StripePy: fast and robust characterization of architectural stripes

Andrea Raffo[1], Roberto Rossini[1], and Jonas Paulsen[1,2]

[1]Department of Biosciences, University of Oslo, 0316, Oslo, Norway
[2]Centre for Bioinformatics, Department of Informatics, 0316, Oslo, Norway

Abstract

Architectural stripes in Hi-C and related data are crucial for gene regulation, development, and DNA repair. Despite their importance, few tools exist for automatic stripe detection. We introduce StripePy, which leverages computational geometry methods to identify and analyze architectural stripes in contact maps from Chromosome Conformation Capture experiments like Hi-C and Micro-C. StripePy outperforms existing tools, as shown through tests on various datasets and a newly developed simulated benchmark, StripeBench, providing a valuable resource for the community.

Eukaryotic genomes are hierarchically folded inside the nucleus into a 3D conformation crucial for gene regulation, cell division, and DNA repair [1–5]. At the scale of the nucleus, individual chromosomes form distinct territories [6, 7], further organized into A (euchromatin) and B (heterochromatin) compartments [8]. At the lower levels of the hierarchy, topologically associated domains (TADs) arise from enriched spatial contacts within domains, mediated by CTCF binding at their boundaries [9, 10].

Chromosome conformation capture sequencing methods like Hi-C and Micro-C have been instrumental for revealing these structural hierarchies across the genome. However, deciphering the resulting patterns would be impossible without computational tools designed to detect and analyze them. The significance of these computational methods is underscored by the numerous tools and techniques developed and made available to the research community for analyzing the 3D genome structure at all levels of the genome hierarchy [11].

In contrast to other genome architectural features, few algorithms are dedicated to the automatic recognition of "stripes", which in Hi-C matrices visually consist of vertical (or horizontal) narrow rectangles rooted at the main matrix diagonal. From a biological standpoint, stripes are thought to arise from asymmetric cohesin-mediated loop extrusion [12, 13]. This occurs when loop-extruding cohesin is unidirectionally blocked by a CTCF protein bound to its binding site with the N-terminal end pointing toward it, while loop extrusion can proceed unobstructed on the other side, resulting in a "stripe" of enriched Hi-C contacts emanating from CTCF bound sites in the genome. Because CTCF binding is enriched at TAD borders, stripes are often found at their edges [13]; this is however not always the case, as it was recently noted that stripes can also appear without a TAD being clearly observed [14].

Stripes are increasingly recognized as important features for epigenetic regulation of transcription and enhancer activity [15, 16], for development [17], and DNA repair [18], highlighting our need to better detect and analyze them. Further, numerous aspects behind loop extrusion are still far from being fully understood, including factors governing the loading of cohesin onto chromatin and the existence of specific targeted loading sites in eukaryotes [19].

Existing stripe detection tools are all rooted in the fields of image processing and analysis. The first method, Zebra [20], identifies pixel tracks of higher interaction frequency at genomic domain boundaries, requiring manual processing to confirm stripe candidates.Zebra's original implementation is not publicly available, but a re-implementation was made available by an independent group on GitHub under the name of StripeCaller[1]; StripeCaller constrains the stripe width to a single bin and does not assume that stripes are anchored to the main diagonal of the matrix. Another tool published in late 2018 is domainClassifyR[2]. The tool detects architectural stripes based on TAD boundaries by defining a stripe score that is based

[1]StripeCaller: https://github.com/XiaoTaoWang/StripeCaller
[2]domainClassifyR: https://github.com/ChristopherBarrington

on the $Z-$statistic; intra-TAD segments remain however undetected. Chromosight [21] works by exploiting template-based pattern recognition, here applied by convolving a set of templates over the contact map. A set of criteria is then applied to filter candidate stripes to deal with, e.g., candidate stripes overlapping with too many empty pixels or that are too close to another detected pattern. Chromosight identifies stripes as a single point, without estimating stripe width and height. Stripenn [22] pre-processes the input Hi-C matrix by performing contrast adjustment followed by noise reduction via the Canny edge detection algorithm. A set of custom criteria is introduced to detect and possibly merge vertical lines. Lastly, the tool computes two coefficients – median p-value and stripiness – to quantitatively evaluate architectural stripes. Of the four methods available at present, Stripenn is the only one capable of estimating both the width and the height of a stripe, effectively transitioning stripes from a 0- or 1-dimensional pattern to a fully realized 2-dimensional representation.

We present StripePy, a stripe recognition method based on concepts rooted in pattern recognition, topological data analysis [23] and simple geometric reasoning. Implemented in Python, StripePy is uniquely capable of reading interactions in both Cooler (.cool and .mcool) and Juicer (.hic) data formats. Beyond the detection of stripes, including their height and width estimates, StripePy provides a collection of descriptors that can be used for subsequent postprocessing and in-depth analyses. One of the computed descriptors is the *relative change*, which is used to discern candidate stripes that exhibit minimal variation when comparing the average signal within stripes with the average signal in their neighborhood. StripePy thus offers an efficient and user-friendly method to detect architectural stripes, while at the same time computing several descriptors that can be used to inform further downstream analyses.

To assess StripePy's performance and compare with existing tools, we developed StripeBench, a novel benchmark consisting of 64 Hi-C contact maps simulated with the computational tool MoDLE [24] at different resolutions, contact densities, and noise levels. These maps come with ground truth annotations. Furthermore, StripeBench defines a set of measures to quantify the performance and compare computational tools in the classification of genomic bins and recognition of stripes. StripeBench is used to compare StripePy against StripeCaller, Chromosight, and Stripenn and assess how these methods behave at increasing levels of resolution, contact density, and noise level.

Finally, the analysis of real contact maps from two cells lines is presented, and the predicted stripes are compared against CTCF chromatin immunoprecipitation sequencing (ChIP-Seq) peaks. These analyses demonstrate that StripePy can increase the number of correctly predicted stripes and detected anchor sites while maintaining high overall evaluation scores, making StripePy a valuable contribution to the community.

Results

Overview of StripePy

StripePy combines pattern recognition, topological persistence, and simple geometric reasoning to detect stripes in contact maps. StripePy's CLI consists of three subcommands: `call`, `view`, and `plot`. `stripepy call` is responsible for running the stripe detection algorithm on a contact matrix in Cooler (.cool or .mcool) or Juicer (.hic) format [25, 26] at a given resolution. The command produces a file in Hierarchical Data Format (.hdf5) [27] containing a list of rectangular regions corresponding to the candidate stripes together with a set of descriptors and complementary information. The genomic coordinates of the candidate stripes identified by `stripepy call` can be extracted from the .hdf5 file using `stripepy view`, which outputs the stripes coordinates in BEDPE format directly to standard output. Finally, `stripepy plot` can be used to visualize architectural stripes overlaid on top of the Hi-C matrix. `stripepy plot` can also generate several graphs showing the general properties of the called stripes. The remaining of this section summarizes the main concepts behind the StripePy algorithm, which are displayed in the form of the graphically-simplified pipeline in Fig. 1; the reader is referred to Methods for the theoretical and technical details.

StripePy's algorithm boils down to four consecutive steps:

Step 1 StripePy starts by applying a pre-processing step: it consists of a log-transformation, followed by the rescaling of the matrix entries such that they are in $[0, 1]$, and then extracts a band around the main diagonal (here set to the default value of 5 Mbp). Then, StripePy searches for vertical stripes

in the lower- and upper-triangular parts of the matrix separately by applying the remaining steps independently.

Step 2 Each of the two triangular matrices is marginalized by summing over the rows, thus obtaining a scalar function. To mitigate uninformative fluctuations due to columns/rows with low interactions, a constraint using the maximum between the scalar function and its weighted moving average is applied. The obtained profile is then scaled so that it has values in $[0, 1]$. All local maxima are filtered with topological persistence, a technique in topological data analysis requiring a local maximum to stand significantly higher than its surroundings to be considered "persistent": this allows to keep loci (here called *seed sites*) that exhibit a more marked linear pattern.

Step 3 Each seed site is analyzed independently to estimate its *horizontal* and *vertical domains* – which consist of the genomic intervals defining the stripe horizontally and vertically, respectively. These can then be used, for example, to compute the stripe width and height and to extract the corresponding regions within the contact map. Given a seed site, the horizontal domain is defined by looking in the neighborhood of the local maximum point where the scalar function from Step 2 is monotonically increasing (resp. decreasing) to the left (resp. to the right) of the maximum point, and then by finding the left (resp. right) bin where the increase (resp. decrease) of the scalar function is the steepest; to prevent excessive stretching of the horizontal domain at low resolutions, a maximum stripe width is introduced (default value: 100 kb). The vertical domain of a stripe is obtained by extracting the columns corresponding to the horizontal domain, marginalizing in a similar manner as in Step 2, and then studying this scalar function. Two criteria are provided: (1) applying topological persistence to the rescaled profile, to identify persistent peaks in the signal and use the location of the furthest peak as a boundary, or (2) thresholding to a minimum value.

