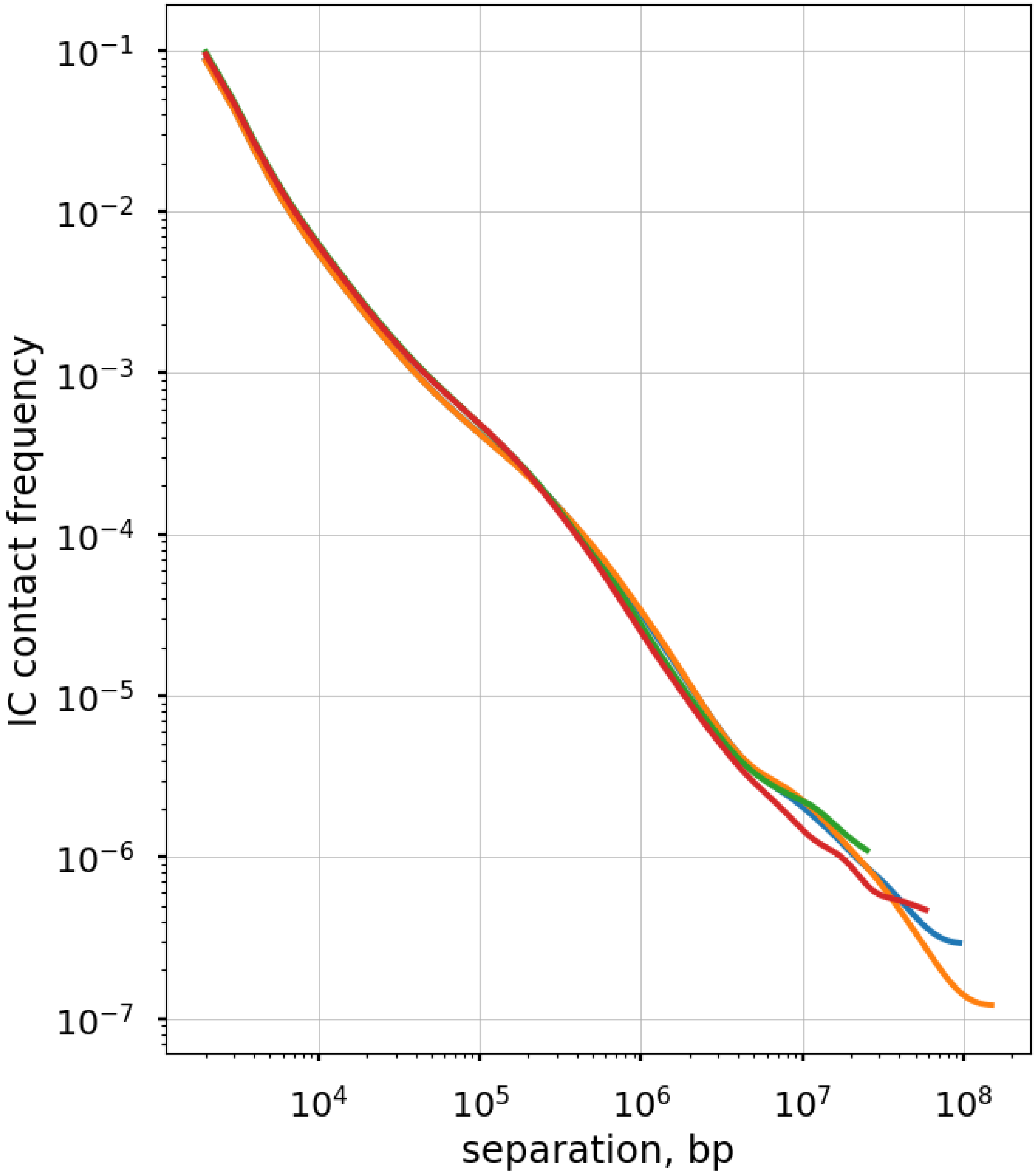
```
  region1 region2  dist  dist_bp  contact_frequency  n_total  n_valid  \
0  chr2_p  chr2_p     0        0                NaN    93900    86055
1  chr2_p  chr2_p     1     1000           0.001060    93899    85282
2  chr2_p  chr2_p     2     2000           0.088615    93898    84918
3  chr2_p  chr2_p     3     3000           0.043947    93897    84649

    count.sum  balanced.sum   count.avg  balanced.avg  balanced.avg.smoothed  \
0         NaN           NaN         NaN           NaN                    NaN
1         NaN           NaN         NaN           NaN               0.001043
2  10842540.0   8344.916674  115.471469      0.098270               0.087228
3   4733321.0   3623.417357   50.409715      0.042805               0.043116

   balanced.avg.smoothed.agg
0                        NaN
1                   0.001060
2                   0.088615
3                   0.043947
```

```
[11]: cvd_smooth_agg['balanced.avg.smoothed'].loc[cvd_smooth_agg['dist'] < 2] = np.nan

      f, ax = plt.subplots(1,1)

      for region in hg38_arms['name']:
          ax.loglog(
              cvd_smooth_agg['dist_bp'].loc[cvd_smooth_agg['region1']==region],
              cvd_smooth_agg['balanced.avg.smoothed'].loc[cvd_smooth_agg['region1']==region],
```

(continues on next page)

(continued from previous page)

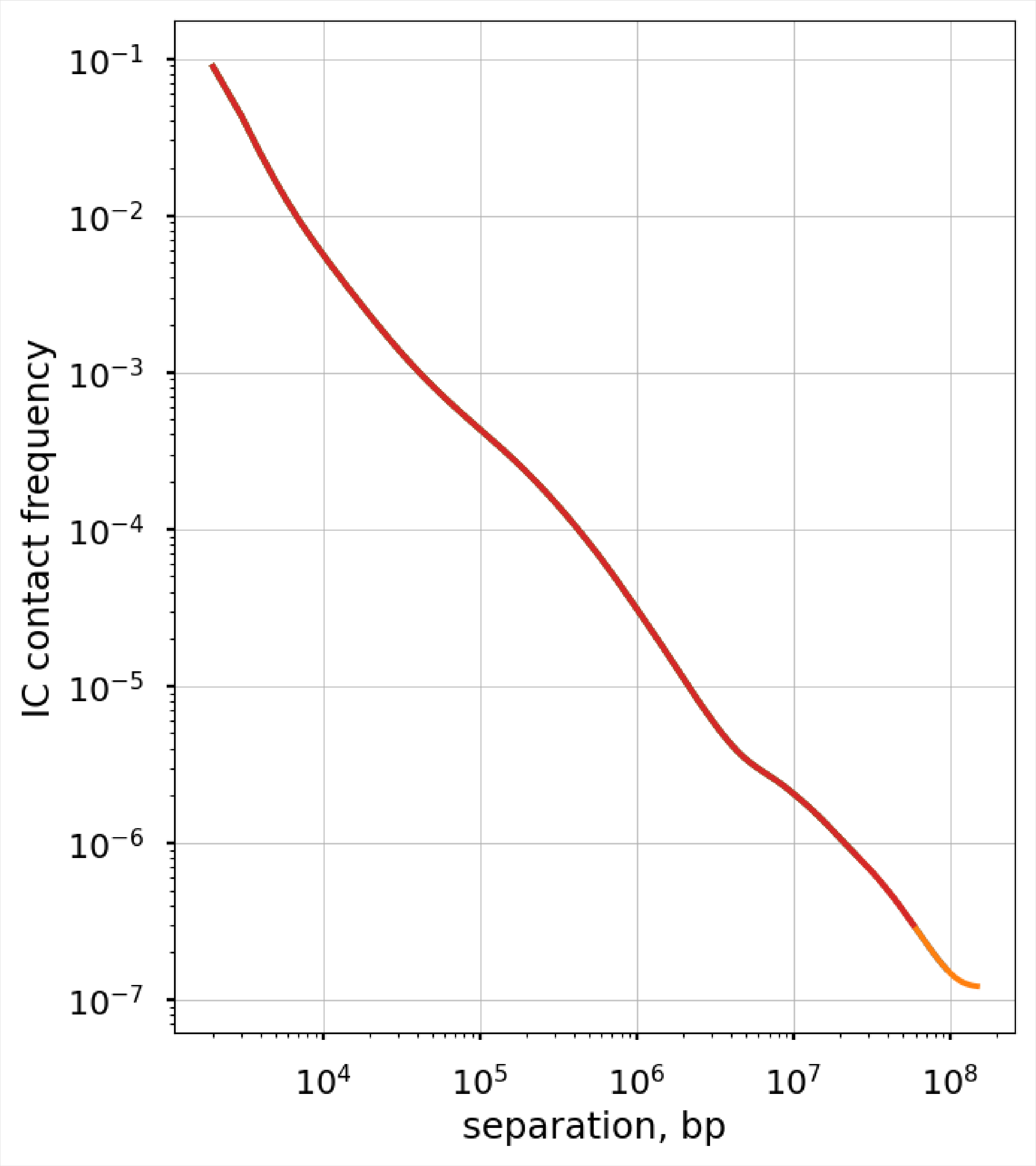
```
    )
    ax.set(
        xlabel='separation, bp',
        ylabel='IC contact frequency')
    ax.set_aspect(1.0)
    ax.grid(lw=0.5)
```



The `balanced.avg.smoothed.agg` is averaged across regions, and shows the same exact curve for each.

[12]:

```
cvd_smooth_agg['balanced.avg.smoothed.agg'].loc[cvd_smooth_agg['dist'] < 2] = np.nan
f, ax = plt.subplots(1,1)

for region in hg38_arms['name']:
    ax.loglog(
        cvd_smooth_agg['dist_bp'].loc[cvd_smooth_agg['region1']==region],
        cvd_smooth_agg['balanced.avg.smoothed.agg'].loc[cvd_smooth_agg['region1
↪']==region],
    )
    ax.set(
        xlabel='separation, bp',
        ylabel='IC contact frequency')
    ax.set_aspect(1.0)
    ax.grid(lw=0.5)
```


IC contact frequency
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^{-6}$
$10^{-7}$
$10^{4}$
$10^{5}$
$10^{6}$
$10^{7}$
$10^{8}$
separation, bp

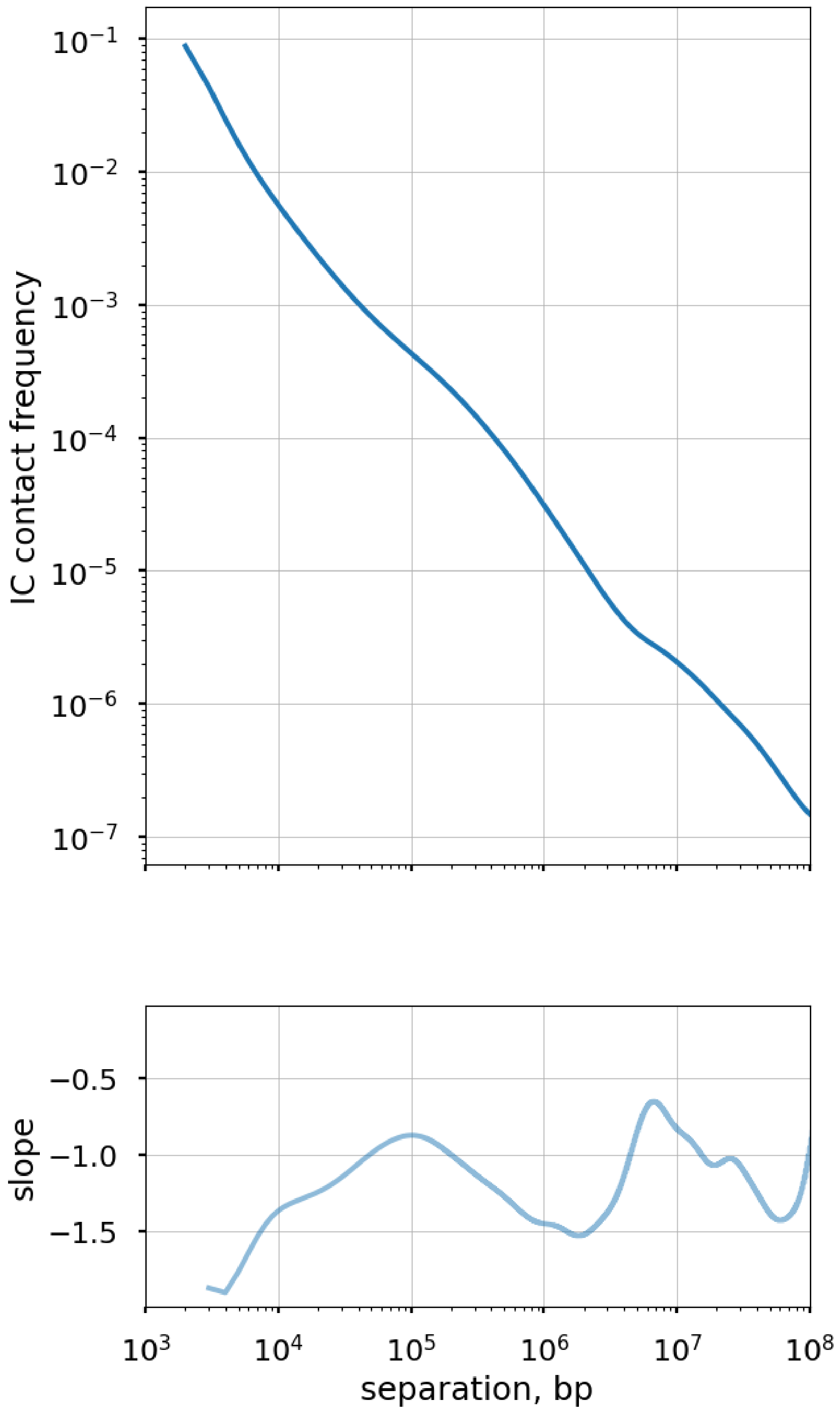
### Plot the smoothed P(s) curve and its derivative

Logbin-smoothing of P(s) reduces the “fanning” at longer s and enables us to plot the derivative of the P(s) curve in the log-log space. This derivative is extremely informative, as it can be compared to predictions from various polymer models.

```
[13]: # Just take a single value for each genomic separation
cvd_merged = cvd_smooth_agg.drop_duplicates(subset=['dist'])[['dist_bp', 'balanced.avg.
↪smoothed.agg']]
```

```
[14]: # Calculate derivative in log-log space
der = np.gradient(np.log(cvd_merged['balanced.avg.smoothed.agg']),
                  np.log(cvd_merged['dist_bp']))
```

```
[15]: f, axs = plt.subplots(
    figsize=(6.5,13),
    nrows=2,
    gridspec_kw={'height_ratios':[6,2]},
    sharex=True)
ax = axs[0]
ax.loglog(
    cvd_merged['dist_bp'],
    cvd_merged['balanced.avg.smoothed.agg'],
    '-'
)

ax.set(
    ylabel='IC contact frequency',
    xlim=(1e3,1e8)
)
ax.set_aspect(1.0)
ax.grid(lw=0.5)


ax = axs[1]
ax.semilogx(
    cvd_merged['dist_bp'],
    der,
    alpha=0.5
)

ax.set(
    xlabel='separation, bp',
    ylabel='slope')

ax.grid(lw=0.5)
```


IC contact frequency
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^{-6}$
$10^{-7}$
slope
−0.5
−1.0
−1.5
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
$10^{7}$
$10^{8}$
separation, bp

### Plot the P(s) curve for interactions between different regions.

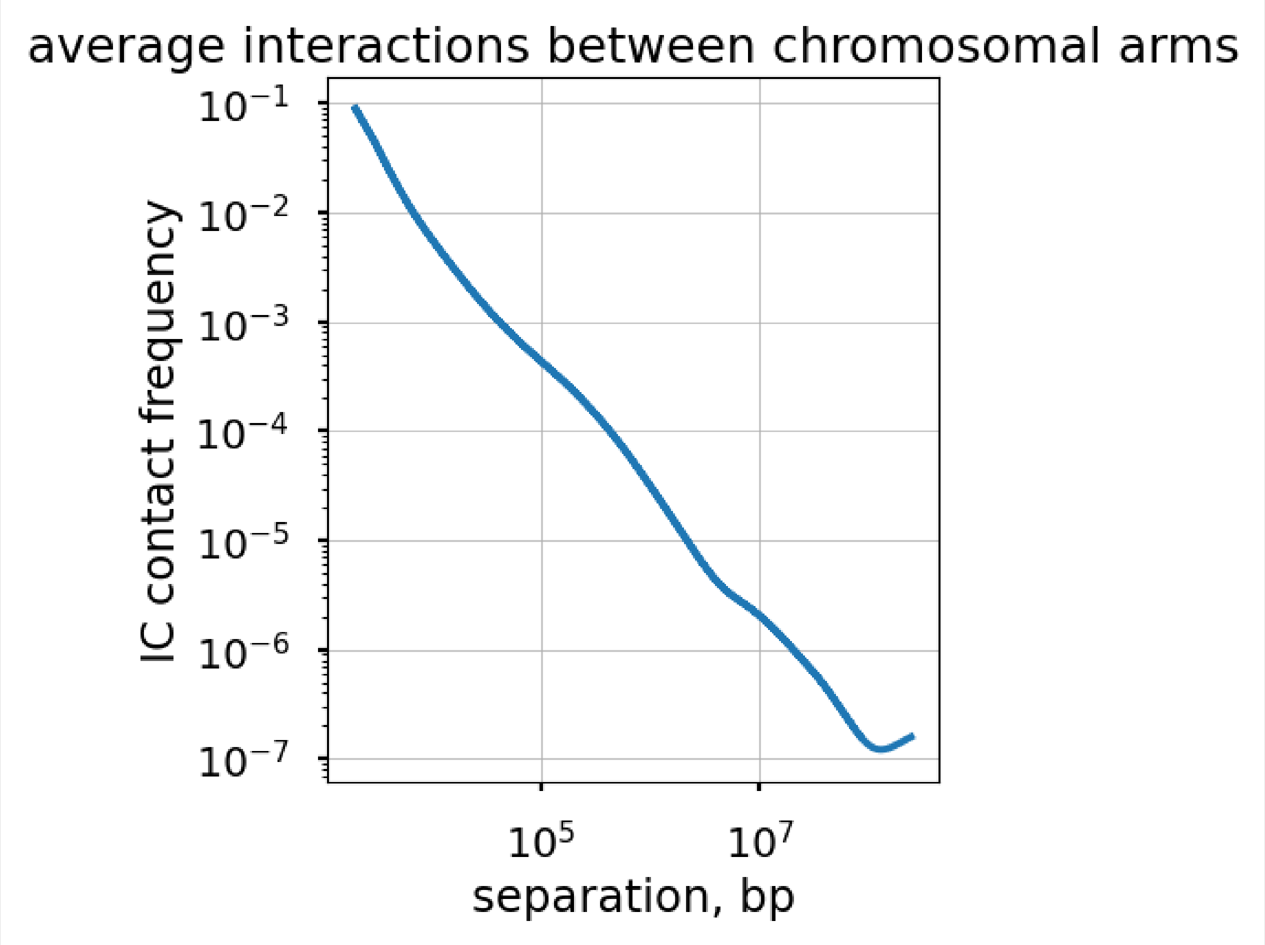
Finally, we can plot P(s) curves for contacts between loci that belong to different regions.

A commonly considered situation is for trans-arm interactions, i.e. contacts between loci on the opposite side of a centromere. Such P(s) can be calculated via `cooltools.expected_cis` by passing `intra_only=False`.

[16]:
```
# cvd_inter == contacts-vs-distance between chromosomal arms
cvd_inter = cooltools.expected_cis(
    clr=clr,
    view_df=hg38_arms,
    intra_only=False,
    nproc=num_cpus
)
# select only inter-arm interactions:
cvd_inter = cvd_inter[ cvd_inter["region1"] != cvd_inter["region2"] ].reset_
↪index(drop=True)
```

```
INFO:root:creating a Pool of 10 workers
```

[17]:
```
cvd_inter['balanced.avg.smoothed.agg'].loc[cvd_inter['dist'] < 2] = np.nan
f, ax = plt.subplots(1,1,
    figsize=(5,5),)

ax.loglog(
    cvd_inter['dist_bp'],
    cvd_inter['balanced.avg.smoothed.agg'],
)

ax.set(
    xlabel='separation, bp',
    ylabel='IC contact frequency',
    title="average interactions between chromosomal arms")
ax.grid(lw=0.5)
ax.set_aspect(1.0)
```



### Averaging interaction frequencies in blocks

For *trans* (i.e. inter-chromosomal) interactions, the notion of "genomic separation" becomes irrelevant, as loci on different chromosomes are not connected by any polymer chain. Thus, the "expected" level of trans interactions is a scalar, the average interaction frequency for a given pair of chromosomes.

Note that this sample dataset only includes two chromosomes– the heatmap of pairwise average contact frequencies can appear much more interesting for a greater number of chromosomes.

```
[18]: # average contacts, in this case between pairs of chromosomal arms:
ac = cooltools.expected_trans(
    clr,
    view_df = None, # full chromosomes as the view
    nproc=num_cpus
)

# average raw interaction counts and normalized contact frequencies are already in the␣
↪result
ac
```

```
INFO:root:creating a Pool of 10 workers
```

[18]:

```
  region1 region2      n_valid  count.sum  balanced.sum  count.avg  \
0    chr2   chr17  16604223614  1307071.0   1021.728671   0.000079

   balanced.avg
0  6.153426e-08
```

[19]:

```
# pivot a table to generate a pair-wise average interaction heatmap:
acp = (ac
    .pivot_table(values="balanced.avg",
                 index="region1",
                 columns="region2",
                 observed=True)
    .reindex(index=clr.chromnames,
             columns=clr.chromnames)
)
```

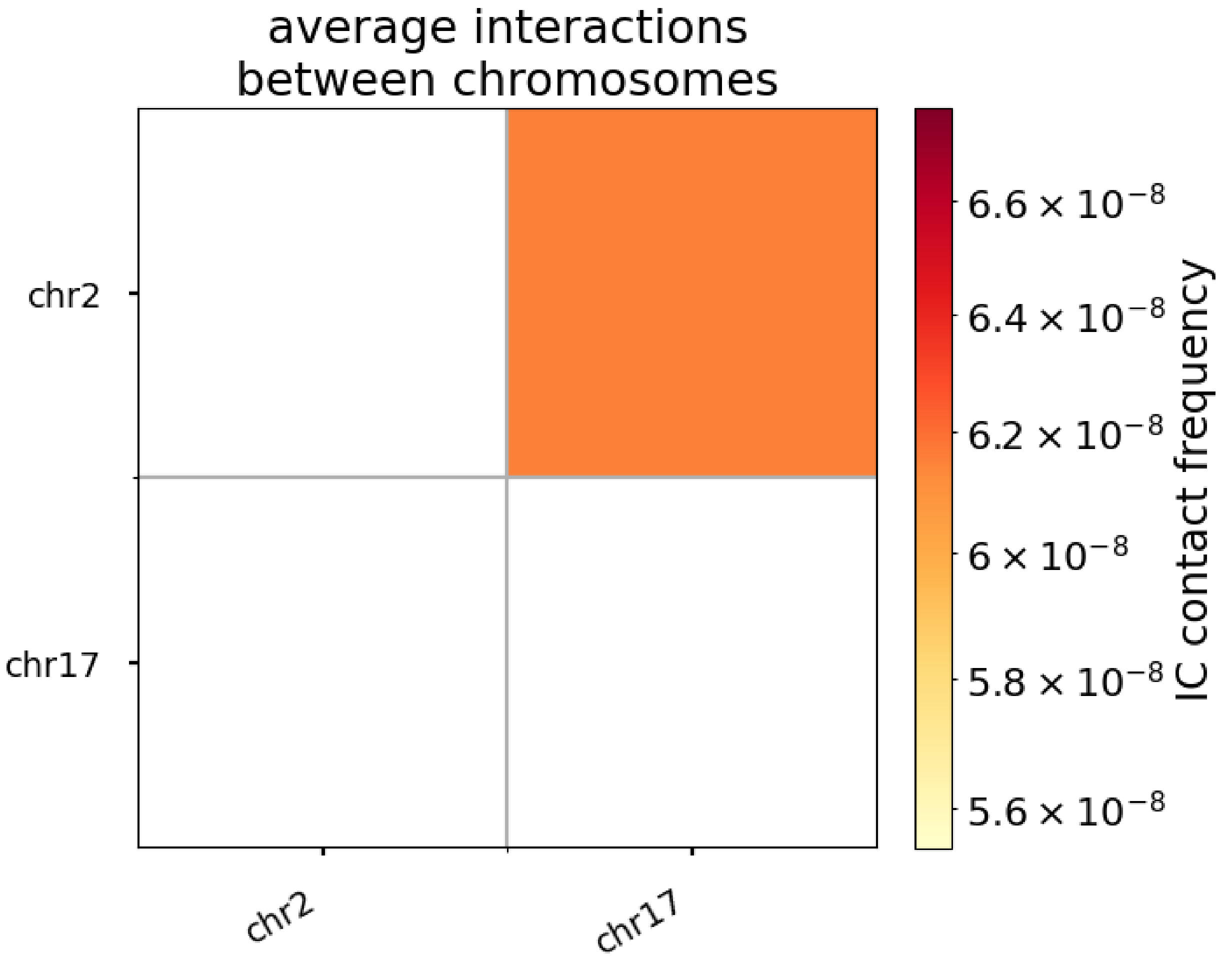
[20]:

```
fs = 14

f, axs = plt.subplots(
    figsize=(6.0,5.5),
    ncols=2,
    gridspec_kw={'width_ratios':[20,1],"wspace":0.1},
)
# assign heatmap and colobar axis:
ax, cax = axs
# draw a heatmap, using log-scale for interaction freq-s:
acpm = ax.imshow(
    acp,
    cmap="YlOrRd",
    norm=colors.LogNorm(),
    aspect=1.0
)
# assign ticks and labels (ordered names of chromosome arms):
ax.set(
    xticks=range(len(clr.chromnames)),
    yticks=range(len(clr.chromnames)),
    title="average interactions\nbetween chromosomes",
)
ax.set_xticklabels(
    clr.chromnames,
    rotation=30,
    horizontalalignment='right',
    fontsize=fs
)
ax.set_yticklabels(
    clr.chromnames,
    fontsize=fs
)
# draw a colorbar of interaction frequencies for the heatmap:
f.colorbar(
    acpm,
    cax=cax,
```

(continues on next page)

(continued from previous page)

```
    label='IC contact frequency'
)

# draw a grid around values:
ax.set_xticks(
    [x-0.5 for x in range(1,len(clr.chromnames))],
    minor=True
)
ax.set_yticks(
    [y-0.5 for y in range(1,len(clr.chromnames))],
    minor=True
)
ax.grid(which="minor")
```



### Impact of sequencing depth on computed P(s)

We explore the influence of sequencing coverage on calculating P(s) curves using cooltools.sample() to generate downsampled coolers from the Kreitenstein microC test dataset. We then compute raw P(s) and smoothed P(s) at various levels of downsampling, for the q arm of chr2.

Raw P(s) appear very noisy even with 0.1% downsampling. Smoothed P(s) are fairly consistent down to 0.01% downsampling. Derivatives, however, are less reliable at 0.01% downsampling.

[31]:

```
# create downsampled test data
downsampling_fracs = [0.0001, 0.001]
num_reads = {}
cvds = {}
cvds_smoothed = {}
derivs_smoothed = {}

p = Pool(num_cpus)
for frac in downsampling_fracs:
    cooltools.sample(clr, out_clr_path=f'./data/test_sampled_{frac}_.cool', frac=frac,␣
↪nproc=num_cpus)

    clr_downsampled = cooler.Cooler(f'./data/test_sampled_{frac}_.cool')
    result = cooler.balance_cooler(clr_downsampled, map=p.map, store=True, min_nnz=0)
p.close()
p.terminate()
```

```
INFO:root:creating a Pool of 10 workers
INFO:root:creating a Pool of 10 workers
```

[32]:

```
# expected & derivative calculation for raw data, using only the second arm of chr2
cvd_smooth_agg_raw = cooltools.expected_cis(
    clr=clr,
    view_df=hg38_arms[1:2],
    clr_weight_name=None,
    smooth=True,
    aggregate_smoothed=True,
    nproc=num_cpus
)

cvd_smooth_agg_raw['count.avg.smoothed'].loc[cvd_smooth_agg_raw['dist'] < 2] = np.nan
cvd_merged = cvd_smooth_agg_raw.drop_duplicates(subset=['dist'])[['dist_bp', 'count.avg.
↪smoothed.agg']]
der = np.gradient(np.log(cvd_merged['count.avg.smoothed.agg']),
                np.log(cvd_merged['dist_bp']))
deriv_smoothed_raw  = der


cvds[1.0] = cvd_smooth_agg_raw.copy()
cvds_smoothed[1.0]= cvd_smooth_agg_raw.copy()
num_reads[1.0] = cvd_smooth_agg_raw['count.sum'].sum().astype(int)
derivs_smoothed[1.0] = deriv_smoothed_raw
```

```
INFO:root:creating a Pool of 10 workers
```

[39]:

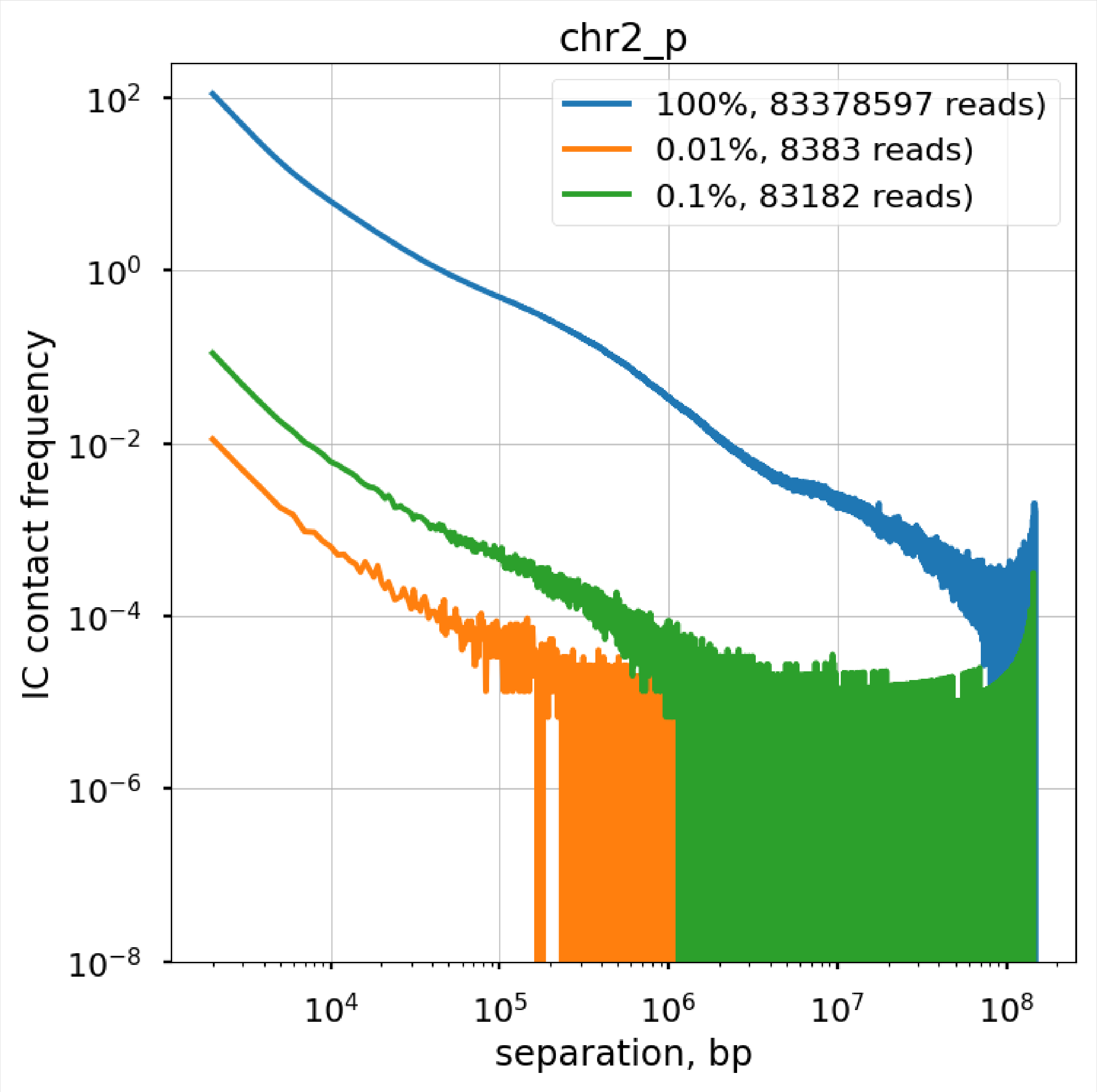
```
# expected calculation for the downsampled data
for frac in downsampling_fracs:
    clr_downsampled = cooler.Cooler(f'./data/test_sampled_{frac}_.cool')

    cvd_downsampled = cooltools.expected_cis(
        clr=clr_downsampled,
        view_df=hg38_arms[1:2],
        clr_weight_name=None,
        smooth=False,
        aggregate_smoothed=False,
        nproc=num_cpus #if you do not have multiple cores available, set to 1
    )
    cvds[frac] = cvd_downsampled
    num_reads[frac] = cvd_downsampled['count.sum'].sum().astype(int)
```

```
INFO:root:creating a Pool of 10 workers
```

```
INFO:root:creating a Pool of 10 workers
```

[42]:

```
f, ax = plt.subplots(1,1, figsize=(8, 8))

region = hg38_arms['name'][0]
for frac in [1]+downsampling_fracs:
    cvd_downsampled = cvds[frac]
    ax.loglog(
        cvd_downsampled['dist_bp'],
        cvd_downsampled['count.avg'],
        label=f'{frac*100}%, {num_reads[frac]} reads',
    )

ax.set(
    xlabel='separation, bp',
    ylabel='contact frequency')
ax.grid(lw=0.5)
ax.legend()
ax.title.set_text(region)
ax.set_ylim(ymin=10**(-8))
f.show()
```



[43]:
```
# expected calculation for the smoothed data
for frac in downsampling_fracs:
    clr_downsampled = cooler.Cooler(f'./data/test_sampled_{frac}_.cool')
    cvd_smooth_agg_downsampled = cooltools.expected_cis(
        clr=clr_downsampled,
        view_df=hg38_arms[1:2],
        clr_weight_name=None,
        smooth=True,
        aggregate_smoothed=True,
        nproc=num_cpus
    )
    cvd_smooth_agg_downsampled['count.avg.smoothed'].loc[cvd_smooth_agg_downsampled['dist
```

(continues on next page)

(continued from previous page)

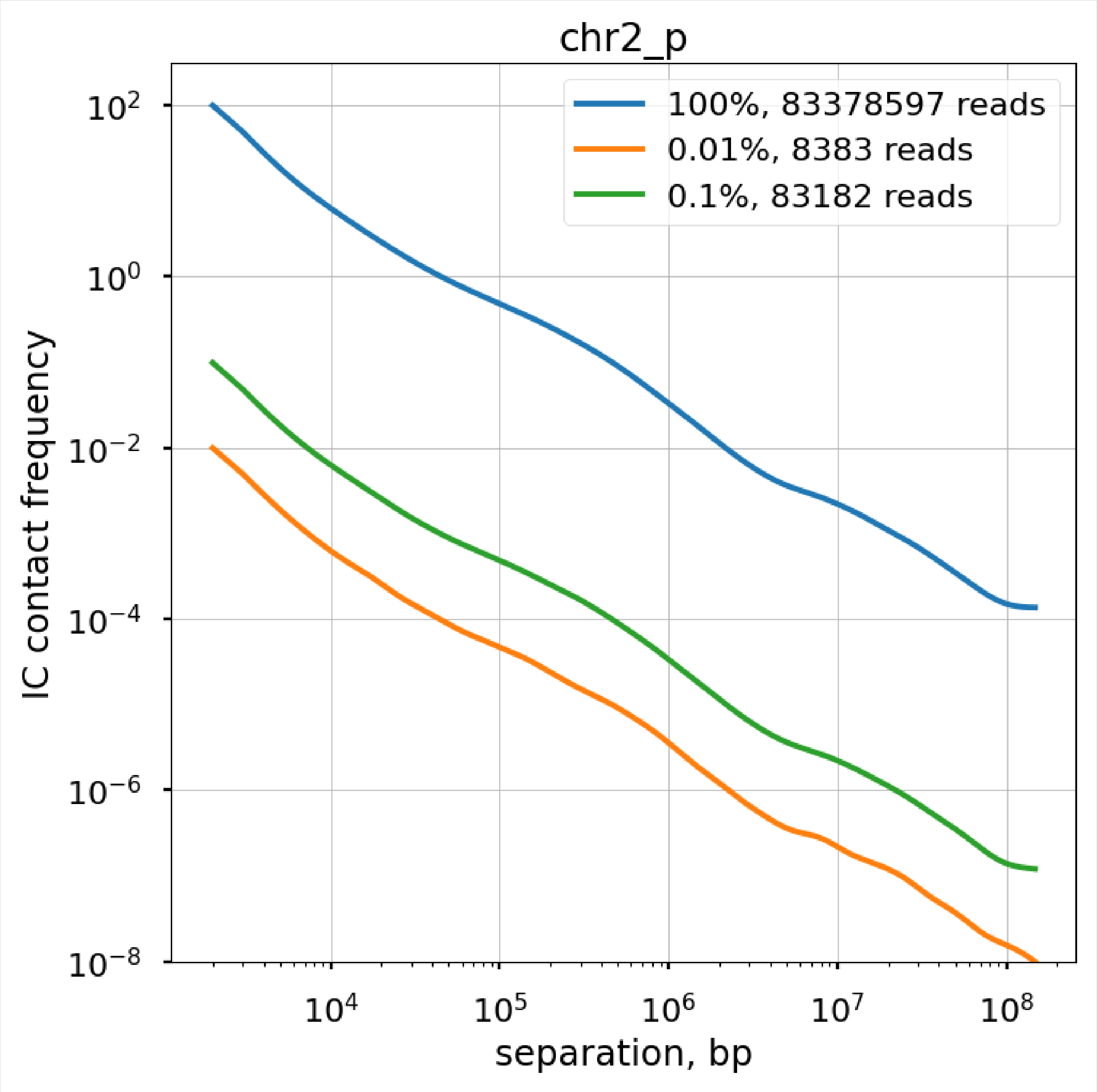
```
↪'] < 2] = np.nan
    cvds_smoothed[frac] = cvd_smooth_agg_downsampled

    cvd_merged = cvd_smooth_agg_downsampled.drop_duplicates(subset=['dist'])[['dist_bp',
↪'count.avg.smoothed.agg']]
    der = np.gradient(np.log(cvd_merged['count.avg.smoothed.agg']),
                    np.log(cvd_merged['dist_bp']))
    derivs_smoothed[frac] = der
```

```
INFO:root:creating a Pool of 10 workers
```

```
INFO:root:creating a Pool of 10 workers
```

[47]:

```
f, ax = plt.subplots(1,1,figsize=(8, 8))

for frac in [1]+downsampling_fracs:
    cvd_smooth_agg_downsampled = cvds_smoothed[frac]
    cvd_smooth_agg_downsampled['count.avg.smoothed'].loc[cvd_smooth_agg_downsampled['dist
↪'] < 2] = np.nan
    ax.loglog(
        cvd_smooth_agg_downsampled['dist_bp'],
        cvd_smooth_agg_downsampled['count.avg.smoothed'],
        label=f'{frac*100}%, {num_reads[frac]} reads')

ax.set(
    xlabel='separation, bp',
    ylabel='contact frequency')
ax.grid(lw=0.5)
ax.title.set_text(region)
ax.legend()
ax.set_ylim(ymin=10**(-8))
f.show()
```



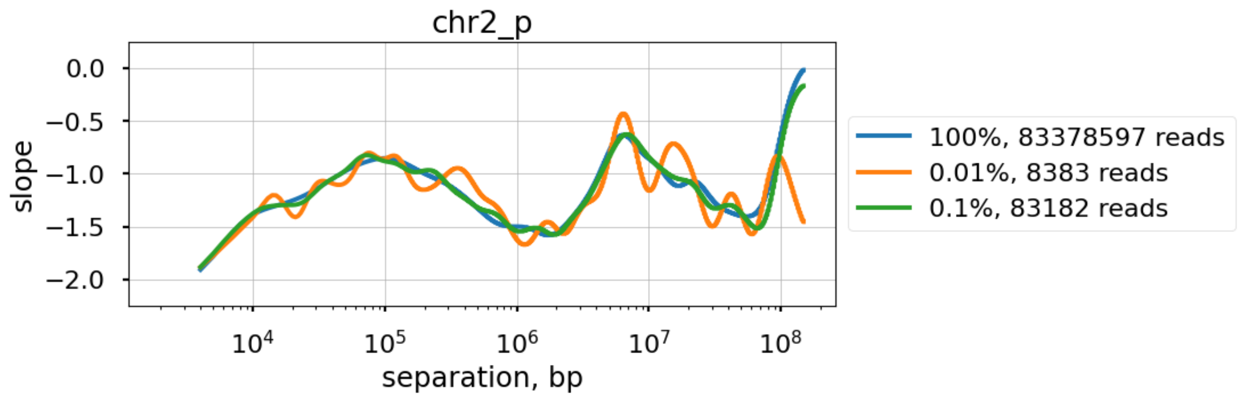
[56]:

```
f, ax = plt.subplots(1,1,figsize=(8,3))
for frac in [1]+downsampling_fracs:
    der = derivs_smoothed[frac]
    der[3] = np.nan
    ax.semilogx(
        cvd_smooth_agg_downsampled['dist_bp'],
        der,
        label=f'{frac*100}%, {num_reads[frac]} reads'
        )

ax.set(
    xlabel='separation, bp',
```

(continues on next page)

(continued from previous page)

```
    ylabel='slope')
ax.grid(lw=0.5)
ax.title.set_text(region)
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.ylim(-2.25,.25)
f.show()
```



This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.3 Compartments & Saddleplots

Welcome to the compartments and saddleplot notebook!

This notebook illustrates cooltools functions used for investigating chromosomal compartments, visible as plaid patterns in mammalian interphase contact frequency maps.

These plaid patterns reflect tendencies of chromosome regions to make more frequent contacts with regions of the same type: active regions have increased contact frequency with other active regions, and intactive regions tend to contact other inactive regions more frequently. The strength of compartmentalization has been show to vary through the cell cycle, across cell types, and after degredation of components of the cohesin complex.

In this notebook we:

- obtain compartment profiles using eigendecomposition
- calculate and visualize strength of compartmentalization using saddleplots

```
[1]: # import standard python libraries
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import os, subprocess
```

```
[2]: # Import python package for working with cooler files and tools for analysis
import cooler
import cooltools.lib.plotting
```

[3]:
```
from packaging import version
if version.parse(cooltools.__version__) < version.parse('0.5.4'):
    raise AssertionError("tutorials rely on cooltools version 0.5.4 or higher,"+
                         "please check your cooltools version and update to the latest")
```

[4]:
```
# download test data
# this file is 145 Mb, and may take a few seconds to download
import cooltools
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir='./data/')
print(cool_file)
```

```
./data/test.mcool
```

### Calculating per-chromosome compartmentalization

We first load the Hi-C data at 100 kbp resolution.