Step 4 To enrich what is now just a purely geometric structure (seed site + horizontal domain + vertical domain) a number of complementary descriptors are computed from the contact map, considering the matrix entries either inside the stripe or outside the stripe (in what we call a *k-neighbour*): minimum, quartiles, maximum, arithmetic mean and standard deviation. We then combine the inner and outer means into the *relative change* parameter, which quantifies to what extent the signal differs between the inside and the outside of the stripe; thresholding the relative change equals to ruling out weaker stripes.

StripeBench: unified benchmarking of stripe identification algorithms for Hi-C data

A critical challenge in evaluating the performance of any pattern recognition method for Hi-C is the absence of a robust benchmarking system. Despite the wide and ready availability of contact maps at different conditions such as sequencing depth, resolution, and noise level, the absence of controlled experiments, scalability, and standardization hinders the quantitative analysis of – as well as the fair comparison between – existing tools. To overcome these limitations we introduce StripeBench, a benchmark that includes: a set of 64 simulated contact maps containing realistic TAD and stripe patterns, a standardized "ground truth" baseline for testing, and a collection of metrics for evaluation and comparison. The remainder of this section presents key points and notations related to the benchmark, with a more detailed explanation provided in the Supplementary Information.

The synthetic contact maps are obtained through a genome-wide run of MoDLE [24], a computational tool that uses fast stochastic simulation to sample DNA-DNA contacts generated by loop extrusion. Here, loop extrusion is modelled using extrusion barriers whose occupancy is based on RAD21 ChIP-Seq data from the H1-hESC cell line. Simulations are run using default settings except for the following three factors: target contact density (i.e., the sequencing depth), scale parameter (which controls the noise), and resolution. The effect of changing these parameters on the contact maps is illustrated in Fig. 2A. The ground truth consists of the extrusion barriers (here, CTCF binding sites) inputted to MoDLE. Each barrier includes a position, a blocking direction, and an occupancy value. These three parameters concur to determine the location, direction, and the strength of the linear pattern observed in the resulting contact maps. The ground

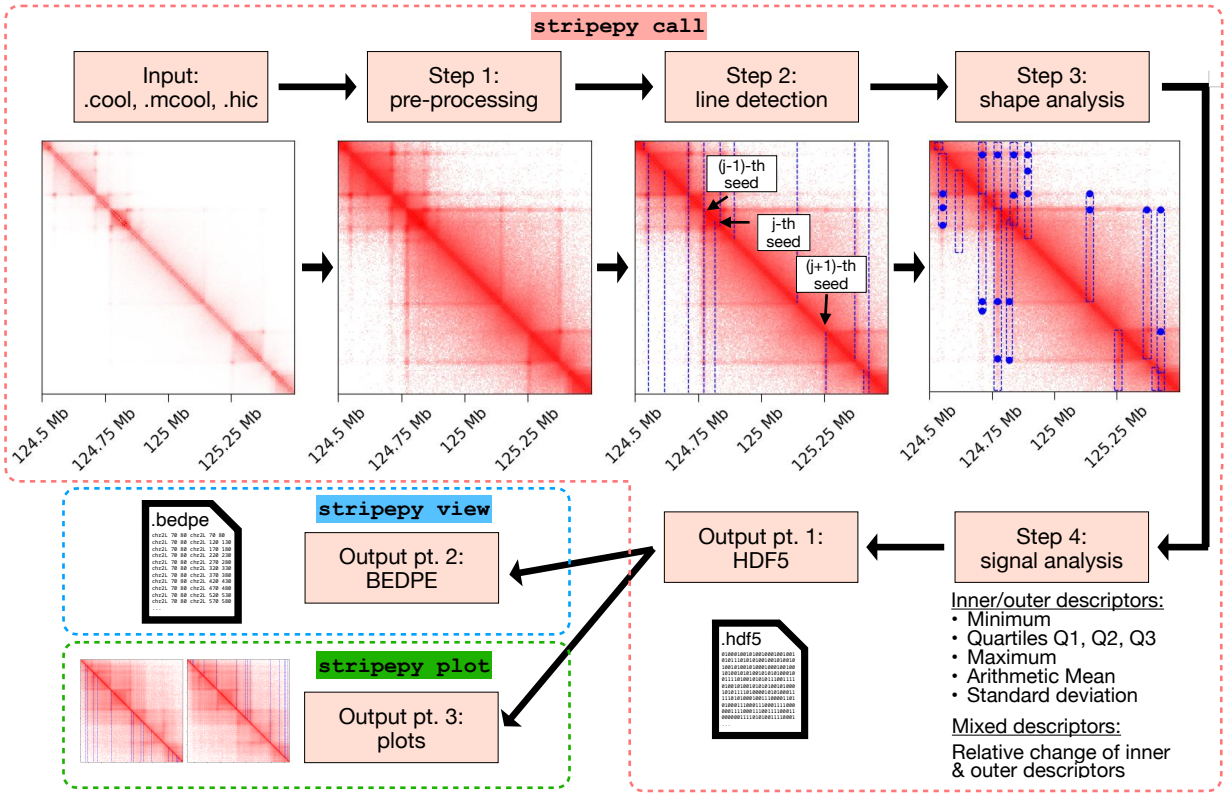


Figure 1: Short overview of StripePy. In `stripepy call`, the input contact map is first pre-processed (Step 1), then studied to detect linear patterns (Step 2) which are subsequently enriched by determining the width and height of each stripe through geometric reasoning (Step 3); the candidates stripes are then post-processed to remove weak occurrences (Step 4). `stripepy view` extracts the genomic coordinates of the candidate stripes and outputs them into a .bedpe file. `stripepy plot` can be used to produce various plots of the steps adopted by `stripepy call`.

truth contains 31,423 barriers at 5 kb, with nearly an equal distribution between lower and upper-triangular occurrences. This number decreases to 30,581, 28,631, and 26,191 at 10 kb, 25 kb and 50 kb, respectively: this reduction occurs because we discard duplicated barriers that overlap with the same genomic bin.

To evaluate and compare stripe callers from multiple directions we introduced two families of measures: *classification measures* target the identification of genomic bins hosting extrusion barriers; *recognition measures* finds out whether predicted stripes include bins *hosting* extrusion barriers, i.e., if it is traversed by it. Fig. 2B illustrates these concepts, see also [28].

We adopt the following "base measures" for the classification problem: sensitivity (TPR), specificity (TNR), and Positive Predictive Value (PPV). Similarly, we introduce the following base recognition measures: *Anchor Hit Rate* (AHR) and *Fraction of Good Candidates* (FGC), which correspond to TPR and PPV in the context of recognition. Despite their popularity and ease of interpretability, these metrics run the risk of being severely biased towards the majority class when applied to highly imbalanced datasets. As a result, they can be misleading if used in isolation [29–31]. To address these concerns and capture a more balanced evaluation, the following single-value metrics are adopted: balanced accuracy (bACC) and Geometric Mean (GM), which combine sensitivity and specificity, respectively; F1-score (F1c) and Fowlkes–Mallows index (FMc), which summarize precision and recall; Jaccard Index (JI).

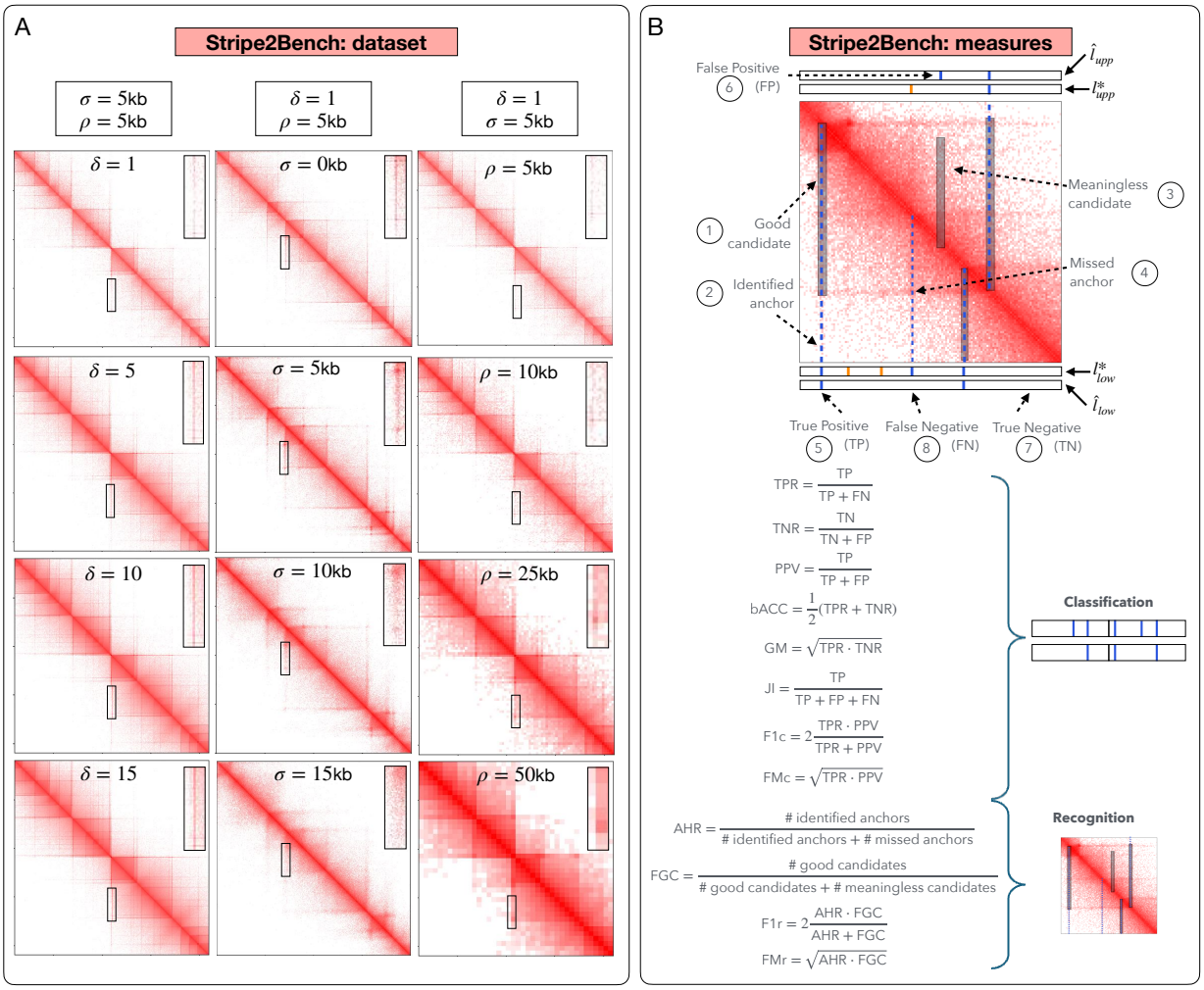


Figure 2: StripeBench. **A** The dataset is composed of 64 contact maps which are generated by varying three factors: contact density δ, noise level σ, resolution ρ; the three columns show the effect of increasing each of these three factors singularly. **B** StripeBench's recognition vs. classification measures. Circled numbers 1-4 point at concepts relevant in recognition: $1-2$) show a predicted stripe (semitransparent rectangle) that can be deemed "good", as it contains a ground truth anchor point (here represented by a dashed blue vertical line); 3) point at a predicted stripe that is not "meaningful" as it does not contain any ground truth anchor point; 4) shows a ground truth anchor point that is not found, as it is not contained in any predicted stripe. Circled numbers 5-8 refer, respectively, to an example of true positive, false positive, true negative and false negative in (bin) classification, where we have represented the lower-triangular ground truth classification vector l^*_{low} and the predicted classification vectors $\hat{l}_{\text{low}}$ with two rectangles where white means 0 (no anchor point) and vertical colored segments denote 1 (anchor point); blue (resp. orange) refers to anchor points with average extrusion barrier occupancy greater or equal (resp. lower) than 0.70. The panel also lists classification and recognition measures.

StripePy: improved stripe classification and recognition

We applied StripePy, Chromosight, StripeCaller, and Stripenn – the currently available stripe callers – to the 64 contacts maps in StripeBench, obtaining a number of predictions that varies with resolution, contact density, and noise level. Specifically, we obtained: from 7,962 to 38,846 stripes with StripePy, from 3,500 to 34,912 with Chromosight, from 0 to 33,002 with StripeCaller, and from 0 to 3,092 with Stripenn. The

number of predicted stripes for each contact map is provided in Supplementary Table 1.

Some representative examples of predicted stripes are provided in Fig. 3A. As mentioned before, Chromosight does not provide estimates for the length or width of a stripe: instead, it returns two genomic pairs – each of which is positioned at a distance determined by the chosen resolution. StripeCaller provides an estimate for the length of a stripe, but, similarly to Chromosight, it sets the width to equal the matrix resolution.

Due to the low predictive capability of Stripenn on this dataset, we decided to exclude this tool from the comparative analysis. Nevertheless, its classification and recognition scores on each contact map are included, together with the scores of the other stripe callers, in Supplementary Tables 2-5.

StripePy improves overall bin classification

First, we conducted an overall analysis by evaluating each tool's classification performance on anchor and non-anchor sites using base measures, considering the entire dataset. Results indicate that StripePy markedly outperforms the other tools in the True Positive Rate (TPR) with a median TPR of 0.2714, compared to Chromosight (0.1957) and StripeCaller (0.0753), see Fig. 3B. Most tools show high True Negative Rate (TNR) due to the overwhelming number of non-anchor bins (median TNR values for StripePy, Chromosight, StripeCaller are 0.9749, 0.9833 and 0.9949, respectively, as shown in Fig. 3C). The strong skewness in label distribution has also been shown to affect negatively Positive Predictive Value (PPV), as pointed out in [31]: the median values for StripePy, Chromosight, and StripeCaller are 0.5270, 0.5395, and 0.5314, respectively, see Fig. 3D).

To address the imbalance in positive and negative bins, we selected a weighted accuracy measure (bACC), which balances sensitivity and specificity through an arithmetic mean. StripePy again outperforms the other tools with a median bACC of 0.6243, compared to Chromosight (0.5886) and StripeCaller (0.5349), see Fig. 3E. Applying the geometric mean, which combines TPR and TNR in an alternative fashion, also confirms this trend: median values for StripePy, Chromosight, and StripeCaller are 0.5161, 0.4389, and 0.2736, respectively (Fig. 3F). Examining the harmonic mean of precision and recall (F1c) reinforces StripePy's improved predictive performance with a median F1c of 0.3571, compared to Chromosight (0.2767) and StripeCaller (0.1291), see Fig. 3G. The alternative combination of precision and recall via the Fowlkes-Mallows index (FMc) also shows StripePy's superiority with a median FMc of 0.3849, versus Chromosight (0.3181) and StripeCaller (0.1966), see Fig. 3H.

Using the Jaccard Index (JI), a summary of classifier performance that considers true positives, false positives, and false negatives, shows that StripePy outperforms the other tools with a median JI of 0.2173, compared to 0.1605 for Chromosight and 0.0690 for StripeCaller, see Fig. 3I.

Regarding interquartile ranges, StripePy provides the lowest values for all classification measures except for TNR (StripePy: 0.0208, Chromosight: 0.0093, StripeCaller: 0.0090, see Fig. 3C) and PPV (StripePy: 0.1332, Chromosight: 0.1252, StripeCaller: 0.1315, see Fig. 3D). The presence of outliers points to possible limit cases where performance drops significantly (see, for example, the case of TPR in Fig. 3B).

To conclude, StripePy consistently outperforms Chromosight and StripeCaller across various classification metrics, including the base measure True Positive Rate (TPR) and all single-value measures: balanced accuracy (bACC), Geometric Mean (GM), F1 score (F1c), Fowlkes-Mallows index (FMc), and Jaccard Index Coefficient (JI). Chromosight and StripeCaller call fewer stripes with a slightly higher Positive Predictive Value (PPV). In terms of stability, interquartile ranges indicate a generally more stable performance for StripePy. However, unhandled edge cases may still occur, as shown by the presence of outliers.

Inclusion of stripe width estimation further increases StripePy's performance

To understand the impact of width estimation, we conducted a similar analysis focused on the recognition problem. The base measure Anchor Hit Rate (AHR), which corresponds to TPR in this task, confirms that StripePy significantly increases the fraction of recognized anchor sites when estimating widths. The median AHR reaches 0.6508, compared to the median classification measure of 0.2714 (Fig. 3J). Analogously, width estimation also enhances the fraction of positives that are true positives: while StripePy's median PPV was 0.5270, its counterpart in the recognition task (Fraction of Good Candidates, FGC) soars to 0.9690 (Fig. 3K). As a result, the single-value measures see a substantial rise: the harmonic mean of AHR and FGC (F1r)