Note that the current implementation of eigendecomposition in cooltools assumes that individual regions can be held in memory– for hg38 at 100kb this is either a 2422x2422 matrix for chr2, or a 3255x3255 matrix for the full cooler here.

[5]:
```
clr = cooler.Cooler('./data/test.mcool::resolutions/100000')
```

Since the orientation of eigenvectors is determined up to a sign, the convention for Hi-C data anaylsis is to orient eigenvectors to be positively correlated with a binned profile of GC content as a ‘phasing track’.

In humans and mice, GC content is useful for phasing because it typically has a strong correlation at the 100kb-1Mb bin level with the eigenvector. In other organisms, other phasing tracks have been used to orient eigenvectors from Hi-C data.

For other data analyses, different conventions are used to consistently orient eigenvectors. For example, spectral clustering as implemented in scikit-learn orients vectors such that the absolute maximum element of each vector is positive.

[ ]:
```
## fasta sequence is required for calculating binned profile of GC conent
if not os.path.isfile('./hg38.fa'):
    ## note downloading a ~1Gb file can take a minute
    subprocess.call('wget --progress=bar:force:noscroll https://hgdownload.cse.ucsc.edu/
↪goldenpath/hg38/bigZips/hg38.fa.gz', shell=True)
    subprocess.call('gunzip hg38.fa.gz', shell=True)
```

[7]:
```
import bioframe
bins = clr.bins()[:]
hg38_genome = bioframe.load_fasta('./hg38.fa');
## note the next command may require installing pysam
gc_cov = bioframe.frac_gc(bins[['chrom', 'start', 'end']], hg38_genome)
gc_cov.to_csv('hg38_gc_cov_100kb.tsv',index=False,sep='\t')
display(gc_cov)
```

```
      chrom      start        end        GC
0      chr2          0     100000  0.435867
1      chr2     100000     200000  0.409530
2      chr2     200000     300000  0.421890
3      chr2     300000     400000  0.431870
4      chr2     400000     500000  0.458610
```

(continues on next page)

(continued from previous page)

```
...     ...        ...       ...       ...
3250  chr17  82800000  82900000  0.528210
3251  chr17  82900000  83000000  0.518530
3252  chr17  83000000  83100000  0.561450
3253  chr17  83100000  83200000  0.535119
3254  chr17  83200000  83257441  0.473451

[3255 rows x 4 columns]
```

Cooltools also allows a view to be passed for eigendecomposition to limit to a certain set of regions. The following code creates the simplest view, of the two chromosomes in this cooler.

[8]:

```
view_df = pd.DataFrame({'chrom': clr.chromnames,
                        'start': 0,
                        'end': clr.chromsizes.values,
                        'name': clr.chromnames}
                      )
display(view_df)
```

```
   chrom  start        end   name
0   chr2      0  242193529   chr2
1  chr17      0   83257441  chr17
```

To capture the pattern of compartmentalization within-chromosomes, in cis, cooltools `eigs_cis` first removes the dependence of contact frequency by distance, and then performs eigenedecompostion.

[9]:

```
# obtain first 3 eigenvectors
cis_eigs = cooltools.eigs_cis(
                        clr,
                        gc_cov,
                        view_df=view_df,
                        n_eigs=3,
                        )

# cis_eigs[0] returns eigenvalues, here we focus on eigenvectors
eigenvector_track = cis_eigs[1][['chrom','start','end','E1']]
```

Plotting the first eigenvector next to the Hi-C map allows us to see how this captures the plaid pattern.

To better visualize this relationship, we overlay the map with a binary segmentation of the eigenvector. Eigenvectors can be segmented by a variety of methods. The simplest segmentation, shown here, is to simply binarize eigenvectors, and term everything above zero the "A-compartment" and everything below 0 the "B-compartment".

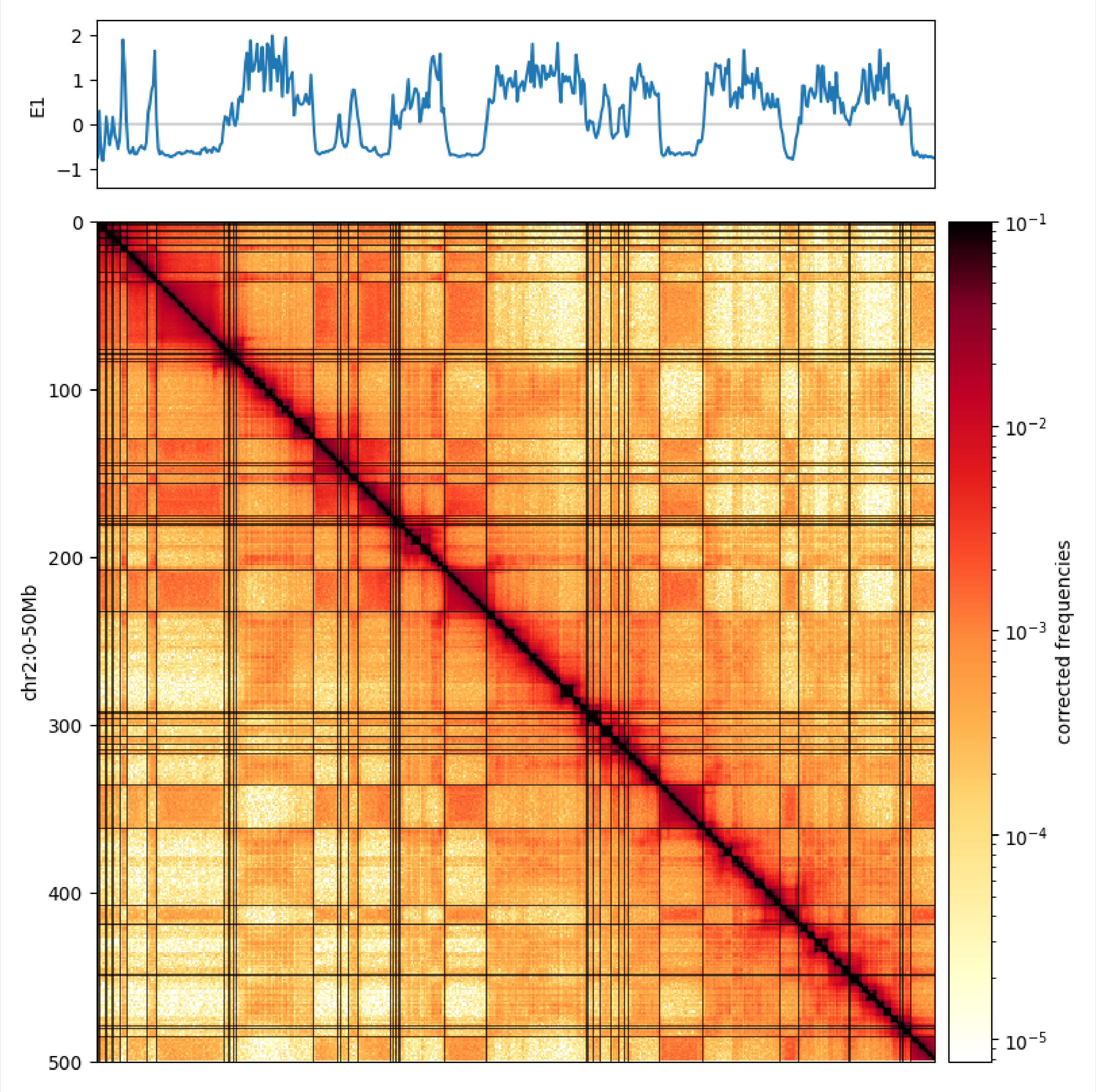
[10]:

```
from matplotlib.colors import LogNorm
from mpl_toolkits.axes_grid1 import make_axes_locatable

f, ax = plt.subplots(
    figsize=(15, 10),
)

norm = LogNorm(vmax=0.1)

im = ax.matshow(
    clr.matrix()[:],
```

(continues on next page)

(continued from previous page)

```
    norm=norm,
    cmap='fall'
);
plt.axis([0,500,500,0])

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.1)
plt.colorbar(im, cax=cax, label='corrected frequencies');
ax.set_ylabel('chr2:0-50Mb')
ax.xaxis.set_visible(False)

ax1 = divider.append_axes("top", size="20%", pad=0.25, sharex=ax)
weights = clr.bins()[:]['weight'].values
ax1.plot([0,500],[0,0],'k',lw=0.25)
ax1.plot( eigenvector_track['E1'].values, label='E1')

ax1.set_ylabel('E1')
ax1.set_xticks([]);


for i in np.where(np.diff( (cis_eigs[1]['E1']>0).astype(int)))[0]:
    ax.plot([0, 500],[i,i],'k',lw=0.5)
    ax.plot([i,i],[0, 500],'k',lw=0.5)
```



### Saddleplots

A common way to visualize preferences captured by the eigenvector is by using saddleplots.

To generate a saddleplot, we first use the eigenvector to stratify genomic regions into groups with similar values of the eigenvector. These groups are then averaged over to create the saddleplot. This process is called "digitizing".

Cooltools will operate with `digitized` bedgraph-like track with four columns. The fourth, or value, column is a categorical, as shown above for the first three bins. Categories have the following encoding:

```
- `1..n` <-> values assigned to bins defined by vrange or qrange
- `0` <-> left outlier values
```

(continues on next page)

(continued from previous page)

```
- `n+1` <-> right outlier values
- `-1` <-> missing data (NaNs)
```

Track values can either be digitized by numeric values, by passing `vrange`, or by quantiles, by passing `qrange`, as above.

To create saddles in cis with `saddle`, cooltools requires: a cooler, a table with expected as function of distance, and parameters for digitizing:

[11]:
```
cvd = cooltools.expected_cis(
        clr=clr,
        view_df=view_df,
)
```

[12]:
```
Q_LO = 0.025 # ignore 2.5% of genomic bins with the lowest E1 values
Q_HI = 0.975 # ignore 2.5% of genomic bins with the highest E1 values
N_GROUPS = 38 # divide remaining 95% of the genome into 38 equisized groups, 2.5% each
```

`saddle` then returns two matrices: one with the sum for each pair of categories, `interaction_sum`, and the other with the number of bins for each pair of categories, `interaction_count`. Typically, `interaction_sum/interaction_count` is visualized.

[13]:
```
interaction_sum, interaction_count =  cooltools.saddle(
        clr,
        cvd,
        eigenvector_track,
        'cis',
        n_bins=N_GROUPS,
        qrange=(Q_LO,Q_HI),
        view_df=view_df
)
```

There are multiple ways to plot saddle data, one useful way is shown below.

This visualization includes histograms of the number of bins contributing to each row/column of the saddleplot.

[14]:
```
import warnings
from cytoolz import merge

def saddleplot(
    track,
    saddledata,
    n_bins,
    vrange=None,
    qrange=(0.0, 1.0),
    cmap="coolwarm",
    scale="log",
    vmin=0.5,
    vmax=2,
    color=None,
    title=None,
    xlabel=None,
    ylabel=None,
```

(continues on next page)

(continued from previous page)

```
    clabel=None,
    fig=None,
    fig_kws=None,
    heatmap_kws=None,
    margin_kws=None,
    cbar_kws=None,
    subplot_spec=None,
):
    """
    Generate a saddle plot.
    Parameters
    ----------
    track : pd.DataFrame
        See cooltools.digitize() for details.
    saddledata : 2D array-like
        Saddle matrix produced by `make_saddle`. It will include 2 flanking
        rows/columns for outlier signal values, thus the shape should be
        `(n+2, n+2)`.
    cmap : str or matplotlib colormap
        Colormap to use for plotting the saddle heatmap
    scale : str
        Color scaling to use for plotting the saddle heatmap: log or linear
    vmin, vmax : float
        Value limits for coloring the saddle heatmap
    color : matplotlib color value
        Face color for margin bar plots
    fig : matplotlib Figure, optional
        Specified figure to plot on. A new figure is created if none is
        provided.
    fig_kws : dict, optional
        Passed on to `plt.Figure()`
    heatmap_kws : dict, optional
        Passed on to `ax.imshow()`
    margin_kws : dict, optional
        Passed on to `ax.bar()` and `ax.barh()`
    cbar_kws : dict, optional
        Passed on to `plt.colorbar()`
    subplot_spec : GridSpec object
        Specify a subregion of a figure to using a GridSpec.
    Returns
    -------
    Dictionary of axes objects.
    """

#     warnings.warn(
#         "Generating a saddleplot will be deprecated in future versions, "
#         + "please see https://github.com/open2c_examples for examples on how to plot␣
↪saddles.",
#         DeprecationWarning,
#     )

    from matplotlib.gridspec import GridSpec, GridSpecFromSubplotSpec
```

(continues on next page)

(continued from previous page)

```
    from matplotlib.colors import Normalize, LogNorm
    from matplotlib import ticker
    import matplotlib.pyplot as plt

    class MinOneMaxFormatter(ticker.LogFormatter):
        def set_locs(self, locs=None):
            self._sublabels = set([vmin % 10 * 10, vmax % 10, 1])

        def __call__(self, x, pos=None):
            if x not in [vmin, 1, vmax]:
                return ""
            else:
                return "{x:g}".format(x=x)

    track_value_col = track.columns[3]
    track_values = track[track_value_col].values

    digitized_track, binedges = cooltools.digitize(
        track, n_bins, vrange=vrange, qrange=qrange
    )
    x = digitized_track[digitized_track.columns[3]].values.astype(int).copy()
    x = x[(x > -1) & (x < len(binedges) + 1)]

    # Old version
    # hist = np.bincount(x, minlength=len(binedges) + 1)

    groupmean = track[track.columns[3]].groupby(digitized_track[digitized_track.
↪columns[3]]).mean()

    if qrange is not None:
        lo, hi = qrange
        binedges = np.linspace(lo, hi, n_bins + 1)

    # Barplot of mean values and saddledata are flanked by outlier bins
    n = saddledata.shape[0]
    X, Y = np.meshgrid(binedges, binedges)
    C = saddledata
    if (n - n_bins) == 2:
        C = C[1:-1, 1:-1]
        groupmean = groupmean[1:-1]

    # Layout
    if subplot_spec is not None:
        GridSpec = partial(GridSpecFromSubplotSpec, subplot_spec=subplot_spec)
    grid = {}
    gs = GridSpec(
        nrows=3,
        ncols=3,
        width_ratios=[0.2, 1, 0.1],
        height_ratios=[0.2, 1, 0.1],
        wspace=0.05,
        hspace=0.05,
```

(continues on next page)

(continued from previous page)

```
    )

    # Figure
    if fig is None:
        fig_kws_default = dict(figsize=(5, 5))
        fig_kws = merge(fig_kws_default, fig_kws if fig_kws is not None else {})
        fig = plt.figure(**fig_kws)

    # Heatmap
    if scale == "log":
        norm = LogNorm(vmin=vmin, vmax=vmax)
    elif scale == "linear":
        norm = Normalize(vmin=vmin, vmax=vmax)
    else:
        raise ValueError("Only linear and log color scaling is supported")

    grid["ax_heatmap"] = ax = plt.subplot(gs[4])
    heatmap_kws_default = dict(cmap="coolwarm", rasterized=True)
    heatmap_kws = merge(
        heatmap_kws_default, heatmap_kws if heatmap_kws is not None else {}
    )
    img = ax.pcolormesh(X, Y, C, norm=norm, **heatmap_kws)
    plt.gca().yaxis.set_visible(False)

    # Margins
    margin_kws_default = dict(edgecolor="k", facecolor=color, linewidth=1)
    margin_kws = merge(margin_kws_default, margin_kws if margin_kws is not None else {})
    # left margin hist
    grid["ax_margin_y"] = plt.subplot(gs[3], sharey=grid["ax_heatmap"])

    plt.barh(
        binedges, height=1/len(binedges), width=groupmean, align="edge", **margin_kws
    )

    plt.xlim(plt.xlim()[1], plt.xlim()[0])  # fliplr
    plt.ylim(hi, lo)
    plt.gca().spines["top"].set_visible(False)
    plt.gca().spines["bottom"].set_visible(False)
    plt.gca().spines["left"].set_visible(False)
    plt.gca().xaxis.set_visible(False)
    # top margin hist
    grid["ax_margin_x"] = plt.subplot(gs[1], sharex=grid["ax_heatmap"])

    plt.bar(
        binedges, width=1/len(binedges), height=groupmean, align="edge", **margin_kws
    )

    plt.xlim(lo, hi)
    # plt.ylim(plt.ylim())  # correct
    plt.gca().spines["top"].set_visible(False)
    plt.gca().spines["right"].set_visible(False)
    plt.gca().spines["left"].set_visible(False)
```

(continues on next page)

(continued from previous page)

```
    plt.gca().xaxis.set_visible(False)
    plt.gca().yaxis.set_visible(False)

#     # Colorbar
    grid["ax_cbar"] = plt.subplot(gs[5])
    cbar_kws_default = dict(fraction=0.8, label=clabel or "")
    cbar_kws = merge(cbar_kws_default, cbar_kws if cbar_kws is not None else {})
    if scale == "linear" and vmin is not None and vmax is not None:
        grid["ax_cbar"] = cb = plt.colorbar(img, **cbar_kws)
        # cb.set_ticks(np.arange(vmin, vmax + 0.001, 0.5))
        # # do linspace between vmin and vmax of 5 segments and trunc to 1 decimal:
        decimal = 10
        nsegments = 5
        cd_ticks = np.trunc(np.linspace(vmin, vmax, nsegments) * decimal) / decimal
        cb.set_ticks(cd_ticks)
    else:
        print('cbar')

        cb = plt.colorbar(img, format=MinOneMaxFormatter(), cax=grid["ax_cbar"], **cbar_
↪kws)
        cb.ax.yaxis.set_minor_formatter(MinOneMaxFormatter())

    # extra settings
    grid["ax_heatmap"].set_xlim(lo, hi)
    grid["ax_heatmap"].set_ylim(hi, lo)
    grid['ax_heatmap'].grid(False)
    if title is not None:
        grid["ax_margin_x"].set_title(title)
    if xlabel is not None:
        grid["ax_heatmap"].set_xlabel(xlabel)
    if ylabel is not None:
        grid["ax_margin_y"].set_ylabel(ylabel)

    return grid
```

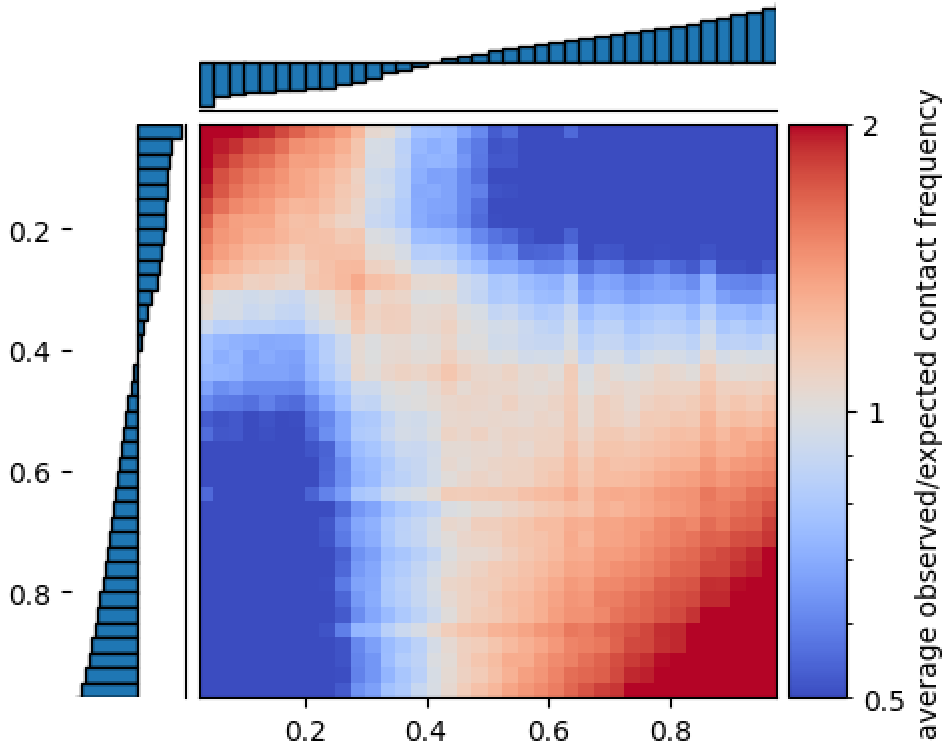
The saddle below shows average observed/expected contact frequency between regions grouped according to their digitized eigenvector values with a blue-to-white-to-red colormap. Inactive regions (i.e. low digitized values) are on the top and left, and active regions (i.e. high digitized values) are on the bottom and right.

The saddleplot shows that inactive regions are enriched for contact frequency with other inactive regions (red area in the upper left), and active regions are enriched for contact frequency with other active regions (red area in the lower right). In contrast, active regions are depleted for contact frequency with inactive regions (blue area in top right and bottom left).

[15]:

```
saddleplot(eigenvector_track,
           interaction_sum/interaction_count,
           N_GROUPS,
           qrange=(Q_LO,Q_HI),
           cbar_kws={'label':'average observed/expected contact frequency'}
          );
```

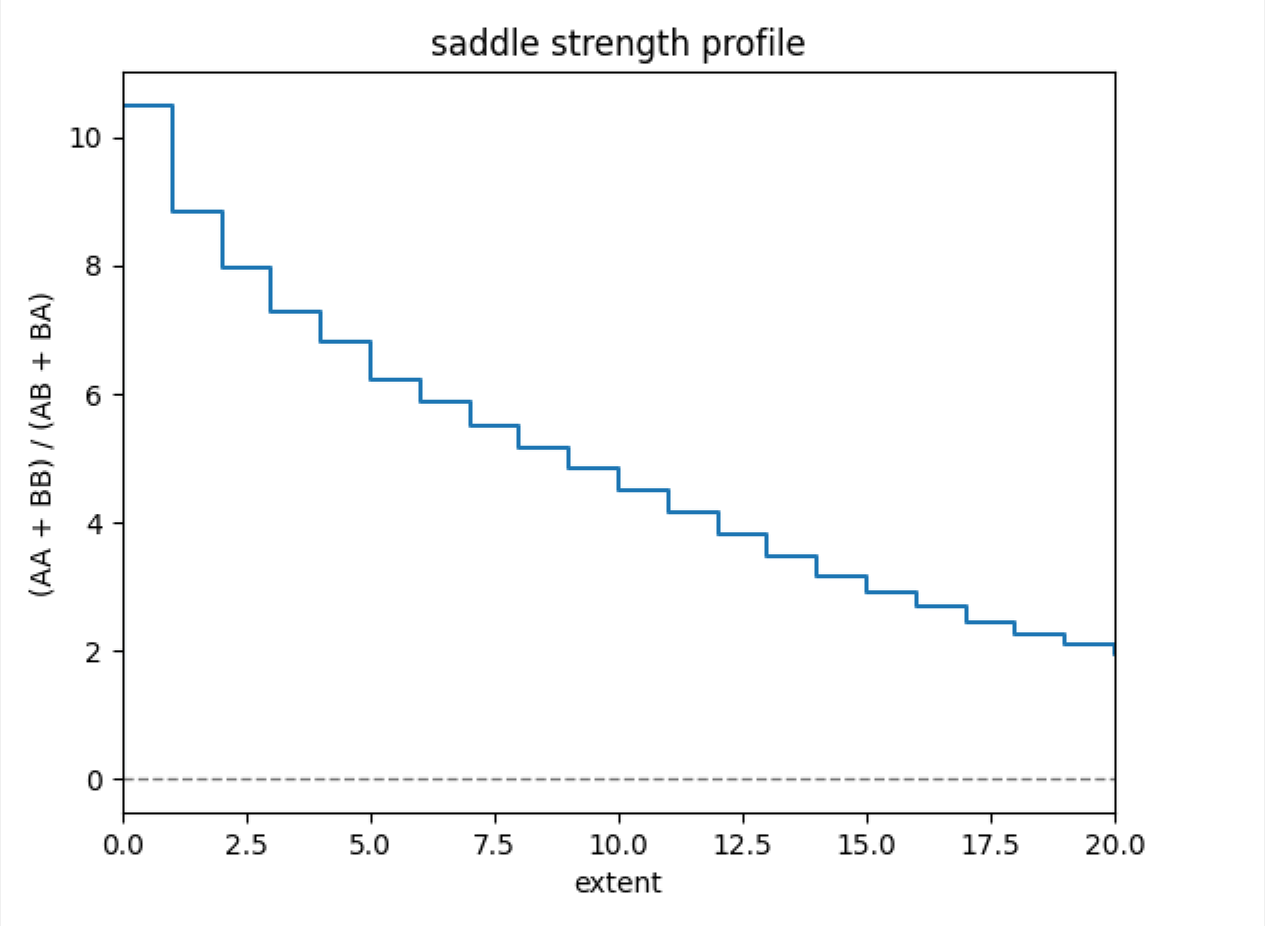
```
cbar
```



### Saddle strength

Comparing the average obs/expected values between active and inactive chromatin, is one useful measure of the strength of compartmentalization. This can be measured with `saddle_strength()`, which can be thought of as taking the ratio between (AA+BB) / (AB+BA). This corresponds visually to the ratio between the upper left and lower right corners, versus the lower left and upper right corners in the plot above.

```
[16]: from cooltools.api.saddle import saddle_strength
      # at extent=0, this reduces to ((S/C)[0,0] + (S/C)[-1,-1]) / (2*(S/C)[-1,0])

      x = np.arange(N_GROUPS + 2)

      plt.step(x, saddle_strength(interaction_sum, interaction_count), where='pre')

      plt.xlabel('extent')
      plt.ylabel('(AA + BB) / (AB + BA)')
      plt.title('saddle strength profile')
      plt.axhline(0, c='grey', ls='--', lw=1) # Q: is there a reason this is 0 not 1?
      plt.xlim(0, len(x)//2); # Q: is this less intuitive than showing for all x, as it␣
      ↪converges to no difference (i.e. 1)?
```



This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.4 Insulation & boundaries

Welcome to the contact insulation notebook!

Insulation is a simple concept, yet a powerful way to look at C data. Insulation is one aspect of locus-specific contact frequency at small genomic distances, and reflects the segmentation of the genome into domains.

Insulation can be computed with multiple methods. One of the most common methods involves using a diamond-window score to generate an **insulation profile**. To compute this profile, slide a diamond-shaped window along the genome, with one of the corners on the main diagonal of the matrix, and sum up the contacts within the window for each position.

Insulation profiles reveal that certain locations have lower scores, reflecting lowered contact frequencies between upstream and downstream loci. These positions are often referred to as **boundaries**, and are also obtained with multiple methods. Here we illustrate one thresholding method for determining boundaries from an insulation profile.

In this notebook we:

- Calculate the insulation score genome-wide and display it alongside an interaction matrix
- Call insulating boundaries
- Filter insulating boundaries based on their strength

- Calculate enrichment of CTCF/genes at boundaries
- Repeat boundary filtering based on enrichmnent of CTCF, a known insulator protein in mammalian genomes

[1]:
```python
# import standard python libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[2]:
```python
# Import python package for working with cooler files and tools for analysis
import cooler
import cooltools.lib.plotting
from cooltools import insulation

from packaging import version
if version.parse(cooltools.__version__) < version.parse('0.5.4'):
    raise AssertionError("tutorials rely on cooltools version 0.5.4 or higher,"+
                         "please check your cooltools version and update to the latest")
```

[3]:
```python
# download test data
# this file is 145 Mb, and may take a few seconds to download
import cooltools
data_dir = './data/'
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir=data_dir)
print(cool_file)
```

```
./data/test.mcool
```

### Calculating genome-wide contact insulation

Here we load the Hi-C data at 10 kbp resolution and calculate insulation score with 4 different window sizes

[4]:
```python
resolution = 10000
clr = cooler.Cooler(f'{data_dir}test.mcool::resolutions/{resolution}')
windows = [3*resolution, 5*resolution, 10*resolution, 25*resolution]
insulation_table = insulation(clr, windows, verbose=True)
```

```
INFO:root:fallback to serial implementation.
INFO:root:Processing region chr2
INFO:root:Processing region chr17
```

This function returns a dataframe where rows correspond to genomic bins of the cooler.

The columns of this insulation dataframe report the insulation score, the number of valid (non-nan) pixels, whether the given bin is valid, the boundary prominence (strength) and whether locus is called as a boundary after thresholding, for each of the window sizes provided to the function.

Below we print the information returned for any window size, as well as the specific information for the largest window used:

[5]:
```python
first_window_summary =insulation_table.columns[[ str(windows[-1]) in i for i in␣
↪insulation_table.columns]]

insulation_table[['chrom','start','end','region','is_bad_bin']+list(first_window_
↪summary)].iloc[1000:1005]
```

[5]:

```
     chrom     start       end region  is_bad_bin  \
1000  chr2  10000000  10010000   chr2       False
1001  chr2  10010000  10020000   chr2       False
1002  chr2  10020000  10030000   chr2       False
1003  chr2  10030000  10040000   chr2       False
1004  chr2  10040000  10050000   chr2       False

      log2_insulation_score_250000  n_valid_pixels_250000  \
1000                      0.309791                  622.0
1001                      0.226045                  622.0
1002                      0.090809                  622.0
1003                     -0.101091                  622.0
1004                     -0.342858                  622.0

      boundary_strength_250000  is_boundary_250000
1000                       NaN               False
1001                       NaN               False
1002                       NaN               False
1003                       NaN               False
1004                       NaN               False
```

[6]:

```
# Functions to help with plotting
def pcolormesh_45deg(ax, matrix_c, start=0, resolution=1, *args, **kwargs):
    start_pos_vector = [start+resolution*i for i in range(len(matrix_c)+1)]
    import itertools
    n = matrix_c.shape[0]
    t = np.array([[1, 0.5], [-1, 0.5]])
    matrix_a = np.dot(np.array([(i[1], i[0])
                                for i in itertools.product(start_pos_vector[::-1],
                                                           start_pos_vector)]), t)
    x = matrix_a[:, 1].reshape(n + 1, n + 1)
    y = matrix_a[:, 0].reshape(n + 1, n + 1)
    im = ax.pcolormesh(x, y, np.flipud(matrix_c), *args, **kwargs)
    im.set_rasterized(True)
    return im

from matplotlib.ticker import EngFormatter
bp_formatter = EngFormatter('b')
def format_ticks(ax, x=True, y=True, rotate=True):
    if y:
        ax.yaxis.set_major_formatter(bp_formatter)
    if x:
        ax.xaxis.set_major_formatter(bp_formatter)
        ax.xaxis.tick_bottom()
    if rotate:
        ax.tick_params(axis='x',rotation=45)
```

Let’s see what the insulation track at the highest resolution looks like, next to a rotated Hi-C matrix.

[7]:

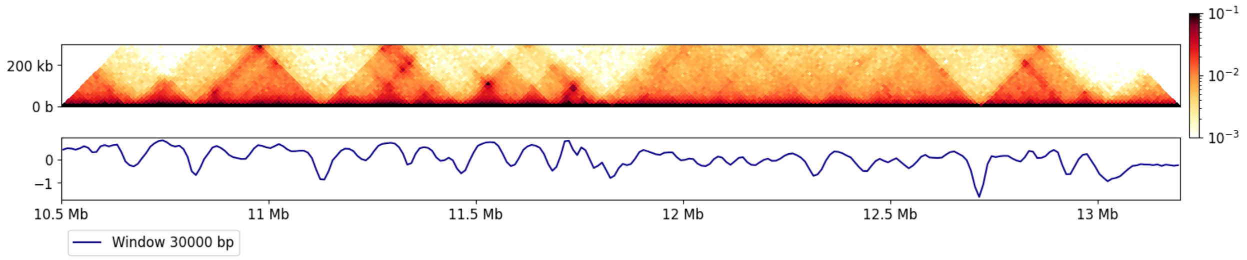
```
from matplotlib.colors import LogNorm
from mpl_toolkits.axes_grid1 import make_axes_locatable
import bioframe
plt.rcParams['font.size'] = 12
```

(continues on next page)

(continued from previous page)

```
start = 10_500_000
end = start+ 90*windows[0]
region = ('chr2', start, end)
norm = LogNorm(vmax=0.1, vmin=0.001)
data = clr.matrix(balance=True).fetch(region)
f, ax = plt.subplots(figsize=(18, 6))
im = pcolormesh_45deg(ax, data, start=region[1], resolution=resolution, norm=norm, cmap=
↪'fall')
ax.set_aspect(0.5)
ax.set_ylim(0, 10*windows[0])
format_ticks(ax, rotate=False)
ax.xaxis.set_visible(False)

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="1%", pad=0.1, aspect=6)
plt.colorbar(im, cax=cax)

insul_region = bioframe.select(insulation_table, region)

ins_ax = divider.append_axes("bottom", size="50%", pad=0., sharex=ax)
ins_ax.set_prop_cycle(plt.cycler("color", plt.cm.plasma(np.linspace(0,1,5))))
ins_ax.plot(insul_region[['start', 'end']].mean(axis=1),
            insul_region['log2_insulation_score_'+str(windows[0])],
            label=f'Window {windows[0]} bp')

ins_ax.legend(bbox_to_anchor=(0., -1), loc='lower left', ncol=4);

format_ticks(ins_ax, y=False, rotate=False)
ax.set_xlim(region[1], region[2])
```

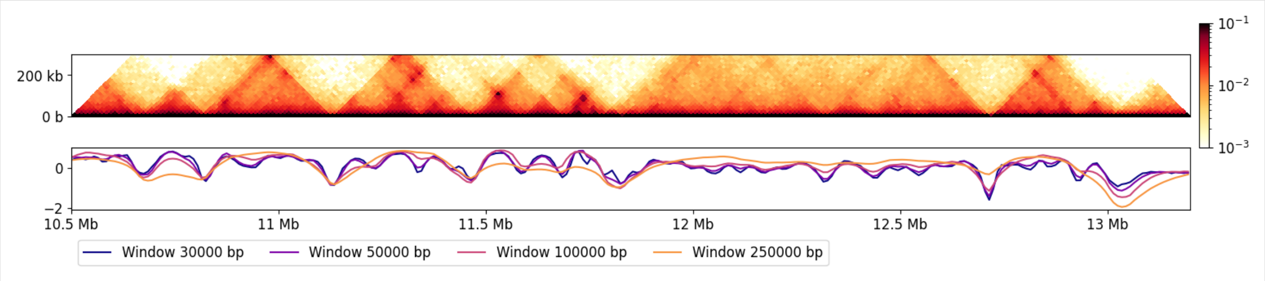
[7]:
```
(10500000.0, 13200000.0)
```



And now let's add the other window sizes.

[8]:
```
for res in windows[1:]:
    ins_ax.plot(insul_region[['start', 'end']].mean(axis=1), insul_region[f'log2_
↪insulation_score_{res}'], label=f'Window {res} bp')
ins_ax.legend(bbox_to_anchor=(0., -1), loc='lower left', ncol=4);
f
```

[8]:



This really highlights how much the result is dependent on window size: smaller windows are sensitive to local structure, whereas large windows capture regions that insulate at larger scales.

### Boundary calling

The insulation table also has annotations for valleys of the insulation score, which correspond to highly insulating regions, such as TAD boundaries. All potential boundaries have an assigned `boundary_strength_` column. Additionally, this strength is thresholded to find regions that insulate particularly strongly, and this is recorded in the `is_boundary_` columns.

Let's repeat the previous plot and show where we found the boundaries:

[9]:

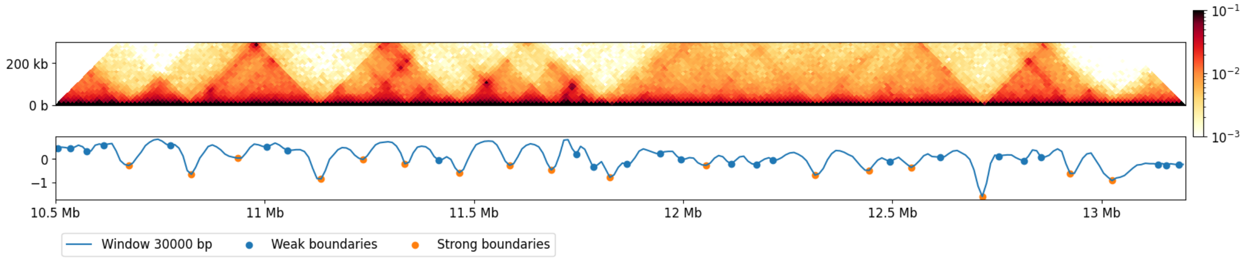
```
f, ax = plt.subplots(figsize=(20, 10))
im = pcolormesh_45deg(ax, data, start=region[1], resolution=resolution, norm=norm, cmap=
↪'fall')
ax.set_aspect(0.5)
ax.set_ylim(0, 10*windows[0])
format_ticks(ax, rotate=False)
ax.xaxis.set_visible(False)

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="1%", pad=0.1, aspect=6)
plt.colorbar(im, cax=cax)

insul_region = bioframe.select(insulation_table, region)

ins_ax = divider.append_axes("bottom", size="50%", pad=0., sharex=ax)

ins_ax.plot(insul_region[['start', 'end']].mean(axis=1),
            insul_region[f'log2_insulation_score_{windows[0]}'], label=f'Window
↪{windows[0]} bp')

boundaries = insul_region[~np.isnan(insul_region[f'boundary_strength_{windows[0]}'])]
weak_boundaries = boundaries[~boundaries[f'is_boundary_{windows[0]}']]
strong_boundaries = boundaries[boundaries[f'is_boundary_{windows[0]}']]
ins_ax.scatter(weak_boundaries[['start', 'end']].mean(axis=1),
            weak_boundaries[f'log2_insulation_score_{windows[0]}'], label='Weak␣
↪boundaries')
ins_ax.scatter(strong_boundaries[['start', 'end']].mean(axis=1),
            strong_boundaries[f'log2_insulation_score_{windows[0]}'], label='Strong␣
↪boundaries')

ins_ax.legend(bbox_to_anchor=(0., -1), loc='lower left', ncol=4);
```

(continues on next page)

(continued from previous page)

```
format_ticks(ins_ax, y=False, rotate=False)
ax.set_xlim(region[1], region[2])
```

```
[9]: (10500000.0, 13200000.0)
```

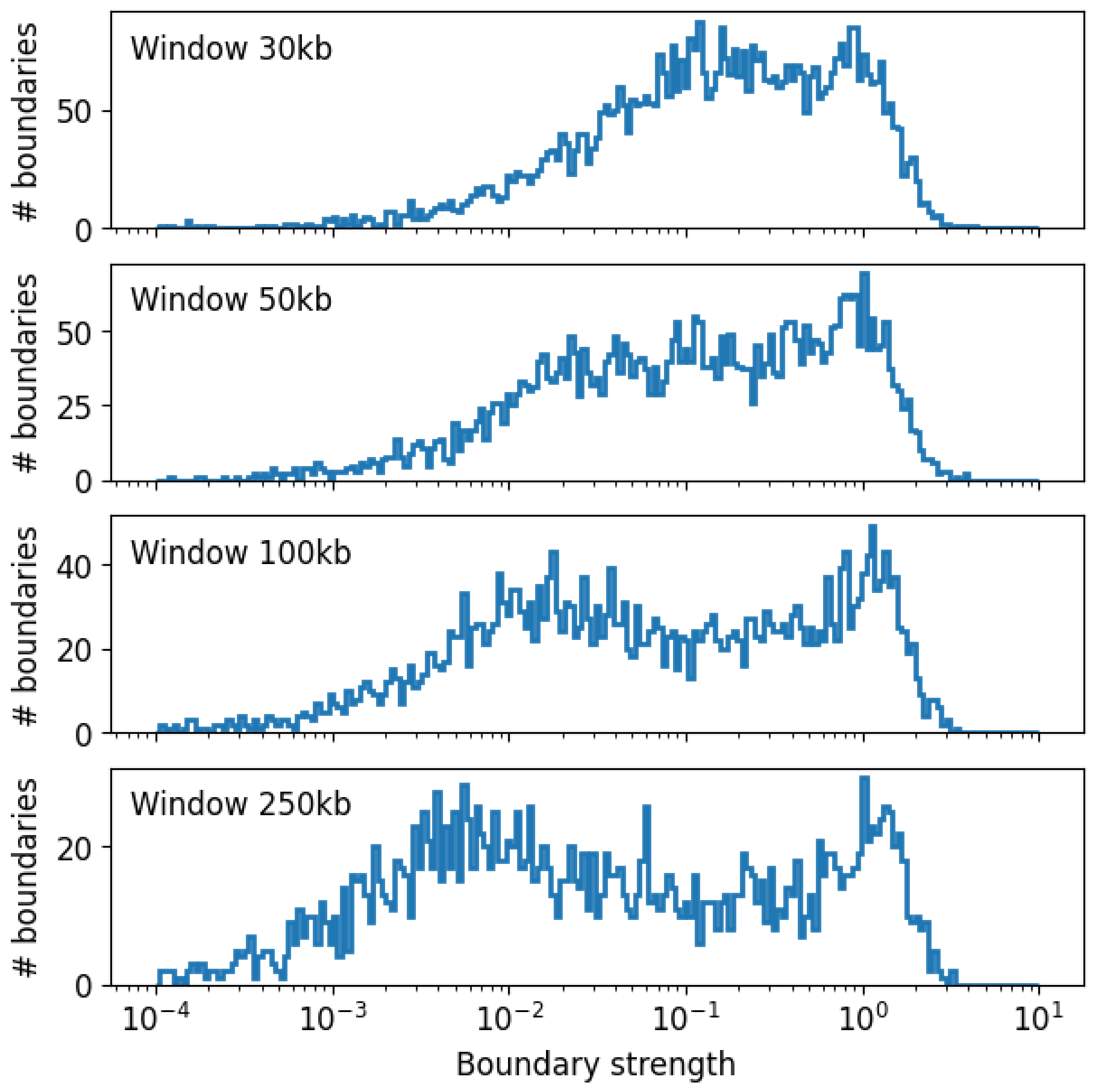


### Calculating boundary strength

Let's inspect the histogram of boundary strengths to show how we selected the strong boundaries.

First, boundary strength is calculated using the peak prominence, on the dips (or minima) of the insulation profile.