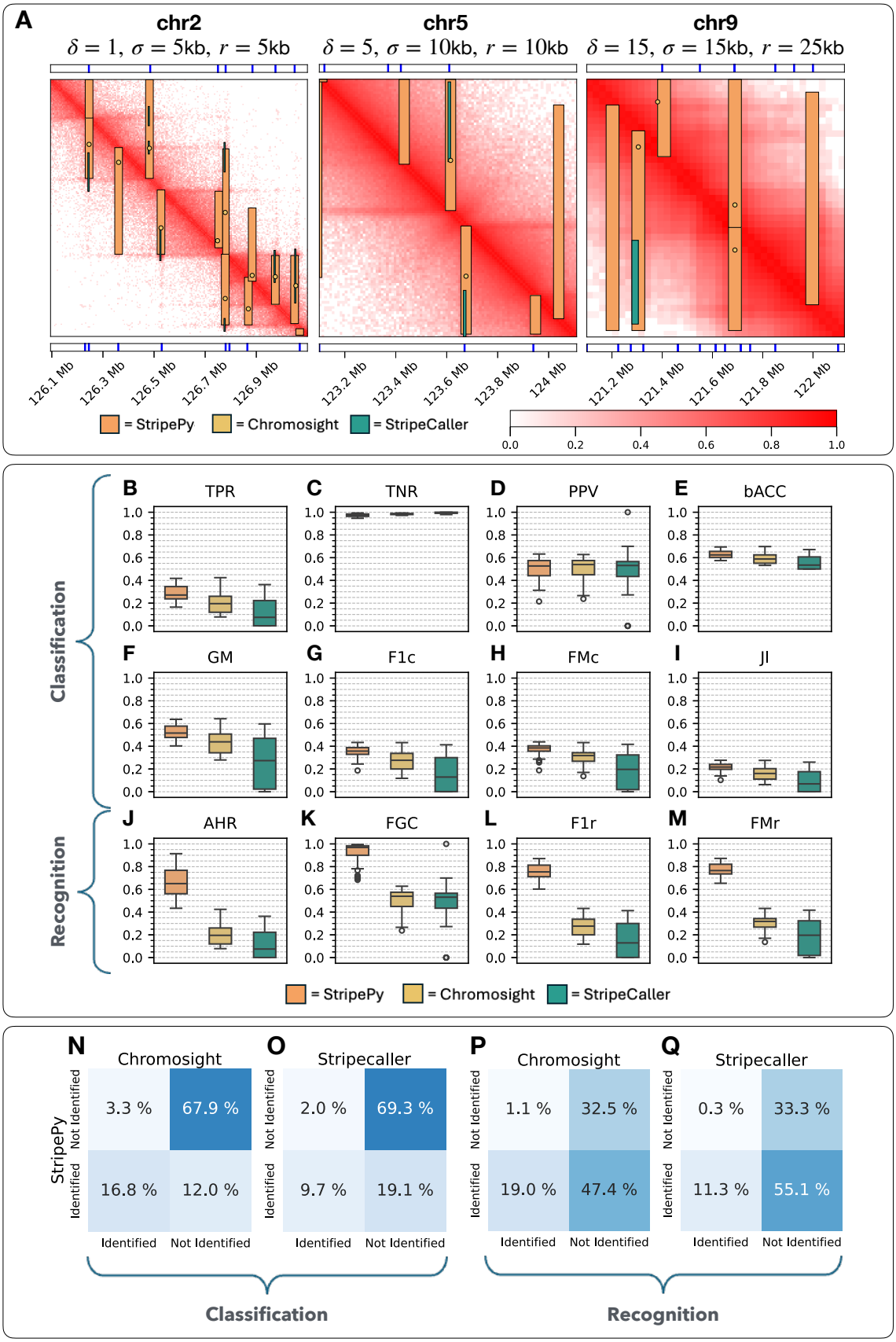

A
chr2
$\delta = 1$, $\sigma = 5$kb, $r = 5$kb
chr5
$\delta = 5$, $\sigma = 10$kb, $r = 10$kb
chr9
$\delta = 15$, $\sigma = 15$kb, $r = 25$kb
126.1 Mb
126.3 Mb
126.5 Mb
126.7 Mb
126.9 Mb
123.2 Mb
123.4 Mb
123.6 Mb
123.8 Mb
124 Mb
121.2 Mb
121.4 Mb
121.6 Mb
121.8 Mb
122 Mb
= StripePy
= Chromosight
= StripeCaller
0.0
0.2
0.4
0.6
0.8
1.0
Classification
Recognition
B TPR
C TNR
D PPV
E bACC
F GM
G F1c
H FMc
I JI
J AHR
K FGC
L F1r
M FMr
= StripePy
= Chromosight
= StripeCaller
N Chromosight
O Stripecaller
P Chromosight
Q Stripecaller
StripePy
Not Identified
Identified
3.3 %
67.9 %
16.8 %
12.0 %
2.0 %
69.3 %
9.7 %
19.1 %
1.1 %
32.5 %
19.0 %
47.4 %
0.3 %
33.3 %
11.3 %
55.1 %
Identified
Not Identified
Classification
Recognition

Figure 3: Benchmarking pt. 1. **A** Three regions extracted from three MoDLE-simulated contact maps. The upper and lower bands show the anchor points from the ground truth as solid vertical blue lines, and they refer to the upper- and the lower-triangular parts, respectively. The images of the contact maps have superimposed the predictions produced by the tools under study. **B-M** Boxplots of the classification (TPR, TNR, PPV, bACC, GM, F1c, FMc, and JI) and recognition (AHR, FGC, F1r, and FMr) measures over the StripeBench contact maps. **N-Q** Heatmaps summarizing how StripePy performs compared to Chromosight and StripeCaller in terms of percentages of ground truth anchor points: (classification) correctly labeled as such; (recognition) found inside a candidate stripe.

attains a median value of 0.7540 for StripePy, compared to a median F1c of 0.3571 (Fig. 3L); the median Fowlkes-Mallows index (FMr) is 0.7655, whereas its corresponding value was 0.3849 in the classification (Fig. 3M). While StripePy drastically improves its performance, Chromosight and StripeCaller merely carry over their TPR, PPV, F1c, and FMc values to AHR, FGC, F1r, and FMr. This is because these tools do not have any notion of widths, and thus their metrics are not impacted by interpreting the problem as recognition rather than classification (Figs. 3J-M).

To summarize, estimating the width of a stripe, not only provides a more realistic description of the stripe properties, but also significantly improves the performance of a stripe caller, reinforcing the point that stripes should not be thought of as simple 1D segments, but rather as narrow rectangles.

StripePy discovers previously detected and novel stripes

We then investigated the degree of overlap between predictions obtained by StripePy and those produced by Chromosight and StripeCaller.

We compared anchors sites detected and missed by StripePy against those by Chromosight (Fig. 3N) and StripeCaller (Fig. 3O). The study reveals that, on average, 67.9% (Fig. 3N) to 69.3% (Fig. 3O) of ground truth anchor points remains undetected by all tools. The vast majority of anchor points correctly identified by Chromosight and StripeCaller, are also found by StripePy: only 3.3% (Fig. 3N) and 2.0% (Fig. 3O) of anchor sites are unique to Chromosight and StripeCaller, respectively. Finally, StripePy recognizes a greater percentage of anchor points, achieving 19.1% detection rate when StripeCaller is considered (Fig. 3O).

When leveraging stripe widths, StripePy's advantage increases significantly. The percentage of anchors undetected by StripePy and Chromosight falls to 32.5% (Fig. 3P), for StripePy and StripeCaller drops to 33.3% (Fig. 3Q). Only 1.1% of anchor points are now unique to Chromosight (Fig. 3P), and just 0.3% are unique to StripeCaller (Fig. 3Q). Meanwhile, a substantial proportion of anchor sites are only found by StripePy: 47.4% compared to Chromosight (Fig. 3P) and 55.1% compared to StripeCaller (Fig. 3Q).

StripePy outperforms existing tools across a diverse set of conditions

Hi-C data is generated with varying noise levels and sequencing depths, which affect the resolution at which the data can be meaningfully studied, and significantly impacts data analysis, such as the identification of architectural stripes [32]. Unlike other functional genomics assays, Hi-C data requires analysis at a user-determined effective resolution, often involving the consideration of multiple bin-sizes on the basis of the pattern under study. However, higher resolution increases technical noise and contributes to data sparseness. These factors are influenced, among other things, by the technology used, such the digestion strategy, e.g., sequence-specific digestion by one or more restriction enzymes, or non-sequence specific digestion by MNase (Micro-C) or DNAse (DNAse Hi-C) [33–36]. To study the effects of contact density, noise, and resolution on the callers, we stratified the dataset by these factors and evaluated the dependency of classification and recognition metrics.

To this end, we grouped contact maps by specifying one factor at a time. For the sake of clarity, here we limit the analysis to the medians of the classification and recognition scores; boxplots of the scores can be found in the Supplementary Information – Extended Data Fig. 1, Extended Data Fig. 2, and Extended Data Fig. 3.

Upon first investigation, we observe that medians yield rankings in line with the overall analysis: StripePy consistently outperforms other tools across all contact densities, noise levels, and resolutions for the majority of measures. Focusing on base measures, StripePy provides the highest TPR scores (Fig. 4, panels A1, B1, and C1), AHR scores (Fig. 4, panels A9, B9, and C9) and FGC scores (Fig. 4, panels A10, B10, and C10); however, StripePy exhibits a generally lower median performance in correctly labeling negative bins, as displayed by TNR (Fig. 4, panels A2, B2, and C2). When it comes to PPV, StripePy achieves median values similar to those of Chromosight and StripeCaller at higher contact density and noise level, and at lower resolution (Fig. 4, panels A3, B3, and C3). StripePy's leads in classification and recognition is confirmed when adopting single-value classification (bACC, GM, F1c, FMc, and JI, see Fig. 4, panels A4-A8, B4-B8, and C4-C8) and recognition measures (F1r, and FMr, see Fig. 4, panels A11-A12, B11-B12, and C11-C12).

We observe that increasing contact frequency does not always result in higher scores. For example, the medians of TPR (Fig. 4A1) and AHR (Fig. 4A9) show that increasing the contact density results in a slight decrease in StripePy performance. Conversely, Chromosight experiences a minor increase in its performance, while it is only StripeCaller that progressively increases its medians, although it appears to asymptotically approach a value well below StripePy and Chromosight.

Regarding changes in noise level, we see that the median of TPR (Fig. 4B1), AHR (Fig. 4B9), and of all single-value metrics (Fig. 4, panels B4-B8 and B11-B12) show drops of different magnitudes as noise increases. This trend is accompanied by a rise in the median TNR scores (Fig. 4B2). The simultaneous rise in median TNR scores and decrease in TPR scores suggests that all callers tend to reduce the number of detected stripes as noise increases, as confirmed by the data in Supplementary Table 1.

With the sole exception of TNR (Fig. 4C2) and AHR (Fig. 4C9), higher resolutions lead to a drop in median scores for StripePy and Chromosight; the same applies to StripeCaller on a smaller number of metrics. For example, all callers exhibit decreases in median FMc (Fig. 4C7) and FMr (Fig. 4C12) when transitioning from 10 kb to 5 kb. This might be due, among other reasons, to the rise in sparsity that accompanies the increase in resolution.

Execution times

Wall-clock execution times were measured on a MacBook Pro running macOS Sonoma, equipped with an Apple M1 processor and 8GB of RAM. The average running time per contact map is: StripePy - 1m24s; Chromosight - 1m56s; StripeCaller - 2m17s; Stripenn - 18m23s.

Analyzing human Hi-C matrices with StripePy reveals fast and robust detection of stripes

To investigate how StripePy behaves on real data, we gathered files from the 4DNucleome and ENCODE, and compared the stripes predicted by various available callers at a resolution of 10 kilo-base pair (kbp) with the CTCF peaks from the corresponding cell lines.

We selected four contact maps: an in-situ Hi-C and a Micro-C map from the H1 human embryonic stem cell line (H1-hESC), and an in-situ Hi-C and an intact Hi-C map from the human lymphoblastoid GM12878 cell line. This choice allows us to compare the tools across different cell lines with distinct contact patterns (e.g., sparsity and number of interactions) and technologies, thus providing a comprehensive evaluation of performance under varied conditions. Each interaction map was paired with a CTCF ChIP-Seq dataset. These datasets were used to generate a list of stripe anchor points by taking the midpoint between the start and end coordinates of a CTCF peak. At a resolution of 10 kb, the Hi-C and Micro-C maps for H1-hESC consist of 215,229,365 and 276,917,669 non-zero entries, which sum up to a total of 2,785,926,565 and 1,186,984,847 interactions, respectively (numbers refer to the cis portion of the contact map). This highlights that although the Hi-C map contains more than twice as many interactions when compared to the Micro-C map, it is sparser. As per the intact Hi-C and in-situ Hi-C maps for GM12878, the first has 278,180,961 non-zero entries and a total of 1,415,005,237 interactions, while the second has 91,009,553 non-zero entries and 338,111,285 interactions (numbers refer to the cis portion of the contact map at 10 kb). This indicates that the first map is less sparse and has a higher number of interactions than the second one.

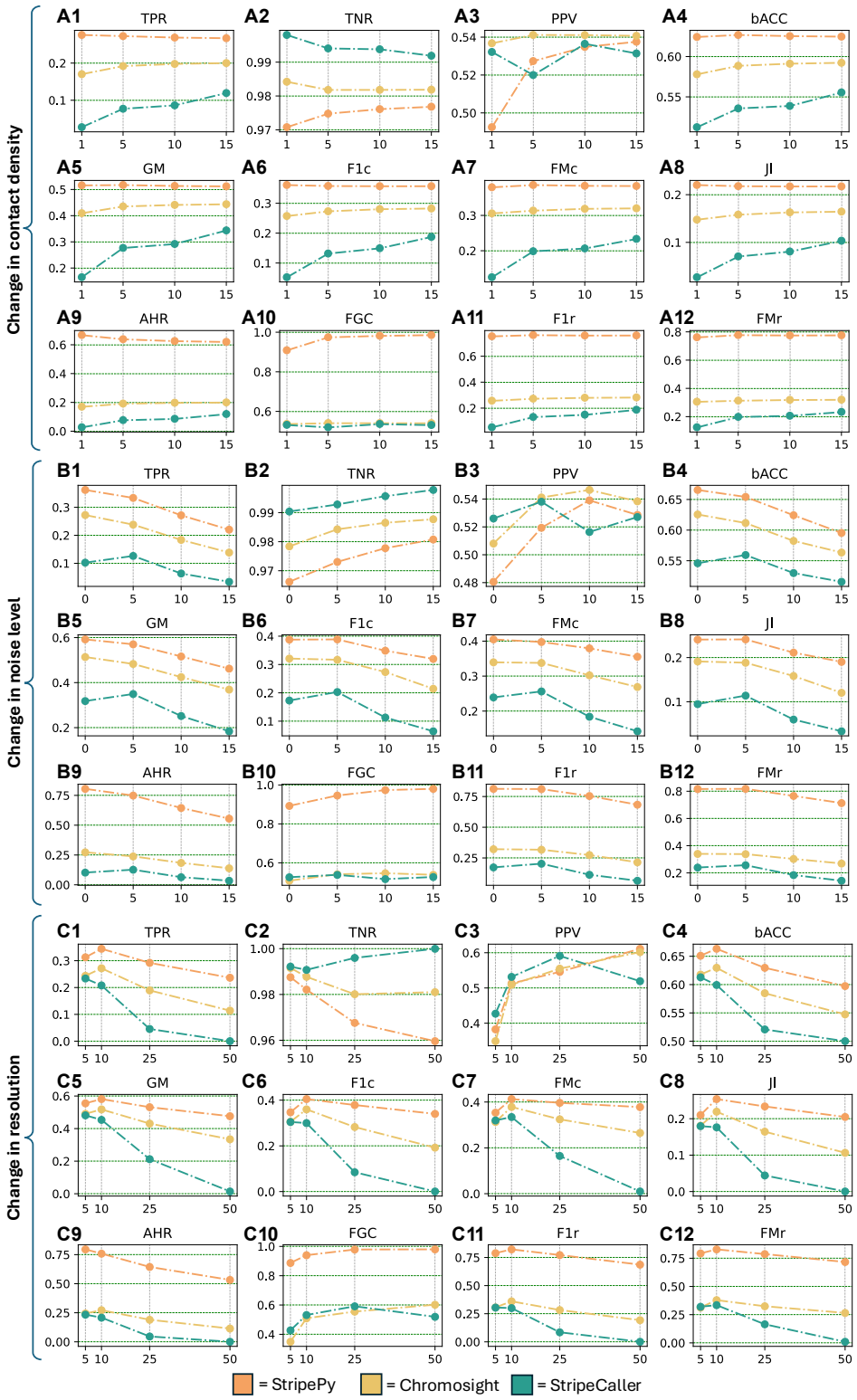


Figure 4: Benchmarking pt.2. Plots of the medians of the classification and recognition measures by single factors: contact density (**A1-A12**), noise level (**B1-B12**), and resolution (**C1-C12**).

Running StripePy on real data confirms its performance in stripe recognition

Since these datasets are also manageable by Stripenn, we have included this tool in the comparative analysis.

Snapshots of neighborhoods from different chromosomes are provided in Fig. 5. Supplementary Table 6 reports the classification and recognition measures for the three maps, complete of: the number of anchors founds (nAF), the number of stripes predicted (nSP), and the elapsed real times measured on a MacBook Pro running macOS Sonoma, equipped with an Apple M1 processor and 8GB of RAM. To ease the comparison, classification and recognition scores are reported as percentages. From plots and statistics we reach a series of conclusions.

First, Chromosight, StripeCaller, and Stripenn focus on visually prominent segments, while StripePy can also locate milder or hidden patterns associated with ChIP-Seq peaks. An example of this is indicated by the black arrows in Fig. 5A1-A3, which shows interactions and stripes for the H1-hESC Micro-C dataset. This point is further highlighted by the number of anchors found (nAF), which is significantly higher for StripePy: for the same contact map, stripes predicted by StripePy contain a total of 31,729 anchor sites, compared to 6,257, 5,294, and 9,263 anchors identified by Chromosight, StripeCaller, and Stripenn, respectively (Supplementary Table 6). The higher number of anchors found is accompanied by a significantly higher number of predictions (nSP): for this contact map, StripePy 23,727 candidates, compared to 11,765, 12,018, and 3,864 predictions from Chromosight, StripeCaller, and Stripenn, respectively (Supplementary Table 6).

Moving to the analysis of the classification and recognition measures, we observe that most of the general findings uncovered with the synthetic data from StripeBench are still valid on these maps (Supplementary Table 6). For classification measures, we indeed observe that StripePy is stronger in correctly predicting positives rather than negatives. When focusing on single-value metrics, StripePy generally achieves the highest scores, making it the best overall classifier. The sole exceptions involve the H1-hESC Micro-C (for bACC and FMc) and GM12878 DpnII in-situ Hi-C datasets (for the sole bACC). A possible explanation for this underperformance is the sparsity of these two maps, which can affect the pseudo-distributions by generating spurious local maxima with little, if any, biological significance. In the most severe case, namely the GM12878 in-situ Hi-C dataset, the sparsity is compounded by a low number of interactions, leading to a significant increase in the number of candidate stripes (see Fig. 5D1-D3). For base recognition measures, StripePy excels in AHR but lags behind Stripenn with respect to FGC. The latter can identify however only between 9.10% (in GM12878 in-situ Hi-C) and 17.00% (in H1-hESC Micro-C) of the anchor sites, which contributes to its relatively low overall performance in recognition measures like F1r and FMr. In other words, Stripenn excels in finding a very limited number of stripes which, in turn, are highly likely to contain anchor sites (see Fig. 5A4, B4, C4, and D4). Conversely, StripePy has a lower percentage of stripes that contain anchor sites but provides a much more balanced performance in terms of stripe recognition (see Fig. 5A1, B1, C1, and D1).