```
[10]: histkwargs = dict(
    bins=10**np.linspace(-4,1,200),
    histtype='step',
    lw=2,
)

f, axs = plt.subplots(len(windows),1, sharex=True, figsize=(6,6), constrained_
↪layout=True)
for i, (w, ax) in enumerate(zip(windows, axs)):
    ax.hist(
        insulation_table[f'boundary_strength_{w}'],
        **histkwargs
    )
    ax.text(0.02, 0.9,
             f'Window {w//1000}kb',
             ha='left',
             va='top',
             transform=ax.transAxes)

    ax.set(
        xscale='log',
        ylabel='# boundaries'
    )

axs[-1].set(xlabel='Boundary strength');
```



As a quick way to automatically threshold the histogram, we borrow the tresholding methods from the image analysis field. These include Li (default `threshold="Li"`) or Otsu, as implemented in scikit-image. Otsu is more conservative, whereas Li is more permissive.

In practice these thresholds work well for a simple parameter-free method for mammalian interphase data, though should be double-checked for any individual dataset.

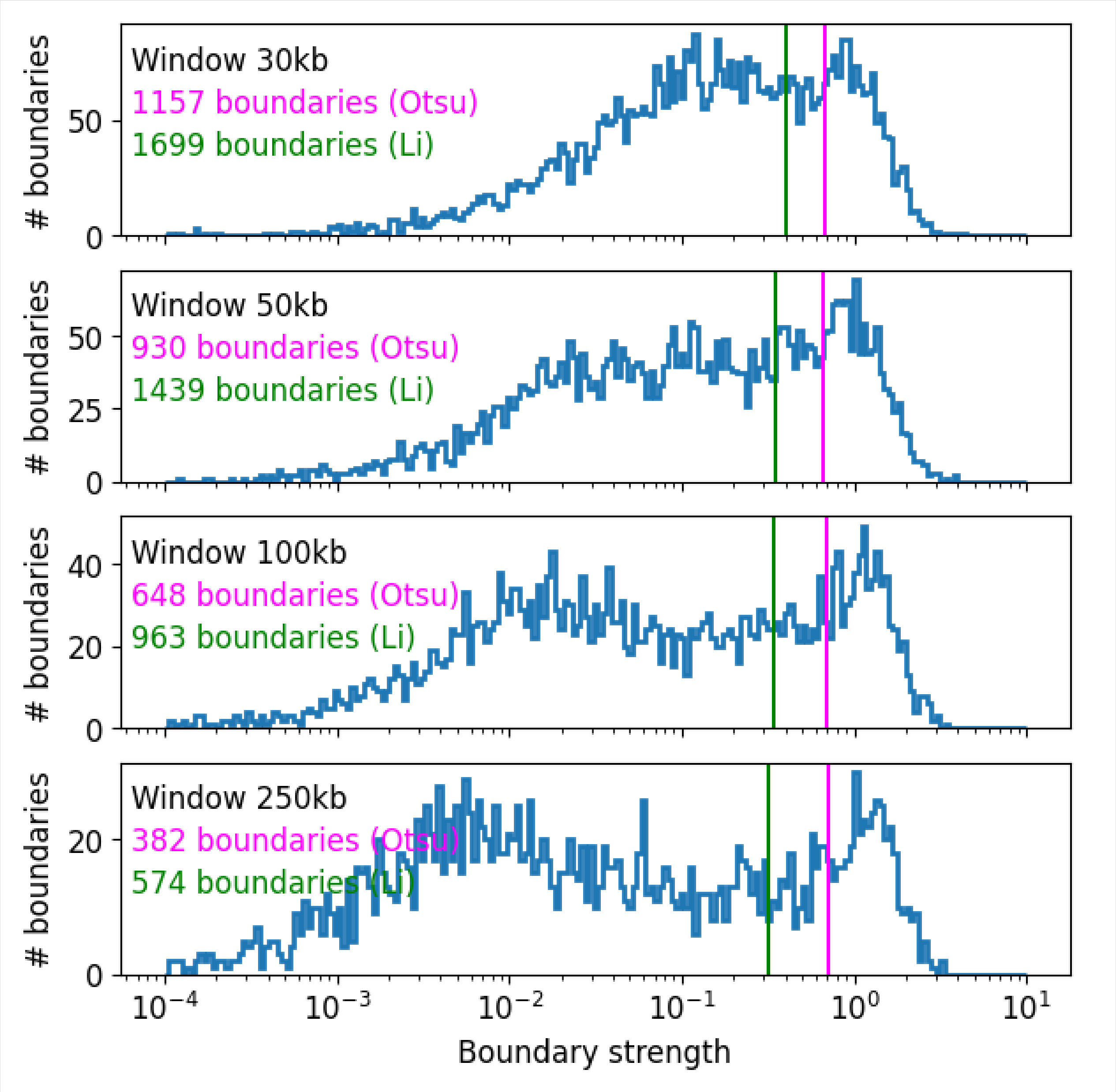
[11]:
```
from skimage.filters import threshold_li, threshold_otsu

f, axs = plt.subplots(len(windows), 1, sharex=True, figsize=(6,6), constrained_
↪layout=True)
thresholds_li = {}
thresholds_otsu = {}
for i, (w, ax) in enumerate(zip(windows, axs)):
```

(continues on next page)

(continued from previous page)

```
    ax.hist(
        insulation_table[f'boundary_strength_{w}'],
        **histkwargs
    )
    thresholds_li[w] = threshold_li(insulation_table[f'boundary_strength_{w}'].dropna().
↪values)
    thresholds_otsu[w] = threshold_otsu(insulation_table[f'boundary_strength_{w}'].
↪dropna().values)
    n_boundaries_li = (insulation_table[f'boundary_strength_{w}'].dropna()>=thresholds_
↪li[w]).sum()
    n_boundaries_otsu = (insulation_table[f'boundary_strength_{w}'].dropna()>=thresholds_
↪otsu[w]).sum()
    ax.axvline(thresholds_li[w], c='green')
    ax.axvline(thresholds_otsu[w], c='magenta')
    ax.text(0.01, 0.9,
             f'Window {w//1000}kb',
             ha='left',
             va='top',
             transform=ax.transAxes)
    ax.text(0.01, 0.7,
            f'{n_boundaries_otsu} boundaries (Otsu)',
            c='magenta',
            ha='left',
            va='top',
            transform=ax.transAxes)
    ax.text(0.01, 0.5,
            f'{n_boundaries_li} boundaries (Li)',
            c='green',
            ha='left',
            va='top',
            transform=ax.transAxes)

    ax.set(
        xscale='log',
        ylabel='# boundaries'
    )

axs[-1].set(xlabel='Boundary strength')
```

```
[11]: [Text(0.5, 0, 'Boundary strength')]
```



### CTCF enrichment at boundaries

TAD boundaries are frequently associated with CTCF binding.

To quantify this, we can aggregate the ChIP-Seq singal around the boundaries, and compare enrichment of CTCF with boundary strength using pybbi (https://github.com/nvictus/pybbi). We provide a test bigWig file with CTCF enrichment over input for the same cell type as the Micro-C data.

We use the bbi.stackup() method with no binning to extract an array of average values for all boundary regions with 1 kbp flanks.

```
[12]: # Download test data. The file size is 592 Mb, so the download might take a while:
ctcf_fc_file = cooltools.download_data("HFF_CTCF_fc", cache=True, data_dir=data_dir)
```

[13]:
```
import bbi
```

[14]:
```
is_boundary = np.any([
        ~insulation_table[f'boundary_strength_{w}'].isnull()
        for w in windows],
    axis=0)
boundaries = insulation_table[is_boundary]
boundaries.head()
```

[14]:
```
   chrom   start     end region  is_bad_bin  log2_insulation_score_30000  \
5   chr2   50000   60000   chr2       False                     0.089080
6   chr2   60000   70000   chr2       False                     0.036906
7   chr2   70000   80000   chr2       False                     0.062353
9   chr2   90000  100000   chr2       False                     0.049426
11  chr2  110000  120000   chr2       False                     0.095762

    n_valid_pixels_30000  log2_insulation_score_50000  n_valid_pixels_50000  \
5                    6.0                     0.059578                  22.0
6                    6.0                     0.134037                  22.0
7                    6.0                     0.122444                  22.0
9                    6.0                     0.198381                  22.0
11                   6.0                     0.190455                  22.0

    log2_insulation_score_100000  ...  log2_insulation_score_250000  \
5                       0.586104  ...                      1.211581
6                       0.547732  ...                      1.161302
7                       0.479052  ...                      1.092480
9                       0.377645  ...                      0.972715
11                      0.320182  ...                      0.867080

    n_valid_pixels_250000  boundary_strength_30000  boundary_strength_50000  \
5                   122.0                      NaN                 0.156397
6                   147.0                 0.150452                      NaN
7                   172.0                      NaN                 0.011593
9                   222.0                 0.029686                      NaN
11                  272.0                      NaN                 0.024922

    boundary_strength_250000  boundary_strength_100000  is_boundary_30000  \
5                        NaN                       NaN              False
6                        NaN                       NaN              False
7                        NaN                       NaN              False
9                        NaN                       NaN              False
11                       NaN                       NaN              False

    is_boundary_50000  is_boundary_100000  is_boundary_250000
5               False               False               False
6               False               False               False
7               False               False               False
9               False               False               False
11              False               False               False

[5 rows x 21 columns]
```

[15]:

```
# Calculate the average ChIP singal/input in the 3kb region around the boundary.
flank = 1000 # Length of flank to one side from the boundary, in basepairs
ctcf_chip_signal = bbi.stackup(
    data_dir+'/test_CTCF.bigWig',
    boundaries.chrom,
    boundaries.start-flank,
    boundaries.end+flank,
    bins=1).flatten()
```

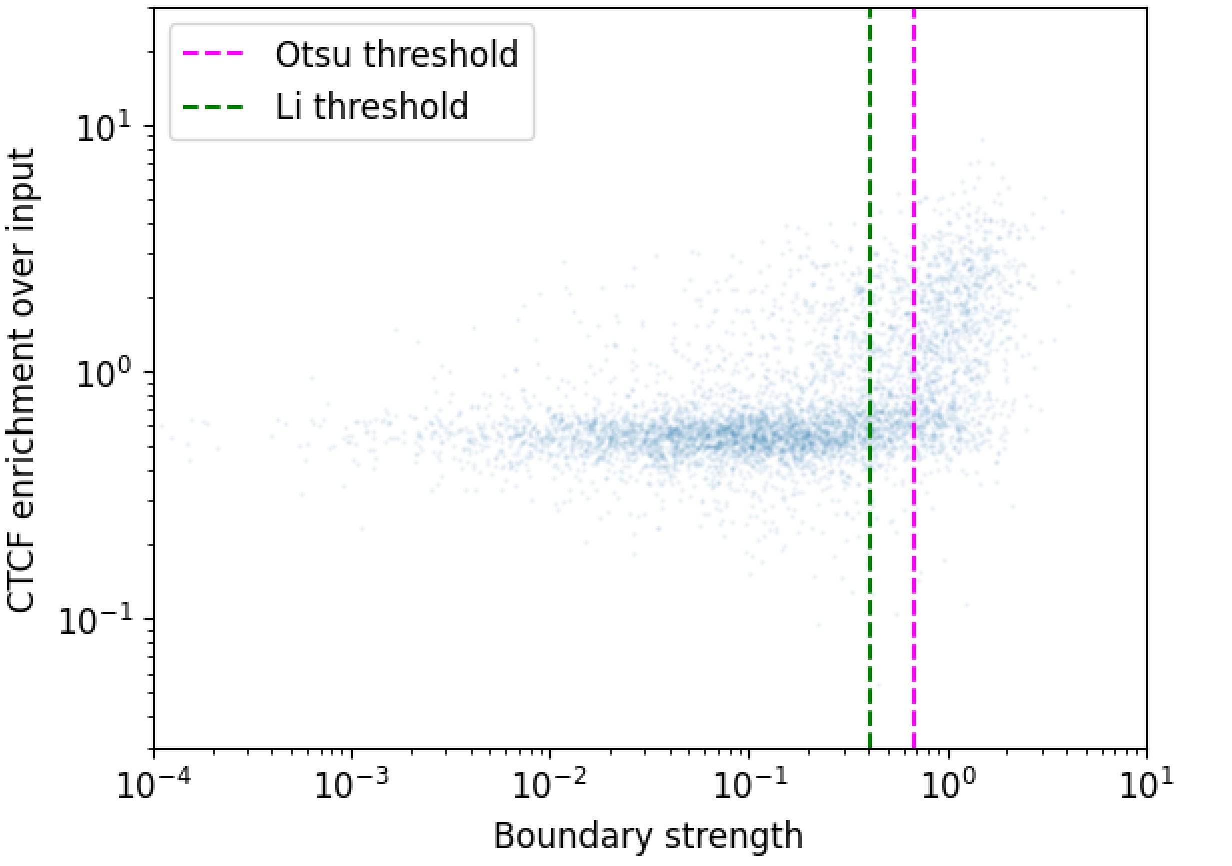
Real boundaries are often enriched in CTCF binding, and this can be used as a guide for thresholding the boundary strength. However note a small population of strong boundaries without CTCF binding.

[16]:

```
w=windows[0]
f, ax = plt.subplots()
ax.loglog(
    boundaries[f'boundary_strength_{w}'],
    ctcf_chip_signal,
    'o',
    markersize=1,
    alpha=0.05
);
ax.set(
    xlim=(1e-4,1e1),
    ylim=(3e-2,3e1),
    xlabel='Boundary strength',
    ylabel='CTCF enrichment over input')

ax.axvline(thresholds_otsu[w], ls='--', color='magenta', label='Otsu threshold')
ax.axvline(thresholds_li[w], ls='--', color='green', label='Li threshold')
ax.legend()
```

[16]:

```
<matplotlib.legend.Legend at 0x1afd95dc0>
```



If we were interested specifically in boundaries with CTCF, we could threshold based on enrichment of CTCF ChIP-seq:

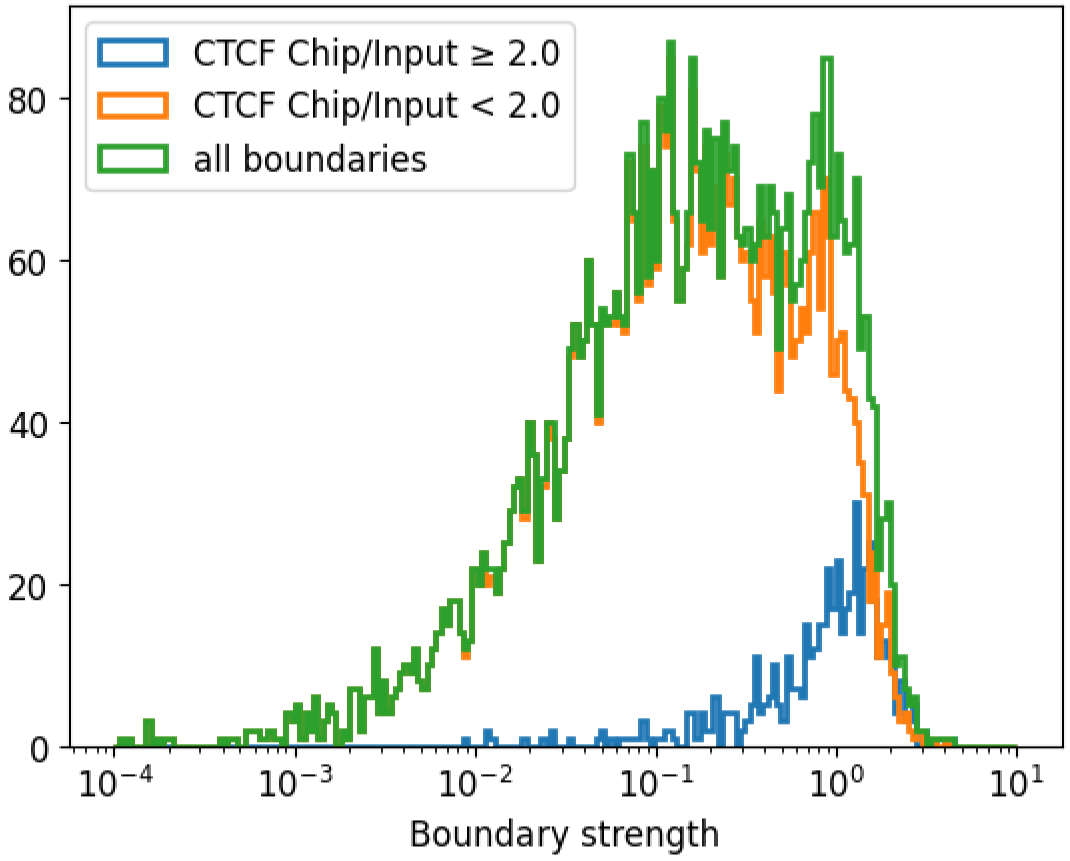
```
[17]: histkwargs = dict(
    bins=10**np.linspace(-4,1,200),
    histtype='step',
    lw=2,
)
f, ax = plt.subplots()
ax.set(xscale='log', xlabel='Boundary strength')
ax.hist(
    boundaries[f'boundary_strength_{windows[0]}'][ctcf_chip_signal>=2],
    label='CTCF Chip/Input  2.0',
    **histkwargs
);
ax.hist(
    boundaries[f'boundary_strength_{windows[0]}'][ctcf_chip_signal<2],
    label='CTCF Chip/Input < 2.0',
    **histkwargs
);
ax.hist(
    boundaries[f'boundary_strength_{windows[0]}'],
    label='all boundaries',
    **histkwargs
);
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc='upper left')
```

[17]:
```
<matplotlib.legend.Legend at 0x1af14eac0>
```



### 1D pileup: CTCF enrichment at boundaries

Additionally, we can create 1D pileup plot of average CTCF enrichment.

First, create a collection of genomic regions of equal size, each centered at the position of the boundary.

Then, create a **stackup** of binned ChIP-Seq signal for these regions. It is based on the same test bigWig file with the CTCF ChIP-Seq log fold change over input.

Finally, create **1D pileup** by averaging each stacked window.

[18]:
```
# Select the strict thresholded boundaries for one window size
top_boundaries = insulation_table[insulation_table[f'boundary_strength_{windows[1]}']>
↪=thresholds_otsu[windows[1]]]
```

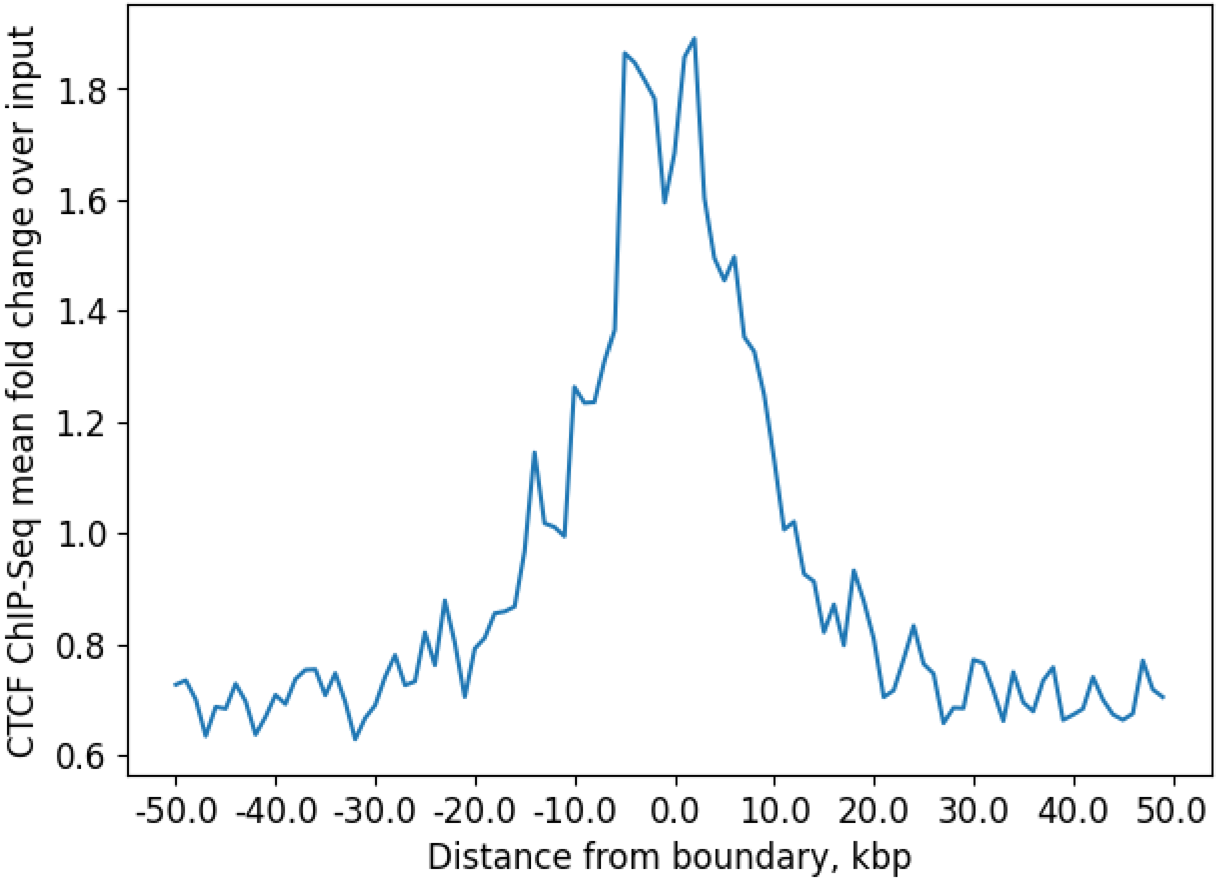
[19]:
```
# Create of the stackup, the flanks are +- 50 Kb, number of bins is 100 :
flank = 50000 # Length of flank to one side from the boundary, in basepairs
nbins = 100   # Number of bins to split the region
stackup = bbi.stackup(data_dir+'/test_CTCF.bigWig',
                      top_boundaries.chrom,
```

(continues on next page)

(continued from previous page)

```
                                    top_boundaries.start+resolution//2-flank,
                                    top_boundaries.start+resolution//2+flank,
                                    bins=nbins)
```

[20]:
```
f, ax = plt.subplots(figsize=[7,5])
ax.plot(np.nanmean(stackup, axis=0) )
ax.set(xticks=np.arange(0, nbins+1, 10),
       xticklabels=(np.arange(0, nbins+1, 10)-nbins//2)*flank*2/nbins/1000,
       xlabel='Distance from boundary, kbp',
       ylabel='CTCF ChIP-Seq mean fold change over input');
```



### Using adjacent boundaries to create a table of TADs

Calling TADs from Hi-C data poses a challenge, in part because domain structures vary greatly in their size, intensity, and can be nested. The number of called TADs varies substantially from tool to tool, and can depend on tool-specific parameters (Forcato, 2017). Below, we show an example of how adjacent boundaries calculated with cooltools can specify a set of intervals that could be analyzed as TADs, using bioframe.merge().

[41]:
```
def extract_TADs(insulation_table, is_boundary_col, max_TAD_length = 3_000_000):
    tads = bioframe.merge(insulation_table[insulation_table[is_boundary_col] == False])
    return tads[ (tads["end"] - tads["start"]) <= MAX_TAD_LENGTH].reset_
```

(continues on next page)

(continued from previous page)

```
↪index(drop=True)[['chrom','start','end']]

TADs_table = extract_TADs(insulation_table, f'is_boundary_{windows[0]}')
TADs_table.head()
```

[46]:

```
TADs_table
```

[46]:

```
      chrom     start       end
0      chr2         0    200000
1      chr2    210000    290000
2      chr2    300000    670000
3      chr2    680000    740000
4      chr2    750000    950000
...     ...       ...       ...
1693  chr17  82460000  82640000
1694  chr17  82650000  82760000
1695  chr17  82770000  82960000
1696  chr17  82970000  83080000
1697  chr17  83090000  83257441

[1698 rows x 3 columns]
```

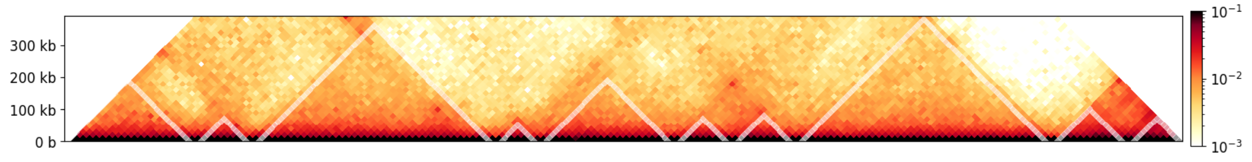
[50]:

```
# Visualizing the first 10 inter-boundary intervals (grey overay) v.s. Hi-C data

region = ('chr2', TADs_table.iloc[0].start, TADs_table.iloc[9].end)
norm = LogNorm(vmax=0.1, vmin=0.001)
data = clr.matrix(balance=True).fetch(region)

f, ax = plt.subplots(figsize=(18, 6))
im = pcolormesh_45deg(ax, data, start=region[1], resolution=resolution, norm=norm, cmap=
↪'fall')
ax.set_aspect(0.5)
ax.set_ylim(0, 13*windows[0])
format_ticks(ax, rotate=False)
ax.xaxis.set_visible(False)

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="1%", pad=0.1, aspect=6)
plt.colorbar(im, cax=cax)

idx = 10
max_pos = TADs_table[:idx]['end'].max()/resolution
contact_matrix = np.zeros((int(max_pos), int(max_pos)))
contact_matrix[:] = np.nan
for _, row in TADs_table[:idx].iterrows():
    contact_matrix[int(row['start']/resolution):int(row['end']/resolution), int(row[
↪'start']/resolution):int(row['end']/resolution)] = 1
    contact_matrix[int(row['start']/resolution + 1):int(row['end']/resolution - 1),␣
↪int(row['start']/resolution + 1):int(row['end']/resolution - 1)] = np.nan

im = pcolormesh_45deg(ax, contact_matrix, start=0, resolution=resolution, cmap='gray',␣
↪vmax=1, vmin=-1, alpha=0.6)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.5 Dots & focal enrichment

Welcome to the dot calling notebook!

Punctate pairwise peaks of enriched contact frequency are a prevalent feature of mammalian interphase contact maps. These features are also referred to as 'dots' or 'loops' in the literature, and can appear either in isolation or as parts of grids and at the corners of domains.

HiCCUPS, proposed in Rao et al. 2014, is a common approach for calling dots in contact maps. HICCUPS uses a multi-step procedure to score and return a filtered list of extracted dots. Scoring is done by convolving a set of **kernels** with the contact map. However, since HICCUPS is written in Java it is challenging to modify the parameters used at specific steps of the calling procedure, which can be important for calling dots at new resolutions or in new organismal or cellular contexts.

*Cooltools* implements a similar approach for calling dots in Python. This enables users to easily vary the parameters and processing steps used for different Hi-C or Micro-C datasets.

```
[1]: import pandas as pd
import numpy as np
from itertools import chain

# Hi-C utilities imports:
import cooler
import bioframe
import cooltools
from cooltools.lib.numutils import fill_diag
from packaging import version
if version.parse(cooltools.__version__) < version.parse('0.5.2'):
    raise AssertionError("tutorials rely on cooltools version 0.5.2 or higher,"+
                         "please check your cooltools version and update to the latest")

# Visualization imports:
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import matplotlib.patches as patches
from matplotlib.ticker import EngFormatter

# helper functions for plotting
bp_formatter = EngFormatter('b')
def format_ticks(ax, x=True, y=True, rotate=True):
    """format ticks with genomic coordinates as human readable"""
```

(continues on next page)

(continued from previous page)

```
    if y:
        ax.yaxis.set_major_formatter(bp_formatter)
    if x:
        ax.xaxis.set_major_formatter(bp_formatter)
        ax.xaxis.tick_bottom()
    if rotate:
        ax.tick_params(axis='x',rotation=45)
```

### Load data and define a genomic view

To call dots, we need an input cooler file with Hi-C data, and regions for calculation of expected (e.g. chromosomes or chromosome arms).

[2]:

```
# Download the test data from osf and define cooler:
data_dir = './data/'
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir=data_dir)
# 10 kb is a resolution at which one can clearly see "dots":
binsize = 10_000
# Open cool file with Micro-C data:
clr = cooler.Cooler(f'{data_dir}/test.mcool::/resolutions/{binsize}')
```

[3]:

```
# define genomic view that will be used to call dots and pre-compute expected

# Use bioframe to fetch the genomic features from the UCSC.
hg38_chromsizes = bioframe.fetch_chromsizes('hg38')
hg38_cens = bioframe.fetch_centromeres('hg38')
hg38_arms = bioframe.make_chromarms(hg38_chromsizes, hg38_cens)

# Select only chromosomes that are present in the cooler.
hg38_arms = hg38_arms.set_index("chrom").loc[clr.chromnames].reset_index()

# intra-arm expected
expected = cooltools.expected_cis(
    clr,
    view_df=hg38_arms,
    nproc=4,
)
```

### Dot-calling with default parameters

We first call dots with default parameters (i.e. similar to HiCCUPs). Later we illustrate the various parameters than can be easily modified in *cooltools*.

Here is a brief description of the steps involved in *cooltools* `dots()`:

- A set of convolution **kernels** are recommended based on the resolution of `clr`, if user-defined convolution kernels are not provided (i.e. `kernels=None`).
- The requested portion of the heatmap (defined by `view_df` and `max_loci_separation`) is split into smaller **tiles** of size `tile_size`. This ensures the entire heatmap is not loaded into memory at once, and computationally intensive steps can be done in parallel using `nproc` workers. `tile_size` and `nproc` do not affect the outcome of the procedure.

- Tiles of the heatmap are convolved with the provided kernels to calculate localy adjusted expected for each pixel. This is in turn used to calculate p-values, assuming a Poisson distribution of pixel counts.
- Pixels are assigned to geometrically-spaced "lambda-bins" of locally-adjusted expected for statistical testing. Within each lambda-bin, signficantly enriched pixels are "caled" using BH-FDR multiple hypothesis testing procedure, and thresholds of significance are calculated for each lambda-bin and each kernel-type (controlled by `lambda_bin_fdr`).
- Significantly-enriched pixels are extracted, based on the thresholds in each lambda bin. Note the *cooltools* implementation of this step involves a second pass with the same convolution kernels to re-score pixels, as this is less costly than storing all such scores in memory.
- Additional clustering and empirical filtering is optionally performed (depending on `clustering_radius` and `cluster_filtering`).

See the `` `dotfinder `` docstring <https://github.com/open2c/cooltools/blob/master/cooltools/api/dotfinder.py>`__ for additional practical details of the implementation.

[4]:
```
dots_df = cooltools.dots(
    clr,
    expected=expected,
    view_df=hg38_arms,
    # how far from the main diagonal to call dots:
    max_loci_separation=10_000_000,
    nproc=4,
)
```

```
INFO:root:Using recommended donut-based kernels with w=5, p=2 for binsize=10000
INFO:root: matrix 9314X9314 to be split into 361 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 14907X14907 to be split into 900 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 2472X2472 to be split into 25 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 5855X5855 to be split into 144 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root:convolving 186 tiles to build histograms for lambda-bins
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done building histograms in 29.498 sec ...
INFO:root:Determined thresholds for every lambda-bin ...
INFO:root:convolving 186 tiles to extract enriched pixels
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done extracting enriched pixels in 22.276 sec ...
INFO:root:Begin post-processing of 15303 filtered pixels
INFO:root:preparing to extract needed q-values ...
INFO:root:clustering enriched pixels in region: chr17_p
INFO:root:detected 341 clusters of 2.96+/-2.60 size
INFO:root:clustering enriched pixels in region: chr17_q
INFO:root:detected 939 clusters of 3.23+/-2.99 size
INFO:root:clustering enriched pixels in region: chr2_p
INFO:root:detected 1400 clusters of 3.12+/-2.93 size
INFO:root:clustering enriched pixels in region: chr2_q
INFO:root:detected 2203 clusters of 3.13+/-2.87 size
INFO:root:Clustering is complete
INFO:root:filtered 3145 out of 4883 centroids to reduce the number of false-positives
```

### Visualizing dot-calling with the default parameters

To visualize the results of this dot calling, we overlay small rectangles at the positions of the called dots over the HiC map.

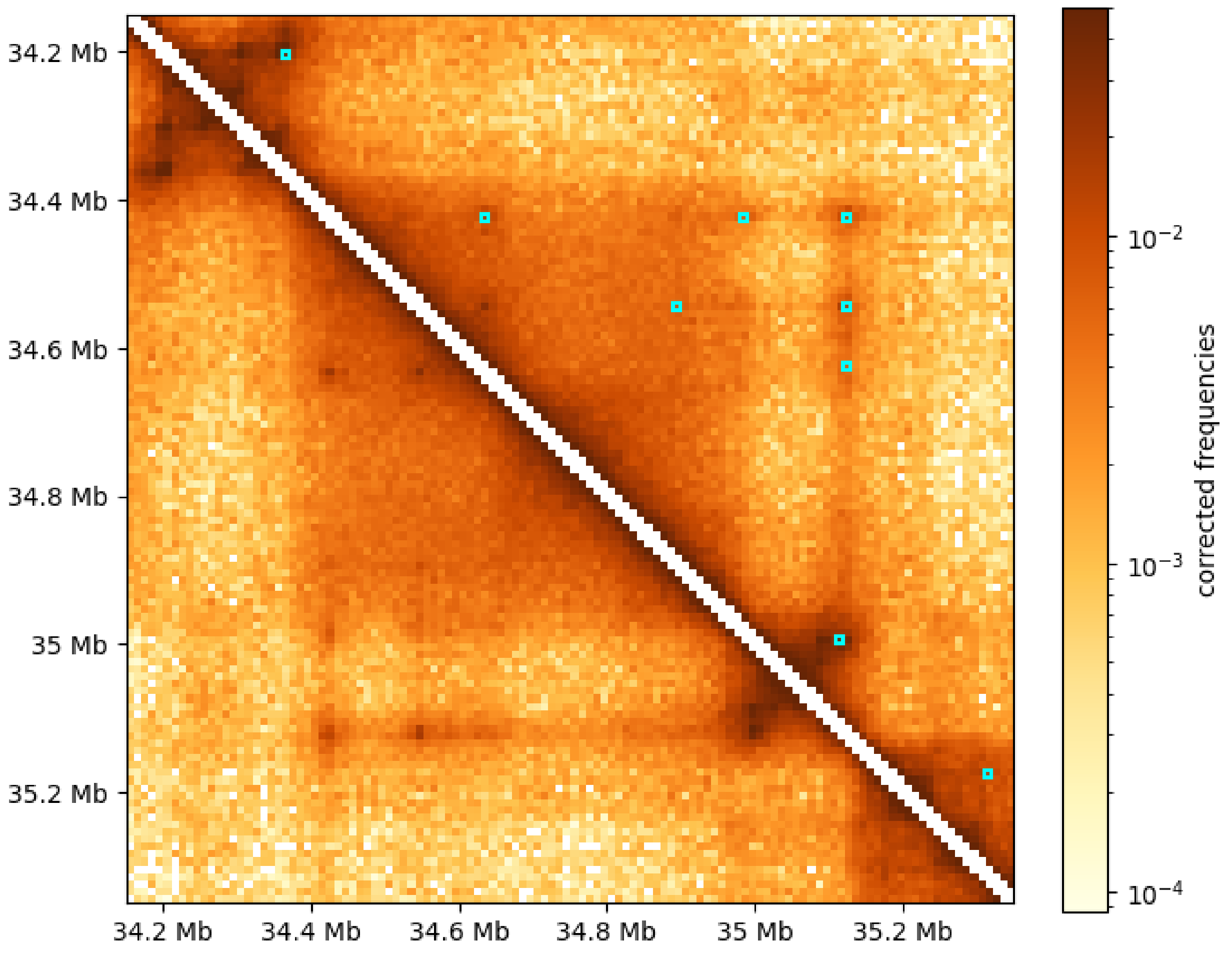
[5]:
```
# create a functions that would return a series of rectangles around called dots
# in a specific region, and exposing importnat plotting parameters
def rectangles_around_dots(dots_df, region, loc="upper", lw=1, ec="cyan", fc="none"):
    """
    yield a series of rectangles around called dots in a given region
    """
    # select dots from the region:
    df_reg = bioframe.select(
        bioframe.select(dots_df, region, cols=("chrom1","start1","end1")),
        region,
        cols=("chrom2","start2","end2"),
    )
    rectangle_kwargs = dict(lw=lw, ec=ec, fc=fc)
    # draw rectangular "boxes" around pixels called as dots in the "region":
    for s1, s2, e1, e2 in df_reg[["start1", "start2", "end1", "end2"]].
↪itertuples(index=False):
        width1 = e1 - s1
        width2 = e2 - s2
        if loc == "upper":
            yield patches.Rectangle((s2, s1), width2, width1, **rectangle_kwargs)
        elif loc == "lower":
            yield patches.Rectangle((s1, s2), width1, width2, **rectangle_kwargs)
        else:
            raise ValueError("loc has to be uppper or lower")
```

[6]:
```
# define a region to look into as an example
start = 34_150_000
end = start + 1_200_000
region = ('chr17', start, end)

# heatmap kwargs
matshow_kwargs = dict(
    cmap='YlOrBr',
    norm=LogNorm(vmax=0.05),
    extent=(start, end, end, start)
)

# colorbar kwargs
colorbar_kwargs = dict(fraction=0.046, label='corrected frequencies')

# compute heatmap for the region
region_matrix = clr.matrix(balance=True).fetch(region)
for diag in [-1,0,1]:
    region_matrix = fill_diag(region_matrix, np.nan, i=diag)

# see viz.ipynb for details of heatmap visualization
f, ax = plt.subplots(figsize=(7,7))
im = ax.matshow( region_matrix, **matshow_kwargs)
```

(continues on next page)

(continued from previous page)

```
format_ticks(ax, rotate=False)
plt.colorbar(im, ax=ax, **colorbar_kwargs)

# draw rectangular "boxes" around pixels called as dots in the "region":
for box in rectangles_around_dots(dots_df, region, lw=1.5):
    ax.add_patch(box)
```



### Skipping clustering and cluster enrichment filtering

Dot-calling returns pixels that are enriched relative to some local background. Such pixels often come in groups ("clusters"). By default `dots()` picks a single representative for each cluster (i.e. centroid). However, *cooltools* users can easily turn clustering off for debugging or alternative clustering approaches:

[7]:
```
dots_df_all = cooltools.dots(
    clr,
    expected=expected,
    view_df=hg38_arms,
    max_loci_separation=10_000_000,
    clustering_radius=None,  # None - implies no clustering
```

(continues on next page)

(continued from previous page)

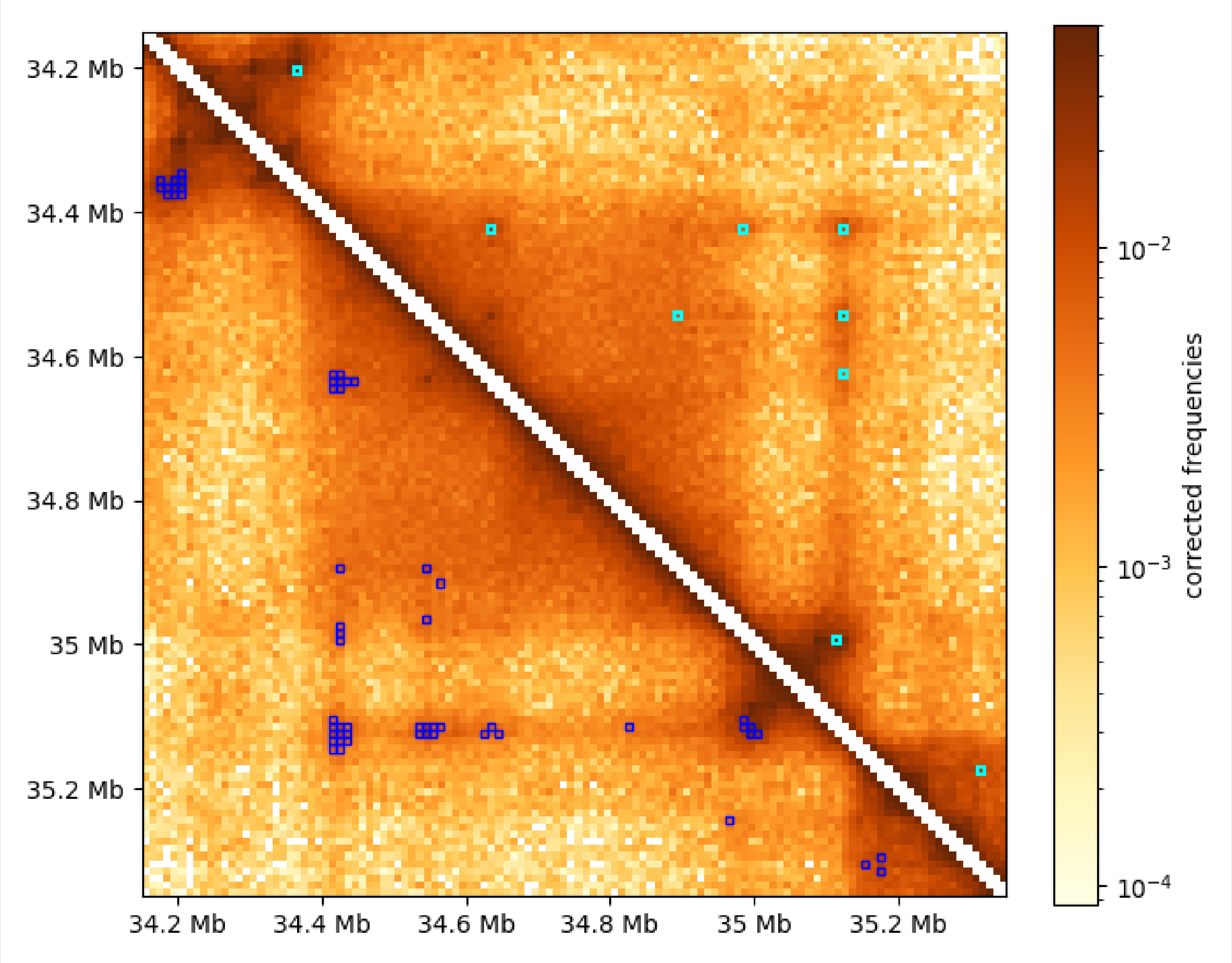
```
    cluster_filtering=False,  # ignored when clustering is off
    nproc=4,
)
```

```
INFO:root:Using recommended donut-based kernels with w=5, p=2 for binsize=10000
INFO:root: matrix 9314X9314 to be split into 361 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 14907X14907 to be split into 900 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 2472X2472 to be split into 25 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 5855X5855 to be split into 144 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root:convolving 186 tiles to build histograms for lambda-bins
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done building histograms in 25.523 sec ...
INFO:root:Determined thresholds for every lambda-bin ...
INFO:root:convolving 186 tiles to extract enriched pixels
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done extracting enriched pixels in 23.327 sec ...
INFO:root:Begin post-processing of 15303 filtered pixels
INFO:root:preparing to extract needed q-values ...
```

The visualization below compares clustered dots (cyan) with all enriched pixels before clustering (blue)

[8]:

```
f, ax = plt.subplots(figsize=(7,7))
# draw heatmap
im = ax.matshow(region_matrix, **matshow_kwargs)
format_ticks(ax, rotate=False)
plt.colorbar(im, ax=ax, **colorbar_kwargs)

# draw rectangular "boxes" around pixels called as dots in the "region":
for rect in chain(
        rectangles_around_dots(dots_df, region, lw=1.5),  # clustered & filtered
        rectangles_around_dots(dots_df_all, region, loc="lower", ec="blue"),  # unclustered
    ):
    ax.add_patch(rect)
```



### Convolution kernels and local expected

A useful local background to calculate the enrichment of pixels was defined in Rao et al. 2014 as a "donut"-shaped surrounding of a given pixel between ~20 to ~50 kb away from that pixel.