In terms of execution times, StripePy proves to be the fastest caller.

Matrix normalization affects StripePy performance

We decided to investigate the effect of contact map normalization on the performance of StripePy. To this end, maps from Supplementary Table 6 were normalized using hictk [37] with two algorithms: genome-wide iterative correction and eigenvector decomposition (ICE) [38] and genome-wide SCALE [39]. To ensure a consistent comparison, StripePy was rerun on the normalized contact maps with all thresholds unchanged. Snapshots of neighborhoods are provided in Extended Data Fig. 4, while classification and recognition measures are reported in Supplementary Table 7.

We first note that applying normalization significantly reduces the number predicted stripes (nSP). This reduction is visually evident when comparing the plots for the case of no normalization (Extended Data Fig. 4 A1, B1, C1, and D1) with those for ICE- (Extended Data Fig. 4 A2, B2, C2, and D2) and SCALE-normalized maps (Extended Data Fig. 4 A3, B3, C3, and D3). This observation is further confirmed by the nSP values in Supplementary Table 7: for example, for the H1-hESC Micro-C map, nSP decreases from 31,729 to 15,114 and 18,472 for raw, and ICE- and SCALE-normalized maps, respectively. This is not surprising, as one of the effects of Hi-C matrix normalization is smoothing of the original data, resulting in smoothed global pseudo-distributions and, consequently, a smaller number of peaks (i.e., stripe seeds).

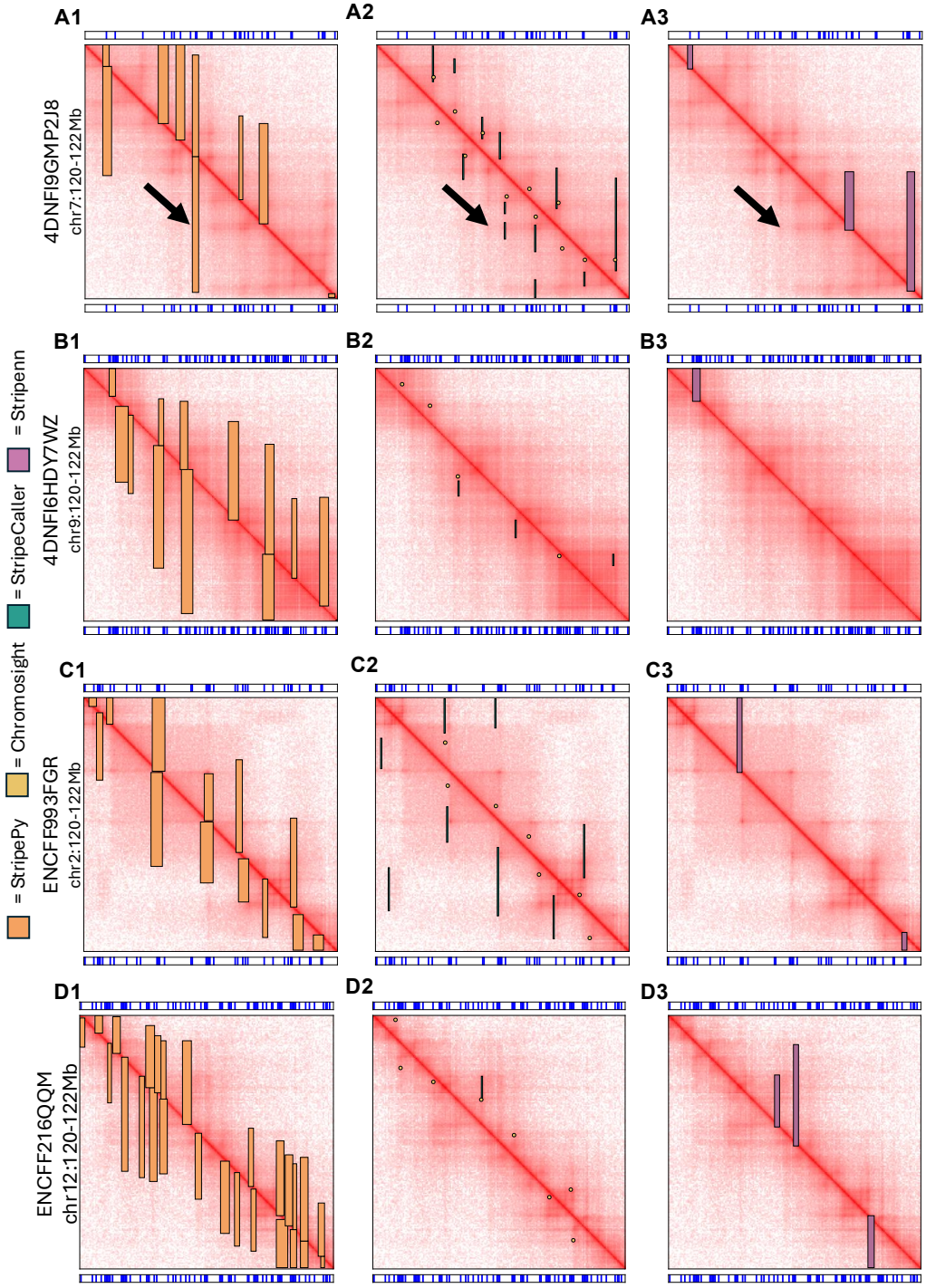


Figure 5: Four regions from as many contact maps. The upper and lower bands point at bin hosting CTCF sites, obtained from ChIP-Seq data. The images of the contact maps have superimposed the predictions produced by the tools under study. For each map, the region is repeated three times to improve clarity: first column consists of StripePy, second column refers to Chromosight and StripeCaller, last column has Stripenn.

Other interesting observations can be made from Supplementary Table 7. First, the decrease in nSP is accompanied by a decrease in the number of found anchors (nAF): for the same map, nAF decreases from 23,727 to 6,478 and 8,574 with ICE- and SCALE-normalized maps, respectively. Moving to the analysis of classification and recognition measures, we observe that for all contact maps, applying normalization increases some base metrics (TNR and FGC) while decreasing others (TPR, PPV and AHR). By examining single-value metrics, it becomes evident that the gain in the former scores is not sufficient to compensate the drop in the latter metrics, resulting in a unanimous decrease in all single-value statistics. Therefore, we conclude that normalizing contact maps before using StripePy for stripe calling might impair performance and should be approached with caution.

Discussion

Hi-C and related techniques have greatly advanced our understanding of 3D genome organization and its role in gene regulation, DNA replication, and repair [14, 22]. While computational methods have been crucial to these insights [40], tools for detecting architectural stripes remain scarce. Here, we developed StripePy, a new CLI application which combines concepts from pattern recognition, algebraic topology, and basic geometry to detect architectural stripes from Hi-C data. StripePy can process contact maps in various formats (.cool, .mcool, and .hic) and outputs its findings in both Hierarchical Data Format (.hdf5) and BEDPE format. Unlike most existing solutions, StripePy does not reduce stripe detection to a mere bin classification problem. Instead, it also computes shape descriptors such as the width and height of stripes, and generates statistics that can be used for postprocessing purposes, such as ranking and classification of candidate stripes to, e.g., filter out unrealistic stripes.

A major challenge in the field is the lack of standardized definitions for architectural elements in Hi-C data, such as TADs, stripes, and individual contacts [11, 41–43]. To address this, we developed StripeBench, a dataset of 64 simulated contact maps with ground truth annotations and a comprehensive list of metrics, facilitating exhaustive quantitative comparisons between callers. We anticipate that this benchmark will aid in the future development of algorithms tailored for Hi-C and related data.

An additional obstacle with the analysis of Hi-C data lies in the variation in resolutions, noise levels, and sequencing depth that characterize this kind of data, all of which hinder standardized detection of chromatin features. Our benchmark tackles this issue by incorporating datasets with diverse conditions, enabling a more comprehensive evaluation of stripe detection methods.

Based on the benchmarks presented in the previous sections, it is clear from the quantitative analysis that all tools struggle, to different degrees, in accurately identifying true positives in the classification task. This difficulty is mainly caused by the skewed distribution of the ground truth labels, but it also stems from the lack of a precise and unambiguous definition of what a stripe is. While incorporating geometric descriptors proves beneficial – as noticeable from the increased F1-scores and Fowlkes–Mallows indexes – further work is needed to better define architectural stripes and their functional implications.

References

1. Van Bortle, K. & Corces, V. G. Nuclear Organization and Genome Function. *Annual Review of Cell and Developmental Biology* **28,** 163–187 (2012).
2. Dekker, J. & Misteli, T. Long-range chromatin interactions. *Cold Spring Harbor perspectives in biology* **7,** a019356 (2015).
3. Bonev, B. & Cavalli, G. Organization and function of the 3D genome. *Nature Reviews Genetics* **17,** 661–678 (Nov. 2016).
4. Zheng, H. & Xie, W. The role of 3D genome organization in development and cell differentiation. *Nature Reviews Molecular Cell Biology* **20,** 535–550 (Sept. 2019).
5. Bouwman, B. A., Crosetto, N. & Bienko, M. The era of 3D and spatial genomics. *Trends in Genetics* **38,** 1062–1075 (2022).