Such a local surrounding is best thought of in the terms of convolutional kernels e.g. as in image processing. In this framework, calcluating the local background for all pixels is simply obtained as the convolution of a contact map with the "donut"-shaped kernel.

Additional kernels can be used to downweight unwanted enrichment types. In addition to the "donut" kernel, the default kernels recommended in cooltools `dots()` are: - "vertical" to avoid calling pixels that are part of vertical stripes as dots - "horizontal" to avoid calling pixels that are part of horizontal stripes as dots - "lowleft" to avoid calling pixels at the corners of domains as dots

These four kernels are illustrated below, where pixels that are included in the calculations are highlighted in yellow, the pixel of interest is highlighted in red, and pixels that are not included in the local background are in purple. A checkerboard pattern is overlayed on the figure to emphasize individual pixels.

[9]:
```
# function to draw kernels:
def draw_kernel(kernel, axis=None, cmap='viridis'):
    if axis is None:
        f, axis = plt.subplots()
```

(continues on next page)

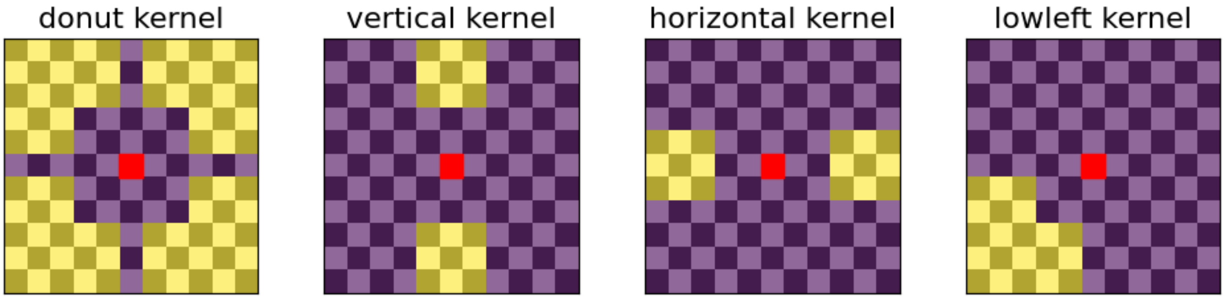
(continued from previous page)

```
    # kernel:
    imk = axis.imshow(
                    kernel[::-1,::-1],  # flip it, as in convolution
                    alpha=0.85,
                    cmap=cmap,
                    interpolation='nearest')
    # draw a square around the target pixel:
    x0 = kernel.shape[0] // 2 - 0.5
    y0 = kernel.shape[1] // 2 - 0.5
    rect = patches.Rectangle((x0, y0), 1, 1, lw=1, ec='r', fc='r')
    axis.add_patch(rect)

    # clean axis:
    axis.set_xticks([])
    axis.set_yticks([])
    axis.set_xticklabels('',visible=False)
    axis.set_yticklabels('',visible=False)
    axis.set_title("{} kernel".format(ktype),fontsize=16)
    # add a checkerboard to highlight pixels:
    checkerboard = np.add.outer(range(kernel.shape[0]),
                                range(kernel.shape[1])) % 2
    # show it:
    axis.imshow(checkerboard,
            cmap='gray',
            interpolation='nearest',
            alpha=0.3)

    return imk
```

[10]:

```
kernels = cooltools.api.dotfinder.recommend_kernels(binsize)

fig, axs = plt.subplots(ncols=4, figsize=(12,2.5))
for ax, (ktype, kernel) in zip(axs, kernels.items()):
    imk = draw_kernel(kernel, ax)
```

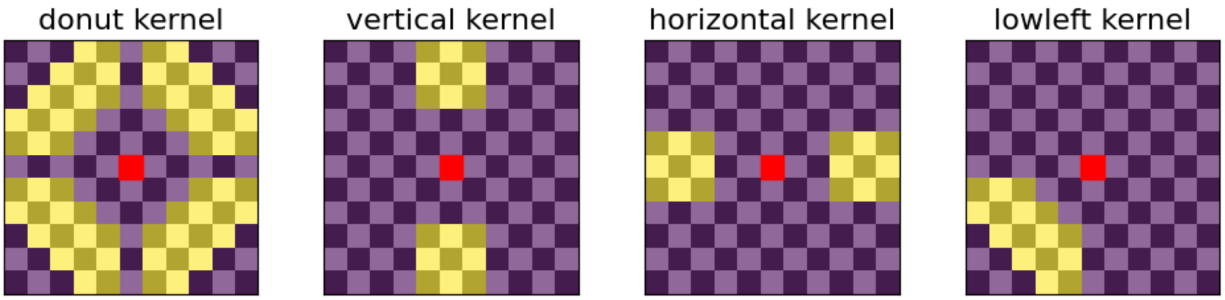
```
INFO:root:Using recommended donut-based kernels with w=5, p=2 for binsize=10000
```



### Calling dots with a "rounded donut" kernel

*Cooltools* enables experimentation and modification of kernels used for determining local enrichment scores. Here we show the result for replacing the default "donut" kernel with a "rounded donut".

[11]:
```
# create a grid of coordinates from -5 to 5, to define round kernels
# see https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html for details
half = 5  # half width of the kernel
x, y = np.meshgrid(
    np.linspace(-half, half, 2*half + 1),
    np.linspace(-half, half, 2*half + 1),
)
# now define a donut-like mask as pixels between 2 radii: sqrt(7) and sqrt(30):
mask = (x**2+y**2 > 7) & (x**2+y**2 <= 30)
mask[:,half] = 0
mask[half,:] = 0

# lowleft mask - zero out neccessary parts
mask_ll = mask.copy()
mask_ll[:,:half] = 0
mask_ll[half:,:] = 0

# new kernels with more round donut and lowleft masks:
kernels_round = {'donut': mask,
 'vertical': kernels["vertical"].copy(),
 'horizontal': kernels["horizontal"].copy(),
 'lowleft': mask_ll}

# plot rounded kernels
fig, axs = plt.subplots(ncols=4, figsize=(12,2.5))
for ax, (ktype, kernel) in zip(axs, kernels_round.items()):
    imk = draw_kernel(kernel, ax)
```



[12]:
```
#### call dots using redefined kernels (without clustering)

dots_round_df_all = cooltools.dots(
    clr,
    expected=expected,
    view_df=hg38_arms,
    kernels=kernels_round, # provide custom kernels
    max_loci_separation=10_000_000,
    clustering_radius=None,
```

(continues on next page)

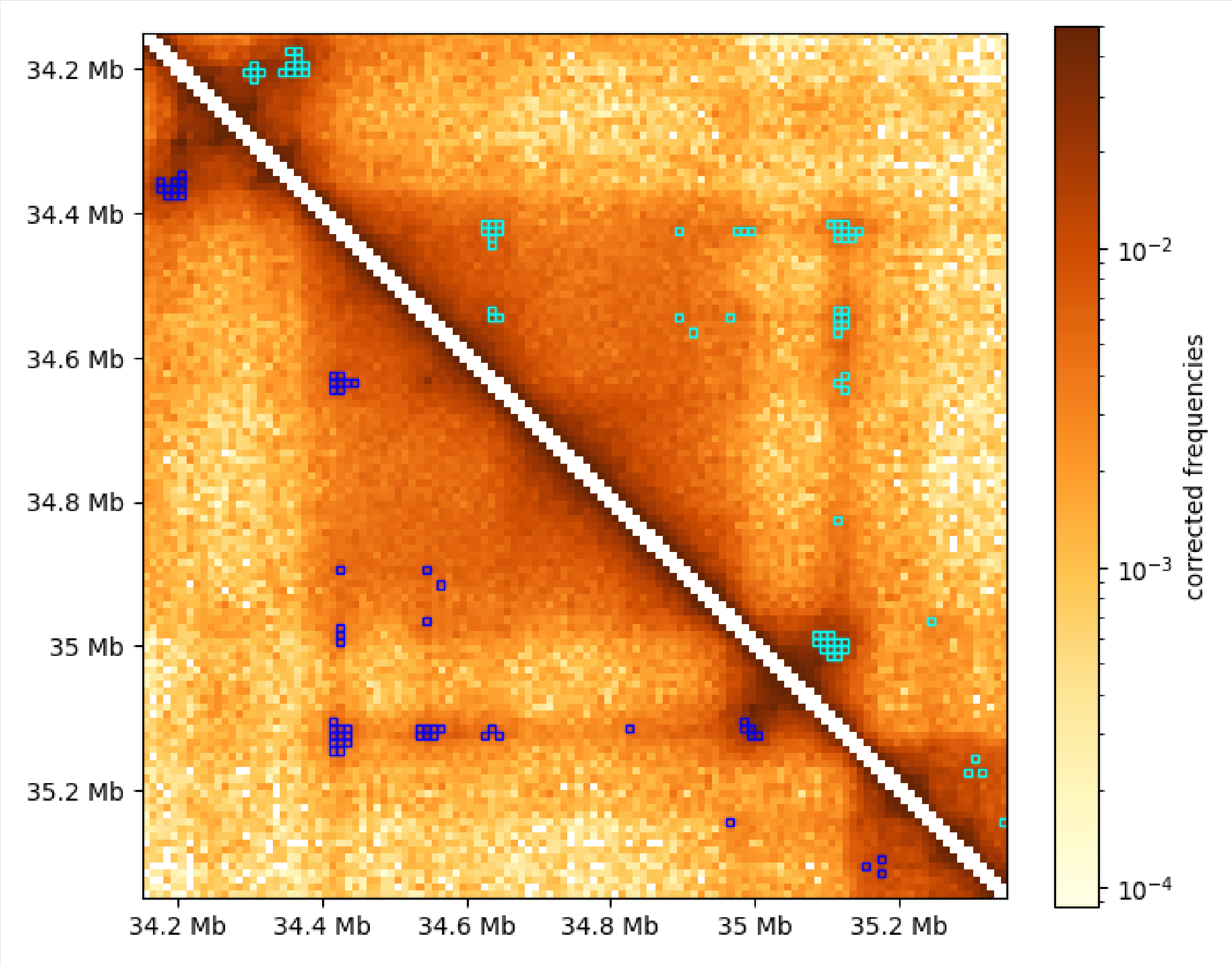
(continued from previous page)

```
    nproc=4,
)
```

```
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/api/
↪dotfinder.py:1571: UserWarning: Compatibility checks for 'kernels' are not fully␣
↪implemented yet, use at your own risk
  warnings.warn(
INFO:root: matrix 9314X9314 to be split into 361 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 14907X14907 to be split into 900 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 2472X2472 to be split into 25 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 5855X5855 to be split into 144 tiles of 500X500.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root:convolving 186 tiles to build histograms for lambda-bins
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done building histograms in 33.029 sec ...
INFO:root:Determined thresholds for every lambda-bin ...
INFO:root:convolving 186 tiles to extract enriched pixels
INFO:root:creating a Pool of 4 workers to tackle 186 tiles
INFO:root:Done extracting enriched pixels in 23.135 sec ...
INFO:root:Begin post-processing of 16716 filtered pixels
INFO:root:preparing to extract needed q-values ...
```

The visualization below compares dots called using "rounded" kernels (cyan) with the dots called using recommended kernels (blue). As one can tell the results are similar, yet the "rounded" kernels allow for calling dots closer to the diagonal because of the shape of the kernel.

[13]:
```
f, ax = plt.subplots(figsize=(7,7))
# draw heatmap
im = ax.matshow(region_matrix, **matshow_kwargs)
format_ticks(ax, rotate=False)
plt.colorbar(im, ax=ax, **colorbar_kwargs)

# draw rectangular "boxes" around pixels called as dots in the "region":
for rect in chain(
        rectangles_around_dots(dots_round_df_all, region),
        rectangles_around_dots(dots_df_all, region, loc="lower", ec="blue"),
    ):
    ax.add_patch(rect)
```



```
[14]: # TODO CTCF-based analysis to look
```

This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.6 Pileups and average features

Welcome to the cooltools pileups notebook!

Averaging Hi-C/Micro-C maps allows the quantification of general patterns observed in the maps. Averaging comes in various forms: contact-vs-distance plots, saddle plots, and pileup plots. **Pileup plots** are the averaged local Hi-C map over the 2D windows (i.e. **snippets**). These are also referred to as "average Hi-C maps". Pileups can be useful for determining the relationship between features (e.g. CTCF and TAD boundaries). Pileups can also be beneficial for reliably observing features in low-coverage Hi-C or single-cell HiC maps.

For pileups, we retrieve local windows that are centered at the **anchors**. We call this procedure **snipping**. Anchors can be ChIP-Seq binding sites, anchors of dots, or any other genomic features. Pileups come in two varieties:

- **On-diagonal pileup**. Each window is centered at the pixel located at the anchor position, at the main diagonal. Both coordinates of the window center are equivalent to the bin of the anchor.
- **Off-diagonal pileup**. Each window is centered at the pixel with one anchor as a left coordinate and another anchor as a right coordinate.

Typically, the sizes of windows are equivalent. After the selection of windows, we average them elementwise.

Content:

1. Download data
2. Load data
   - Load genomic regions
   - Load features for anchors
3. On-diagonal pipeup of CTCF
   - On-diagonal pileup of ICed Hi-C interactions
   - On-diagonal pileup of observed over expected interactions
   - Inspect the snips
4. Off-diagonal pileup of CTCF

```
[1]: # If you are a developer, you may want to reload the packages on a fly.
     # Jupyter has a magic for this particular purpose:
     %load_ext autoreload
     %autoreload 2
```

```
[2]: # import standard python libraries
     import numpy as np
     import matplotlib.pyplot as plt
     import pandas as pd
     import seaborn as sns
```

```
[3]: # import libraries for biological data analysis
     import cooler
     import bioframe

     import cooltools

     from packaging import version
     if version.parse(cooltools.__version__) < version.parse('0.5.2'):
         raise AssertionError("tutorials rely on cooltools version 0.5.2 or higher,"+
                              "please check your cooltools version and update to the latest")
```

### Download data

For this example notebook, we collected the data from immortalized human foreskin fibroblast cell line HFFc6:

- Micro-C data from Krietenstein et al. 2020
- ChIP-Seq for CTCF from ENCODE ENCSR000DWQ

You can automatically download test datasets with cooltools. More information on the files and how they were obtained is available from the datasets description.

```
[4]: # Print available datasets for download
     cooltools.print_available_datasets()
```

```
1) HFF_MicroC : Micro-C data from HFF human cells for two chromosomes (hg38) in a multi-
↪resolution mcool format.
	Downloaded from https://osf.io/3h9js/download
	Stored as test.mcool
	Original md5sum: e4a0fc25c8dc3d38e9065fd74c565dd1

2) HFF_CTCF_fc : ChIP-Seq fold change over input with CTCF antibodies in HFF cells␣
↪(hg38). Downloaded from ENCODE ENCSR000DWQ, ENCFF761RHS.bigWig file
	Downloaded from https://osf.io/w92u3/download
	Stored as test_CTCF.bigWig
	Original md5sum: 62429de974b5b4a379578cc85adc65a3

3) HFF_CTCF_binding : Binding sites called from CTCF ChIP-Seq peaks for HFF cells (hg38).
↪ Peaks are from ENCODE ENCSR000DWQ, ENCFF498QCT.bed file. The motifs are called with␣
↪gimmemotifs (options --nreport 1 --cutoff 0), with JASPAR pwm MA0139.
	Downloaded from https://osf.io/c9pwe/download
	Stored as test_CTCF.bed.gz
	Original md5sum: 61ecfdfa821571a8e0ea362e8fd48f63
```

[5]:

```
# Downloading test data for pileups
# cache = True will doanload the data only if it was not previously downloaded
data_dir = './data/'
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir=data_dir)
ctcf_peaks_file = cooltools.download_data("HFF_CTCF_binding", cache=True, data_dir=data_
↪dir)
ctcf_fc_file = cooltools.download_data("HFF_CTCF_fc", cache=True, data_dir=data_dir)
```

### Load data

### Load genomic regions

The pileup function needs **genomic regions**. Why?

- First, pileup uses regions for parallelization of snipping. Different genomic regions are loaded simultaneously by different processes, and the snipping can be done in parallel.
- Second, the observed over expected pileup requires calculating expected interactions before snipping (P(s), in other words). Typically, P(s) is calculated separately for each chromosome arm as inter-arms interactions might be affected by strong insulation of centromeres or Rabl configuration.

For species that do not have information on chromosome arms, or have *telocentric chromosomes* (e.g., mouse), you may want to use full chromosomes instead.

[6]:

```
# Open cool file with Micro-C data:
clr = cooler.Cooler(data_dir+'/test.mcool::/resolutions/10000')
# Set up selected data resolution:
resolution = clr.binsize
```

[7]:

```
# Use bioframe to fetch the genomic features from the UCSC.
hg38_chromsizes = bioframe.fetch_chromsizes('hg38')
hg38_cens = bioframe.fetch_centromeres('hg38')
```

(continues on next page)

(continued from previous page)

```
hg38_arms = bioframe.make_chromarms(hg38_chromsizes, hg38_cens)

# Select only chromosomes that are present in the cooler.
# This step is typically not required! we call it only because the test data are reduced.
hg38_arms = hg38_arms.set_index("chrom").loc[clr.chromnames].reset_index()
```

### Load features for anchors

Construction of the pileup requires genomic **features** that will be used for centering of the **snippets**. In this example, we will use *positions of motifs in CTCF peaks* as features.

[8]:

```
# Read CTCF peaks data and select only chromosomes present in cooler:
ctcf = bioframe.read_table(ctcf_peaks_file, schema='bed').query(f'chrom in {clr.
↪chromnames}')
ctcf['mid'] = (ctcf.end+ctcf.start)//2
ctcf.head()
```

[8]:

```
       chrom   start     end                name      score strand     mid
17271  chr17  118485  118504  MA0139.1_CTCF_human  12.384042      -  118494
17272  chr17  144002  144021  MA0139.1_CTCF_human  11.542617      +  144011
17273  chr17  163676  163695  MA0139.1_CTCF_human   5.294219      -  163685
17274  chr17  164711  164730  MA0139.1_CTCF_human  11.889376      +  164720
17275  chr17  309416  309435  MA0139.1_CTCF_human   7.879575      -  309425
```

### Feature inspection and filtering

Since we have both the list of strongest motifs of CTCF located in CTCF ChIP-Seq and the fold change over input for the genome, we have two characteristics of each feature:

- score of the motif
- CTCF ChIP-Seq fold-change over input

Let's take a look at joint distribution of these scores:

[9]:

```
import bbi
from scipy.stats import linregress
```

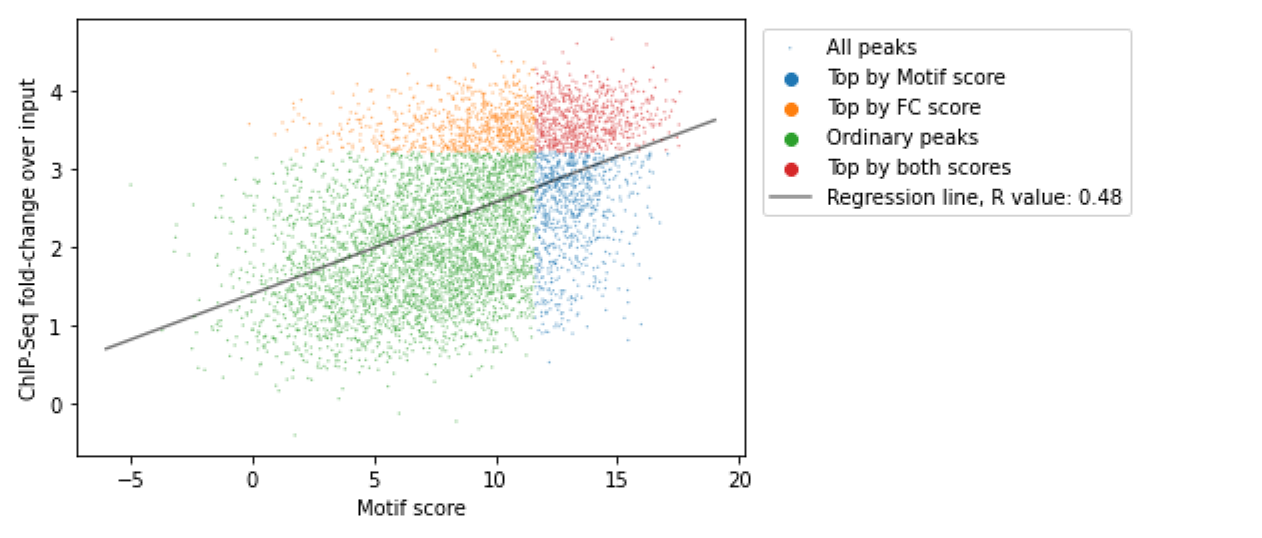
[10]:

```
# Get CTCF ChIP-Seq fold-change over input for genomic regions centered at the positions␣
↪of the motifs

flank = 250 # Length of flank to one side from the boundary, in basepairs
ctcf_chip_signal = bbi.stackup(
    ctcf_fc_file,
    ctcf.chrom,
    ctcf.mid-flank,
    ctcf.mid+flank,
    bins=1)

ctcf['FC_score'] = ctcf_chip_signal
```

[11]:
```
ctcf['quartile_score']    = pd.qcut(ctcf['score'], 4, labels=False) + 1
ctcf['quartile_FC_score'] = pd.qcut(ctcf['FC_score'], 4, labels=False) + 1
ctcf['peaks_importance'] = ctcf.apply(
    lambda x: 'Top by both scores' if x.quartile_score==4 and x.quartile_FC_score==4 else
                'Top by Motif score' if x.quartile_score==4 else
                'Top by FC score' if x.quartile_FC_score==4 else 'Ordinary peaks', axis=1
)
```

[12]:
```
x = ctcf['score']
y = np.log(ctcf['FC_score'])

fig, ax = plt.subplots()

sns.scatterplot(x=x, y=y, hue=ctcf['peaks_importance'],
    s=2,
    alpha=0.5,
    label='All peaks',
    ax=ax
)

slope, intercept, r, p, se = linregress(x, y)

ax.plot([-6, 19], [intercept-6*slope, intercept+19*slope],
        alpha=0.5,
        color='black',
        label=f"Regression line, R value: {r:.2f}")

ax.set(
    xlabel='Motif score',
    ylabel='ChIP-Seq fold-change over input')

ax.legend(bbox_to_anchor=(1.01,1), loc="upper left")

plt.show()
```

```
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.8/site-packages/seaborn/_
↪decorators.py:36: FutureWarning: Pass the following variables as keyword args: x, y.␣
↪From version 0.12, the only valid positional argument will be `data`, and passing␣
↪other arguments without an explicit keyword will result in an error or␣
↪misinterpretation.
  warnings.warn(
```



```
[13]: # Select the CTCF sites that are in top quartile by both the ChIP-Seq data and motif␣
      ↪score

      sites = ctcf[ctcf['peaks_importance']=='Top by both scores']\
          .sort_values('FC_score', ascending=False)\
          .reset_index(drop=True)
      sites.tail()
```

```
[13]:      chrom      start        end                name      score strand  \
      659  chr17    8158938    8158957  MA0139.1_CTCF_human  13.276979      -
      660   chr2  176127201  176127220  MA0139.1_CTCF_human  12.820343      +
      661  chr17   38322364   38322383  MA0139.1_CTCF_human  13.534864      -
      662   chr2  119265336  119265355  MA0139.1_CTCF_human  13.739862      -
      663   chr2  118003514  118003533  MA0139.1_CTCF_human  12.646685      -

                 mid   FC_score  quartile_score  quartile_FC_score  \
      659    8158947  25.056849               4                  4
      660  176127210  25.027294               4                  4
      661   38322373  25.010430               4                  4
      662  119265345  24.980141               4                  4
      663  118003523  24.957502               4                  4

             peaks_importance
      659  Top by both scores
      660  Top by both scores
      661  Top by both scores
      662  Top by both scores
      663  Top by both scores
```

```
[14]: # Some CTCF sites might be located too close in the genome and interfere with analysis.
      # We will collapse the sites falling into the same size genomic bins as the resolution␣
      ↪of our micro-C data:
      sites = bioframe.cluster(sites, min_dist=resolution)\
          .drop_duplicates('cluster')\
          .reset_index(drop=True)
```

(continues on next page)

(continued from previous page)

```
sites.tail()
```

[14]:

```
     chrom      start        end                 name      score strand  \
608  chr17    8158938    8158957  MA0139.1_CTCF_human  13.276979      -   
609   chr2  176127201  176127220  MA0139.1_CTCF_human  12.820343      +   
610  chr17   38322364   38322383  MA0139.1_CTCF_human  13.534864      -   
611   chr2  119265336  119265355  MA0139.1_CTCF_human  13.739862      -   
612   chr2  118003514  118003533  MA0139.1_CTCF_human  12.646685      -   

           mid   FC_score  quartile_score  quartile_FC_score  \
608    8158947  25.056849               4                  4   
609  176127210  25.027294               4                  4   
610   38322373  25.010430               4                  4   
611  119265345  24.980141               4                  4   
612  118003523  24.957502               4                  4   

       peaks_importance  cluster  cluster_start  cluster_end  
608  Top by both scores       34        8158938      8158957  
609  Top by both scores      515      176127201    176127220  
610  Top by both scores      104       38322364     38322383  
611  Top by both scores      465      119265336    119265355  
612  Top by both scores      462      118003514    118003533  
```

### On-diagonal pileup

On-diagonal pileup is the simplest, you need the positions of **features** (middlepoints of CTCF motifs) and the size of flanks aroung each motif. cooltools will create a snippet of Hi-C map for each feature. Then you can combine them into a single 2D pileup.

### On-diagonal pileup of ICed Hi-C interactions

[15]:

```
stack = cooltools.pileup(clr, sites, view_df=hg38_arms, flank=300_000)
# Mirror reflect snippets when the feature is on the opposite strand
mask = np.array(sites.strand == '-', dtype=bool)
stack[:, :, mask] = stack[::-1, ::-1, mask]

# Aggregate. Note that some pixels might be converted to NaNs after IC, thus we aggregate by nanmean:
mtx = np.nanmean(stack, axis=2)
```

[16]:

```
# Load colormap with large number of distinguishable intermediary tones,
# The "fall" colormap in cooltools is exactly for this purpose.
# After this step, you can use "fall" as cmap parameter in matplotlib:
import cooltools.lib.plotting
```

[17]:

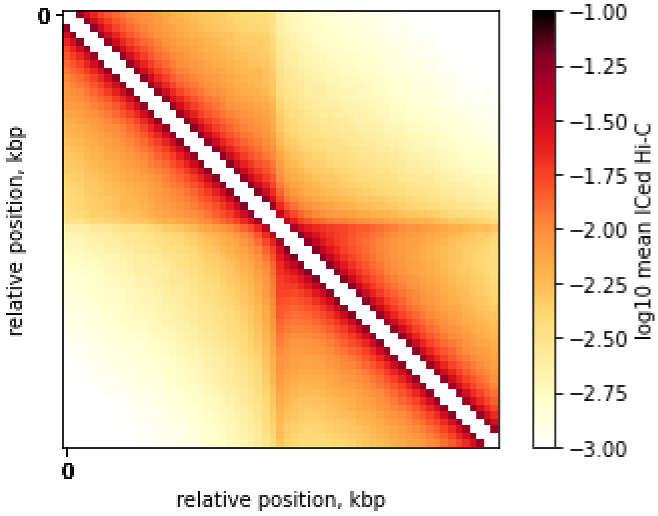
```
plt.imshow(
    np.log10(mtx),
    vmin = -3,
    vmax = -1,
```

(continues on next page)

(continued from previous page)

```
    cmap='fall',
    interpolation='none')

plt.colorbar(label = 'log10 mean ICed Hi-C')
ticks_pixels = np.linspace(0, flank*2//resolution,5)
ticks_kbp = ((ticks_pixels-ticks_pixels[-1]/2)*resolution//1000).astype(int)
plt.xticks(ticks_pixels, ticks_kbp)
plt.yticks(ticks_pixels, ticks_kbp)
plt.xlabel('relative position, kbp')
plt.ylabel('relative position, kbp')

plt.show()
```

```
/var/folders/4s/d866wm3s4zbc9m41334fxfwr0000gp/T/ipykernel_33615/2426526626.py:2:␣
↪RuntimeWarning: divide by zero encountered in log10
  np.log10(mtx),
```



### On-diagonal pileup of observed over expected interactions

Sometimes you don't want to include the **distance decay** P(s) in your pileups. For example, when you make comparison of pileups between experiments and they have different P(s). Even if these differences are slight, they might affect the pileup of raw ICed Hi-C interactions.

In this case, the observed over expected pileup is your choice. Prior to running the pileup function, you need to calculate expected interactions for chromosome arms.

[18]:
```
expected = cooltools.expected_cis(clr, view_df=hg38_arms, nproc=2, chunksize=1_000_000)
```

[19]:
```
expected
```

[19]:
```
      region1  region2  dist  n_valid  count.sum  balanced.sum   count.avg  \
0      chr2_p   chr2_p     0     8771        NaN           NaN         NaN
1      chr2_p   chr2_p     1     8753        NaN           NaN         NaN
2      chr2_p   chr2_p     2     8745  2656738.0    406.013088  303.800800
3      chr2_p   chr2_p     3     8741  1563363.0    237.585271  178.854021
```

(continues on next page)

(continued from previous page)

```
4       chr2_p   chr2_p     4     8738  1125674.0    169.308714  128.825132
...        ...      ...   ...      ...        ...           ...         ...
32543  chr17_q  chr17_q  5850        0        0.0      0.000000         NaN
32544  chr17_q  chr17_q  5851        0        0.0      0.000000         NaN
32545  chr17_q  chr17_q  5852        0        0.0      0.000000         NaN
32546  chr17_q  chr17_q  5853        0        0.0      0.000000         NaN
32547  chr17_q  chr17_q  5854        0        0.0      0.000000         NaN

       balanced.avg  balanced.avg.smoothed  balanced.avg.smoothed.agg
0               NaN                    NaN                        NaN
1               NaN               0.000495                   0.000520
2          0.046428               0.042469                   0.044728
3          0.027181               0.026796                   0.028226
4          0.019376               0.019097                   0.020152
...             ...                    ...                        ...
32543           NaN               0.000010                   0.000006
32544           NaN               0.000010                   0.000006
32545           NaN               0.000010                   0.000006
32546           NaN               0.000010                   0.000006
32547           NaN               0.000010                   0.000006

[32548 rows x 10 columns]
```

[20]:

```
# Create the stack of snips:
stack = cooltools.pileup(clr, sites, view_df=hg38_arms, expected_df=expected, flank=300_
↪000)

# Mirror reflect snippets when the feature is on the opposite strand
mask = np.array(sites.strand == '-', dtype=bool)
stack[:, :, mask] = stack[::-1, ::-1, mask]

mtx = np.nanmean(stack, axis=2)
```

[21]:

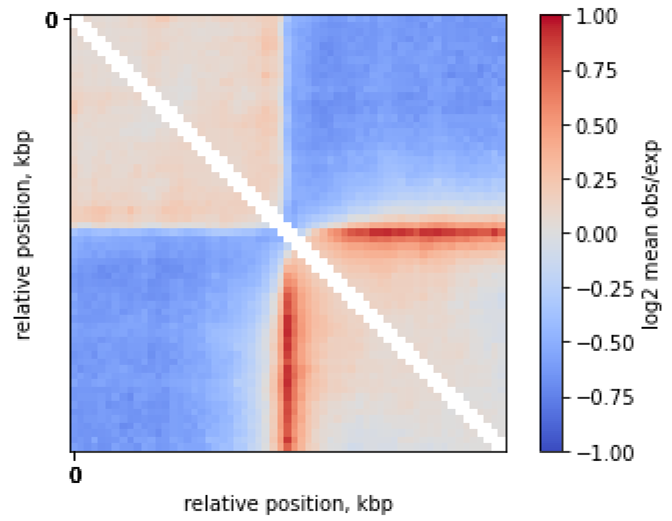
```
plt.imshow(
    np.log2(mtx),
    vmax = 1.0,
    vmin = -1.0,
    cmap='coolwarm',
    interpolation='none')

plt.colorbar(label = 'log2 mean obs/exp')
ticks_pixels = np.linspace(0, flank*2//resolution,5)
ticks_kbp = ((ticks_pixels-ticks_pixels[-1]/2)*resolution//1000).astype(int)
plt.xticks(ticks_pixels, ticks_kbp)
plt.yticks(ticks_pixels, ticks_kbp)
plt.xlabel('relative position, kbp')
plt.ylabel('relative position, kbp')

plt.show()
```

```
/var/folders/4s/d866wm3s4zbc9m41334fxfwr0000gp/T/ipykernel_33615/2557000624.py:2:␣
↪RuntimeWarning: divide by zero encountered in log2
```

(continues on next page)

(continued from previous page)

```
  np.log2(mtx),
```



### Inspect the snips

Aggregation is a convenient though dangerous step. It averages your data so that you cannot distinguish whether the signal is indeed average, or there is a single dataset that introduces a bias to your analysis. To make sure there are no outliers, you may want to use inspection of individual snippets.

The cell below shows one way to interactively investigate snippets contributing to a pileup. Note that this is not interactive on readthedocs, but can be run if the notebook is obtained from open2c_examples. This widget sorts the dataframe with CTCF motifs by the strength of binding. This allows us to inspect the Micro-C maps at the positions of the strongest and weakest CTCF sites. Run the cell below and try to compare snippets with the lowest score to the snippets with the largest score.

```
[22]: from ipywidgets import interact
      from matplotlib.gridspec import GridSpec

      n_examples = len(sites)

      @interact(i=(0, n_examples-1))
      def f(i):
          fig, ax = plt.subplots(figsize=[5,5])
          img = ax.matshow(
              np.log2(stack[:, :, i]),
              vmin=-1,
              vmax=1,
              extent=[-flank//1000, flank//1000, -flank//1000, flank//1000],
              cmap='coolwarm'
          )
          ax.xaxis.tick_bottom()
          if i > 0:
              ax.yaxis.set_visible(False)
          plt.title(f'{i+1}-th snippet from top \n FC score: {sites.loc[i, "FC_score"]:.2f}\n␣
      ↪and motif score: {sites.loc[i, "score"]:.2f}')
```

(continues on next page)

(continued from previous page)

```
    plt.axvline(0, c='g', ls=':')
    plt.axhline(0, c='g', ls=':')
```

```
interactive(children=(IntSlider(value=306, description='i', max=612), Output()), _dom_
↪classes=('widget-interac...
```

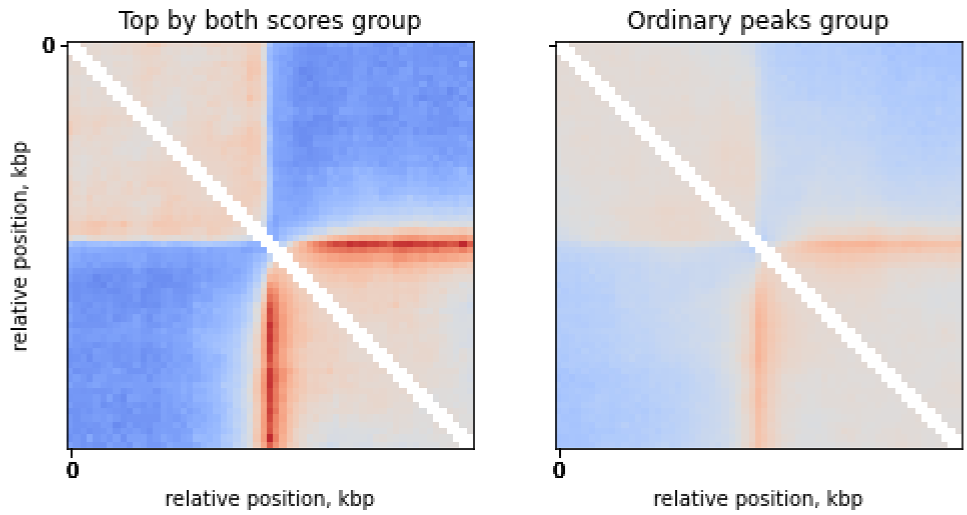
### Compare top strongest peaks with others

Compare the top peaks with both motif score and FC score to the rest of the peaks:

```
[23]: # Create the stack of snips:
stack = cooltools.pileup(clr, ctcf, view_df=hg38_arms, expected_df=expected, flank=300_
↪000
            )

# Mirror reflect snippets where the feature is on the opposite strand
mask = np.array(ctcf.strand == '-', dtype=bool)
stack[:, :, mask] = stack[::-1, ::-1, mask]

mtx = np.nanmean(stack, axis=2)
```

```
[24]: # TODO: add some strength of insulation for the pileup?

groups = ['Top by both scores', 'Ordinary peaks']
n_groups = len(groups)

ticks_pixels = np.linspace(0, flank*2//resolution,5)
ticks_kbp = ((ticks_pixels-ticks_pixels[-1]/2)*resolution//1000).astype(int)

fig, axs = plt.subplots(1, n_groups, sharex=True, sharey=True, figsize=(4*n_groups, 4))
for i in range(n_groups):
    mtx = np.nanmean( stack[:, :, ctcf['peaks_importance']==groups[i]], axis=2)
    ax = axs[i]
    ax.imshow(
        np.log2(mtx),
        vmax = 1.0,
        vmin = -1.0,
        cmap='coolwarm',
        interpolation='none')

    ax.set(title=f'{groups[i]} group',
           xticks=ticks_pixels,
           xticklabels=ticks_kbp,
           xlabel='relative position, kbp')

axs[0].set(yticks=ticks_pixels,
       yticklabels=ticks_kbp,
       ylabel='relative position, kbp')

plt.show()
```

```
/var/folders/4s/d866wm3s4zbc9m41334fxfwr0000gp/T/ipykernel_33615/2210978640.py:14:␣
↪RuntimeWarning: divide by zero encountered in log2
  np.log2(mtx),
```



### Off-diagonal pileup

**Off-diagonal pileups** are the averaged Hi-C maps around double anchors. In this case, the anchors are CTCF sites in the genome.

[25]:
```
paired_sites = bioframe.pair_by_distance(sites, min_sep=200000, max_sep=1000000,␣
↪suffixes=('1', '2'))
paired_sites.loc[:, 'mid1'] = (paired_sites['start1'] + paired_sites['end1'])//2
paired_sites.loc[:, 'mid2'] = (paired_sites['start2'] + paired_sites['end2'])//2
```

[26]:
```
print(len(paired_sites))
paired_sites.head()
```

```
1634
```

[26]:
```
  chrom1  start1    end1               name1      score1 strand1    mid1  \
0  chr17  412407  412426  MA0139.1_CTCF_human  12.212548       +  412416
1  chr17  412407  412426  MA0139.1_CTCF_human  12.212548       +  412416
2  chr17  412407  412426  MA0139.1_CTCF_human  12.212548       +  412416
3  chr17  412407  412426  MA0139.1_CTCF_human  12.212548       +  412416
4  chr17  412407  412426  MA0139.1_CTCF_human  12.212548       +  412416

   FC_score1  quartile_score1  quartile_FC_score1  ...     score2  strand2  \
0  41.645123                4                   4  ...  13.272118        -
1  41.645123                4                   4  ...  13.996208        -
2  41.645123                4                   4  ...  14.735101        +
3  41.645123                4                   4  ...  13.983562        +
4  41.645123                4                   4  ...  12.045221        +

      mid2  FC_score2 quartile_score2 quartile_FC_score2   peaks_importance2  \
0  1056231  35.072572               4                  4  Top by both scores
```

(continues on next page)

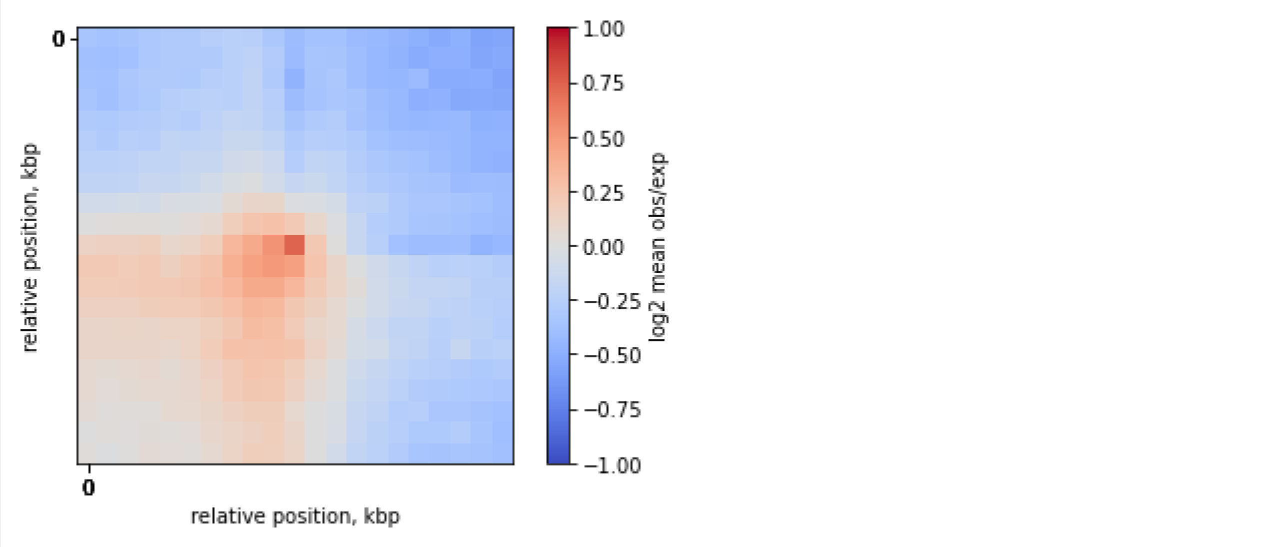
(continued from previous page)

```
1  1187374  69.994562                4                   4  Top by both scores
2  1259280  31.643758                4                   4  Top by both scores
3  1276338  35.440247                4                   4  Top by both scores
4  1365609  36.746886                4                   4  Top by both scores

  cluster2  cluster_start2 cluster_end2
0        1         1056222      1056241
1        2         1187365      1187384
2        3         1259271      1259290
3        4         1276329      1276348
4        5         1365600      1365619

[5 rows x 28 columns]
```

For pileup, we will use the expected calculated above:

[27]:
```
# create the stack of snips:
stack = cooltools.pileup(clr, paired_sites, view_df=hg38_arms, expected_df=expected,␣
↪flank=100_000)

mtx = np.nanmean(stack, axis=2)
```

[28]:
```
plt.imshow(
    np.log2(mtx),
    vmax = 1,
    vmin = -1,
    cmap='coolwarm')

plt.colorbar(label = 'log2 mean obs/exp')
ticks_pixels = np.linspace(0, flank*2//resolution,5)
ticks_kbp = ((ticks_pixels-ticks_pixels[-1]/2)*resolution//1000).astype(int)
plt.xticks(ticks_pixels, ticks_kbp)
plt.yticks(ticks_pixels, ticks_kbp)
plt.xlabel('relative position, kbp')
plt.ylabel('relative position, kbp')

plt.show()
```



This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

### 1.3.7 Command line interface

Welcome to the cooltools command line interface (CLI) notebook!

Cooltools features a paired python API & CLI that enables user-facing functions to be run from the command line.

[1]:
```
import os, subprocess
import pandas as pd
import bioframe
import cooltools
import cooler
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
plt.rcParams['font.size']=12

from packaging import version
if version.parse(cooltools.__version__) < version.parse('0.5.2'):
    raise AssertionError("tutorials rely on cooltools version 0.5.2 or higher,"+
                         "please check your cooltools version and update to the latest")

# We can use this function to display a file within the notebook
from IPython.display import Image


# download test data
# this file is 145 Mb, and may take a few seconds to download
cool_file = cooltools.download_data("HFF_MicroC", cache=True, data_dir='./data/')
print(cool_file)

# To use this variable in a bash call from jupyter just use $cool_file
```

```
./data/test.mcool
```

[2]:

```
%%bash
mkdir -p data
mkdir -p outputs
```

[3]:

```
# Note for the correct bash environment to be recognized from jupyter with the ! magic,
# jupyter notebook must be initialized from a conda environment with cooler and␣
↪cooltools installed.
```

To see a list of CLI commands for cooltools, see the help:

[4]:

```
!cooltools -h
```

```
Usage: cooltools [OPTIONS] COMMAND [ARGS]...

  Type -h or --help after any subcommand for more information.

Options:
  -v, --verbose  Verbose logging
  -d, --debug    Post mortem debugging
  -V, --version  Show the version and exit.
  -h, --help     Show this message and exit.

Commands:
  coverage        Calculate the sums of cis and genome-wide contacts (aka...
  dots            Call dots on a Hi-C heatmap that are not larger than...
  eigs-cis        Perform eigen value decomposition on a cooler matrix to...
  eigs-trans      Perform eigen value decomposition on a cooler matrix to...
  expected-cis    Calculate expected Hi-C signal for cis regions of...
  expected-trans  Calculate expected Hi-C signal for trans regions of...
  genome          Utilities for binned genome assemblies.
  insulation      Calculate the diamond insulation scores and call...
  pileup          Perform retrieval of the snippets from .cool file.
  random-sample   Pick a random sample of contacts from a Hi-C map.
  saddle          Calculate saddle statistics and generate saddle plots...
  virtual4c       Generate virtual 4C profile from a contact map by...
```

### Visualization

[5]:

```
!cooler show $cool_file::resolutions/1000000 'chr2' -o 'outputs/chr2.png'
```

```
Traceback (most recent call last):
  File "/Users/geoffudenberg/anaconda3/envs/open2c/bin/cooler", line 8, in <module>
    sys.exit(cli())
  File "/Users/geoffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/click/
↪core.py", line 1130, in __call__
    return self.main(*args, **kwargs)
  File "/Users/geoffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/click/
↪core.py", line 1055, in main
    rv = self.invoke(ctx)
  File "/Users/geoffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/click/
```

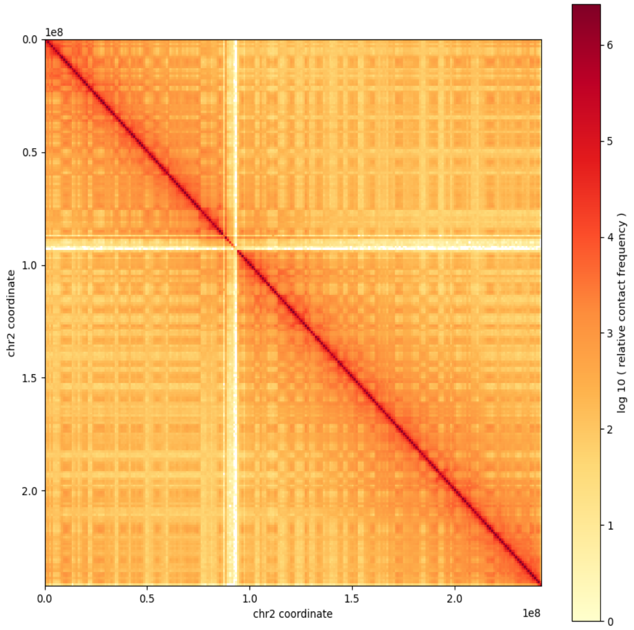
(continues on next page)

(continued from previous page)

```
↪core.py", line 1657, in invoke
    return _process_result(sub_ctx.command.invoke(sub_ctx))
  File "/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/click/
↪core.py", line 1404, in invoke
    return ctx.invoke(self.callback, **ctx.params)
  File "/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/click/
↪core.py", line 760, in invoke
    return __callback(*args, **kwargs)
  File "/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooler/
↪cli/show.py", line 230, in show
    plt.gcf().canvas.set_window_title("Contact matrix".format())
AttributeError: 'FigureCanvasAgg' object has no attribute 'set_window_title'
```

[6]:
```
Image('outputs/chr2.png', width=400, height=400)
```

[6]:



## Expected

Tables of expected counts, either in cis or trans, are key inputs for many downstream analyses in cooltools. For more details, see the contacts_vs_dist notebook.

Typically, we specify a view to define regions under analysis. Here we quickly create a view that specifies chromosome arms using tables of chromosome sizes and centromere positions.

[7]:
```
# create a view of hg38 chromosome arms using chromosome sizes and definition of␣
↪centromeres
hg38_chromsizes = bioframe.fetch_chromsizes('hg38')
```

(continues on next page)

(continued from previous page)

```
hg38_cens = bioframe.fetch_centromeres('hg38')
view_hg38 = bioframe.make_chromarms(hg38_chromsizes,  hg38_cens)

# select only those chromosomes available in cooler
clr = cooler.Cooler(f'{cool_file}::/resolutions/1000000')
view_hg38 = view_hg38[view_hg38.chrom.isin(clr.chromnames)].reset_index(drop=True)
view_hg38.to_csv("data/view_hg38.tsv", index=False, header=False, sep='\t')
```

[8]:
```
! cooltools expected-cis $cool_file::resolutions/100000 --nproc 6 -o 'outputs/test.
↪expected.cis.100000.tsv' --view "data/view_hg38.tsv"
```

Note expected for the first two distances are not defined with default settings, due to masking of near-diagonals in the cooler.

[9]:
```
display(
    pd.read_table("outputs/test.expected.cis.100000.tsv")[0:5]
)
```

```
  region1 region2  dist  n_valid  count.sum  balanced.sum    count.avg  \
0  chr2_p  chr2_p     0      878        NaN           NaN          NaN
1  chr2_p  chr2_p     1      876        NaN           NaN          NaN
2  chr2_p  chr2_p     2      874  2738583.0     65.287351  3133.390160
3  chr2_p  chr2_p     3      872  1739972.0     41.011675  1995.380734
4  chr2_p  chr2_p     4      870  1184707.0     28.473626  1361.732184

   balanced.avg
0           NaN
1           NaN
2      0.074699
3      0.047032
4      0.032728
```

### Compartments & saddles

Many contact maps display plaid patterns of interactions. For more detail see the compartments notebook.

Since the orientation of eigenvectors is determined up to a sign, we often use GC content to orient or "phase" eigenvectors. Before calculating comparments, this notebook generates a binned GC content profile for the relevant region.

[10]:
```
## fasta sequence is required for calculating binned profile of GC conent
if not os.path.isfile('./data/hg38.fa'):
    ## note downloading a ~1Gb file can take a minute
    subprocess.call('wget -P ./data --progress=bar:force:noscroll https://hgdownload.cse.
↪ucsc.edu/goldenpath/hg38/bigZips/hg38.fa.gz', shell=True)
    subprocess.call('gunzip ./data/hg38.fa.gz', shell=True)
```

Bins can be fetched from the cooler, dropping any weights columns & keeping the header.

[11]:
```
!cooler dump --header -t bins $cool_file::resolutions/100000 | cut -f1-3 > outputs/bins.
↪100000.tsv
```

```
[12]: !cooltools genome gc outputs/bins.100000.tsv data/hg38.fa > outputs/gc.100000.tsv
```

```
[13]: !cooltools eigs-cis -o outputs/test.eigs.100000 --view data/view_hg38.tsv --phasing-
      ↪track outputs/gc.100000.tsv --n-eigs 1 $cool_file::resolutions/100000
```

```
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/lib/
↪checks.py:550: FutureWarning: In a future version of pandas, a length 1 tuple will be␣
↪returned when iterating over a groupby with a grouper equal to a list of length 1. Don
↪'t supply a list with a single grouper to avoid this warning.
  for name, group in track.groupby([track.columns[0]]):
```

```
[14]: display(
          pd.read_table('outputs/test.eigs.100000.cis.vecs.tsv')[0:5]
      )
```

```
  chrom   start     end    weight        E1
0  chr2       0  100000  0.006754 -1.564658
1  chr2  100000  200000  0.006767 -1.747567
2  chr2  200000  300000  0.004638 -0.370827
3  chr2  300000  400000  0.006034 -1.326894
4  chr2  400000  500000  0.006153 -1.434981
```

Pairwise class averaging is a typical way to reveal preferences in contact frequencies between regions.

```
[15]: !cooltools saddle --qrange 0.02 0.98 --fig png -o outputs/test.saddle.cis.100000 --view␣
      ↪data/view_hg38.tsv $cool_file::resolutions/100000 outputs/test.eigs.100000.cis.vecs.
      ↪tsv outputs/test.expected.cis.100000.tsv
```

```
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/lib/
↪checks.py:550: FutureWarning: In a future version of pandas, a length 1 tuple will be␣
↪returned when iterating over a groupby with a grouper equal to a list of length 1. Don
↪'t supply a list with a single grouper to avoid this warning.
  for name, group in track.groupby([track.columns[0]]):
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/lib/
↪checks.py:550: FutureWarning: In a future version of pandas, a length 1 tuple will be␣
↪returned when iterating over a groupby with a grouper equal to a list of length 1. Don
↪'t supply a list with a single grouper to avoid this warning.
  for name, group in track.groupby([track.columns[0]]):
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/lib/
↪checks.py:550: FutureWarning: In a future version of pandas, a length 1 tuple will be␣
↪returned when iterating over a groupby with a grouper equal to a list of length 1. Don
↪'t supply a list with a single grouper to avoid this warning.
  for name, group in track.groupby([track.columns[0]]):
/Users/geofffudenberg/anaconda3/envs/open2c/lib/python3.9/site-packages/cooltools/lib/
↪checks.py:550: FutureWarning: In a future version of pandas, a length 1 tuple will be␣
↪returned when iterating over a groupby with a grouper equal to a list of length 1. Don
↪'t supply a list with a single grouper to avoid this warning.
  for name, group in track.groupby([track.columns[0]]):
```

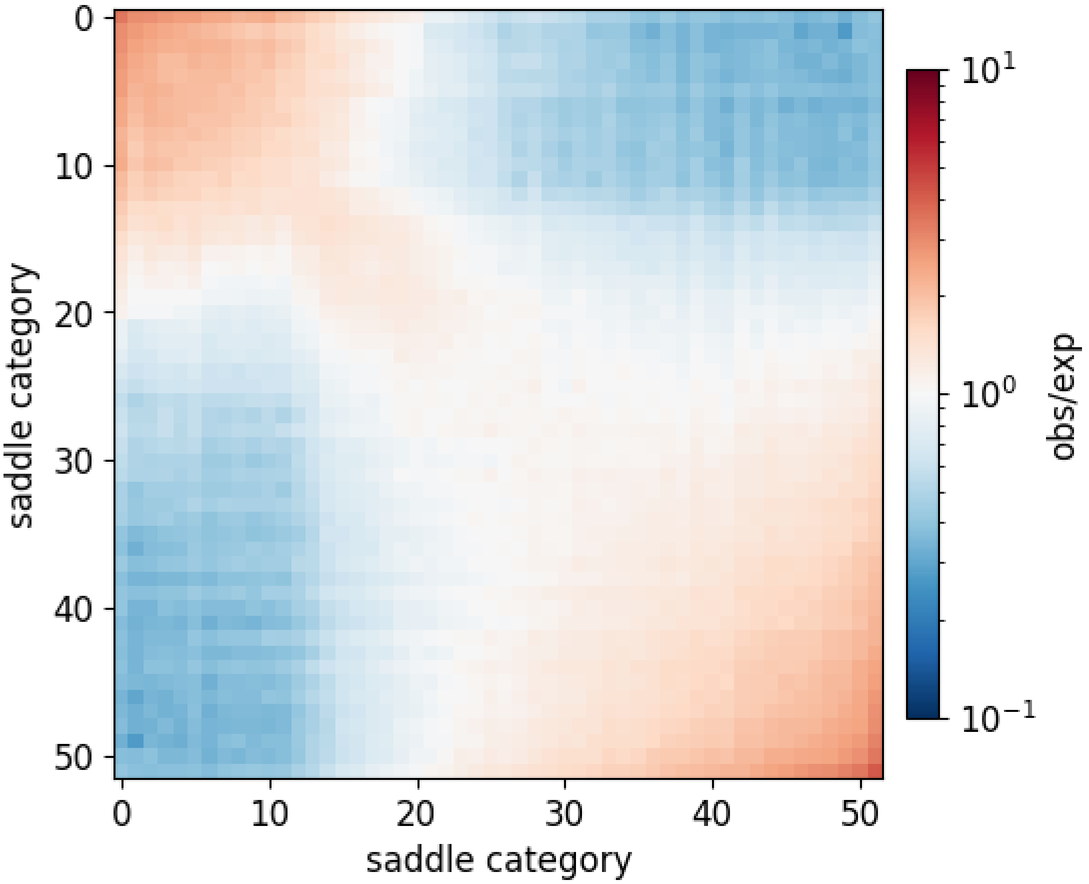
```
[16]: saddle = np.load('outputs/test.saddle.cis.100000.saddledump.npz', allow_pickle=True)
```

```
[17]: plt.figure(figsize=(6,6))
      norm = LogNorm(    vmin=10**(-1), vmax=10**1)
```

(continues on next page)

(continued from previous page)

```
im = plt.imshow(
    saddle['saddledata'],
    cmap='RdBu_r',
    norm = norm
);
plt.xlabel("saddle category")
plt.ylabel("saddle category")
plt.colorbar(im, label='obs/exp', pad=0.025, shrink=0.7);
```



### Insulation & boundaries

A common strategy to summarize the near-diagonal structure of a contact map is by computing insulation scores. See the insulation notebook for more details.

The command below uses the Li method to threshold the insulation score for boundary calling, so the resulting table has both log2 insulation scores and whether a given region is a boundary. The boundary_strength_{window} column has a value for all local minima of the insulation profile, and the is_boundary_{window} column indicates whether this strength passed the threshold. Note that {window} indicates the chosen size of the sliding diamond, and the command below uses windows of size 100kb and 200kb.

```
[18]: ! cooltools insulation --threshold Li -o 'outputs/test.insulation.10000.tsv' --view
      ↪"data/view_hg38.tsv" $cool_file::resolutions/10000 100000 200000
```

[19]:

```
display(
    pd.read_table('outputs/test.insulation.10000.tsv')[0:5]
)
```

```
  chrom  start    end  region  is_bad_bin  log2_insulation_score_100000  \
0  chr2      0  10000  chr2_p        True                           NaN
1  chr2  10000  20000  chr2_p       False                      0.692051
2  chr2  20000  30000  chr2_p       False                      0.760561
3  chr2  30000  40000  chr2_p       False                      0.766698
4  chr2  40000  50000  chr2_p       False                      0.674906

   n_valid_pixels_100000  log2_insulation_score_200000  n_valid_pixels_200000  \
0                    0.0                           NaN                    0.0
1                    8.0                      1.123245                   18.0
2                   17.0                      1.196643                   37.0
3                   27.0                      1.211748                   57.0
4                   37.0                      1.135037                   77.0

   boundary_strength_100000  boundary_strength_200000  is_boundary_100000  \
0                       NaN                       NaN               False
1                       NaN                       NaN               False
2                       NaN                       NaN               False
3                       NaN                       NaN               False
4                       NaN                       NaN               False

   is_boundary_200000
0               False
1               False
2               False
3               False
4               False
```

### Dots & focal enrichment

Punctate pairwise peaks of enriched contact frequency are a prevalent feature of mammalian interphase contact maps. See the dots notebook for more details.

Since dots are evident at higher resolutions, we first calculate the 10kb expected, and we used multiple cores to speed up the calculation.

[20]:

```
! cooltools expected-cis --nproc 6 -o 'outputs/test.expected.cis.10000.tsv' --view "data/
↪view_hg38.tsv" $cool_file::resolutions/10000
```

[21]:

```
! cooltools dots --nproc 6 -o 'outputs/test.dots.10000.tsv' --view "data/view_hg38.tsv"
↪$cool_file::resolutions/10000 outputs/test.expected.cis.10000.tsv
```

```
INFO:root:Using recommended donut-based kernels with w=5, p=2 for binsize=10000
INFO:root: matrix 9314X9314 to be split into 256 tiles of 600X600.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 14907X14907 to be split into 625 tiles of 600X600.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root: matrix 2472X2472 to be split into 25 tiles of 600X600.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
```

(continues on next page)

(continued from previous page)

```
INFO:root: matrix 5855X5855 to be split into 100 tiles of 600X600.
INFO:root: tiles are padded (width=5) to enable convolution near the edges
INFO:root:convolving 108 tiles to build histograms for lambda-bins
INFO:root:creating a Pool of 6 workers to tackle 108 tiles
INFO:root:Done building histograms in 17.489 sec ...
INFO:root:Determined thresholds for every lambda-bin ...
INFO:root:convolving 108 tiles to extract enriched pixels
INFO:root:creating a Pool of 6 workers to tackle 108 tiles
INFO:root:Done extracting enriched pixels in 18.971 sec ...
INFO:root:Begin post-processing of 11284 filtered pixels
INFO:root:preparing to extract needed q-values ...
INFO:root:clustering enriched pixels in region: chr17_p
INFO:root:detected 198 clusters of 3.69+/-3.15 size
INFO:root:clustering enriched pixels in region: chr17_q
INFO:root:detected 584 clusters of 3.87+/-3.44 size
INFO:root:clustering enriched pixels in region: chr2_p
INFO:root:detected 841 clusters of 3.82+/-3.56 size
INFO:root:clustering enriched pixels in region: chr2_q
INFO:root:detected 1364 clusters of 3.72+/-3.24 size
INFO:root:Clustering is complete
INFO:root:filtered 2548 out of 2987 centroids to reduce the number of false-positives
```

[22]:

```
display(
    pd.read_table('outputs/test.dots.10000.tsv')[0:5]
)
```

```
  chrom1    start1      end1 chrom2    start2      end2  count  \
0  chr17  12250000  12260000  chr17  12740000  12750000    186
1  chr17  10640000  10650000  chr17  11980000  11990000     64
2  chr17   9920000   9930000  chr17  10610000  10620000    444
3  chr17  10640000  10650000  chr17  10840000  10850000    340
4  chr17  12180000  12190000  chr17  13000000  13010000    215

   la_exp.donut.value  la_exp.vertical.value  la_exp.horizontal.value  ...  \
0           72.485199              79.603089                72.129599  ...
1           20.549081              23.891314                25.102296  ...
2           40.054241              75.104470                41.227788  ...
3           34.137182              46.658333                48.204555  ...
4           28.235042              36.990608                38.266379  ...

   la_exp.vertical.qval  la_exp.horizontal.qval  la_exp.lowleft.qval   region  \
0          2.210430e-22            2.032096e-22         1.506741e-33  chr17_p
1          1.368050e-08            1.294897e-08         8.847705e-09  chr17_p
2         3.310219e-171           7.071107e-246        2.515747e-246  chr17_p
3         2.637572e-154           1.768044e-154        4.524004e-183  chr17_p
4          7.841810e-80            1.045765e-79         7.967002e-80  chr17_p

        cstart1       cstart2  c_label  c_size  region1  region2
0  1.220571e+07  1.273857e+07        0       7  chr17_p  chr17_p
1  1.063000e+07  1.195500e+07        1       4  chr17_p  chr17_p
2  9.920000e+06  1.061200e+07        2       4  chr17_p  chr17_p
3  1.063667e+07  1.083889e+07        3       9  chr17_p  chr17_p
```

(continues on next page)

(continued from previous page)

```
4  1.218571e+07  1.299143e+07        4        7  chr17_p  chr17_p

[5 rows x 22 columns]
```

### Pileups & average features

A common method for quantifying contact maps is by creating pileup plots, which are averages over a set of “snippets”, or 2D windows, from the genome-wide map. See the pileups notebook for more details.

Below shows pileup for dots that were computed above.

[23]:

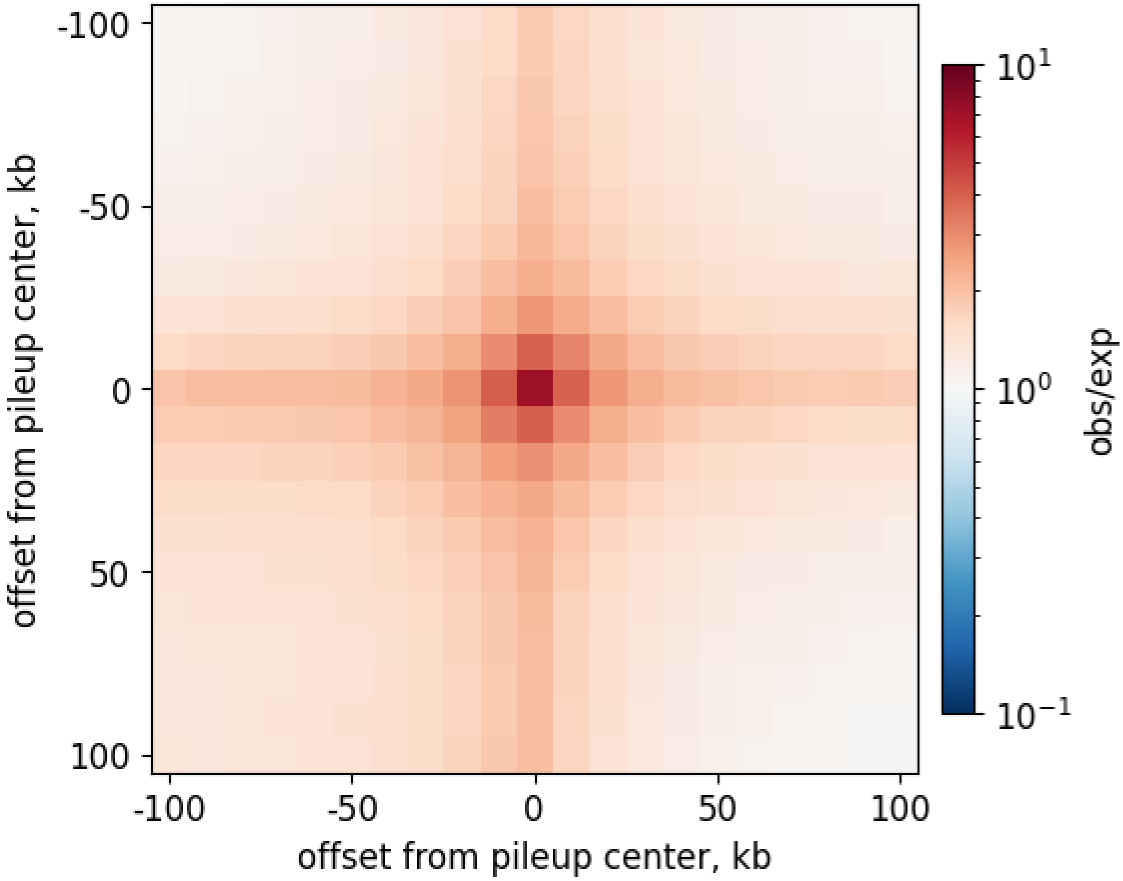
```
!cooltools pileup --features-format BEDPE --nproc 6 -o 'outputs/test.pileup.10000.npz' --
↪view "data/view_hg38.tsv" --expected outputs/test.expected.cis.10000.tsv $cool_file::
↪resolutions/10000  outputs/test.dots.10000.tsv
```

[24]:

```
## output is saved as an npz with keys 'pileup' and 'stack'
resolution = 10000
pile = np.load( f'outputs/test.pileup.{resolution}.npz')
```

[25]:

```
plt.figure(figsize=(6,6))
norm = LogNorm(    vmin=10**(-1), vmax=10**1)

im = plt.imshow(
    pile['pileup'],
    cmap='RdBu_r',
    norm = norm
);

plt.xticks(np.arange(0,25,5).astype(int),
           (np.arange(0,25,5).astype(int)-10)*resolution//1000,
)
plt.yticks(np.arange(0,25,5).astype(int),
           (np.arange(0,25,5).astype(int)-10)*resolution//1000,
)
plt.xlabel("offset from pileup center, kb")
plt.ylabel("offset from pileup center, kb")

plt.colorbar(im, label='obs/exp', pad=0.025, shrink=0.7);
```



This page was generated with nbsphinx from /home/docs/checkouts/readthedocs.org/user_builds/cooltools/checkouts/stable/docs/noteboo

Note that these notebooks currently focus on mammalian interphase Hi-C analysis, but are readily extendible to other organisms and cellular contexts. To clone and work interactively with these notebooks, visit: https://github.com/open2c/open2c_examples.

## 1.3.8 CLI Reference

### cooltools

Type -h or –help after any subcommand for more information.

```
cooltools [OPTIONS] COMMAND [ARGS]...
```

### Options

**-v, --verbose**
Verbose logging

**-d, --debug**
Post mortem debugging

**-V, --version**
Show the version and exit.

### coverage

Calculate the sums of cis and genome-wide contacts (aka coverage aka marginals) for a sparse Hi-C contact map in Cooler HDF5 format. Note that the sum(tot_cov) from this function is two times the number of reads contributing to the cooler, as each side contributes to the coverage.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map.

```
cooltools coverage [OPTIONS] COOL_PATH
```

### Options

**-o, --output** <output>
Specify output file name to store the coverage in a tsv format.

**--ignore-diags** <ignore_diags>
The number of diagonals to ignore. By default, equals the number of diagonals ignored during IC balancing.

**--store**
Append columns with coverage (cov_cis_raw, cov_tot_raw), or (cov_cis_clr_weight_name, cov_tot_clr_weight_name) if calculating balanced coverage, to the cooler bin table. If clr_weight_name=None, also stores total cis counts in the cooler info

**--chunksize** <chunksize>
Split the contact matrix pixel records into equally sized chunks to save memory and/or parallelize. Default is 10^7

**Default**
`10000000.0`

**--bigwig**
Also save output as bigWig files for cis and total coverage with the names <output>.<cis/tot>.bw

**--clr_weight_name** <clr_weight_name>
Name of the weight column. Specify to calculate coverage of balanced cooler.

**-p, --nproc** <nproc>
Number of processes to split the work between. [default: 1, i.e. no process pool]

### Arguments

**COOL_PATH**

Required argument

## dots

Call dots on a Hi-C heatmap that are not larger than max_loci_separation.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map.

EXPECTED_PATH : The paths to a tsv-like file with expected signal, including a header. Use the '::' syntax to specify a column name.

Analysis will be performed for chromosomes referred to in EXPECTED_PATH, and therefore these chromosomes must be a subset of chromosomes referred to in COOL_PATH. Also chromosomes refered to in EXPECTED_PATH must be non-trivial, i.e., contain not-NaN signal. Thus, make sure to prune your EXPECTED_PATH before applying this script.

COOL_PATH and EXPECTED_PATH must be binned at the same resolution.

EXPECTED_PATH must contain at least the following columns for cis contacts: 'region1/2', 'dist', 'n_valid', value_name. value_name is controlled using options. Header must be present in a file.

```
cooltools dots [OPTIONS] COOL_PATH EXPECTED_PATH
```

### Options

**--view, --regions** <view>

Path to a BED file with the definition of viewframe (regions) used in the calculation of EXPECTED_PATH. Dot-calling will be performed for these regions independently e.g. chromosome arms. Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--clr-weight-name** <clr_weight_name>

Use cooler balancing weight with this name.

**Default**
`weight`

**-p, --nproc** <nproc>

Number of processes to split the work between. [default: 1, i.e. no process pool]

**--max-loci-separation** <max_loci_separation>

Limit loci separation for dot-calling, i.e., do not call dots for loci that are further than max_loci_separation basepair apart. 2-20MB is reasonable and would capture most of CTCF-dots.

**Default**
`2000000`

**--max-nans-tolerated** <max_nans_tolerated>

Maximum number of NaNs tolerated in a footprint of every used filter. Must be controlled with caution, as large max-nans-tolerated, might lead to pixels scored in the padding area of the tiles to "penetrate" to the list of scored pixels for the statistical testing. [max-nans-tolerated <= 2*w ]

**Default**
`1`

**--tile-size** <tile_size>

Tile size for the Hi-C heatmap tiling. Typically on order of several mega-bases, and <= max_loci_separation.

**Default**
6000000

**--num-lambda-bins** <num_lambda_bins>

Number of log-spaced bins to divide your adjusted expected between. Same as HiCCUPS_W1_MAX_INDX (40) in the original HiCCUPS.

**Default**
45

**--fdr** <fdr>

False discovery rate (FDR) to control in the multiple hypothesis testing BH-FDR procedure.

**Default**
0.02

**--clustering-radius** <clustering_radius>

Radius for clustering dots that have been called too close to each other.Typically on order of 40 kilo-bases, and >= binsize.

**Default**
39000

**-v, --verbose**

Enable verbose output

**-o, --output** <output>

**Required** Specify output file name to store called dots in a BEDPE-like format

### Arguments

**COOL_PATH**

Required argument

**EXPECTED_PATH**

Required argument

### eigs-cis

Perform eigen value decomposition on a cooler matrix to calculate compartment signal by finding the eigenvector that correlates best with the phasing track.

COOL_PATH : the paths to a .cool file with a balanced Hi-C map. Use the '::' syntax to specify a group path in a multicooler file.

TRACK_PATH : the path to a BedGraph-like file that stores phasing track as track-name named column.

BedGraph-like format assumes tab-separated columns chrom, start, stop and track-name.

```
cooltools eigs-cis [OPTIONS] COOL_PATH
```

### Options

**--phasing-track** <TRACK_PATH>

Phasing track for orienting and ranking eigenvectors,provided as /path/to/track::track_value_column_name.

**--view, --regions** <view>

Path to a BED file which defines which regions of the chromosomes to use (only implemented for cis contacts). Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--n-eigs** <n_eigs>

Number of eigenvectors to compute.

**Default**
`3`

**--clr-weight-name** <clr_weight_name>

Use balancing weight with this name. Using raw unbalanced data is not currently supported for eigenvectors.

**Default**
`weight`

**--ignore-diags** <ignore_diags>

The number of diagonals to ignore. By default, equals the number of diagonals ignored during IC balancing.

**-v, --verbose**

Enable verbose output

**-o, --out-prefix** <out_prefix>

**Required** Save compartment track as a BED-like file. Eigenvectors and corresponding eigenvalues are stored in out_prefix.contact_type.vecs.tsv and out_prefix.contact_type.lam.txt

**--bigwig**

Also save compartment track (E1) as a bigWig file with the name out_prefix.contact_type.bw

### Arguments

**COOL_PATH**

Required argument

### eigs-trans

Perform eigen value decomposition on a cooler matrix to calculate compartment signal by finding the eigenvector that correlates best with the phasing track.

COOL_PATH : the paths to a .cool file with a balanced Hi-C map. Use the '::' syntax to specify a group path in a multicooler file.

TRACK_PATH : the path to a BedGraph-like file that stores phasing track as track-name named column.

BedGraph-like format assumes tab-separated columns chrom, start, stop and track-name.

```
cooltools eigs-trans [OPTIONS] COOL_PATH
```

### Options

**--phasing-track** <TRACK_PATH>

Phasing track for orienting and ranking eigenvectors,provided as /path/to/track::track_value_column_name.

**--view, --regions** <view>

Path to a BED file which defines which regions of the chromosomes to use (only implemented for cis contacts). Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--n-eigs** <n_eigs>

Number of eigenvectors to compute.

**Default**
3

**--clr-weight-name** <clr_weight_name>

Use balancing weight with this name. Using raw unbalanced data is not supported for saddles.

**Default**
weight

**-v, --verbose**

Enable verbose output

**-o, --out-prefix** <out_prefix>

**Required** Save compartment track as a BED-like file. Eigenvectors and corresponding eigenvalues are stored in out_prefix.contact_type.vecs.tsv and out_prefix.contact_type.lam.txt

**--bigwig**

Also save compartment track (E1) as a bigWig file with the name out_prefix.contact_type.bw

### Arguments

**COOL_PATH**

Required argument

### expected-cis

Calculate expected Hi-C signal for cis regions of chromosomal interaction map: average of interactions separated by the same genomic distance, i.e. are on the same diagonal on the cis-heatmap.

When balancing weights are not applied to the data, there is no masking of bad bins performed.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map.

```
cooltools expected-cis [OPTIONS] COOL_PATH
```

### Options

**-p, --nproc** <nproc>

Number of processes to split the work between.[default: 1, i.e. no process pool]

**-c, --chunksize** <chunksize>

Control the number of pixels handled by each worker process at a time.

**Default**
`10000000`

**-o, --output** <output>

Specify output file name to store the expected in a tsv format.

**--view, --regions** <view>

Path to a 3 or 4-column BED file with genomic regions to calculated cis-expected on. When region names are not provided (no 4th column), UCSC-style region names are generated. Cis-expected is calculated for all chromosomes, when this is not specified. Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--smooth**

If set, cis-expected is smoothed and result stored in an additional column e.g. balanced.avg.smoothed

**--aggregate-smoothed**

If set, cis-expected is averaged over all regions and then smoothed. Result is stored in an additional column, e.g. balanced.avg.smoothed.agg. Ignored without smoothing

**--smooth-sigma** <smooth_sigma>

Control smoothing with the standard deviation of the smoothing Gaussian kernel, ignored without smoothing.

**Default**
`0.1`

**--clr-weight-name** <clr_weight_name>

Use balancing weight with this name stored in cooler.Provide empty argument to calculate cis-expected on raw data

**Default**
`weight`

**--ignore-diags** <ignore_diags>

Number of diagonals to neglect for cis contact type

**Default**
`2`

### Arguments

**COOL_PATH**

Required argument

### expected-trans

Calculate expected Hi-C signal for trans regions of chromosomal interaction map: average of interactions in a rectangular block defined by a pair of regions, e.g. inter-chromosomal blocks.

When balancing weights are not applied to the data, there is no masking of bad bins performed.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map.

```
cooltools expected-trans [OPTIONS] COOL_PATH
```

#### Options

**-p, --nproc** <nproc>

Number of processes to split the work between.[default: 1, i.e. no process pool]

**-c, --chunksize** <chunksize>

Control the number of pixels handled by each worker process at a time.

**Default**
`10000000`

**-o, --output** <output>

Specify output file name to store the expected in a tsv format.

**--view, --regions** <view>

Path to a 3 or 4-column BED file with genomic regions. Trans-expected is calculated on all pairwise combinations of these regions. When region names are not provided (no 4th column), UCSC-style region names are generated. Trans-expected is calculated for all inter-chromosomal pairs, when view is not specified. Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--clr-weight-name** <clr_weight_name>

Use balancing weight with this name stored in cooler.Provide empty argument to calculate cis-expected on raw data

**Default**
`weight`

#### Arguments

**COOL_PATH**

Required argument

### genome

Utilities for binned genome assemblies.

```
cooltools genome [OPTIONS] COMMAND [ARGS]...
```

### binnify

```
cooltools genome binnify [OPTIONS] CHROMSIZES_PATH BINSIZE
```

#### Options

**--all-names**

Parse all chromosome names from file, not only default r"^chr[0-9]+$", r"^chr[XY]$", r"^chrM$".

#### Arguments

**CHROMSIZES_PATH**

Required argument

**BINSIZE**

Required argument

### digest

```
cooltools genome digest [OPTIONS] CHROMSIZES_PATH FASTA_PATH ENZYME_NAME
```

#### Arguments

**CHROMSIZES_PATH**

Required argument

**FASTA_PATH**

Required argument

**ENZYME_NAME**

Required argument

### fetch-chromsizes

```
cooltools genome fetch-chromsizes [OPTIONS] DB
```

#### Arguments

**DB**

Required argument

### gc

```
cooltools genome gc [OPTIONS] BINS_PATH FASTA_PATH
```

#### Options

**--mapped-only**

#### Arguments

**BINS_PATH**

Required argument

**FASTA_PATH**

Required argument

### genecov

BINS_PATH is the path to bintable.

DB is the name of the genome assembly. The gene locations will be automatically downloaded from teh UCSC goldenPath.

```
cooltools genome genecov [OPTIONS] BINS_PATH DB
```

#### Arguments

**BINS_PATH**

Required argument

**DB**

Required argument

### insulation

Calculate the diamond insulation scores and call insulating boundaries.

IN_PATH : The path to a .cool file with a balanced Hi-C map.

**WINDOW**

[The window size for the insulation score calculations.] Multiple space-separated values can be provided. By default, the window size must be provided in units of bp. When the flag –window-pixels is set, the window sizes must be provided in units of pixels instead.

```
cooltools insulation [OPTIONS] IN_PATH WINDOW
```

### Options

**-p, --nproc** <nproc>

Number of processes to split the work between.[default: 1, i.e. no process pool]

**-o, --output** <output>

Specify output file name to store the insulation in a tsv format.

**--view, --regions** <view>

Path to a BED file containing genomic regions for which insulation scores will be calculated. Region names can be provided in a 4th column and should match regions and their names in expected. Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--ignore-diags** <ignore_diags>

The number of diagonals to ignore. By default, equals the number of diagonals ignored during IC balancing.

**--clr-weight-name** <clr_weight_name>

Use balancing weight with this name. Provide empty argument to calculate insulation on raw data (no masking bad pixels).

**Default**
`weight`

**--min-frac-valid-pixels** <min_frac_valid_pixels>

The minimal fraction of valid pixels in a sliding diamond. Used to mask bins during boundary detection.

**Default**
`0.66`

**--min-dist-bad-bin** <min_dist_bad_bin>

The minimal allowed distance to a bad bin. Use to mask bins after insulation calculation and during boundary detection.

**Default**
`0`

**--threshold** <threshold>

Rule used to threshold the histogram of boundary strengths to exclude weakboundaries. 'Li' or 'Otsu' use corresponding methods from skimage.thresholding.Providing a float value will filter by a fixed threshold

**Default**
`0`

**--window-pixels**

If set then the window sizes are provided in units of pixels.

**--append-raw-scores**

Append columns with raw scores (sum_counts, sum_balanced, n_pixels) to the output table.

**--chunksize** <chunksize>

**Default**
`20000000`

**--verbose**

Report real-time progress.

**--bigwig**

Also save insulation tracks as a bigWig files for different window sizes with the names output.<window-size>.bw

### Arguments

**IN_PATH**

Required argument

**WINDOW**

Optional argument(s)

## pileup

Perform retrieval of the snippets from .cool file.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map. Use the '::' syntax to specify a group path in a multicooler file.

FEATURES_PATH : the path to a BED or BEDPE-like file that contains features for snipping windows. If BED, then the features are on-diagonal. If BEDPE, then the features can be off-diagonal (but not in trans or between different regions in the view).

```
cooltools pileup [OPTIONS] COOL_PATH FEATURES_PATH
```

### Options

**--view, --regions** `<view>`

Path to a BED file which defines which regions of the chromosomes to use. Required if EXPECTED_PATH is provided Note that '–regions' is the deprecated name of the option. Use '–view' instead.

**--expected** `<expected>`

Path to the expected table. If provided, outputs OOE pileup. if not provided, outputs regular pileup.

**--flank** `<flank>`

Size of flanks.

**Default**
`100000`

**--features-format** `<features_format>`

Input features format.

**Options**
auto | BED | BEDPE

**--clr-weight-name** `<clr_weight_name>`

Use balancing weight with this name.

**Default**
`weight`

**-o, --out** `<out>`

**Required** Save output pileup as NPZ/HDF5 file.

**--out-format** `<out_format>`

Type of output.

**Default**
`NPZ`

**Options**
NPZ | HDF5

**--store-snips**
Flag indicating whether snips should be stored.

**-p, --nproc** <nproc>
Number of processes to split the work between. [default: 1, i.e. no process pool]

**--ignore-diags** <ignore_diags>
The number of diagonals to ignore. By default, equals the number of diagonals ignored during IC balancing.

**--aggregate** <aggregate>
Function for calculating aggregate signal.

**Default**
none

**Options**
none | mean | median | std | min | max

**-v, --verbose**
Enable verbose output

#### Arguments

**COOL_PATH**
Required argument

**FEATURES_PATH**
Required argument

### random-sample

Pick a random sample of contacts from a Hi-C map.

IN_PATH : Input cooler path or URI.

OUT_PATH : Output cooler path or URI.

Specify the target sample size with either –count or –frac.

```
cooltools random-sample [OPTIONS] IN_PATH OUT_PATH
```

#### Options

**-c, --count** <count>
The target number of contacts in the sample. The resulting sample size will not match it precisely. Mutually exclusive with –frac and –cis-count

**--cis-count** <cis_count>
The target number of cis contacts in the sample. The resulting sample size will not match it precisely. Mutually exclusive with –count and –frac

`-f, --frac <frac>`

The target sample size as a fraction of contacts in the original dataset. Mutually exclusive with –count and –cis-count

`--exact`

If specified, use exact sampling that guarantees the size of the output sample. Otherwise, binomial sampling will be used and the sample size will be distributed around the target value.

`-p, --nproc <nproc>`

Number of processes to split the work between.[default: 1, i.e. no process pool]

`--chunksize <chunksize>`

The number of pixels loaded and processed per step of computation.

**Default**
`10000000`

### Arguments

`IN_PATH`

Required argument

`OUT_PATH`

Required argument

### rearrange

Rearrange data from a cooler according to a new genomic view

### Parameters

**IN_PATH**
[str] .cool file (or URI) with data to rearrange.

**OUT_PATH**
[str] .cool file (or URI) to save the rearrange data.

**view**
[str] Path to a BED-like file which defines which regions of the chromosomes to use and in what order. Has to be a valid viewframe (columns corresponding to region coordinates followed by the region name), with potential additional columns. Using –new-chrom-col and –orientation-col you can specify the new chromosome names and whether to invert each region (optional). If has no header with column names, assumes the *new-chrom-col* is the fifth column and *–orientation-col* is the sixth, if they exist.

**new_chrom_col**
[str] Column name in the view with new chromosome names. If not provided and there is no column named 'new_chrom' in the view file, uses original chromosome names.

**orientation_col**
[str] Columns name in the view with orientations of each region (+ or -). - means the region will be inverted. If not providedand there is no column named 'strand' in the view file, assumes all are forward oriented.