6. Rabl, C. Uber zellthilung. *Morphol Jahrb* **10,** 214–330 (1885).

7. Cremer, T. & Cremer, M. Chromosome territories. *Cold Spring Harbor perspectives in biology* **2,** a003889 (2010).

8. Lieberman-Aiden, E. *et al.* Comprehensive Mapping of Long-Range Interactions Reveals Folding Principles of the Human Genome. *Science* **326,** 289–293 (2009).

9. Nora, E. P. *et al.* Spatial partitioning of the regulatory landscape of the X-inactivation centre. *Nature* **485,** 381–385 (May 2012).

10. Dixon, J. R. *et al.* Topological domains in mammalian genomes identified by analysis of chromatin interactions. *Nature* **485,** 376–380 (May 2012).

11. Raffo, A. & Paulsen, J. The shape of chromatin: insights from computational recognition of geometric patterns in Hi-C data. *Briefings in Bioinformatics* **24,** bbad302 (2023).

12. Fudenberg, G. *et al.* Formation of Chromosomal Domains by Loop Extrusion. *Cell Reports* **15,** 2038–2049 (2016).

13. Chang, L.-H., Ghosh, S. & Noordermeer, D. TADs and Their Borders: Free Movement or Building a Wall? *Journal of Molecular Biology* **432.** Perspectives on Chromosome Folding, 643–652 (2020).

14. Gupta, K. *et al.* StripeDiff: Model-based algorithm for differential analysis of chromatin stripe. *Science Advances* **8,** eabk2246 (2022).

15. Kraft, K. *et al.* Serial genomic inversions induce tissue-specific architectural stripes, gene misexpression and congenital malformations. *Nature cell biology* **21,** 305–310 (2019).

16. Kim, S. & Shendure, J. Mechanisms of Interplay between Transcription Factors and the 3D Genome. *Molecular Cell* **76,** 306–319 (2019).

17. Kraft, K. *et al.* Serial genomic inversions induce tissue-specific architectural stripes, gene misexpression and congenital malformations. *Nature Cell Biology* **21,** 305–310 (Mar. 2019).

18. Arnould, C. *et al.* Loop extrusion as a mechanism for formation of DNA damage repair foci. *Nature* **590,** 660–665 (Feb. 2021).

19. Drayton, J. A. & Hansen, A. S. Right on target: Chromatin jets arise from targeted cohesin loading in wild-typecells. *Molecular Cell* **82,** 3755–3757 (2022).

20. Vian, L. *et al.* The Energetics and Physiological Impact of Cohesin Extrusion. *Cell* **173,** 292–294 (2018).

21. Matthey-Doret, C. *et al.* Computer vision for pattern detection in chromosome contact maps. *Nature Communications* **11,** 5795–5795 (2020).

22. Sora Yoon, Aditi Chandra & Vahedi, G. Stripenn detects architectural stripes from chromatin conformation data using computer vision. *Nature Communications* **13** (2022).

23. Carlsson, G. Topology and data. *Bulletin of the American Mathematical Society* **46,** 255–308 (2009).

24. Rossini, R., Kumar, V., Mathelier, A., Rognes, T. & Paulsen, J. MoDLE: high-performance stochastic modeling of DNA loop extrusion interactions. *Genome Biology* **23,** 247 (Nov. 2022).

25. Abdennur, N. & Mirny, L. A. Cooler: scalable storage for Hi-C data and other genomically labeled arrays. *Bioinformatics* **36,** 311–316 (2020).

26. Durand, N. C. *et al.* Juicebox provides a visualization system for Hi-C contact maps with unlimited zoom. *Cell systems* **3,** 99–101 (2016).

27. Folk, M., Heber, G., Koziol, Q., Pourmal, E. & Robinson, D. *An overview of the HDF5 technology suite and its applications* in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (Association for Computing Machinery, Uppsala, Sweden, 2011), 36–47.

28. Kuhn Max;Johnson, K. *Applied Predictive Modeling* (Springer New York, 2018).

29. Gu, Q., Zhu, L. & Cai, Z. *Evaluation Measures of the Classification Performance of Imbalanced Data Sets* en. in *Computational Intelligence and Intelligent Systems* (eds Cai, Z., Li, Z., Kang, Z. & Liu, Y.) (Springer, Berlin, Heidelberg, 2009), 461–471.

30. Luque, A., Carrasco, A., Martín, A. & de las Heras, A. The impact of class imbalance in classification performance metrics based on the binary confusion matrix. *Pattern Recognition* **91,** 216–231. (2024) (July 2019).

31. Aguilar-Ruiz, J. S. & Michalak, M. Classification performance assessment for imbalanced multiclass data. *Scientific Reports* **14,** 10759 (May 2024).

32. Lajoie, B. R., Dekker, J. & Kaplan, N. The Hitchhiker's guide to Hi-C analysis: Practical guidelines. *Methods* **72,** 65–75 (2015).

33. Krietenstein, N. *et al.* Ultrastructural Details of Mammalian Chromosome Architecture. *Molecular Cell* **78,** 554–565.e7 (May 2020).

34. Ma, W. *et al.* Fine-scale chromatin interaction maps reveal the cis-regulatory landscape of human lincRNA genes. *Nature methods* **12,** 71–78 (2015).

35. Ramani, V. *et al.* Mapping 3D genome architecture through in situ DNase Hi-C. *Nature protocols* **11,** 2104–2121 (2016).

36. Yardımcı, G. G. *et al.* Measuring the reproducibility and quality of Hi-C data. *Genome Biology* **20,** 57–57 (2019).

37. Rossini, R. & Paulsen, J. hictk: blazing fast toolkit to work with .hic and .cool files. *Bioinformatics* **40,** btae408 (June 2024).

38. Imakaev, M. *et al.* Iterative correction of Hi-C data reveals hallmarks of chromosome organization. *Nature Methods* **9,** 999–1003 (Oct. 2012).

39. Robinson, J. T. *et al.* Juicebox.js Provides a Cloud-Based Visualization System for Hi-C Data. *Cell Systems* **6,** 256–258.e1 (2018).

40. Stefano, M. D., Paulsen, J., Jost, D. & Marti-Renom, M. A. 4D nucleome modeling. *Current Opinion in Genetics & Development* **67,** 25–32 (2021).

41. Carty, M. *et al.* An integrated model for detecting significant chromatin interactions from high-resolution Hi-C data. *Nature Communications* **8,** 15454 (May 2017).

42. Wit, E. d. TADs as the Caller Calls Them. *Journal of Molecular Biology* **432,** 638–642 (2020).

43. Liu, N. *et al.* Seeing the forest through the trees: prioritising potentially functional interactions from Hi-C. *Epigenetics & Chromatin* **14,** 41 (Aug. 2021).

Methods

Extended description of StripePy

StripePy runs through four consecutive steps: a pre-processing (Step 1), followed by the identification of genomic bins exhibiting linear patterns (Step 2), the subsequent estimation of geometric descriptors (Step 3), and the final definition of biological descriptors (Step 4). A compact overview of the algorithm is given in Extended Data Fig. 5.

Step 1: Laying the groundwork for data exploration

To mitigate the bias of the *genomic distance effect* – namely the tendency of higher prevalence of crosslinks between genomic loci close together along the genome even in the absence of any specific higher order structure – we perform global enhancement via the well-known log-transformation. The log-transformation is a global intensity transformation which expands narrow ranges of low intensity values and compresses wide ranges of high intensity values. The general form of the log-transformation is

$$P(x, y) = c \log(1 + H(x, y)),$$

being $H(x, y) \geq 0$ the initial intensity at bin pair (x, y), $P(x, y)$ the output intensity at the same location, and c a constant. Here, we set c so that intensities are projected onto $[0, 1]$. Furthermore, we retain only entries that fall within a specific band around the main diagonal (default bandwidth: 5 Mbp), as we are specifically targeting patterns anchored at the main diagonal. As an example, Extended Data Fig. 5A-B provides a graphical illustration of how global enhancement acts on a specific square sub-region along the main diagonal (what we will call a *region of interest*).

We proceed by splitting the pre-processed Hi-C matrix P into lower- and upper-triangular matrices, henceforth denoted as L and U, respectively; see Extended Data Fig. 5C-D.

Step 2: Line detection via pattern recognition and Topological Data Analysis

For the sake of conciseness, we here adopt two simplifications. First and foremost, we restrict our explanation on the lower-triangular matrix, i.e., L – as the processing of the upper-triangular part U is analogous; for visualization purposes, however, we will often show the full Hi-C matrix P. We choose to display the full matrix as it can help the visual analysis of an algorithm's output, as any vertical pattern can be found horizontally given that Hi-C matrices are naturally symmetric. Secondly, we limit ourselves to recognizing vertical patterns, again because of symmetry.