**assembly**
[str] The name of the assembly for the new cooler. If None, uses the same as in the original cooler.

**chunksize**
[int] The number of pixels loaded and processed per step of computation.

**mode**
[str] (w)rite or (a)ppend to the output file (default: w)

```
cooltools rearrange [OPTIONS] IN_PATH OUT_PATH
```

### Options

**--view** <view>

**Required** Path to a BED-like file which defines which regions of the chromosomes to use and in what order. Using –new-chrom-col and –orientation-col you can specify the new chromosome names and whether to invert each region (optional)

**--new-chrom-col** <new_chrom_col>

Column name in the view with new chromosome names. If not provided and there is no column named 'new_chrom' in the view file, uses original chromosome names

**--orientation-col** <orientation_col>

Columns name in the view with orientations of each region (+ or -). If not providedand there is no column named 'strand' in the view file, assumes all are forward oriented

**--assembly** <assembly>

The name of the assembly for the new cooler. If None, uses the same as in the original cooler.

**--chunksize** <chunksize>

The number of pixels loaded and processed per step of computation.

**Default**
`10000000`

**--mode** <mode>

(w)rite or (a)ppend to the output file (default: w)

**Options**
w | a

### Arguments

**IN_PATH**

Required argument

**OUT_PATH**

Required argument

### saddle

Calculate saddle statistics and generate saddle plots for an arbitrary signal track on the genomic bins of a contact matrix.

COOL_PATH : The paths to a .cool file with a balanced Hi-C map. Use the '::' syntax to specify a group path in a multicooler file.

TRACK_PATH : The path to bedGraph-like file with a binned compartment track (eigenvector), including a header. Use the '::' syntax to specify a column name.

EXPECTED_PATH : The paths to a tsv-like file with expected signal, including a header. Use the '::' syntax to specify a column name.

Analysis will be performed for chromosomes referred to in TRACK_PATH, and therefore these chromosomes must be a subset of chromosomes referred to in COOL_PATH and EXPECTED_PATH.

COOL_PATH, TRACK_PATH and EXPECTED_PATH must be binned at the same resolution (expect for EXPECTED_PATH in case of trans contact type).

EXPECTED_PATH must contain at least the following columns for cis contacts: 'chrom', 'diag', 'n_valid', value_name and the following columns for trans contacts: 'chrom1', 'chrom2', 'n_valid', value_name value_name is controlled using options. Header must be present in a file.

```
cooltools saddle [OPTIONS] COOL_PATH TRACK_PATH EXPECTED_PATH
```

#### Options

**-t, --contact-type** <contact_type>

Type of the contacts to aggregate

**Default**
`cis`

**Options**
cis | trans

**--min-dist** <min_dist>

Minimal distance between bins to consider, bp. If negative, removesthe first two diagonals of the data. Ignored with –contact-type trans.

**Default**
`-1`

**--max-dist** <max_dist>

Maximal distance between bins to consider, bp. Ignored, if negative. Ignored with –contact-type trans.

**Default**
`-1`

**-n, --n-bins** <n_bins>

Number of bins for digitizing track values.

**Default**
`50`

**--vrange** <vrange>

Low and high values used for binning genome-wide track values, e.g. if *range`=(-0.05, 0.05), `n-bins* equidistant bins would be generated. Use to prevent extreme track values from exploding the bin range and to ensure consistent bins across several runs of *compute_saddle* command using different track files.

`--qrange <qrange>`

Low and high values used for quantile bins of genome-wide track values,e.g. if `qrange`=(0.02, 0.98) the lower bin would start at the 2nd percentile and the upper bin would end at the 98th percentile of the genome-wide signal. Use to prevent the extreme track values from exploding the bin range.

**Default**
`None, None`

`--clr-weight-name <clr_weight_name>`

Use balancing weight with this name.

**Default**
`weight`

`--strength, --no-strength`

Compute and save compartment 'saddle strength' profile

`--view, --regions <view>`

Path to a BED file containing genomic regions for which saddleplot will be calculated. Region names can be provided in a 4th column and should match regions and their names in expected. Note that '–regions' is the deprecated name of the option. Use '–view' instead.

`-o, --out-prefix <out_prefix>`

**Required** Dump 'saddledata', 'binedges' and 'hist' arrays in a numpy-specific .npz container. Use numpy.load to load these arrays into a dict-like object. The digitized signal values are saved to a bedGraph-style TSV.

`--fig <fig>`

Generate a figure and save to a file of the specified format. If not specified - no image is generated. Repeat for multiple output formats.

**Options**
png | jpg | svg | pdf | ps | eps

`--scale <scale>`

Value scale for the heatmap

**Default**
`log`

**Options**
linear | log

`--cmap <cmap>`

Name of matplotlib colormap

**Default**
`coolwarm`

`--vmin <vmin>`

Low value of the saddleplot colorbar. Note: value in original units irrespective of used scale, and therefore should be positive for both vmin and vmax.

`--vmax <vmax>`

High value of the saddleplot colorbar

`--hist-color <hist_color>`

Face color of histogram bar chart

`-v, --verbose`

Enable verbose output

### Arguments

**COOL_PATH**

Required argument

**TRACK_PATH**

Required argument

**EXPECTED_PATH**

Required argument

### virtual4c

Generate virtual 4C profile from a contact map by extracting all interactions of a given viewpoint with the rest of the genome.

COOL_PATH : the paths to a .cool file with a Hi-C map. Use the '::' syntax to specify a group path in a multicooler file.

VIEWPOINT : the viewpoint to use for the virtual 4C profile. Provide as a UCSC-string (e.g. chr1:1-1000)

Note: this is a new (experimental) tool, the interface or output might change in a future version.

```
cooltools virtual4c [OPTIONS] COOL_PATH VIEWPOINT
```

### Options

**--clr-weight-name** `<clr_weight_name>`

Use balancing weight with this name. Provide empty argument to calculate insulation on raw data (no masking bad pixels).

**Default**
`weight`

**-o, --out-prefix** `<out_prefix>`

**Required** Save virtual 4C track as a BED-like file. Contact frequency is stored in out_prefix.v4C.tsv

**--bigwig**

Also save virtual 4C track as a bigWig file with the name out_prefix.v4C.bw

**-p, --nproc** `<nproc>`

Number of processes to split the work between. [default: 1, i.e. no process pool]

### Arguments

**COOL_PATH**

Required argument

**VIEWPOINT**

Required argument

### 1.3.9 API Reference

#### subpackages

#### cooltools.lib package

#### common

cooltools.lib.common.**align_track_with_cooler**(*track*, *clr*, *view_df=None*, *clr_weight_name='weight'*, *mask_clr_bad_bins=True*, *drop_track_na=True*)

Sync a track dataframe with a cooler bintable.

Checks that bin sizes match between a track and a cooler, merges the cooler bintable with the track, and propagates masked regions from a cooler bintable to a track.

**Parameters**

- **track** (*pd.DataFrame*) – bedGraph-like track DataFrame to check
- **clr** (*cooler*) – cooler object to check against
- **view_df** (*bioframe.viewframe or None*) – Optional viewframe of regions to check for their number of bins with assigned track values. If None, constructs a view_df from cooler chromsizes.
- **clr_weight_name** (*str*) – Name of the column in the bin table with weight
- **mask_clr_bad_bins** (*bool*) – Whether to propagate null bins from cooler bintable column clr_weight_name to the 'value' column of the output clr_track. Default True.
- **drop_track_na** (*bool*) – Whether to ignore missing values in the track (as if they are absent). Important for raising errors for unassigned regions and warnings for partial assignment. Default True, so NaN values are treated as not assigned. False means that NaN values are treated as assigned.

**Returns**

*clr_track* – track dataframe that has been aligned with the cooler bintable and has columns ['chrom','start','end','value']

cooltools.lib.common.**assign_regions**(*features*, *supports*)

DEPRECATED. Will be removed in the future versions and replaced with bioframe.overlap() For each feature in features dataframe assign the genomic region (support) that overlaps with it. In case if feature overlaps multiple supports, the region with largest overlap will be reported.

cooltools.lib.common.**assign_regions_to_bins**(*bin_ids*, *regions_span*)

cooltools.lib.common.**assign_supports**(*features*, *supports*, *labels=False*, *suffix=''*)

Assign support regions to a table of genomic intervals. Obsolete, replaced by assign_regions now.

**Parameters**

- **features** (*DataFrame*) – Dataframe with columns *chrom*, *start*, *end* or *chrom1*, *start1*, *end1*, *chrom2*, *start2*, *end2*
- **supports** (*array-like*) – Support areas

cooltools.lib.common.**assign_view_auto**(*features*, *view_df*, *cols_unpaired=['chrom', 'start', 'end']*, *cols_paired=['chrom1', 'start1', 'end1', 'chrom2', 'start2', 'end2']*, *cols_view=['chrom', 'start', 'end']*, *features_view_col_unpaired='region'*, *features_view_cols_paired=['region1', 'region2']*, *view_name_col='name'*, *drop_unassigned=False*, *combined_assignments_column='region'*, *force=True*)

Assign region names from the view to each feature

Determines whether the *features* are unpaired (1D, bed-like) or paired (2D, bedpe-like) based on presence of column names (*cols_unpaired* vs *cols_paired*) Assigns a regular 1D view, independently to each side in case of paired features. Will add one or two columns with region names (*features_view_col_unpaired* or *features_view_cols_paired*) respectively, in case of unpaired and paired features.

**Parameters**

- **features** (*pd.DataFrame*) – bedpe-style dataframe
- **view_df** (*pandas.DataFrame*) – ViewFrame specifying region start and ends for assignment. Attempts to convert dictionary and pd.Series formats to viewFrames.
- **cols_unpaired** (*list of str*) – The names of columns containing the chromosome, start and end of the genomic intervals for unpaired features. The default values are *"chrom", "start", "end"*.
- **cols_paired** (*list of str*) – The names of columns containing the chromosome, start and end of the genomic intervals for paired features. The default values are *"chrom1", "start1", "end1", "chrom2", "start2", "end2"*.
- **cols_view** (*list of str*) – The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are *"chrom", "start", "end"*.
- **features_view_col_unpaired** (*str*) – Name of the column where to save the assigned region name for unpaired features
- **features_view_cols_paired** (*list of str*) – Names of the columns where to save the assigned region names for paired features
- **view_name_col** (*str*) – Column of `view_df` with region names. Default "name".
- **drop_unassigned** (*bool*) – If True, drop intervals in *features* that do not overlap a region in the view. Default False.
- **combined_assignments_column** (*str or None*) – If set to a string value, will combine assignments from two sides of paired features when they match into column with this name: region name when regions assigned to both sides match, np.nan if not. Default "region"
- **force** (*bool, True or False*) – if features already have features_view_col (paired or not, depending on the feature types), should we re-wrtie region columns or keep them.

cooltools.lib.common.**assign_view_paired**(*features*, *view_df*, *cols_paired=['chrom1', 'start1', 'end1', 'chrom2', 'start2', 'end2']*, *cols_view=['chrom', 'start', 'end']*, *features_view_cols=['region1', 'region2']*, *view_name_col='name'*, *drop_unassigned=False*)

Assign region names from the view to each feature

Assigns a regular 1D view independently to each side of a bedpe-style dataframe. Will add two columns with region names (*features_view_cols*)

**Parameters**

- **features** (*pd.DataFrame*) – bedpe-style dataframe

- **view_df** (*pandas.DataFrame*) – ViewFrame specifying region start and ends for assignment. Attempts to convert dictionary and pd.Series formats to viewFrames.
- **cols_paired** (*list of str*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are *"chrom1", "start1", "end1", "chrom2", "start2", "end2"*.
- **cols_view** (*list of str*) – The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are *"chrom", "start", "end"*.
- **features_view_cols** (*list of str*) – Names of the columns where to save the assigned region names
- **view_name_col** (*str*) – Column of `view_df` with region names. Default "name".
- **drop_unassigned** (*bool*) – If True, drop intervals in df that do not overlap a region in the view. Default False.

`cooltools.lib.common.`**make_cooler_view**(*clr*, *ucsc_names=False*)

Generate a full chromosome viewframe using cooler's chromsizes

**Parameters**

- **clr** (*cooler*) – cooler-object to extract chromsizes
- **ucsc_names** (*bool*) – Use full UCSC formatted names instead of short chromosome names.

**Returns**

**cooler_view** (*viewframe*) – full chromosome viewframe

`cooltools.lib.common.`**mask_cooler_bad_bins**(*track*, *bintable*)

Mask (set to NaN) values in track where bin is masked in bintable.

Currently used in *cli.get_saddle()*. TODO: determine if this should be used elsewhere.

**Parameters**

- **track** (*tuple of (DataFrame, str)*) – bedGraph-like dataframe along with the name of the value column.
- **bintable** (*tuple of (DataFrame, str)*) – bedGraph-like dataframe along with the name of the weight column.

**Returns**

**track** (*DataFrame*) – New bedGraph-like dataframe with bad bins masked in the value column

`cooltools.lib.common.`**pool_decorator**(*func*)

A decorator function that enables multiprocessing for a given function. The function must have a `map_functor` keyword argument. It works by hijacking map_functor argument and substituting it with the parallel one: multiprocess.Pool(nproc).imap, when nproc > 1

**Parameters**

**func** (*callable*) – The function to be decorated.

**Returns**

*A wrapper function that enables multiprocessing for the given function.*

`cooltools.lib.common.`**view_from_track**(*track_df*)

### numutils

`cooltools.lib.numutils.`**`COMED`**(*xs*, *ys*, *has_nans=False*)

Calculate the comedian - the robust median-based counterpart of Pearson's r.

```
comedian = median((xs-median(xs))*(ys-median(ys))) / MAD(xs) / MAD(ys)
```

**Parameters**

**has_nans** (*bool*) – if True, mask (x,y) pairs with at least one NaN

#### Notes

Citations: "On MAD and comedians" by Michael Falk (1997), "Robust Estimation of the Correlation Coefficient: An Attempt of Survey" by Georgy Shevlyakov and Pavel Smirnov (2011)

`cooltools.lib.numutils.`**`MAD`**(*arr*, *axis=None*, *has_nans=False*)

Calculate the Median Absolute Deviation from the median.

**Parameters**

- **arr** (*np.ndarray*) – Input data.
- **axis** (*int*) – The axis along which to calculate MAD.
- **has_nans** (*bool*) – If True, use the slower NaN-aware method to calculate medians.

`cooltools.lib.numutils.`**`adaptive_coarsegrain`**(*ar*, *countar*, *cutoff=5*, *max_levels=8*, *min_shape=8*)

Adaptively coarsegrain a Hi-C matrix based on local neighborhood pooling of counts.

**Parameters**

- **ar** (*array_like, shape (n, n)*) – A square Hi-C matrix to coarsegrain. Usually this would be a balanced matrix.
- **countar** (*array_like, shape (n, n)*) – The raw count matrix for the same area. Has to be the same shape as the Hi-C matrix.
- **cutoff** (*float, optional*) – A minimum number of raw counts per pixel required to stop 2x2 pooling. Larger cutoff values would lead to a more coarse-grained, but smoother map. 3 is a good default value for display purposes, could be lowered to 1 or 2 to make the map less pixelated. Setting it to 1 will only ensure there are no zeros in the map.
- **max_levels** (*int, optional*) – How many levels of coarsening to perform. It is safe to keep this number large as very coarsened map will have large counts and no substitutions would be made at coarser levels.
- **min_shape** (*int, optional*) – Stop coarsegraining when coarsegrained array shape is less than that.

**Returns**

*Smoothed array, shape (n, n)*

### Notes

The algorithm works as follows:

First, it pads an array with NaNs to the nearest power of two. Second, it coarsens the array in powers of two until the size is less than minshape.

Third, it starts with the most coarsened array, and goes one level up. It looks at all 4 pixels that make each pixel in the second-to-last coarsened array. If the raw counts for any valid (non-NaN) pixel are less than `cutoff`, it replaces the values of the valid (4 or less) pixels with the NaN-aware average. It is then applied to the next (less coarsened) level until it reaches the original resolution.

In the resulting matrix, there are guaranteed to be no zeros, unless very large zero-only areas were provided such that zeros were produced `max_levels` times when coarsening.

### Examples

```
>>> c = cooler.Cooler("/path/to/some/cooler/at/about/2000bp/resolution")
```

```
>>> # sample region of about 6000x6000
>>> mat = c.matrix(balance=True).fetch("chr1:10000000-22000000")
>>> mat_raw = c.matrix(balance=False).fetch("chr1:10000000-22000000")
>>> mat_cg = adaptive_coarsegrain(mat, mat_raw)
```

```
>>> plt.figure(figsize=(16,7))
>>> ax = plt.subplot(121)
>>> plt.imshow(np.log(mat), vmax=-3)
>>> plt.colorbar()
>>> plt.subplot(122, sharex=ax, sharey=ax)
>>> plt.imshow(np.log(mat_cg), vmax=-3)
>>> plt.colorbar()
```

`cooltools.lib.numutils.`**`coarsen`**(*reduction*, *x*, *axes*, *trim_excess=False*)

Coarsen an array by applying reduction to fixed size neighborhoods. Adapted from *dask.array.coarsen* to work on regular numpy arrays.

**Parameters**

- **reduction** (*function*) – Function like np.sum, np.mean, etc…
- **x** (*np.ndarray*) – Array to be coarsened
- **axes** (*dict*) – Mapping of axis to coarsening factor
- **trim_excess** (*bool, optional*) – Remove excess elements. Default is False.

### Examples

Provide dictionary of scale per dimension

```
>>> x = np.array([1, 2, 3, 4, 5, 6])
>>> coarsen(np.sum, x, {0: 2})
array([ 3,  7, 11])
```

```
>>> coarsen(np.max, x, {0: 3})
array([3, 6])
```

```
>>> x = np.arange(24).reshape((4, 6))
>>> x
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

```
>>> coarsen(np.min, x, {0: 2, 1: 3})
array([[ 0,  3],
       [12, 15]])
```

**See also:**

`dask.array.coarsen`

`cooltools.lib.numutils.`**`dist_to_mask`**(*mask*, *side='min'*)

Calculate the distance to the nearest True element of an array.

**Parameters**

- **mask** (*iterable of bool*) – A boolean array.
- **side** (*str*) – The side . Accepted values are: 'left' : calculate the distance to the nearest True value on the left 'right' : calculate the distance to the nearest True value on the right 'min' : calculate the distance to the closest True value 'max' : calculate the distance to the furthest of the two neighbouring True values

**Returns**

**dist** (*array of int*)

### Notes

The solution is borrowed from https://stackoverflow.com/questions/18196811/cumsum-reset-at-nan

`cooltools.lib.numutils.`**`fill_diag`**(*arr*, *x*, *i=0*, *copy=True*)

Identical to set_diag, but returns a copy by default

`cooltools.lib.numutils.`**`fill_inf`**(*arr*, *pos_value=0*, *neg_value=0*, *copy=True*)

Replaces positive and negative infinity entries in an array with the provided values.

**Parameters**

- **arr** (*np.array*) –
- **pos_value** (*float*) – Fill value for np.inf
- **neg_value** (*float*) – Fill value for -np.inf

- **copy** (*bool, optional*) – If True, creates a copy of x, otherwise replaces values in-place. By default, True.

`cooltools.lib.numutils.`**`fill_na`**(*arr*, *value=0*, *copy=True*)

Replaces np.nan entries in an array with the provided value.

**Parameters**

- **arr** (*np.array*) –
- **value** (*float*) –
- **copy** (*bool, optional*) – If True, creates a copy of x, otherwise replaces values in-place. By default, True.

`cooltools.lib.numutils.`**`fill_nainf`**(*arr*, *value=0*, *copy=True*)

Replaces np.nan and np.inf entries in an array with the provided value.

**Parameters**

- **arr** (*np.array*) –
- **value** (*float*) –
- **copy** (*bool, optional*) – If True, creates a copy of x, otherwise replaces values in-place. By default, True.

### Notes

Differs from np.nan_to_num in that it replaces np.inf with the same number as np.nan.

`cooltools.lib.numutils.`**`get_diag`**(*arr*, *i=0*)

Get the i-th diagonal of a matrix. This solution was borrowed from http://stackoverflow.com/questions/9958577/changing-the-values-of-the-diagonal-of-a-matrix-in-numpy

`cooltools.lib.numutils.`**`get_eig`**(*mat*, *n=3*, *mask_zero_rows=False*, *subtract_mean=False*, *divide_by_mean=False*)

Perform an eigenvector decomposition.

**Parameters**

- **mat** (*np.ndarray*) – A square matrix, must not contain nans, infs or zero rows.
- **n** (*int*) – The number of eigenvectors to return. Output is backfilled with NaNs when n exceeds the size of the input matrix.
- **mask_zero_rows** (*bool*) – If True, mask empty rows/columns before eigenvector decomposition. Works only with symmetric matrices.
- **subtract_mean** (*bool*) – If True, subtract the mean from the matrix.
- **divide_by_mean** (*bool*) – If True, divide the matrix by its mean.

**Returns**

- **eigvecs** (*np.ndarray*) – An array of eigenvectors (in rows), sorted by a decreasing absolute eigenvalue.
- **eigvals** (*np.ndarray*) – An array of sorted eigenvalues.

`cooltools.lib.numutils.`**`get_finite`**(*arr*)

Select only finite elements of an array.

cooltools.lib.numutils.**get_kernel**(*w*, *p*, *ktype*)

Return typical kernels given size parameteres w, p,and kernel type.

**Parameters**

- **w** (*int*) – Outer kernel size (actually half of it).
- **p** (*int*) – Inner kernel size (half of it).
- **ktype** (*str*) – Name of the kernel type, could be one of the following: 'donut', 'vertical', 'horizontal', 'lowleft', 'upright'.

**Returns**

**kernel** (*ndarray*) – A square matrix of int type filled with 1 and 0, according to the kernel type.

cooltools.lib.numutils.**infer_mask2D**(*mat*)

cooltools.lib.numutils.**interp_nan**(*a_init*, *pad_zeros=True*, *method='linear'*, *verbose=False*)

Linearly interpolate to fill NaN rows and columns in a matrix. Also interpolates NaNs in 1D arrays.

**Parameters**

- **a_init** (*np.array*) –
- **pad_zeros** (*bool, optional*) – If True, pads the matrix with zeros to fill NaNs at the edges. By default, True.
- **method** (*str, optional*) – For 2D: "linear", "nearest", or "splinef2d" For 1D: "linear", "nearest", "zero", "slinear", "quadratic", "cubic"

**Returns**

*array with NaNs linearly interpolated*

## Notes

1D case adapted from: https://stackoverflow.com/a/39592604 2D case assumes that entire rows or columns are masked & edges to be NaN-free, but is much faster than griddata implementation.

cooltools.lib.numutils.**interpolate_bad_singletons**(*mat*, *mask=None*, *fillDiagonal=True*, *returnMask=False*, *secondPass=True*, *verbose=False*)

Interpolate singleton missing bins for visualization

## Examples

```
>>> ax = plt.subplot(121)
>>> maxval =  np.log(np.nanmean(np.diag(mat,3))*2 )
>>> plt.matshow(np.log(mat)), vmax=maxval, fignum=False)
>>> plt.set_cmap('fall');
>>> plt.subplot(122, sharex=ax, sharey=ax)
>>> plt.matshow(
...     np.log(interpolate_bad_singletons(remove_good_singletons(mat))),
...     vmax=maxval,
...     fignum=False
... )
>>> plt.set_cmap('fall');
>>> plt.show()
```

`cooltools.lib.numutils.`**`is_symmetric`**`(`*mat*`)`

Check if a matrix is symmetric.

`cooltools.lib.numutils.`**`normalize_score`**`(`*arr*, *norm='z'*, *axis=None*, *has_nans=True*`)`

Normalize an array by subtracting the first moment and dividing the residual by the second.

**Parameters**

- **`arr`** (*`np.ndarray`*) – Input data.
- **`norm`** (*`str`*) – The type of normalization. 'z' - report z-scores, norm_arr = (arr - mean(arr)) / std(arr)

  'mad' - report deviations from the median in units of MAD (Median Absolute Deviation from the median), norm_arr = (arr - median(arr)) / MAD(arr)

  'madz' - report robust z-scores, i.e. estimate the mean as the median and the standard error as MAD / 0.67499, norm_arr = (arr - median(arr)) / MAD(arr) * 0.67499
- **`axis`** (*`int`*) – The axis along which to calculate the normalization parameters.
- **`has_nans`** (*`bool`*) – If True, use slower NaN-aware methods to calculate the normalization parameters.

`cooltools.lib.numutils.`**`persistent_log_bins`**`(`*end=10*, *bins_per_order_magnitude=10*`)`

Creates most nicely looking log-spaced integer bins starting at 1, with the defined number of bins per order of magnitude.

**Parameters**

- **`end`** (*`number (int recommended) log10 of the last value. It is safe to put a`*) –
- **`later.`** (*`large value here and select your range of bins`*) –
- **`bins_per_order_magnitude`** (*`int >0 how many bins per order of magnitude`*) –

## Notes

This is not a replacement for logbins, and it has a different purpose.

### Difference between this and logbins

Logbins creates bins from lo to hi, spaced logarithmically with an appriximate ratio. Logbins makes sure that the last bin is large (i.e. hi/ratio … hi), and will not allow the last bin to be much less than ratio. It would slightly adjust the ratio to achieve that. As a result, by construciton, logbins bins are different for different lo or hi.

This function is designed to create exactly the same bins that only depend on one parameter, bins_per_order_magnitude. The goal is to make things calculated for different datasets/organisms/etc. comparable. For example, if these bins are used, it would allow us to divide P(s) for two different organisms by each other because it was calculated for the same bins.

The price you pay for such versatility is that the last bin can be much less than others in real application. For example, if you have 10 bins per order of magnitude (ratio of 1.25), but your data ends at 10500, then the only points in the last bin would be 10000..10500, 1/5 of what could be. This may make the last point noisy.

The main part is done using np.logspace and rounding to the nearest integer, followed by unique. The gaps are then re-sorted to ensure that gaps are strictly increasing. The re-sorting of gaps was essential, and produced better results than manual adjustment.

### Alternatives that produce irregular bins

Using np.unique(np.logspace(a,b,N,dtype=int)) can be sub-optimal For example, np.unique(np.logspace(0,1,11,dtype=int)) = [ 1, 2, 3, 5, 6, 7, 10] Note the gaps jump from 1 to 2 back to 1

Similarly using np.unique(np.rint(np.logspace..)) can be suboptimal np.unique(np.array(np.rint(np.logspace(0,1,9)),dtype=int)) = [ 1, 2, 3, 4, 6, 7, 10]

for bins_per_order_of_magnitude=16, 10 is not in bins. Other than that, 10, 100, 1000, etc. are always included.

`cooltools.lib.numutils.`**`remove_good_singletons`**(*mat*, *mask=None*, *returnMask=False*)

`cooltools.lib.numutils.`**`robust_gauss_filter`**(*ar*, *sigma=2*, *functon=<Mock name='mock.gaussian_filter1d' id='139638368253312'>*, *kwargs=None*)

Implements an edge-handling mode for gaussian filter that basically ignores the edge, and also handles NaNs.

**Parameters**

- **ar** (*array-like*) – Input array
- **sigma** (*float*) – sigma to be passed to the filter
- **function** (*callable*) – Filter to use. Default is gauusian_filter1d
- **kwargs** (*dict*) – Additional args to pass to the filter. Default:None

### Notes

Available edge-handling modes in ndimage.filters attempt to somehow “extrapolate” the edge value and then apply the filter (see https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html).

That’s likely because convolve uses fast fourier transform, which requires

the kernel to be constant. Here we design a better edge-handling for the gaussian smoothing.

In a gaussian-filtered array, a pixel away from the edge is a mean of nearby pixels with gaussian weights. With this mode, pixels near start/end are also a mean of nearby pixels with gaussian weights. That’s it. If we encounter NANs, we also simply ignore them, following the same definition: mean of nearby valid pixels. Yes, it raises the weights for the first/last pixels, because now only a part of the whole gaussian is being used (up to 1/2 for the first/last pixel and large sigma). But it preserves the “mean of nearby pixels” definition. It is different from padding with zeros (it would drag the first pixel down to be more like zero). It is also different from “nearest” - that gives too much weight to the first/last pixel.

To achieve this smoothing, we preform regular gaussian smoothing using mode=”constant” (pad with zeros). Then we take an array of valid pixels and smooth it the same way. This calculates how many “average valid pixels” contributed to each point of a smoothed array. Dividing one by the other achieves the desired result.

`cooltools.lib.numutils.`**`set_diag`**(*arr*, *x*, *i=0*, *copy=False*)

Rewrite the i-th diagonal of a matrix with a value or an array of values. Supports 2D arrays, square or rectangular. In-place by default.

**Parameters**

- **arr** (*2-D array*) – Array whose diagonal is to be filled.
- **x** (*scalar or 1-D vector of correct length*) – Values to be written on the diagonal.

- **i** (*int, optional*) – Which diagonal to write to. Default is 0. Main diagonal is 0; upper diagonals are positive and lower diagonals are negative.
- **copy** (*bool, optional*) – Return a copy. Diagonal is written in-place if false. Default is False.

**Returns**

*Array with diagonal filled.*

### Notes

Similar to numpy.fill_diagonal, but allows for kth diagonals as well. This solution was borrowed from http://stackoverflow.com/questions/9958577/changing-the-values-of-the-diagonal-of-a-matrix-in-numpy

`cooltools.lib.numutils.`**`slice_sorted`**(*arr*, *lo*, *hi*)

Get the subset of a sorted array with values >=lo and <hi. A faster version of arr[(arr>=lo) & (arr<hi)]

`cooltools.lib.numutils.`**`smooth`**(*y*, *box_pts*)

`cooltools.lib.numutils.`**`stochastic_sd`**(*arr*, *n=10000*, *seed=0*)

Estimate the standard deviation of an array by considering only the subset of its elements.

**Parameters**

- **n** (*int*) – The number of elements to consider. If the array contains fewer elements, use all.
- **seed** (*int*) – The seed for the random number generator.

`cooltools.lib.numutils.`**`weighted_groupby_mean`**(*df*, *group_by*, *weigh_by*, *mode='mean'*)

Weighted mean, std, and std in log space for a dataframe.groupby

**Parameters**

- **df** (*dataframe*) – Dataframe to groupby
- **group_by** (*str or list*) – Columns to group by
- **weight_by** (*str*) – Column to use as weights
- **mode** (*"mean", "std" or "logstd"*) – Do the weighted mean, the weighted standard deviaton, or the weighted std in log-space from the mean-log value (useful for P(s) etc.)

`cooltools.lib.numutils.`**`zoom_array`**(*in_array*, *final_shape*, *same_sum=False*, *zoom_function=functools.partial(<Mock name='mock.zoom' id='139638368254656'>*, *order=1)*, ***zoom_kwargs*)

Rescale an array or image.

Normally, one can use scipy.ndimage.zoom to do array/image rescaling. However, scipy.ndimage.zoom does not coarsegrain images well. It basically takes nearest neighbor, rather than averaging all the pixels, when coarsegraining arrays. This increases noise. Photoshop doesn't do that, and performs some smart interpolation-averaging instead.

If you were to coarsegrain an array by an integer factor, e.g. 100x100 -> 25x25, you just need to do block-averaging, that's easy, and it reduces noise. But what if you want to coarsegrain 100x100 -> 30x30?

Then my friend you are in trouble. But this function will help you. This function will blow up your 100x100 array to a 120x120 array using scipy.ndimage zoom Then it will coarsegrain a 120x120 array by block-averaging in 4x4 chunks.

It will do it independently for each dimension, so if you want a 100x100 array to become a 60x120 array, it will blow up the first and the second dimension to 120, and then block-average only the first dimension.

(Copied from mirnylib.numutils)

**Parameters**

- `in_array` (*ndarray*) – n-dimensional numpy array (1D also works)
- `final_shape` (*shape tuple*) – resulting shape of an array
- `same_sum` (*bool, optional*) – Preserve a sum of the array, rather than values. By default, values are preserved
- `zoom_function` (*callable*) – By default, scipy.ndimage.zoom with order=1. You can plug your own.
- `**zoom_kwargs` – Options to pass to zoomFunction.

**Returns**

**rescaled** (*ndarray*) – Rescaled version of in_array

## peaks

cooltools.lib.peaks.**find_peak_prominence**(*arr*, *max_dist=None*)

Find the local maxima of an array and their prominence. The prominence of a peak is defined as the maximal difference between the height of the peak and the lowest point in the range until a higher peak.

**Parameters**

- `arr` (*array_like*) –
- `max_dist` (*int*) – If specified, the distance to the adjacent higher peaks is limited by *max_dist*.

**Returns**

- **loc_max_poss** (*numpy.array*) – The positions of local maxima of a given array.
- **proms** (*numpy.array*) – The prominence of the detected maxima.

cooltools.lib.peaks.**find_peak_prominence_iterative**(*arr*, *min_prom=None*, *max_prom=None*, *steps_prom=1000*, *log_space_proms=True*, *min_n_peak_pairs=5*)

Finds the minima/maxima of an array using the peakdet algorithm at different values of the threshold prominence. For each location, returns the maximal threshold prominence at which it is called as a minimum/maximum.

Note that this function is inferior in every aspect to find_peak_prominence. We keep it for testing purposes and will remove in the future.

**Parameters**

- `arr` (*array_like*) –
- `min_prom` (*float*) – The minimal and the maximal values of prominence to probe. If None, these values are inferred as the minimal and the maximal non-zero difference between any two elements of *v*.
- `max_prom` (*float*) – The minimal and the maximal values of prominence to probe. If None, these values are inferred as the minimal and the maximal non-zero difference between any two elements of *v*.
- `steps_prom` (*int*) – The number of threshold prominence values to probe in the range between *min_prom* and *max_prom*.

- **log_space_proms** (*bool*) – If True, probe logarithmically spaced values of the threshold prominence in the range between *min_prom* and *max_prom*.
- **min_n_peak_pairs** (*int*) – If the number of detected minima/maxima at a certain threshold prominence is < *min_n_peak_pairs*, the detected peaks are ignored.

**Returns**

**minproms, maxproms** (*numpy.array*) – The prominence of detected minima and maxima.

`cooltools.lib.peaks.`**`peakdet`**`(`*arr*, *min_prominence*`)`

Detect local peaks in an array. Finds a sequence of minima and maxima such that the two consecutive extrema have a value difference (i.e. a prominence) >= *min_prominence*. This is analogous to the definition of prominence in topography: https://en.wikipedia.org/wiki/Topographic_prominence

The original peakdet algorithm was designed by Eli Billauer and described in http://billauer.co.il/peakdet.html (v. 3.4.05, Explicitly not copyrighted). This function is released to the public domain; Any use is allowed. The Python implementation was published by endolith on Github: https://gist.github.com/endolith/250860 .

Here, we use the endolith's implementation with minimal to none modifications to the algorithm, but with significant changes in the interface and the documentation

**Parameters**

- **arr** (*array_like*) –
- **min_prominence** (*float*) – The minimal prominence of detected extrema.

**Returns**

**maxidxs, minidx** (*numpy.array*) – The indices of the maxima and minima in *arr*.

### plotting

Migrated from `mirnylib.plotting`.

`cooltools.lib.plotting.`**`get_cmap`**`(`*name*`)`

`cooltools.lib.plotting.`**`gridspec_inches`**`(`*wcols*, *hrows*, *fig_kwargs={}*`)`

`cooltools.lib.plotting.`**`list_to_colormap`**`(`*color_list*, *name=None*`)`

### schemas

### cooltools.api.coverage module

`cooltools.api.coverage.`**`coverage`**`(`*clr*, *ignore_diags=None*, *chunksize=10000000*, *use_lock=False*, *clr_weight_name=None*, *store=False*, *store_prefix='cov'*, *nproc=1*, *map_functor=<class 'map'>*`)`

Calculate the sums of cis and genome-wide contacts (aka coverage aka marginals) for a sparse Hi-C contact map in Cooler HDF5 format. Note that for raw coverage (i.e. clr_weight_name=None) the sum(tot_cov) from this function is two times the number of reads contributing to the cooler, as each side contributes to the coverage.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **ignore_diags** (*int, optional*) – Drop elements occurring on the first `ignore_diags` diagonals of the matrix (including the main diagonal). If None, equals the number of diagonals ignored during IC balancing.

- **chunksize** (*int, optional*) – Split the contact matrix pixel records into equally sized chunks to save memory and/or parallelize. Default is 10^7
- **clr_weight_name** (*str*) – Name of the weight column. Specify to calculate coverage of balanced cooler.
- **store** (*bool, optional*) – If True, store the results in the input cooler file when finished. If clr_weight_name=None, also stores total cis counts in the cooler info. Default is False.
- **store_prefix** (*str, optional*) – Name prefix of the columns of the bin table to save cis and total coverages. Will add suffixes _cis and _tot, as well as _raw in the default case or _clr_weight_name if specified.
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

- **cis_cov** (1D array, whose shape is the number of bins in `h5`. Vector of bin sums in cis.)
- **tot_cov** (1D array, whose shape is the number of bins in `h5`. Vector of bin sums.)

### cooltools.api.directionality module

`cooltools.api.directionality.`**`directionality`**(*clr*, *window_bp=100000*, *balance='weight'*, *min_dist_bad_bin=2*, *ignore_diags=None*, *chromosomes=None*)

Calculate the diamond insulation scores and call insulating boundaries.

**Parameters**

- **clr** (*cooler.Cooler*) – A cooler with balanced Hi-C data.
- **window_bp** (*int*) – The size of the sliding diamond window used to calculate the insulation score.
- **min_dist_bad_bin** (*int*) – The minimal allowed distance to a bad bin. Do not calculate insulation scores for bins having a bad bin closer than this distance.
- **ignore_diags** (*int*) – The number of diagonals to ignore. If None, equals the number of diagonals ignored during IC balancing.

**Returns**

**ins_table** (*pandas.DataFrame*) – A table containing the insulation scores of the genomic bins and the insulating boundary strengths.

### cooltools.api.dotfinder module

Collection of functions related to dot-calling

The main user-facing API function is:

```
dots(
    clr,
    expected,
    expected_value_col="balanced.avg",
    clr_weight_name="weight",
    view_df=None,
    kernels=None,
    max_loci_separation=10_000_000,
    max_nans_tolerated=1,
    n_lambda_bins=40,
    lambda_bin_fdr=0.1,
    clustering_radius=20_000,
    cluster_filtering=None,
    tile_size=5_000_000,
    nproc=1,
)
```

This function implements HiCCUPS-style dot calling, but enables user-specified modifications at multiple steps. The current implementation makes two passes over the input data, first to create a histogram of pixel enrichment values, and second to extract significantly enriched pixels.

- The function starts with compatibility verifications
- Recommendation or verification for *kernels* is done next. Custom kernels must satisfy properties including: square shape, equal sizes, odd sizes, zeros in the middle, etc. By default, HiCCUPS-style kernels are recommended based on the binsize.
- Lambda bins are defined for multiple hypothesis testing separately for different value ranges of the locally adjusted expected. Currently, log-binned lambda-bins are hardcoded using a pre-defined BASE of 2^(1/3). *n_lambda_bins* controls the total number of bins. for the *clr*, *expected* and *view* of interest.
- Genomic regions in the specified *view`(all chromosomes by default) are split into smaller tiles of size `tile_size*.
- *scoring_and_histogramming_step()* is performed independently on the genomic tiles. In this step, locally adjusted expected is calculated using convolution kernels for each pixel in the tile. All surveyed pixels are histogrammed according to their adjusted expected and raw observed counts. Locally adjusted expected is not stored in memory.
- Chunks of histograms are aggregated together and a modified BH-FDR procedure is applied to the result in *determine_thresholds()*. This returns thresholds for statistical significance in each lambda-bin (for observed counts), along with the adjusted p-values (q-values).
- Calculated thresholds are used to extract statistically significant pixels in *scoring_and_extraction_step()*. Because locally adjusted expected is not stored in memory, it is re-caluclated during this step, which makes it computationally intensive. Locally adjusted expected values are required in order to apply different thresholds of significance depending on the lambda-bin.
- Returned filtered pixels, or ‘dots’, are significantly enriched relative to their locally adjusted expecteds and thus have potential biological interest. Dots are further annotated with their genomic coordinates and q-values (adjusted p-values) for all applied kernels.
- All further steps perform optional post-processing on called dots

  - enriched pixels that are within *clustering_radius* of each other are clustered together and the brightest one is selected as the representative position of a dot.
  - cluster-representatives along with "singletons" (enriched pixels that are not part of any cluster) can be subjected to further empirical enrichment filtering in *cluster_filtering_hiccups()*. This both requires clustered dots exceed prescribed enrichment thresholds relative to their local neighborhoods and that singletons pass an even more stringent q-value threshold.

`cooltools.api.dotfinder.`**`adjusted_exp_name`**`(`*kernel_name*`)`

`cooltools.api.dotfinder.`**`annotate_pixels_with_qvalues`**`(`*pixels_df*, *qvalues*, *obs_raw_name='count'*`)`

Add columns with the qvalues to a DataFrame of scored pixels

**Parameters**

- **`pixels_df`** (*`pandas.DataFrame`*) – a DataFrame with pixel coordinates that must have at least 2 columns named 'bin1_id' and 'bin2_id', where first is pixels's row and the second is pixel's column index.
- **`qvalues`** (*`dict of DataFrames`*) – A dictionary with keys being kernel names and values DataFrames storing q-values for each observed count values in each lambda- bin. Colunms are Intervals defined by 'ledges' boundaries. Rows corresponding to a range of observed count values.
- **`obs_raw_name`** (*`str`*) – Name of the column/field that carry number of counts per pixel, i.e. observed raw counts.

**Returns**

**pixels_qvalue_df** (*pandas.DataFrame*) – DataFrame of pixels with additional columns la_exp.{k}.qval, storing q-values (adjusted p-values) corresponding to the count value of a pixel, its kernel, and a lambda-bin it belongs to.

`cooltools.api.dotfinder.`**`bp_to_bins`**`(`*basepairs*, *binsize*`)`

`cooltools.api.dotfinder.`**`clust_2D_pixels`**`(`*pixels_df*, *threshold_cluster=2*, *bin1_id_name='bin1_id'*, *bin2_id_name='bin2_id'*, *clust_label_name='c_label'*, *clust_size_name='c_size'*`)`

Group significant pixels by proximity using Birch clustering. We use "n_clusters=None", which implies no AgglomerativeClustering, and thus simply reporting "blobs" of pixels of radii <="threshold_cluster" along with corresponding blob-centroids as well.

**Parameters**

- **`pixels_df`** (*`pandas.DataFrame`*) – a DataFrame with pixel coordinates that must have at least 2 columns named 'bin1_id' and 'bin2_id', where first is pixels's row and the second is pixel's column index.
- **`threshold_cluster`** (*`int`*) – clustering radius for Birch clustering derived from ~40kb radius of clustering and bin size.
- **`bin1_id_name`** (*`str`*) – Name of the 1st coordinate (row index) in 'pixel_df', by default 'bin1_id'. 'start1/end1' could be usefull as well.
- **`bin2_id_name`** (*`str`*) – Name of the 2nd coordinate (column index) in 'pixel_df', by default 'bin2_id'. 'start2/end2' could be usefull as well.
- **`clust_label_name`** (*`str`*) – Name of the cluster of pixels label. "c_label" by default.
- **`clust_size_name`** (*`str`*) – Name of the cluster of pixels size. "c_size" by default.

**Returns**

**peak_tmp** (*pandas.DataFrame*) – DataFrame with the following columns: [c+bin1_id_name, c+bin2_id_name, clust_label_name, clust_size_name] row/col (bin1/bin2) are coordinates of centroids, label and sizes are unique pixel-cluster labels and their corresponding sizes.

cooltools.api.dotfinder.**cluster_filtering_hiccups**(*centroids*, *obs_raw_name='count'*, *enrichment_factor_vh=1.5*, *enrichment_factor_d_and_ll=1.75*, *enrichment_factor_d_or_ll=2.0*, *FDR_orphan_threshold=0.02*)

Centroids of enriched pixels can be filtered to further minimize the amount of false-positive dot-calls.

First, centroids are filtered on enrichment relative to the locally-adjusted expected for the "donut", "lowleft", "vertical", and "horizontal" kernels. Additionally, singleton pixels (i.e. pixels that do not belong to a cluster) are filtered based on a combined q-values for all kernels. This empirical filtering approach was developed in Rao et al 2014 and results in a conservative dot-calls with the low rate of false-positive calls.

**Parameters**

- **centroids** (*pd.DataFrame*) – DataFrame that stores enriched and clustered pixels.
- **obs_raw_name** (*str*) – name of the column with raw observed pixel counts
- **enrichment_factor_vh** (*float*) – minimal enrichment factor for pixels relative to both "vertical" and "horizontal" kernel.
- **enrichment_factor_d_and_ll** (*float*) – minimal enrichment factor for pixels relative to both "donut" and "lowleft" kernels.
- **enrichment_factor_d_or_ll** (*float*) – minimal enrichment factor for pixels relative to either "donut" or" "lowleft" kenels.
- **FDR_orphan_threshold** (*float*) – minimal combined q-value for singleton pixels.

**Returns**

**filtered_centroids** (*pd.DataFrame*) – filtered dot-calls

cooltools.api.dotfinder.**clustering_step**(*scored_df*, *dots_clustering_radius*, *assigned_regions_name='region'*, *obs_raw_name='count'*)

Group together adjacent significant pixels into clusters after the lambda-binning multiple hypothesis testing by iterating over assigned regions and calling *clust_2D_pixels*.

**Parameters**

- **scored_df** (*pandas.DataFrame*) – DataFrame with enriched pixels that are ready to be clustered and are annotated with their genomic coordinates.
- **dots_clustering_radius** (*int*) – Birch-clustering threshold.
- **assigned_regions_name** (*str | None*) – Name of the column in scored_df to use for grouping pixels before clustering. When None, full chromosome clustering is done.
- **obs_raw_name** (*str*) – name of the column with raw observed pixel counts

**Returns**

**centroids** (*pandas.DataFrame*) – Pixels from 'scored_df' annotated with clustering information.

### Notes

'dots_clustering_radius' in Birch clustering algorithm corresponds to a double the clustering radius in the "greedy"-clustering used in HiCCUPS

`cooltools.api.dotfinder.`**`determine_thresholds`**`(`*gw_hist*, *fdr*`)`

given a 'gw_hist' histogram of observed counts for each lambda-bin and for each kernel-type, and also given a FDR, calculate q-values for each observed count value in each lambda-bin for each kernel-type.

**Parameters**

- **gw_hist_kernels** (*dict*) – dictionary {kernel_name : 2D_hist}, where '2D_hist' is a pd.DataFrame
- **fdr** (*float*) – False Discovery Rate level

**Returns**

- **threshold_df** (*dict*) – each threshold_df[k] is a Series indexed by la_exp intervals (IntervalIndex) and it is all we need to extract "good" pixels from each chunk …
- **qvalues** (*dict*) – A dictionary with keys being kernel names and values pandas.DataFrames storing q-values: each column corresponds to a lambda-bin, while rows correspond to observed pixels values.

`cooltools.api.dotfinder.`**`dots`**`(`*clr*, *expected*, *expected_value_col='balanced.avg'*, *clr_weight_name='weight'*, *view_df=None*, *kernels=None*, *max_loci_separation=10000000*, *max_nans_tolerated=1*, *n_lambda_bins=40*, *lambda_bin_fdr=0.1*, *clustering_radius=20000*, *cluster_filtering=None*, *tile_size=5000000*, *nproc=1*`)`

Call dots on a cooler {clr}, using {expected} defined in regions specified in {view_df}.

All convolution kernels specified in {kernels} will be all applied to the {clr}, and statistical testing will be performed separately for each kernel. A convolutional kernel is a small squared matrix (e.g. 7x7) of zeros and ones that defines a "mask" to extract local expected around each pixel. Since the enrichment is calculated relative to the central pixel, kernel width should be an odd number >=3.

**Parameters**

- **clr** (*cooler.Cooler*) – A cooler with balanced Hi-C data.
- **expected** (*DataFrame in expected format*) – Diagonal summary statistics for each chromosome, and name of the column with the values of expected to use.
- **expected_value_col** (*str*) – Name of the column in expected that holds the values of expected
- **clr_weight_name** (*str*) – Name of the column in the clr.bins to use as balancing weights. Using raw unbalanced data is not supported for dot-calling.
- **view_df** (*viewframe*) – Viewframe with genomic regions, at the moment the view has to match the view used for generating expected. If None, generate from the cooler.
- **kernels** (*{ str:np.ndarray } | None*) – A dictionary of convolution kernels to be used for calculating locally adjusted expected. If None the default kernels from HiCCUPS are going to be recommended based on the resolution of the cooler.
- **max_loci_separation** (*int*) – Miaximum loci separation for dot-calling, i.e., do not call dots for loci that are further than max_loci_separation basepair apart. default 10Mb.

- **max_nans_tolerated** (*int*) – Maximum number of NaNs tolerated in a footprint of every used kernel Adjust with caution, as large max_nans_tolerated, might lead to artifacts in pixels scoring.
- **n_lambda_bins** (*int*) – Number of log-spaced bins, where FDR-testing will be performed independently. TODO: generate lambda-bins on the fly based on the dynamic range of the data (i.e. maximum pixel count)
- **lambda_bin_fdr** (*float*) – False discovery rate (FDR) for multiple hypothesis testing BH-FDR procedure, applied per lambda bin.
- **clustering_radius** (*None | int*) – Cluster enriched pixels with a given radius. "Brightest" pixels in each group will be reported as the final dot-calls. If None, no clustering is performed.
- **cluster_filtering** (*bool*) – whether to apply additional filtering to centroids after clustering, using cluster_filtering_hiccups()
- **tile_size** (*int*) – Tile size for the Hi-C heatmap tiling. Typically on order of several mega-bases, and <= max_loci_separation. Controls tradeoff between memory consumption and speed of execution.
- **nproc** (*int*) – Number of processes to use for multiprocessing.

**Returns**

**dots** (*pandas.DataFrame*) – BEDPE-style dataFrame with genomic coordinates of called dots and additional annotations.

### Notes

'clustering_radius' in Birch clustering algorithm corresponds to a double the clustering radius in the "greedy"-clustering used in HiCCUPS (to be tested).

TODO describe sequence of processing steps

`cooltools.api.dotfinder.`**`extract_scored_pixels`**(*scored_df*, *thresholds*, *ledges*, *obs_raw_name='count'*)

Implementation of HiCCUPS-like lambda-binning statistical procedure. Use FDR thresholds for different "classes" of hypothesis (classified by their locally-adjusted expected (la_exp) scores), in order to extract "enriched" pixels.

**Parameters**

- **scored_df** (*pd.DataFrame*) – A table with the scoring information for a group of pixels.
- **thresholds** (*dict*) – A dictionary {kernel_name : lambda_thresholds}, where 'lambda_thresholds' are pd.Series with FDR thresholds indexed by lambda-bin intervals
- **ledges** (*ndarray*) – An ndarray with bin lambda-edges for groupping locally adjusted expecteds, i.e., classifying statistical hypothesis into lambda-bins. Left-most bin (-inf, 1], and right-most one (value,+inf].
- **obs_raw_name** (*str*) – Name of the column/field with number of counts per pixel, i.e. observed raw counts.

**Returns**

**scored_df_slice** (*pandas.DataFrame*) – Filtered DataFrame of pixels that satisfy thresholds.

`cooltools.api.dotfinder.`**`generate_tiles_diag_band`**(*clr*, *view_df*, *pad_size*, *tile_size*, *band_to_cover*)

A generator yielding corrdinates of heatmap tiles that are needed to cover the requested band_to_cover around diagonal. Each tile is "padded" with the pad of size 'pad_size' to allow for convolution near the boundary of a tile.

**Parameters**

- **clr** (*cooler*) – Cooler object to use to extract chromosome extents.
- **view_df** (*viewframe*) – Viewframe with genomic regions to process, chrom, start, end, name.
- **pad_size** (*int*) – Size of padding around each tile. Typically the outer size of the kernel.
- **tile_size** (*int*) – Size of the heatmap tile.
- **band_to_cover** (*int*) – Size of the diagonal band to be covered by the generated tiles. Typically correspond to the max_loci_separation for called dots.

**Returns**

**tile_coords** (*tuple*) – Generator of tile coordinates, i.e. tuples of three: (region_name, tile_span_i, tile_span_j), where 'tile_span_i/j' each is a tuple of bin ids (bin_start, bin_end).

cooltools.api.dotfinder.**get_adjusted_expected_tile_some_nans**(*origin_ij*, *observed*, *expected*, *bal_weights*, *kernels*)

Get locally adjusted expected for a collection of local-filters (kernels).

Such locally adjusted expected, 'Ek' for a given kernel, can serve as a baseline for deciding whether a given pixel is enriched enough to call it a feature (dot-loop, flare, etc.) in a downstream analysis.

For every pixel of interest [i,j], locally adjusted expected is a product of a global expected in that pixel E_bal[i,j] and an enrichment of local environ- ment of the pixel, described with a given kernel:

```
                             KERNEL[i,j](O_bal)
Ek_bal[i,j] = E_bal[i,j]* ------------------
                             KERNEL[i,j](E_bal)
```

where KERNEL[i,j](X) is a result of convolution between the kernel and a slice of matrix X centered around (i,j). See link below for details: https://en.wikipedia.org/wiki/Kernel_(image_processing)

Returned values for observed and all expecteds are rescaled back to raw-counts, for the sake of downstream statistical analysis, which is using Poisson test to decide is a given pixel is enriched. (comparison between balanced values using Poisson- test is intractable):

```
                             KERNEL[i,j](O_bal)
Ek_raw[i,j] = E_raw[i,j]* ------------------ ,
                             KERNEL[i,j](E_bal)
```

where E_raw[i,j] is:

```
      1                1
-------------- * -------------- * E_bal[i,j]
bal_weights[i]   bal_weights[j]
```

**Parameters**

- **origin_ij** (*(int,int) tuple*) – tuple of interegers that specify the location of an observed matrix slice. Measured in bins, not in nucleotides.
- **observed** (*numpy.ndarray*) – square symmetrical dense-matrix that contains balanced observed O_bal
- **expected** (*numpy.ndarray*) – square symmetrical dense-matrix that contains expected, calculated based on balanced observed: E_bal.

- `bal_weights` (*numpy.ndarray or (numpy.ndarray, numpy.ndarray)*) – 1D vector used to turn raw observed into balanced observed for a slice of a matrix with the origin_ij on the diagonal; and a tuple/list of a couple of 1D arrays in case it is a slice with an arbitrary origin_ij.
- `kernels` (*dict of (str, numpy.ndarray)*) – dictionary of kernels/masks to perform convolution of the heatmap. Kernels describe the local environment, and used to estimate baseline for finding enriched/prominent peaks. Peak must be enriched with respect to all local environments (all kernels), to be considered significant. Dictionay keys must contain names for each kernel. Note, scipy.ndimage.convolve first flips kernel and only then applies it to matrix.

**Returns**

**peaks_df** (*pandas.DataFrame*) – DataFrame with the results of locally adjusted calculations for every kernel for a given slice of input matrix.

### Notes

**Reported columns:**

bin1_id - bin1_id index (row), adjusted to tile_start_i bin2_id - bin bin2_id index, adjusted to tile_start_j la_exp - locally adjusted expected (for each kernel) la_nan - number of NaNs around (each kernel's footprint) exp.raw - global expected, rescaled to raw-counts obs.raw(counts) - observed values in raw-counts.

Depending on the intial tiling of the interaction matrix, concatened *peaks_df* may require "deduplication", as some pixels can be evaluated in several tiles (e.g. near the tile edges). Default tilitng in the *dots* functions, should avoid this problem.

`cooltools.api.dotfinder.histogram_scored_pixels`(*scored_df*, *kernels*, *ledges*, *obs_raw_name='count'*)

An attempt to implement HiCCUPS-like lambda-binning statistical procedure. This function aims at building up a histogram of locally adjusted expected scores for groups of characterized pixels.

Such histograms are subsequently used to compute FDR thresholds for different "classes" of hypothesis (classified by their locally-adjusted expected (la_exp)).

**Parameters**

- `scored_df` (*pd.DataFrame*) – A table with the scoring information for a group of pixels.
- `kernels` (*dict*) – A dictionary with keys being kernels names and values being ndarrays representing those kernels.
- `ledges` (*ndarray*) – An ndarray with bin lambda-edges for groupping locally adjusted expecteds, i.e., classifying statistical hypothesis into lambda-bins. Left-most bin (-inf, 1], and right-most one (value,+inf].
- `obs_raw_name` (*str*) – Name of the column/field that carry number of counts per pixel, i.e. observed raw counts.

**Returns**

**hists** (*dict of pandas.DataFrame*) – A dictionary of pandas.DataFrame with lambda/observed 2D histogram for every kernel-type.

### Notes

returning histograms corresponding to the chunks of scored pixels.

`cooltools.api.dotfinder.`**`is_compatible_kernels`**`(`*kernels*, *binsize*, *max_nans_tolerated*`)`

**TODO implement checks for kernels:**

- matrices are of the same size
- they should be squared (too restrictive ? maybe pad with 0 as needed)
- dimensions are odd, to have a center pixel to refer to
- they can be turned into int 1/0 ones (too restrictive ? allow weighted kernels ?)
- the central pixel should be zero perhaps (unless weights are allowed 4sure)
- maybe introduce an upper limit to the size - to avoid crazy long calculations
- check relative to the binsize maybe ? what's the criteria ?

`cooltools.api.dotfinder.`**`nans_inkernel_name`**`(`*kernel_name*`)`

`cooltools.api.dotfinder.`**`recommend_kernels`**`(`*binsize*`)`

Return a recommended set of convolution kernels for dot-calling based on the resolution, or binsize, of the input data.

This function currently recommends the four kernels used in the HiCCUPS method: donut, horizontal, vertical, lowerleft. Kernels are recommended for resolutions near 5 kb, 10 kb, and 25 kb. Dots are not typically visible at lower resolutions (binsize >28kb) and the majority of datasets are too sparse for dot-calling at very high resolutions (<4kb). Given this, default kernels are not recommended for resolutions outside this range.

**Parameters**

**binsize** (*integer*) – binsize of the provided cooler

**Returns**

**kernels** (*{str:ndarray}*) – dictionary of convolution kernels as ndarrays, with their names as keys.

`cooltools.api.dotfinder.`**`score_tile`**`(`*tile_cij*, *clr*, *expected_indexed*, *expected_value_col*, *clr_weight_name*, *kernels*, *max_nans_tolerated*, *band_to_cover*`)`

The main working function that given a tile of a heatmap, applies kernels to perform convolution to calculate locally-adjusted expected and then calculates a p-value for every meaningfull pixel against these locally-adjusted expected (la_exp) values.

**Parameters**

- **tile_cij** (*tuple*) – Tuple of 3: region name, tile span row-wise, tile span column-wise: (region, tile_span_i, tile_span_j), where tile_span_i = (start_i, end_i), and tile_span_j = (start_j, end_j).
- **clr** (*cooler*) – Cooler object to use to extract Hi-C heatmap data.
- **expected_indexed** (*pandas.DataFrame*) – DataFrame with cis-expected, indexed with 'region1', 'region2', 'dist'.
- **expected_value_col** (*str*) – Name of a value column in expected DataFrame
- **clr_weight_name** (*str*) – Name of a value column with balancing weights in a cooler.bins() DataFrame. Typically 'weight'.
- **kernels** (*dict*) – A dictionary with keys being kernels names and values being ndarrays representing those kernels.

- **max_nans_tolerated** (*int*) – Number of NaNs tolerated in a footprint of every kernel.
- **band_to_cover** (*int*) – Results would be stored only for pixels connecting loci closer than 'band_to_cover'.

**Returns**

**res_df** (*pandas.DataFrame*) – results: annotated pixels with calculated locally adjusted expected for every kernels, observed, precalculated pvalues, number of NaNs in footprint of every kernels, all of that in a form of an annotated pixels DataFrame for eligible pixels of a given tile.

cooltools.api.dotfinder.**scoring_and_extraction_step**(*clr*, *expected_indexed*, *expected_value_col*, *clr_weight_name*, *tiles*, *kernels*, *ledges*, *thresholds*, *max_nans_tolerated*, *loci_separation_bins*, *nproc*, *bin1_id_name='bin1_id'*, *bin2_id_name='bin2_id'*, *map_functor=<class 'map'>*)

This implements the 2nd step of the lambda-binning scoring procedure, extracting pixels that are FDR compliant.

In short, this combines scoring with with extraction into a single pipeline of per-chunk operations/transforms.

cooltools.api.dotfinder.**scoring_and_histogramming_step**(*clr*, *expected_indexed*, *expected_value_col*, *clr_weight_name*, *tiles*, *kernels*, *ledges*, *max_nans_tolerated*, *loci_separation_bins*, *nproc*, *map_functor=<class 'map'>*)

This implements the 1st step of the lambda-binning scoring procedure - histogramming.

In short, this pipes a scoring operation together with histogramming into a single pipeline of per-chunk operations/transforms.

cooltools.api.dotfinder.**tile_square_matrix**(*matrix_size*, *offset*, *tile_size*, *pad=0*)

Generate a stream of coordinates of tiles that cover a matrix of a given size. Matrix has to be square, on-digaonal one: e.g. corresponding to a chromosome or a chromosomal arm.

**Parameters**

- **matrix_size** (*int*) – Size of a squared matrix
- **offset** (*int*) – Offset coordinates of generated tiles by 'offset'
- **tile_size** (*int*) – Requested size of the tiles. Tiles near the right and botoom edges could be rectangular and smaller then 'tile_size'
- **pad** (*int*) – Small padding around each tile to be included in the yielded coordinates.

**Yields**

**Pairs of indices/coordinates of every tile** (*(start_i, end_i), (start_j, end_j)*)

### Notes

Generated tiles coordinates [start_i,end_i) , [start_i,end_i) can be used to fetch heatmap tiles from cooler: >>> clr.matrix()[start_i:end_i, start_j:end_j]

'offset' is useful when a given matrix is part of a larger matrix (a given chromosome or arm), and thus all coordinated needs to be offset to get absolute coordinates.

Tiles are non-overlapping (pad=0), but tiles near the right and bottom edges could be rectangular:

- – * ·

  ·

.
.
.
.
• – * .
• – * .
• – * .
.
.
.
.
• – *
• – … *
• – * .
.
• –
• – * .
.
.
.
.
.

### cooltools.api.eigdecomp module

`cooltools.api.eigdecomp.`**`cis_eig`**(*A*, *n_eigs=3*, *phasing_track=None*, *ignore_diags=2*, *clip_percentile=0*, *sort_metric=None*)

Compute compartment eigenvector on a dense cis matrix.

Note that the amplitude of compartment eigenvectors is weighted by their corresponding eigenvalue

**Parameters**

- **A** (*2D array*) – balanced dense contact matrix
- **n_eigs** (*int*) – number of eigenvectors to compute
- **phasing_track** (*1D array, optional*) – if provided, eigenvectors are flipped to achieve a positive correlation with *phasing_track*.
- **ignore_diags** (*int*) – the number of diagonals to ignore
- **clip_percentile** (*float*) – if >0 and <100, clip pixels with diagonal-normalized values higher than the specified percentile of matrix-wide values.
- **sort_metric** (*str*) – If provided, re-sort *eigenvecs* and *eigvals* in the order of decreasing correlation between phasing_track and eigenvector, using the specified measure of correlation. Possible values: 'pearsonr' - sort by decreasing Pearson correlation. 'var_explained' - sort by decreasing absolute amount of variation in *eigvecs* explained

by *phasing_track* (i.e. R^2 * var(eigvec)) 'MAD_explained' - sort by decreasing absolute amount of Median Absolute Deviation from the median of *eigvecs* explained by *phasing_track* (i.e. COMED(eigvec, phasing_track) * MAD(eigvec)). 'spearmanr' - sort by decreasing Spearman correlation. This option is designed to report the most "biologically" informative eigenvectors first, and prevent eigenvector swapping caused by translocations. In reality, however, sometimes it shows poor performance and may lead to reporting of non-informative eigenvectors. Off by default.

**Returns**

- *eigenvalues, eigenvectors*
- *.. note:: ALWAYS check your EVs by eye. The first one occasionally does* – not reflect the compartment structure, but instead describes chromosomal arms or translocation blowouts.

`cooltools.api.eigdecomp.`**`eigs_cis`**(*clr*, *phasing_track=None*, *view_df=None*, *n_eigs=3*, *clr_weight_name='weight'*, *ignore_diags=None*, *clip_percentile=99.9*, *sort_metric=None*, *map=<class 'map'>*)

Compute compartment eigenvector for a given cooler *clr* in a number of symmetric intra chromosomal regions defined in view_df (cis-regions), or for each chromosome.

Note that the amplitude of compartment eigenvectors is weighted by their corresponding eigenvalue. Eigenvectors can be oriented by passing a binned *phasing_track* with the same resolution as the cooler.

**Parameters**

- **clr** (*cooler*) – cooler object to fetch data from
- **phasing_track** (*DataFrame*) – binned track with the same resolution as cooler bins, the fourth column is used to phase the eigenvectors, flipping them to achieve a positive correlation.
- **view_df** (*iterable or DataFrame, optional*) – if provided, eigenvectors are calculated for the regions of the view only, otherwise chromosome-wide eigenvectors are computed, for chromosomes specified in phasing_track.
- **n_eigs** (*int*) – number of eigenvectors to compute
- **clr_weight_name** (*str*) – name of the column with balancing weights to be used.
- **ignore_diags** (*int, optional*) – the number of diagonals to ignore. Derived from cooler metadata if not specified.
- **clip_percentile** (*float*) – if >0 and <100, clip pixels with diagonal-normalized values higher than the specified percentile of matrix-wide values.
- **sort_metric** (*str*) – If provided, re-sort *eigenvecs* and *eigvals* in the order of decreasing correlation between phasing_track and eigenvector, using the specified measure of correlation. Possible values: 'pearsonr' - sort by decreasing Pearson correlation. 'var_explained' - sort by decreasing absolute amount of variation in *eigvecs* explained by *phasing_track* (i.e. R^2 * var(eigvec)) 'MAD_explained' - sort by decreasing absolute amount of Median Absolute Deviation from the median of *eigvecs* explained by *phasing_track* (i.e. COMED(eigvec, phasing_track) * MAD(eigvec)). 'spearmanr' - sort by decreasing Spearman correlation. This option is designed to report the most "biologically" informative eigenvectors first, and prevent eigenvector swapping caused by translocations. In reality, however, sometimes it shows poor performance and may lead to reporting of non-informative eigenvectors. Off by default.
- **map** (*callable, optional*) – Map functor implementation.

**Returns**

- *eigvals, eigvec_table -> DataFrames with eigenvalues for each region and*
- a table of eigenvectors filled in the *bins* table.
- *.. note:: ALWAYS check your EVs by eye. The first one occasionally does* – not reflect the compartment structure, but instead describes chromosomal arms or translocation blowouts. Possible mitigations: employ *view_df* (e.g. arms) to avoid issues with chromosomal arms, consider blacklisting regions with translocations during balancing.

cooltools.api.eigdecomp.**eigs_trans**(*clr*, *phasing_track=None*, *n_eigs=3*, *partition=None*, *clr_weight_name='weight'*, *sort_metric=None*, ***kwargs*)

cooltools.api.eigdecomp.**trans_eig**(*A*, *partition*, *n_eigs=3*, *perc_top=99.95*, *perc_bottom=1*, *phasing_track=None*, *sort_metric=False*)

Compute compartmentalization eigenvectors on trans contact data

**Parameters**

- **A** (*2D array*) – balanced whole genome contact matrix
- **partition** (*sequence of int*) – bin offset of each contiguous region to treat separately (e.g., chromosomes or chromosome arms)
- **n_eigs** (*int*) – number of eigenvectors to compute; default = 3
- **perc_top** (*float (percentile)*) – filter - clip trans blowout contacts above this cutoff; default = 99.95
- **perc_bottom** (*float (percentile)*) – filter - remove bins with trans coverage below this cutoff; default=1
- **phasing_track** (*1D array, optional*) – if provided, eigenvectors are flipped to achieve a positive correlation with *phasing_track*.
- **sort_metric** (*str*) – If provided, re-sort *eigenvecs* and *eigvals* in the order of decreasing correlation between phasing_track and eigenvector, using the specified measure of correlation. Possible values: 'pearsonr' - sort by decreasing Pearson correlation. 'var_explained' - sort by decreasing absolute amount of variation in *eigvecs* explained by *phasing_track* (i.e. R^2 * var(eigvec)) 'MAD_explained' - sort by decreasing absolute amount of Median Absolute Deviation from the median of *eigvecs* explained by *phasing_track* (i.e. COMED(eigvec, phasing_track) * MAD(eigvec)). 'spearmanr' - sort by decreasing Spearman correlation. This option is designed to report the most "biologically" informative eigenvectors first, and prevent eigenvector swapping caused by translocations. In reality, however, sometimes it shows poor performance and may lead to reporting of non-informative eigenvectors. Off by default.

**Returns**

- *eigenvalues, eigenvectors*
- *.. note:: ALWAYS check your EVs by eye. The first one occasionally does* – not reflect the compartment structure, but instead describes chromosomal arms or translocation blowouts.

### cooltools.api.expected module

cooltools.api.expected.**blocksum_pairwise**(*clr*, *view_df*, *transforms={}*, *clr_weight_name='weight'*, *chunksize=1000000*, *map=<class 'map'>*)

Summary statistics on rectangular blocks of all (trans-)pairwise combinations of genomic regions in the view_df (aka trans-expected).

**Note:** This is a special case of asymmetric block-level summary stats, that can be calculated very efficiently. Regions in view_df are assigned to pixels only once and pixels falling into a given asymmetric block i != j are summed up.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **view_df** (*viewframe*) – view_df of regions defining blocks for summary calculations, has to be sorted according to the order of chromosomes in clr.
- **transforms** (*dict of str -> callable, optional*) – Transformations to apply to pixels. The result will be assigned to a temporary column with the name given by the key. Callables take one argument: the current chunk of the (annotated) pixel dataframe.

- **clr_weight_name** (*str*) – name of the balancing weight column in cooler bin-table used to count "bad" pixels per block. Set to *None* not ot mask "bad" pixels (raw data only).
- **chunksize** (*int, optional*) – Size of pixel table chunks to process
- **map** (*callable, optional*) – Map functor implementation.

**Returns**

**DataFrame with entries for each blocks** (*region1, region2, n_valid, count.sum*)

cooltools.api.expected.**combine_binned_expected**(*binned_exp*, *binned_exp_slope=None*, *Pc_name='balanced.avg'*, *der_smooth_function_combined=<function <lambda>>*, *spread_funcs='logstd'*, *spread_funcs_slope='std'*, *minmax_drop_bins=2*, *concat_original=False*)

Combines by-region log-binned expected and slopes into genome-wide averages, handling small chromosomes and "corners" in an optimal fashion, robust to outliers. Calculates spread of by-chromosome P(s) and slopes, also in an optimal fashion.

**Parameters**

- **binned_exp** (*dataframe*) – binned expected as outputed by logbin_expected
- **binned_exp_slope** (*dataframe or None*) – If provided, estimates spread of slopes. Is necessary if concat_original is True
- **Pc_name** (*str*) – Name of the column with the probability of contacts. Defaults to "balanced.avg".
- **der_smooth_function_combined** (*callable*) – A smoothing function for calculating slopes on combined data
- **spread_funcs** (*"minmax", "std", "logstd" or a function (see below)*) – A way to estimate the spread of the P(s) curves between regions. * "minmax" - use the minimum/maximum of by-region P(s) * "std" - use weighted standard deviation of P(s) curves (may produce negative results) * "logstd" (recommended) weighted standard deviation in logspace (as seen on the plot)
- **spread_funcs_slope** (*"minmax", "std" or a funciton*) – Similar to spread_func, but for slopes rather than P(s)
- **concat_original** (*bool (default = False)*) – Append original dataframe, and put combined under region "combined"

**Returns**

*scal, slope_df*

## Notes

This function does not calculate errorbars. The spread is not the deviation of the mean, and rather is representative of variability between chromosomes.

Calculating errorbars/spread

1. Take all by-region P(s)
2. For "minmax", remove the last var_drop_last_bins bins for each region (by default two. They are most noisy and would inflate the spread for the last points). Min/max are most susceptible to this.
3. Groupby P(s) by region
4. Apply spread_funcs to the pd.GroupBy object. Options are: * minimum and maximum ("minmax"), * weighted standard deviation ("std"), * weighted standard deviation in logspace ("logstd", default) or two custom functions We do not remove the last bins for "std" / "logstd" because we are doing weighted standard deviation. Therefore, noisy "ends" of regions would contribute very little to this.
5. Append them to the P(s) for the same bin.

As a result, by for minmax, we do not estimate spread for the last two bins. This is because there are often very few chromosomal arms there, and different arm measurements are noisy. For other methods, we do estimate the spread there, and noisy last bins are taken care of by the weighted standard deviation. However, the spread in the last bins may be noisy, and may become a 0 if only one region is contributing to the last pixel.

`cooltools.api.expected.`**`count_all_pixels_per_block`**`(`*x*, *y*`)`

Calculate total number of pixels in a rectangular block

**Parameters**

- **x** (*int*) – block width in pixels
- **y** (*int*) – block height in pixels

**Returns**

**number_of_pixels** (*int*) – total number of pixels in a block

`cooltools.api.expected.`**`count_all_pixels_per_diag`**`(`*n*`)`

Total number of pixels on each upper diagonal of a square matrix.

**Parameters**

**n** (*int*) – total number of bins (dimension of square matrix)

**Returns**

**dcount** (*1D array of length n*) – dcount[d] == total number of pixels on diagonal d

`cooltools.api.expected.`**`count_bad_pixels_per_block`**`(`*x*, *y*, *bad_bins_x*, *bad_bins_y*`)`

Calculate number of "bad" pixels per rectangular block of a contact map

**Parameters**

- **x** (*int*) – block width in pixels
- **y** (*int*) – block height in pixels
- **bad_bins_x** (*int*) – number of bad bins on x-side
- **bad_bins_y** (*int*) – number of bad bins on y-side

**Returns**

**number_of_pixes** (*int*) – number of "bad" pixels in a block

`cooltools.api.expected.`**`count_bad_pixels_per_diag`**`(`*n*, *bad_bins*`)`

Efficiently count the number of bad pixels on each upper diagonal of a matrix assuming a sequence of bad bins forms a "grid" of invalid pixels.

Each bad bin bifurcates into two a row and column of bad pixels, so an upper bound on number of bad pixels per diagonal is 2*k, where k is the number of bad bins. For a given diagonal, we need to subtract from this upper estimate the contribution from rows/columns reaching "out-of-bounds" and the contribution of the intersection points of bad rows with bad columns that get double counted.

```
o : bad bin
* : bad pixel
x : intersection bad pixel
$ : out of bounds bad pixel
     $    $     $
 *--------------------------+
  *  *    *     *           |
   * *    *     *           |
    **    *     *           |
     o****x*****x***********|$
      *   *     *           |
       *  *     *           |
        * *     *           |
         o******x***********|$
          *     *           |
           *    *           |
            *   *           |
             *  *           |
```

(continues on next page)

(continued from previous page)

```
              * *           |
               **           |
                o***********|$
                 *          |
                  *         |
```

**Parameters**

- **n** (*int*) – total number of bins
- **bad_bins** (*1D array of int*) – sorted array of bad bin indexes

**Returns**

**dcount** (*1D array of length n*) – dcount[d] == number of bad pixels on diagonal d

`cooltools.api.expected.`**`diagsum_from_array`**(*A*, *counts=None*, ***, *offset=0*, *ignore_diags=2*, *filter_counts=False*, *region_name=None*)

Calculates Open2C-formatted expected for a dense submatrix of a whole genome contact map.

**Parameters**

- **A** (*2D array*) – Normalized submatrix to calculate expected (`balanced.sum`).
- **counts** (*2D array or None, optional*) – Corresponding raw contacts to populate `count.sum`.
- **offset** (*int or (int, int)*) – i- and j- bin offsets of A relative to the parent matrix. If a single offset is provided it is applied to both axes.
- **ignore_diags** (*int, optional*) – Number of initial diagonals to ignore.
- **filter_counts** (*bool, optional*) – Apply the validity mask from balanced matrix to the raw one. Ignored when counts is None.
- **region_name** (*str or (str, str), optional*) – A custom region name or pair of region names. If provided, region columns will be included in the output.

### Notes

For regions that cross the main diagonal of the whole-genome contact map, the lower triangle “overhang” is ignored.

### Examples

```
>>> A = clr.matrix()[:, :]  # whole genome balanced
>>> C = clr.matrix(balance=False)[:, :]  # whole genome raw
```

Using only balanced data: >>> exp = diagsum_from_array(A)

Using balanced and raw counts: >>> exp1 = diagsum_from_array(A, C)

Using an off-diagonal submatrix >>> exp2 = diagsum_from_array(A[:50, 50:], offset=(0, 50))

`cooltools.api.expected.`**`diagsum_pairwise`**(*clr*, *view_df*, *transforms={}*, *clr_weight_name='weight'*, *ignore_diags=2*, *chunksize=10000000*, *map=<class 'map'>*)

Intra-chromosomal diagonal summary statistics for asymmetric blocks of contact matrix defined as pairwise combinations of regions in “view_df.

**Note:** This is a special case of asymmetric diagonal summary statistic that is efficient and covers the most important practical case of inter-chromosomal arms “expected” calculation.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **view_df** (*viewframe*) – view_df of regions for intra-chromosomal diagonal summation, has to be sorted according to the order of chromosomes in cooler.
- **transforms** (*dict of str -> callable, optional*) – Transformations to apply to pixels. The result will be assigned to a temporary column with the name given by the key. Callables take one argument: the current chunk of the (annotated) pixel dataframe.
- **clr_weight_name** (*str*) – name of the balancing weight vector used to count "bad" pixels per diagonal. Set to *None* not to mask "bad" pixels (raw data only).
- **chunksize** (*int, optional*) – Size of pixel table chunks to process
- **map** (*callable, optional*) – Map functor implementation.

**Returns**

- *Dataframe of diagonal statistics for all intra-chromosomal blocks defined as*
- *pairwise combinations of regions in the view*

`cooltools.api.expected.`**diagsum_symm**(*clr*, *view_df*, *transforms={}*, *clr_weight_name='weight'*, *ignore_diags=2*, *chunksize=10000000*, *map=<class 'map'>*)

Intra-chromosomal diagonal summary statistics.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **view_df** (*viewframe*) – view_dfof regions for intra-chromosomal diagonal summation
- **transforms** (*dict of str -> callable, optional*) – Transformations to apply to pixels. The result will be assigned to a temporary column with the name given by the key. Callables take one argument: the current chunk of the (annotated) pixel dataframe.
- **clr_weight_name** (*str*) – name of the balancing weight vector used to count "bad" pixels per diagonal. Set to *None* not to mask "bad" pixels (raw data only).
- **chunksize** (*int, optional*) – Size of pixel table chunks to process
- **ignore_diags** (*int, optional*) – Number of intial diagonals to exclude from statistics
- **map** (*callable, optional*) – Map functor implementation.

**Returns**

*Dataframe of diagonal statistics for all regions in the view*

`cooltools.api.expected.`**expected_cis**(*clr*, *view_df=None*, *intra_only=True*, *smooth=True*, *aggregate_smoothed=True*, *smooth_sigma=0.1*, *clr_weight_name='weight'*, *ignore_diags=2*, *chunksize=10000000*, *nproc=1*, *map_functor=<class 'map'>*)

Calculate average interaction frequencies as a function of genomic separation between pixels i.e. interaction decay with distance. Genomic separation aka "dist" is measured in the number of bins, and defined as an index of a diagonal on which pixels reside (bin1_id - bin2_id).

Average values are reported in the columns with names {}.avg, and they are calculated as a ratio between a corresponding sum {}.sum and the total number of "valid" pixels on the diagonal "n_valid".

When balancing weights (clr_weight_name=None) are not applied to the data, there is no masking of bad bins performed.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **view_df** (*viewframe*) – a collection of genomic intervals where expected is calculated otherwise expected is calculated for full chromosomes. view_df has to be sorted, when inter-regions expected is requested, i.e. intra_only is False.
- **intra_only** (*bool*) – Return expected only for symmetric intra-regions defined by view_df, i.e. chromosomes, chromosomal-arms, intra-domains, etc. When False returns expected both for symmetric intra-regions and assymetric inter-regions.

- smooth (*bool*) – Apply smoothing to cis-expected. Will be stored in an additional column
- aggregate_smoothed (*bool*) – When smoothing, average over all regions, ignored without smoothing.
- smooth_sigma (*float*) – Control smoothing with the standard deviation of the smoothing Gaussian kernel. Ignored without smoothing.
- clr_weight_name (*str or None*) – Name of balancing weight column from the cooler to use. Use raw unbalanced data, when None.
- ignore_diags (*int, optional*) – Number of intial diagonals to exclude results
- chunksize (*int, optional*) – Size of pixel table chunks to process
- nproc (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- map_functor (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

- *DataFrame with summary statistic of every diagonal of every symmetric*
- *or asymmetric block*

### Notes

When clr_weight_name=None, smooth=False, aggregate_smoothed=False, the minimum output DataFrame includes the following quantities (columns):

**dist:**
Distance in bins.

**dist_bp:**
Distance in basepairs.

**contact_freq:**
The "most processed" contact frequency value. For example, if balanced & smoothing then this will return the balanced.avg.smooth.agg; if aggregated+smoothed, then balanced.avg.smooth.agg; if nothing then count.avg.

**n_total:**
Number of total pixels at a given distance.

**n_valid:**
Number of valid pixels (with non-NaN values after balancing) at a given distance.

**count.sum:**
Sum up raw contact counts of all pixels at a given distance.

**count.avg:**
The average raw contact count of pixels at a given distance. count.sum / n_total.

When clr_weigh_name is provided (by default, clr_weigh_name="weight"), the following quantities (columns) will be added into the DataFrame:

**balanced.sum:**
Sum up balanced contact values of valid pixels at a given distance. Returned if clr_weight_name is not None.

**balanced.avg:**
The average balanced contact values of valid pixels at a given distance. balanced.sum / n_valid. Returned if clr_weight_name is not None.

When smooth=True, the following quantities (columns) will be added into the DataFrame:

**count.avg.smoothed:**
Log-smoothed count.avg. Returned if smooth=True and clr_weight_name=None.

**balanced.avg.smoothed:**
Log-smoothed balanced.avg. Returned if smooth=True and clr_weight_name is not None.

**When aggregate_smoothed=True, the following quantities (columns) will be added into the DataFrame:**

**count.avg.smoothed.agg:**
Aggregate Log-smoothed count.avg of all genome regions. Returned if smooth=True and aggregate_smoothed=True and clr_weight_name=None.

**balanced.avg.smoothed.agg:**
Aggregate Log-smoothed balanced.avg of all genome regions. Returned if smooth=True and aggregate_smoothed=True and clr_weight_name is not None.

By default, clr_weight_name="weight", smooth=True, aggregate_smoothed=True, the output DataFrame includes all quantities (columns).

`cooltools.api.expected.`**`expected_trans`**(*clr*, *view_df=None*, *clr_weight_name='weight'*, *chunksize=10000000*, *nproc=1*, *map_functor=<class 'map'>*)

Calculate average interaction frequencies for inter-chromosomal blocks defined as pairwise combinations of regions in view_df.

An expected level of interactions between disjoint chromosomes is calculated as a simple average, as there is no notion of genomic separation for a pair of chromosomes and contact matrix for these regions looks "flat".

Average values are reported in the columns with names {}.avg, and they are calculated as a ratio between a corresponding sum {}.sum and the total number of "valid" pixels on the diagonal "n_valid".

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler object
- **view_df** (*viewframe*) – a collection of genomic intervals where expected is calculated otherwise expected is calculated for full chromosomes, has to be sorted.
- **clr_weight_name** (*str or None*) – Name of balancing weight column from the cooler to use. Use raw unbalanced data, when None.
- **chunksize** (*int, optional*) – Size of pixel table chunks to process
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

- *DataFrame with summary statistic for every trans-blocks*
- *region1, region2, n_valid, count.sum count.avg, etc*

`cooltools.api.expected.`**`genomewide_smooth_cvd`**(*cvd*, *sigma_log10=0.1*, *window_sigma=5*, *points_per_sigma=10*, *cols=None*, *suffix='.smoothed'*)

Smooth the contact-vs-distance curve aggregated across all regions in log-space.

**Parameters**

- **cvd** (*pandas.DataFrame*) – A dataframe with the expected values in the cooltools.expected format.
- **sigma_log10** (*float, optional*) – The standard deviation of the smoothing Gaussian kernel, applied over log10(diagonal), by default 0.1
- **window_sigma** (*int, optional*) – Width of the smoothing window, expressed in sigmas, by default 5
- **points_per_sigma** (*int, optional*) – If provided, smoothing is done only for *points_per_sigma* points per sigma and the rest of the values are interpolated (this results in a major speed-up). By default 10
- **cols** (*dict, optional*) – If provided, use the specified column names instead of the standard ones. See DEFAULT_CVD_COLS variable for the format of this argument.
- **suffix** (*string, optional*) – If provided, use the specified string as the suffix of the output column's name

**Returns**

**cvd** (*pandas.DataFrame*) – A cvd table with extra column for the log-smoothed contact frequencies (by default, "balanced.avg.smoothed.agg" if balanced, or "count.avg.smoothed.agg" if raw).

### Notes

Parameters in "cols" will be used:

**dist:**
Name of the column that stores distance values (by default, "dist").

**n_pixels:**
Name of the column that stores number of pixels (by default, "n_valid" if balanced, or "n_total" if raw).

**n_contacts:**
Name of the column that stores the sum of contacts (by default, "balanced.sum" if balanced, or "count.sum" if raw).

**output_prefix:**
Name prefix of the column that will store output value (by default, "balanced.avg" if balanced, or "count.avg" if raw).

`cooltools.api.expected.`**`interpolate_expected`**`(`*expected*, *binned_expected*, *columns=['balanced.avg']*, *kind='quadratic'*, *by_region=True*, *extrapolate_small_s=False*`)`

Interpolates expected to match binned_expected. Basically, this function smoothes the original expected according to the logbinned expected. It could either use by-region expected (each region will have different expected) or use combined binned_expected (all regions will have the same expected after that)

Such a smoothed expected should be used to calculate observed/expected for downstream analysis.

**Parameters**

- **`expected`** (*`pd.DataFrame`*) – expected as returned by diagsum_symm
- **`binned_expected`** (*`pd.DataFrame`*) – binned expected (combined or not)
- **`columns`** (*`list[str] (optional)`*) – Columns to interpolate. Must be present in binned_expected, but not necessarily in expected.
- **`kind`** (*`str (optional)`*) – Interpolation type, according to scipy.interpolate.interp1d
- **`by_region`** (*`bool or str (optional)`*) – Whether to do interpolation by-region (default=True). False means use one expected for all regions (use entire table). If a region name is provided, expected for that region is used.

`cooltools.api.expected.`**`lattice_pdist_frequencies`**`(`*n*, *points*`)`

Distribution of pairwise 1D distances among a collection of distinct integers ranging from 0 to n-1.

**Parameters**

- **`n`** (*`int`*) – Size of the lattice on which the integer points reside.
- **`points`** (*`sequence of int`*) – Arbitrary integers between 0 and n-1, inclusive, in any order but with no duplicates.

**Returns**

**h** (*1D array of length n*) – h[d] counts the number of integer pairs that are exactly d units apart

### Notes

This is done using a convolution via FFT. Thanks to Peter de Rivaz; see http://stackoverflow.com/questions/42423823/distribution-of-pairwise-distances-between-many-integers.

`cooltools.api.expected.`**`logbin_expected`**`(`*exp*, *summary_name='balanced.sum'*, *bins_per_order_magnitude=10*, *bin_layout='fixed'*, *smooth=<function <lambda>>*, *min_nvalid=200*, *min_count=50*`)`

Logarithmically bins expected as produced by diagsum_symm method.

**Parameters**

- **exp** (*DataFrame*) – DataFrame produced by diagsum_symm
- **summary_name** (*str, optional*) – Name of the column of exp-DataFrame to use as a diagonal summary. Default is "balanced.sum".
- **bins_per_order_magnitude** (*int, optional*) – How many bins per order of magnitude. Default of 10 has a ratio of neighboring bins of about 1.25
- **bin_layout** (*"fixed", "longest_region", or array*) – "fixed" means that bins are exactly the same for different datasets, and only depend on bins_per_order_magnitude

  "longest_region" means that the last bin will end at size of the longest region.

  > GOOD: the last bin will have as much data as possible. BAD: bin edges will end up different for different datasets, you can't divide them by each other

  array: provide your own bin edges. Can be of any size, and end at any value. Bins exceeding the size of the largest region will be simply ignored.
- **smooth** (*callable*) – A smoothing function to be applied to log(P(s)) and log(x) before calculating P(s) slopes for by-region data
- **min_nvalid** (*int*) – For each region, throw out bins (log-spaced) that have less than min_nvalid valid pixels This will ensure that each entree in Pc_by_region has at least n_valid valid pixels Don't set it to zero, or it will introduce bugs. Setting it to 1 is OK, but not recommended.
- **min_count** (*int*) – If counts are found in the data, then for each region, throw out bins (log-spaced) that have more than min_counts of counts.sum (raw Hi-C counts). This will ensure that each entree in Pc_by_region has at least min_count raw Hi-C reads

**Returns**

- **Pc** (*DataFrame*) – dataframe of contact probabilities and spread across regions
- **slope** (*ndarray*) – slope of Pc(s) on a log-log plot and spread across regions
- **bins** (*ndarray*) – an array of bin edges used for calculating P(s)

### Notes

For main Pc and slope, the algorithm is the following

1. concatenate all the expected for all regions into a large dataframe.
2. create logarithmically-spaced bins of diagonals (or use provided)
3. pool together n_valid and balanced.sum for each region and for each bin
4. calculate the average diagonal for each bucket, weighted by n_valid
5. divide balanced.sum by n_valid after summing for each bucket (not before)
6. calculate the slope in log space (for each region)

### X values are not midpoints of bins

In step 4, we calculate the average diag index weighted by n_valid. This seems counter-intuitive, but it actually is justified.

Let's take the worst case scenario. Let there be a bin from 40MB to 44MB. Let there be a region that is exactly 41 MB long. The midpoint of the bin is at 42MB. But the only part of this region belonging to this bin is actually between 40MB and 41MB. Moreover, the "average" read in this little triangle of the heatmap is actually not coming even from 40.5 MB because the triangle is getting narrower towards 41MB. The center of mass of a triangle is 1/3 of the way up, or 40.33 MB. So an average read for this region in this bin is coming from 40.33.

Consider the previous bin, say, from 36MB to 40MB. The heatmap there is a trapezoid with a long side of 5MB, the short side of 1MB, and height of 4MB. The center of mass of this trapezoid is at 36 + 14/9 = 37.55MB, and not at 38MB. So the last bin center is definitely mis-assigned, and the second-to-last bin center is off by some 25%. This would lead to a 25% error of the P(s) slope estimated between the third-to-last and second-to-last bin.

In presence of missing bins, this all becomes more complex, but this kind of averaging should take care of everything. It follows a general principle: when averaging the y values with some weights, one needs to average the x values with the same weights. The y values here are being added together, so per-diag means are effectively averaged with the weight of n_valid. Therefore, the x values (diag) should be averaged with the same weights.

### Other considerations

Steps #3 and #5 are important because the ratio of sums does not equal to the sum of ratios, and the former is more correct (the latter is more susceptible to noise). It is generally better to divide at the very end, rather than dividing things for each diagonal.

Here we divide at the end twice: first we divide balanced.sum by n_valid for each region, then we effectively multiply it back up and divide it for each bin when combining different regions (see weighted average in the next function).

### Smoothing P(s) for the slope

For calcuating the slope, we apply smoothing to the P(s) to ensure the slope is not too noisy. There are several caveats here: the P(s) has to be smoothed in logspace, and both P and s have to be smoothed. It is discussed in detail here

https://gist.github.com/mimakaev/4becf1310ba6ee07f6b91e511c531e73

### Examples

For example, see this gist: https://gist.github.com/mimakaev/e9117a7fcc318e7904702eba5b47d9e6

`cooltools.api.expected.`**`make_block_table`**`(`*clr*, *regions1*, *regions2*, *clr_weight_name='weight'*`)`

Creates a table of total and valid pixels for a set of rectangular genomic blocks defined by regions1 and regions2. For every block calculate its “area” in pixels (“n_total”), and calculate number of “valid” pixels (“n_valid”). Valid pixels exclude “bad” pixels, which are inferred from the balancing weight column *clr_weight_name*.

When *clr_weight_name* is None, raw data is used, and no “bad” pixels are exclued.

**Parameters**

- **clr** (*cooler.Cooler*) – Input cooler
- **regions1** (*viewframe-like dataframe*) – viewframe-like dataframe, where repeated entries are allowed
- **regions2** (*viewframe-like dataframe*) – viewframe-like dataframe, where repeated entries are allowed
- **clr_weight_name** (*str*) – name of the weight column in the cooler bins-table, used for masking bad pixels. When clr_weight_name is None, no bad pixels are masked.

**Returns**

**block_table** (*dict*) – dictionary for blocks that are 0-indexed

`cooltools.api.expected.`**`make_diag_table`**`(`*bad_mask*, *span1*, *span2*`)`

Compute the total number of elements `n_total` and the number of bad elements `n_bad` per diagonal for a single contact area encompassing `span1` and `span2` on the same genomic scaffold (cis matrix).

Follows the same principle as the algorithm for finding contact areas for computing scalings.

**Parameters**

- **bad_mask** (*1D array of bool*) – Mask of bad bins for the whole genomic scaffold containing the regions of interest.
- **span1** (*pair of ints*) – The bin spans (not genomic coordinates) of the two regions of interest.

- **span2** (*pair of ints*) – The bin spans (not genomic coordinates) of the two regions of interest.

**Returns**
**diags** (*pandas.DataFrame*) – Table indexed by 'diag' with columns ['n_total', 'n_bad'].

cooltools.api.expected.**make_diag_tables**(*clr*, *regions*, *regions2=None*, *clr_weight_name='weight'*)

For every region infer diagonals that intersect this region and calculate the size of these intersections in pixels, both "total" and "n_valid", where "n_valid" does not count "bad" pixels.

"Bad" pixels are inferred from the balancing weight column *clr_weight_name*. When *clr_weight_name* is None, raw data is used, and no "bad" pixels are exclued.

When *regions2* are provided, all intersecting diagonals are reported for each rectangular and asymmetric block defined by combinations of matching elements of *regions* and *regions2*. Otherwise only *regions*-based symmetric square blocks are considered. Only intra-chromosomal regions are supported.

**Parameters**

- **clr** (*cooler.Cooler*) – Input cooler
- **regions** (*viewframe or viewframe-like dataframe*) – viewframe without repeated entries or viewframe-like dataframe with repeated entries
- **regions2** (*viewframe or viewframe-like dataframe*) – viewframe without repeated entries or viewframe-like dataframe with repeated entries
- **clr_weight_name** (*str*) – name of the weight column in the clr bin-table, Balancing weight is used to infer bad bins, set to *None* is masking bad bins is not desired for raw data.

**Returns**
**diag_tables** (*dict*) – dictionary with DataFrames of relevant diagonals for every region.

cooltools.api.expected.**per_region_smooth_cvd**(*cvd*, *sigma_log10=0.1*, *window_sigma=5*, *points_per_sigma=10*, *cols=None*, *suffix=''*)

Smooth the contact-vs-distance curve for each region in log-space.

**Parameters**

- **cvd** (*pandas.DataFrame*) – A dataframe with the expected values in the cooltools.expected format.
- **sigma_log10** (*float, optional*) – The standard deviation of the smoothing Gaussian kernel, applied over log10(diagonal), by default 0.1
- **window_sigma** (*int, optional*) – Width of the smoothing window, expressed in sigmas, by default 5
- **points_per_sigma** (*int, optional*) – If provided, smoothing is done only for *points_per_sigma* points per sigma and the rest of the values are interpolated (this results in a major speed-up). By default 10
- **cols** (*dict, optional*) – If provided, use the specified column names instead of the standard ones. See DEFAULT_CVD_COLS variable for the format of this argument.
- **suffix** (*string, optional*) – If provided, use the specified string as the suffix of the output column's name

**Returns**
**cvd** (*pandas.DataFrame*) – A cvd table with extra column for the log-smoothed contact frequencies (by default, "balanced.avg.smoothed" if balanced, or "count.avg.smoothed" if raw).

### Notes

Parameters in "cols" will be used:

**region1:**
Name of the column that stores region1's locations (by default, "region1").

**region2:**
Name of the column that stores region2's locations (by default, "region2").

**dist:**
Name of the column that stores distance values (by default, "dist").

**n_pixels:**
Name of the column that stores number of pixels (by default, "n_valid" if balanced, or "n_total" if raw).

**n_contacts:**
Name of the column that stores the sum of contacts (by default, "balanced.sum" if balanced, or "count.sum" if raw).

**output_prefix:**
Name prefix of the column that will store output value (by default, "balanced.avg" if balanced, or "count.avg" if raw).

## cooltools.api.insulation module

`cooltools.api.insulation.`**`calculate_insulation_score`**`(`*clr*, *window_bp*, *view_df=None*, *ignore_diags=None*, *min_dist_bad_bin=0*, *is_bad_bin_key='is_bad_bin'*, *append_raw_scores=False*, *chunksize=20000000*, *clr_weight_name='weight'*, *verbose=False*, *nproc=1*, *map_functor=<class 'map'>*`)`

Calculate the diamond insulation scores for all bins in a cooler.

**Parameters**

- **clr** (*cooler.Cooler*) – A cooler with balanced Hi-C data.
- **window_bp** (*int or list of integers*) – The size of the sliding diamond window used to calculate the insulation score. If a list is provided, then a insulation score if calculated for each value of window_bp.
- **view_df** (*bioframe.viewframe or None*) – Viewframe for independent calculation of insulation scores for regions
- **ignore_diags** (*int | None*) – The number of diagonals to ignore. If None, equals the number of diagonals ignored during IC balancing.
- **min_dist_bad_bin** (*int*) – The minimal allowed distance to a bad bin to report insulation score. Fills bins that have a bad bin closer than this distance by nans.
- **is_bad_bin_key** (*str*) – Name of the output column to store bad bins
- **append_raw_scores** (*bool*) – If True, append columns with raw scores (sum_counts, sum_balanced, n_pixels) to the output table.
- **clr_weight_name** (*str or None*) – Name of the column in the bin table with weight. Using unbalanced data with *None* will avoid masking "bad" pixels.
- **verbose** (*bool*) – If True, report real-time progress.
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

**ins_table** (*pandas.DataFrame*) – A table containing the insulation scores of the genomic bins cooltools.api.insulation.**find_boundaries**(*ins_table*, *min_frac_valid_pixels=0.66*, *min_dist_bad_bin=0*, *log2_ins_key='log2_insulation_score_{WINDOW}'*, *n_valid_pixels_key='n_valid_pixels_{WINDOW}'*, *is_bad_bin_key='is_bad_bin'*)

Call insulating boundaries.

Find all local minima of the log2(insulation score) and calculate their chromosome-wide topographic prominence.

**Parameters**

- **ins_table** (*pandas.DataFrame*) – A bin table with columns containing log2(insulation score), annotation of regions (required), the number of valid pixels per diamond and (optionally) the mask of bad bins. Normally, this should be an output of calculate_insulation_score.
- **view_df** (*bioframe.viewframe or None*) – Viewframe for independent boundary calls for regions
- **min_frac_valid_pixels** (*float*) – The minimal fraction of valid pixels in a diamond to be used in boundary picking and prominence calculation.
- **min_dist_bad_bin** (*int*) – The minimal allowed distance to a bad bin to be used in boundary picking. Ignore bins that have a bad bin closer than this distance.
- **log2_ins_key** (*str*) – The names of the columns containing log2_insulation_score and the number of valid pixels per diamond. When a template containing *{WINDOW}* is provided, the calculation is repeated for all pairs of columns matching the template.
- **n_valid_pixels_key** (*str*) – The names of the columns containing log2_insulation_score and the number of valid pixels per diamond. When a template containing *{WINDOW}* is provided, the calculation is repeated for all pairs of columns matching the template.

**Returns**

**ins_table** (*pandas.DataFrame*) – A bin table with appended columns with boundary prominences.

cooltools.api.insulation.**get_n_pixels**(*bad_bin_mask*, *window=10*, *ignore_diags=2*)

Calculate the number of “good” pixels in a diamond at each bin.

cooltools.api.insulation.**insul_diamond**(*pixel_query*, *bins*, *window=10*, *ignore_diags=2*, *norm_by_median=True*, *clr_weight_name='weight'*)

Calculates the insulation score of a Hi-C interaction matrix.

**Parameters**

- **pixel_query** (*RangeQuery object <TODO:update description>*) – A table of Hi-C interactions. Must follow the Cooler columnar format: bin1_id, bin2_id, count, balanced (optional)).
- **bins** (*pandas.DataFrame*) – A table of bins, is used to determine the span of the matrix and the locations of bad bins.
- **window** (*int*) – The width (in bins) of the diamond window to calculate the insulation score.
- **ignore_diags** (*int*) – If > 0, the interactions at separations < *ignore_diags* are ignored when calculating the insulation score. Typically, a few first diagonals of the Hi-C map should be ignored due to contamination with Hi-C artifacts.
- **norm_by_median** (*bool*) – If True, normalize the insulation score by its NaN-median.
- **clr_weight_name** (*str or None*) – Name of balancing weight column from the cooler to use. Using raw unbalanced data is not supported for insulation.

cooltools.api.insulation.**insulation**(*clr*, *window_bp*, *view_df=None*, *ignore_diags=None*, *clr_weight_name='weight'*, *min_frac_valid_pixels=0.66*, *min_dist_bad_bin=0*, *threshold='Li'*, *append_raw_scores=False*, *chunksize=20000000*, *verbose=False*, *nproc=1*)

Find insulating boundaries in a contact map via the diamond insulation score.

For a given cooler, this function (a) calculates the diamond insulation score track, (b) detects all insulating boundaries, and (c) removes weak boundaries via an automated thresholding algorithm.

**Parameters**

- **clr** (*cooler.Cooler*) – A cooler with balanced Hi-C data.
- **window_bp** (*int or list of integers*) – The size of the sliding diamond window used to calculate the insulation score. If a list is provided, then a insulation score if done for each value of window_bp.
- **view_df** (*bioframe.viewframe or None*) – Viewframe for independent calculation of insulation scores for regions
- **ignore_diags** (*int | None*) – The number of diagonals to ignore. If None, equals the number of diagonals ignored during IC balancing.
- **clr_weight_name** (*str*) – Name of the column in the bin table with weight
- **min_frac_valid_pixels** (*float*) – The minimal fraction of valid pixels in a diamond to be used in boundary picking and prominence calculation.
- **min_dist_bad_bin** (*int*) – The minimal allowed distance to a bad bin to report insulation score. Fills bins that have a bad bin closer than this distance by nans.
- **threshold** (*"Li", "Otsu" or float*) – Rule used to threshold the histogram of boundary strengths to exclude weak boundaries. “Li” or “Otsu” use corresponding methods from skimage.thresholding. Providing a float value will filter by a fixed threshold
- **append_raw_scores** (*bool*) – If True, append columns with raw scores (sum_counts, sum_balanced, n_pixels) to the output table.
- **verbose** (*bool*) – If True, report real-time progress.
- **nproc** (*int, optional*) – How many processes to use for calculation

**Returns**

**ins_table** (*pandas.DataFrame*) – A table containing the insulation scores of the genomic bins

## cooltools.api.saddle module

cooltools.api.saddle.**digitize**(*track*, *n_bins*, *vrange=None*, *qrange=None*, *digitized_suffix='.d'*)

Digitize genomic signal tracks into integers between *1* and *n*.

**Parameters**

- **track** (*4-column DataFrame*) – bedGraph-like dataframe with columns understood as (chrom,start,end,value).
- **n_bins** (*int*) – number of bins for signal quantization.
- **vrange** (*tuple*) – Low and high values used for binning track values. E.g. if `vrange`=(-0.05, 0.05), equal width bins would be generated between the value -0.05 and 0.05.
- **qrange** (*tuple*) – Low and high values for quantile binning track values. E.g., if `qrange`=(0.02, 0.98) the lower bin would start at the 2nd percentile and the upper bin would end at the 98th percentile of the track value range. Low must be 0.0 or more, high must be 1.0 or less.
- **digitized_suffix** (*str*) – suffix to append to the track value name in the fourth column.

**Returns**

- **digitized** (*DataFrame*) – New track dataframe (bedGraph-like) with digitized value column with name suffixed by ‘.d’ The digized column is returned as a categorical.
- **binedges** (*1D array (length n + 1)*) – Bin edges used in quantization of track. For *n* bins, there are *n + 1* edges. See encoding details in Notes.

### Notes

The digital encoding is as follows:

- *1..n* <-> values assigned to bins defined by vrange or qrange
- *0* <-> left outlier values
- *n+1* <-> right outlier values
- *-1* <-> missing data (NaNs)

`cooltools.api.saddle.saddle`(*clr*, *expected*, *track*, *contact_type*, *n_bins*, *vrange=None*, *qrange=None*, *view_df=None*, *clr_weight_name='weight'*, *expected_value_col='balanced.avg'*, *view_name_col='name'*, *min_diag=3*, *max_diag=-1*, *trim_outliers=False*, *verbose=False*, *drop_track_na=False*)

Get a matrix of average interactions between genomic bin pairs as a function of a specified genomic track.

The provided genomic track is either: (a) digitized inside this function by passing 'n_bins', and one of 'v_range' or 'q_range' (b) passed as a pre-digitized track with a categorical value column as generated by *get_digitized()*.

**Parameters**

- **clr** (*cooler.Cooler*) – Observed matrix.
- **expected** (*DataFrame in expected format*) – Diagonal summary statistics for each chromosome, and name of the column with the values of expected to use.
- **contact_type** (*str*) – If 'cis' then only cis interactions are used to build the matrix. If 'trans', only trans interactions are used.
- **track** (*DataFrame*) – A track, i.e. BedGraph-like dataframe, which is digitized with the options n_bins, vrange and qrange. Can optionally be passed as a pre-digitized dataFrame with a categorical value column, as generated by get_digitzied(), also passing n_bins as None.
- **n_bins** (*int or None*) – number of bins for signal quantization. If None, then track must be passed as a pre-digitized track.
- **vrange** (*tuple*) – Low and high values used for binning track values. See get_digitized().
- **qrange** (*tuple*) – Low and high values for quantile binning track values. Low must be 0.0 or more, high must be 1.0 or less. Only one of vrange or qrange can be passed. See get_digitzed().
- **view_df** (*viewframe*) – Viewframe with genomic regions. If none, generate from track chromosomes.
- **clr_weight_name** (*str*) – Name of the column in the clr.bins to use as balancing weights. Using raw unbalanced data is not supported for saddles.
- **expected_value_col** (*str*) – Name of the column in expected used for normalizing.
- **view_name_col** (*str*) – Name of column in view_df with region names.
- **min_diag** (*int*) – Smallest diagonal to include in computation. Ignored with contact_type=trans.
- **max_diag** (*int*) – Biggest diagonal to include in computation. Ignored with contact_type=trans.
- **trim_outliers** (*bool, optional*) – Remove first and last row and column from the output matrix.
- **verbose** (*bool, optional*) – If True then reports progress.
- **drop_track_na** (*bool, optional*) – If True then drops NaNs in input track (as if they were missing), If False then counts NaNs as present in dataframe. In general, this only adds check form chromosomes that have all missing values, but does not affect the results.

**Returns**

- **interaction_sum** (*2D array*) – The matrix of summed interaction probability between two genomic bins given their values of the provided genomic track.
- **interaction_count** (*2D array*) – The matrix of the number of genomic bin pairs that contributed to the corresponding pixel of `interaction_sum`.

`cooltools.api.saddle.`**`saddle_strength`**`(`*S*, *C*`)`

**Parameters**

- **S** (*2D arrays, square, same shape*) – Saddle sums and counts, respectively
- **C** (*2D arrays, square, same shape*) – Saddle sums and counts, respectively

**Returns**

- *1D array*
- *Ratios of cumulative corner interaction scores, where the saddle data is*
- *grouped over the AA+BB corners and AB+BA corners with increasing extent.*

`cooltools.api.saddle.`**`saddleplot`**`(`*track*, *saddledata*, *n_bins*, *vrange=None*, *qrange=(0.0, 1.0)*, *cmap='coolwarm'*, *scale='log'*, *vmin=0.5*, *vmax=2*, *color=None*, *title=None*, *xlabel=None*, *ylabel=None*, *clabel=None*, *fig=None*, *fig_kws=None*, *heatmap_kws=None*, *margin_kws=None*, *cbar_kws=None*, *subplot_spec=None*`)`

Generate a saddle plot.

**Parameters**

- **track** (*pd.DataFrame*) – See get_digitized() for details.
- **saddledata** (*2D array-like*) – Saddle matrix produced by *make_saddle*. It will include 2 flanking rows/columns for outlier signal values, thus the shape should be *(n+2, n+2)*.
- **cmap** (*str or matplotlib colormap*) – Colormap to use for plotting the saddle heatmap
- **scale** (*str*) – Color scaling to use for plotting the saddle heatmap: log or linear
- **vmin** (*float*) – Value limits for coloring the saddle heatmap
- **vmax** (*float*) – Value limits for coloring the saddle heatmap
- **color** (*matplotlib color value*) – Face color for margin bar plots
- **fig** (*matplotlib Figure, optional*) – Specified figure to plot on. A new figure is created if none is provided.
- **fig_kws** (*dict, optional*) – Passed on to *plt.Figure()*
- **heatmap_kws** (*dict, optional*) – Passed on to *ax.imshow()*
- **margin_kws** (*dict, optional*) – Passed on to *ax.bar()* and *ax.barh()*
- **cbar_kws** (*dict, optional*) – Passed on to *plt.colorbar()*
- **subplot_spec** (*GridSpec object*) – Specify a subregion of a figure to using a GridSpec.

**Returns**

*Dictionary of axes objects.*

### cooltools.api.sample module

`cooltools.api.sample.`**`sample`**`(`*clr*, *out_clr_path*, *count=None*, *cis_count=None*, *frac=None*, *exact=False*, *chunksize=10000000*, *nproc=1*, *map_functor=<class 'map'>*`)`

Pick a random subset of contacts from a Hi-C map.

**Parameters**

- **clr** (*cooler.Cooler or str*) – A Cooler or a path/URI to a Cooler with input data.
- **out_clr_path** (*str*) – A path/URI to the output.
- **count** (*int*) – The target number of contacts in the sample. Mutually exclusive with *cis_count* and *frac*.
- **cis_count** (*int*) – The target number of cis contacts in the sample. Mutually exclusive with *count* and *frac*.
- **frac** (*float*) – The target sample size as a fraction of contacts in the original dataset. Mutually exclusive with *count* and *cis_count*.
- **exact** (*bool*) – If True, the resulting sample size will exactly match the target value. Exact sampling will load the whole pixel table into memory! If False, binomial sam-

pling will be used instead and the sample size will be randomly distributed around the target value.
- **chunksize** (*int*) – The number of pixels loaded and processed per step of computation.
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

cooltools.api.sample.**sample_pixels_approx**(*pixels*, *frac*)

cooltools.api.sample.**sample_pixels_exact**(*pixels*, *count*)

### cooltools.api.snipping module

Collection of classes and functions used for snipping and creation of pileups (averaging of multiple small 2D regions) The main user-facing function of this module is *pileup*, it performs pileups using snippers and other functions defined in the module. The concept is the following:

- First, the provided features are annotated with the regions from a view (or simply whole chromosomes, if no view is provided). They are assigned to the region that contains it, or the one with the largest overlap.
- Then the features are expanded using the *flank* argument, and aligned to the bins of the cooler
- Depending on the requested operation (whether the normalization to expected is required), the appropriate snipper object is created
- A snipper can *select* a particular region of a genome-wide matrix, meaning it stores its sparse representation in memory. This could be whole chromosomes or chromosome arms, for example
- A snipper can *snip* a small area of a selected region, meaning it will extract and return a dense representation of this area
- For each region present, it is first `select`ed, and then all features within it are `snip`ped, creating a stack: a 3D array containing all snippets for this region
- For features that are not assigned to any region, an empty snippet is returned
- All per-region stacks are then combined into one, which then can be averaged to create a single pileup
- The order of snippets in the stack matches the order of features, this way the stack can also be used for analysis of any subsets of original features

This procedure achieves a good tradeoff between speed and RAM. Extracting each individual snippet directly from disk would be extremely slow due to slow IO. Extracting the whole chromosomes into dense matrices is not an option due to huge memory requirements. As a warning, deeply sequenced data can still require a substantial amount of RAM at high resolution even as a sparse matrix, but typically it's not a problem.

class cooltools.api.snipping.**CoolerSnipper**(*clr*, *cooler_opts=None*, *view_df=None*, *min_diag=2*)

Bases: object

**select**(*region1*, *region2*)

Select a portion of the cooler for snipping based on two regions in the view

In addition to returning the selected portion of the data, stores necessary information about it in the snipper object for future snipping

**Parameters**
- **region1** (*str*) – Name of a region from the view
- **region2** (*str*) – Name of another region from the view.

**Returns**

*CSR matrix* – Sparse matrix of the selected portion of the data from the cooler

`snip`(*matrix*, *region1*, *region2*, *tup*)

Extract a snippet from the matrix

Returns a NaN-filled array for out-of-bounds regions. Fills in NaNs based on the cooler weight, if using balanced data. Fills NaNs in all diagonals below min_diag

**Parameters**

- **matrix** (*SCR matrix*) – Output of the .select() method
- **region1** (*str*) – Name of a region from the view corresponding to the matrix
- **region2** (*str*) – Name of the other regions from the view corresponding to the matrix
- **tup** (*tuple*) – (start1, end1, start2, end2) coordinates of the requested snippet in bp

**Returns**

*np.array* – Requested snippet.

`class cooltools.api.snipping.ExpectedSnipper`(*clr*, *expected*, *view_df=None*, *min_diag=2*, *expected_value_col='balanced.avg'*)

Bases: `object`

`select`(*region1*, *region2*)

Select a portion of the expected matrix for snipping based on two regions in the view

In addition to returning the selected portion of the data, stores necessary information about it in the snipper object for future snipping

**Parameters**

- **region1** (*str*) – Name of a region from the view
- **region2** (*str*) – Name of another region from the view.

**Returns**

*CSR matrix* – Sparse matrix of the selected portion of the data from the cooler

`snip`(*exp*, *region1*, *region2*, *tup*)

Extract an expected snippet

Returns a NaN-filled array for out-of-bounds regions. Fills NaNs in all diagonals below min_diag

**Parameters**

- **exp** (*SCR matrix*) – Output of the .select() method
- **region1** (*str*) – Name of a region from the view corresponding to the matrix
- **region2** (*str*) – Name of the other regions from the view corresponding to the matrix
- **tup** (*tuple*) – (start1, end1, start2, end2) coordinates of the requested snippet in bp

**Returns**

*np.array* – Requested snippet.

`class cooltools.api.snipping.ObsExpSnipper`(*clr*, *expected*, *cooler_opts=None*, *view_df=None*, *min_diag=2*, *expected_value_col='balanced.avg'*)

Bases: `object`

`select`(*region1*, *region2*)

Select a portion of the cooler for snipping based on two regions in the view

In addition to returning the selected portion of the data, stores necessary information about it in the snipper object for future snipping

**Parameters**

- **region1** (*str*) – Name of a region from the view
- **region2** (*str*) – Name of another region from the view.

**Returns**

*CSR matrix* – Sparse matrix of the selected portion of the data from the cooler

**snip**(*matrix*, *region1*, *region2*, *tup*)

Extract an expected-normalised snippet from the matrix

Returns a NaN-filled array for out-of-bounds regions. Fills in NaNs based on the cooler weight, if using balanced data. Fills NaNs in all diagonals below min_diag

**Parameters**

- **matrix** (*SCR matrix*) – Output of the .select() method
- **region1** (*str*) – Name of a region from the view corresponding to the matrix
- **region2** (*str*) – Name of the other regions from the view corresponding to the matrix
- **tup** (*tuple*) – (start1, end1, start2, end2) coordinates of the requested snippet in bp

**Returns**

*np.array* – Requested snippet.

cooltools.api.snipping.**expand_align_features**(*features_df*, *flank*, *resolution*, *format='bed'*)

Short summary.

**Parameters**

- **features_df** (*pd.DataFrame*) – Dataframe with feature coordinates.
- **flank** (*int*) – Flank size to add to the central bin of each feature.
- **resolution** (*int*) – Size of the bins to use.
- **format** (*str*) – “bed” or “bedpe” format: has to have ‘chrom’, ‘start’, ‘end’ or ‘chrom1’, ‘start1’, ‘end1’, ‘chrom2’, ‘start2’, ‘end1’ columns, repectively.

**Returns**

*pd.DataFrame* –

**DataFrame with features with new columns**

”center”, “orig_start” “orig_end”

**or “center1”, “orig_start1”, “orig_end1”,**

”center2”, “orig_start2”, “orig_rank_end2”, depending on format.

cooltools.api.snipping.**make_bin_aligned_windows**(*binsize*, *chroms*, *centers_bp*, *flank_bp=0*, *region_start_bp=0*, *ignore_index=False*)

Convert genomic loci into bin spans on a fixed bin-segmentation of a genomic region. Window limits are adjusted to align with bin edges.

**Parameters**

- **binsize** (*int*) – Bin size (resolution) in base pairs.
- **chroms** (*1D array-like*) – Column of chromosome names.
- **centers_bp** (*1D or nx2 array-like*) – If 1D, center points of each window. If 2D, the starts and ends.
- **flank_bp** (*int*) – Distance in base pairs to extend windows on either side.
- **region_start_bp** (*int, optional*) – If region is a subset of a chromosome, shift coordinates by this amount. Default is 0.

**Returns**

*DataFrame with columns* – ‘chrom’ - chromosome ‘start’, ‘end’ - window limits in base pairs ‘lo’, ‘hi’ - window limits in bins

cooltools.api.snipping.**pileup**(*clr*, *features_df*, *view_df=None*, *expected_df=None*, *expected_value_col='balanced.avg'*, *flank=100000*, *min_diag='auto'*, *clr_weight_name='weight'*, *nproc=1*, *map_functor=<class 'map'>*)

Pileup features over the cooler.

**Parameters**

- **clr** (*cooler.Cooler*) – Cooler with Hi-C data
- **features_df** (*pd.DataFrame*) – Dataframe in bed or bedpe format: has to have 'chrom', 'start', 'end' or 'chrom1', 'start1', 'end1', 'chrom2', 'start2', 'end2' columns.
- **view_df** (*pd.DataFrame*) – Dataframe with the genomic view for this operation (has to match the expected_df, if provided)
- **expected_df** (*pd.DataFrame*) – Dataframe with the expected level of interactions at different genomic separations
- **expected_value_col** (*str*) – Name of the column in expected used for normalizing.
- **flank** (*int*) – How much to flank the center of the features by, in bp
- **min_diag** (*str or int*) – All diagonals of the matrix below this value are ignored. 'auto' tries to extract the value used during the matrix balancing, if it fails defaults to 2
- **clr_weight_name** (*str*) – Value of the column that contains the balancing weights
- **force** (*bool*) – Allows start>end in the features (not implemented)
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

- **np.ndarray** (*a stackup of all snippets corresponding to the features, with shape*)
- *(n, D, D), where n is the number of snippets and (D, D) is the shape of each*
- *snippet*

### cooltools.api.virtual4c module

cooltools.api.virtual4c.**virtual4c**(*clr*, *viewpoint*, *clr_weight_name='weight'*, *nproc=1*, *map_functor=<class 'map'>*)

Generate genome-wide contact profile for a given viewpoint.

Extract all contacts of a given viewpoint from a cooler file.

**Parameters**

- **clr** (*cooler.Cooler*) – A cooler with balanced Hi-C data.
- **viewpoint** (*tuple or str*) – Coordinates of the viewpoint.
- **clr_weight_name** (*str*) – Name of the column in the bin table with weight
- **nproc** (*int, optional*) – How many processes to use for calculation. Ignored if map_functor is passed.
- **map_functor** (*callable, optional*) – Map function to dispatch the matrix chunks to workers. If left unspecified, pool_decorator applies the following defaults: if nproc>1 this defaults to multiprocess.Pool; If nproc=1 this defaults the builtin map.

**Returns**

**v4C_table** (*pandas.DataFrame*) – A table containing the interaction frequency of the viewpoint with the rest of the genome

**Note:** Note: this is a new (experimental) function, the interface or output might change in a future version.

## 1.3.10 Release notes

### Upcoming release

### v0.7.0

### New features

- Add pool decorator to functions for supporting multiprocess
- `expected_cis` now accepts unbalanced cool file too

### API changes

- `expected_cis`
  - output cvd table now also includes "dist_bp", "contact_frequency", and "n_valid" columns
  - now returns "count.avg.smoothed" and "count.avg.smoothed.agg", when `clr_weight_name=None, smooth=True, aggregate_smoothed=True`

### Maintenance

- Replaced np.int with int in adaptive_coarsegrain
- OE update in sandbox
- Cross score sandbox fixes
- Support for pandas 2

### v0.6.1

### Maintenance

- Bug fix in CLI pileup

### v0.6.0

### New features

- New function/tool `rearrange_cooler` to reorder/subset/flip regions of the genome in a cooler
- New test dataset for micro-C from hESCs

### API changes

- snipping: reorder the axes of the output snipper array to (snippet_idx, i, j).

### Maintenance

- snipping: fix spurious nan->0 conversion of bad bins in on-diagonal pileups
- snipping: fix snipping without provided view
- snipping: fix for storing the stack in a file
- virtual4c: fix for the case when viewpoint has no contacts
- fix: Fix numba deprecation warnings by adding `nopython=True`
- Other small bugfixes

## v0.5.4

### Maintenance

- Updated import statements and requirements to use cooler 0.9.

## v0.5.3

### Maintenance

- Improvements for `read_expected_from_file`
- Bug fix for dot caller 0/0 occurrences
- Remove `cytoolz` dependency
- Pin cooler <0.9 until compatibility

## v0.5.2

### API changes

- remove custom bad_bins from expected & eigdecomp #336
- coverage can store total cis counts in the cooler, and sampling can use cis counts #332
- can now calculate coverge for balanced data #385
- new drop_track_na argument for align_track_with_cooler, allows calcultions that that missing data in tracks as absent #360
- multi-thread insulation by chromosome (TODO: by chunk)
- Virtual 4C tool #378

### CLI changes

- CLI tool for `coverage()`

### Documentation

- snipping documentation
- dots tutorial
- CLI tutorial

### Maintenance

- Dropped support for Python 3.7 (due to Pandas compatability issues)
- Added support for Python 3.10
- Minor bugfixes and compatibility updates
  - Pandas compatibility, pinned above 1.5.1
  - bioframe compatability
  - scikit-learn, pinned above >=1.1.2
  - saddle binedges, value limits #361
  - pileup CLI bugfix for reading features

### Other

- Code of conduct

## v0.5.1

### API changes

- cooltools.dots is the new user-facing function for calling dots

### Maintenance

- Compatibility with pandas 1.4
- Strict dictinary typing for new numba versions
- Update to bioframe 0.3.3

### v0.5.0

**NOTE: THIS RELEASE BREAKS BACKWARDS COMPATIBILITY!**

This release addresses two major issues:

- Integration with bioframe viewframes defined as of bioframe v0.3.
- Synchronization of the CLI and Python API

Additionally, the documentation has been greatly improved and now includes detailed tutorials that show how to use the `cooltools` API in conjunction with other Open2C libraries. These tutorials are automatically re-built from notebooks copied from https://github.com/open2c/open2c_examples repository.

#### API changes

- More clear separation of top-level user-facing functions and low-level API.
  - Most standard analyses can be performed using just the user-facing functions which are imported into the top-level namespace. Some of them are new or heavily modified from earlier versions.
    - `cooltools.expected_cis` and `cooltools.expected_trans` for average by-diagonal contact frequency in intra-chromosomal data and in inter-chromosomal data, respectively
    - `cooltools.eigs_cis` and `cooltools.eigs_trans` for eigenvectors (compartment profiles) of cis and trans data, repectively
    - `cooltools.digitize` and `cooltools.saddle` can be used together for creation of 2D summary tables of Hi-C interactions in relation to a digitized genomic track, such as eigenvectors
    - `cooltools.insulation` for insulation score and annotation of insulating boundaries
    - `cooltools.directionality` for directionality index
    - `cooltools.pileup` for average signal at 1D or 2D genomic features, including APA
    - `cooltools.coverage` for calculation of per-bin sequencing depth
    - `cooltools.sample` for random downsampling of cooler files
    - For non-standard analyses that require custom algorithms, a lower level API is available under `cooltools.api`
- Most functions now take an optional `view_df` argument. A pandas dataframe defining a genomic view (https://bioframe.readthedocs.io/en/latest/guide-technical-notes.html) can be provided to limit the analyses to regions included in the view. If not provided, the analysis is performed on whole chromosomes according to what's stored in the cooler.
- All functions apart from `coverage` now take a `clr_weight_name` argument to specify how the desired balancing weight column is named. Providing a `None` value allows one to use unbalanced data (except the `eigs_cis`, `eigs_trans` methods, since eigendecomposition is only defined for balanced Hi-C data).
- The output of `expected-cis` function has changed: it now contains `region1` and `region2` columns (with identical values in case of within-region expected). Additionally, it now allows smoothing of the result to avoid noisy values at long distances (enabled by default and result saved in additional columns of the dataframe)
- The new `cooltools.insulation` method includes a thresholding step to detect strong boundaries, using either the Li or the Otsu method (from `skimage.thresholding`), or a fixed float value. The result of thresholding for each window size is stored as a boolean in a new column `is_boundary_{window}`.
- New subpackage `sandbox` for experimental codes that are either candidates for merging into cooltools or candidates for removal. No documentation and tests are expected, proceed at your own risk.

- New subpackage `lib` for auxiliary modules

### CLI changes

- CLI tools are renamed with prefixes dropped (e.g. `diamond-insulation` is now `insulation`), to align with names of user-facing API functions.
- The CLI tool for expected has been split in two for intra- and inter-chromosomal data (`expected-cis` and `expected-trans`, repectively).
- Similarly, the compartment profile calculation is now separate for cis and trans (`eigs-cis` and `eigs-trans`).
- New CLI tool `cooltools pileup` for creation of average features based on Hi-C data. It takes a .bed- or .bedpe-style file to create average on-diagonal or off-diagonal pileups, respectively.

### Maintenance

Support for Python 3.6 dropped

### v0.4.0

Date: 2021-04-06

Maintenance

- Make saddle strength work with NaNs
- Add output option to diamond-insulation
- Upgrade bioframe dependency
- Parallelize random sampling
- Various compatibility fixes to expected, saddle and snipping and elsewhere to work with standard formats for "expected" and "regions": https://github.com/open2c/cooltools/issues/217

New features

- New dataset download API
- New functionality for smoothing P(s) and derivatives (API is not yet stable): `logbin_expected`, `interpolate_expected`

### v0.3.2

Date: 2020-05-05

Updates and bug fixes

- Error checking for vmin/vmax in compute-saddle
- Various updates and fixes to expected and dot-caller code

Project health

- Added docs on RTD, tutorial notebooks, code formatting, linting, and contribution guidelines.

### v0.3.0

Date: 2019-11-04

- Several library utilities added: `plotting.gridspec_inches`, `adaptive_coarsegrain`, singleton interpolation, and colormaps.
- New tools: `cooltools sample` for random downsampling, `cooltools coverage` for marginalization.

Improvements to saddle functions:

- `compute-saddle` now saves saddledata without transformation, and the `scale` argument (with options `log` or `linear`) now only determines how the saddle is plotted. Consequently, `saddleplot` function now expects untransformed `saddledata`, and plots it directly or with log-scaling of the colormap. (https://github.com/open2c/cooltools/pull/105)
- Added `saddle.mask_bad_bins` method to filter bins in a track based on Hi-C bin-level filtering - improves saddle and histograms when using ChIP-seq and similar tracks. It is automatically applied in the CLI interface. Shouldn't affect the results when using eigenvectors calculated from the same data.
- `make_saddle` Python function and `compute-saddle` CLI now allow setting min and max distance to use for calculating saddles.

### v0.2.0

Date: 2019-05-02

- New tagged release for DCIC. Many updates, including more memory-efficient insulation score calling. Next release should include docs.

### v0.1.0

Date: 2018-05-07

- First official release

# PYTHON MODULE INDEX

# INDEX