We sum L over the rows, thus obtaining a scalar function, or 1D profile, which can be interpreted as a *pseudo-distribution* π_{glob}. The function is smoothed by using weighted quasi-interpolation [1, 2], a weighted moving average technique for curve approximation. A graphical example is given in Extended Data Fig. 5E. The term "pseudo" used in this context is necessary as π_{glob} is not, strictly speaking, a probability distribution. The location of each local maximum points at a possible linear pattern. Note that the pseudo-distribution bears resemblance to the accumulator function derived by applying the Hough Transform [3, 4], but in this case the search for vertical linear patterns is conducted without any edge detection or binarization to speed up computation and avoid the introduction of additional parameters.

We filter local maxima by measuring their significance via *topological persistence*. In the analogy of the water level, the pseudo-distribution is a submerged bank. Imagine the water level to gradually go down: when it reaches a local maximum, a new island is formed, and its height is called *birth level*; when it reaches a local minimum, two islands merge. The higher island is, by convention, considered to subsume the lower island, and the local minimum is the lower island's *death level*. The topological persistence of an island, i.e., its significance, is the difference between its birth and death levels. The local maximum points of π_{glob} that survive the filtration process will be here referred to as the *seed sites* of candidate stripes, and will be denoted as $s_1 < s_2 < \cdots < s_n$. An example of seed sites is provided in Extended Data Fig. 5F, while the corresponding locations are superimposed to the contact map in Extended Data Fig. 5G.

To avoid spurious seed sites from sparse regions, we apply the following additional check. For each seed site, we consider a neighborhood of pre-defined size and calculate the ratio of bins – within this neighborhood – where π_{glob} exceeds 0.1 (i.e., 10% of the maximum height); if this ratio is greater than 85%, the seed site is retained, otherwise it is discarded.

Step 3: Shape analysis

Once the seed sites (i.e., sites hosting linear patterns) $s_1, \ldots, s_n$ are detected in the lower-triangular pre-processed Hi-C matrix L, we proceed by estimating width and height of each candidate stripe S_j independently as follows.

Width estimation We start by defining a sub-interval $I_j := [i_j^L, i_j^R] \subset I$ so that I_j is the largest interval with the following properties: (1) s_j is contained in I_j; (2) π_{glob} is monotonically increasing in $[i_j^L, s_j]$; (3) π_{glob} is monotonically decreasing in $[s_j, i_j^R]$. To define the width of of S_j, we look at the global pseudo-distribution π_{glob} in the interval I_j. The right boundary of the candidate stripe is defined as

$$b^R(s_j) := \min\{\tilde{b}^R(s_j), b + w_{\max}/2\},$$

where

$$\tilde{b}^R(s_j) := \arg\max_b \left\{ \pi_{\text{glob}}(b) - \pi_{\text{glob}}(b-1) \middle| s_j < b \leq i_j^R \right\};$$

the left boundary is given by

$$b^L(s_j) := \max\{\tilde{b}^L(s_j), b - w_{\max}/2\},$$

where

$$\tilde{b}^L(s_j) := \arg\max_b \left\{ \pi_{\text{glob}}(b+1) - \pi_{\text{glob}}(b) \middle| i_j^L \leq b < s_j \right\}.$$

Here, $w_{\max}$ denotes the maximum width, which by default is set to a value of 100 kb.

These formulas correspond to defining the boundaries as the genomic bins exhibiting largest values of (approximate) first-order derivative of π_{glob} in a left or right neighbourhood of s_j, with a maximum width as additional constraint. An exemplification of this description is shown in Extended Data Fig. 5H. Given a seed site, the left and right boundaries naturally define horizontal intervals, sometimes referred to as horizontal domains: see Extended Data Fig. 5I for a graphical display of what the single horizontal domains mean in terms of (lower-triangular) matrix slicing.

The width of a candidate stripe is computed by $b^R(s_j) - b^L(s_j)$.

Height estimation Given the j-th seed site and the corresponding horizontal domain made of left and right boundaries, we define a vertical domain (lower and upper boundaries) by introducing a *local* pseudo-distribution. Such a profile is obtained by applying the following procedure (also see Extended Data Fig. 5I-J):

- Slice L: filter out columns that do not belong to the current horizontal domain; discard rows from the upper-triangular part of the matrix.
- Sum (i.e., marginalize) over the surviving columns, then scale by dividing by the global maximum. Smooth the local pseudo-distribution by using, again, the weighted quasi-interpolation.

Two criteria are available: (1) By applying topological persistence to find persistent maxima, we can determine where the local 1D pseudo-distribution presents significant peaks; we use the peak that is farthest from the main diagonal to define the stripe boundary. (2) The vertical domain of a candidate stripe is retrieved by finding the smallest interval containing all coordinates where the smooth approximation is above a user-threshold. When studying a candidate stripe in the lower-triangular matrix, the top-boundary $b^U(s_j)$ is naturally anchored to the main diagonal; for bottom-boundary $b^D(s_j)$, the steepest the local pseudo-distribution, the closer $b^D(s_j)$ is to the main diagonal. The height of a stripe is then computed as $b^D(s_j) - b^U(s_j)$.

At this point, each quintuplet consisting of seed site, left-, right-, top-, and bottom-boundaries, defines uniquely a candidate stripe from a geometric viewpoint. We will denote candidate stripes as $S_1, \ldots, S_n$ to distinguish them from seed sites $s_1, \ldots, s_n$; an example is shown in Extended Data Fig. 5K.

Step 4: Signal analysis

To enrich a purely geometric structure with biological information, we compute descriptors that summarize information about the signal distribution – both inside and outside of candidate stripes.

- *Inner descriptors* We adopt the five-number summary, a set of descriptive statistics consisting of five sample percentiles:
  - The minimum, sometimes called Q_0 or 0th percentile.
  - First (Q_1), second (Q_2), and third (Q_3) quartiles – which consists of the 25th, 50th, and 75th percentile, respectively. The second quartile corresponds to the median; the first and third quartiles correspond to the median of the lower and the upper half of the dataset, respectively.
  - The maximum, sometimes called Q_4 or 100th percentile.

In addition, we consider the arithmetic mean and standard deviation of the signal within the stripe boundaries. We will call the latter as *mean inner signal*.

- *Outer descriptors*. We compute the statistics from the previous point using regions neighbouring the candidate stripe. More precisely, the *left k-neighbour* $\mathcal{N}_j^R$ consists of the first k columns to the left of S_j; analogously, the *right k-neighbour* $\mathcal{N}_j^R$ consists of the first k columns to the right of S_j. We will denote as $\mathcal{N}_j := \mathcal{N}_j^L \cup \mathcal{N}_j^R$ the *k-neighbour* of the j-th candidate stripe. The arithmetic mean of the signal inside the k-outer neighbour will be referred to as *mean k-outer signal*.

- *Mixed descriptors*. We define the *k-relative change* as the difference between mean inner and mean k-outer signals, divided by the mean k-outer signal.

Extended description of StripeBench

This section is devoted to provide additional information on the StripeBench benchmark, intended as a dataset equipped of a ground truth and a set of performance measures.

Dataset

To generate Hi-C matrices we consider a genome-wide run of MoDLE with loop extrusion simulated by using barriers imputed from RAD21 ChIP-Seq H1-hESC data, and with default settings except for the following three factors:

- Target contact density δ, i.e., the average contact density to be reached before the simulation is halted, here set to 1, 5, 10, and 15.

- Scale parameter σ of the generalized extreme value distribution, used to randomize contacts by additively introducing noise to molecular contacts, is set to 0, 5000, 10000, and 15000.

- Resolution r in base-pairs for the output contact matrix in cooler format, set to 5k, 10k, 25k and 50k.

The values chosen for the three factors lead to 16 Hi-C matrices, each one available at four resolutions for a total of 64 simulated matrices. The choice of target contact density and scale parameters in our study is driven by the objective of considering Hi-C matrices with diverse sequencing depths and increasing levels of noise, while still maintaining a manageable number of matrices. For a graphical illustration of the effects of these parameters on the contact maps, the reader is referred to Fig. 2A.

Ground truth

For each Hi-C matrix, the ground truth is obtained by retrieving loci with average extrusion barrier occupancy greater or equal than 0.70. Extrusion barrier occupancy consists of a scalar in $[0, 1]$, and is therefore interpretable as a probability value; intuitively, the higher this value, the more visible the stripe is, as shown in visual examples in Fig. 2B.

We then proceed by merging extrusion barriers mapping to the same bin at a given resolution – so that each locus appears only once; however, we consider the lower- and the upper-triangular parts of the contact map separately, so that a bin can yet be deemed relevant for both parts of the matrix, either individual parts or neither of them. Each of loci will be henceforth referred to as *anchor site* or *anchor point*.

Different measures for different problems

The capability of a tool to find meaningful stripes is here quantified by using classification and recognition measures. We hereby adopt the following terminology:

- *Classification* is the task of assigning labels to input data, such as categorizing genomic bins as either 0 or 1 based on whether or not they contain an anchor point. We therefore discard the notion of width and evaluate a method's ability to identify whether a certain bin *hosts (i.e., contains)* an anchor point or

not. For Chromosight and StripeCaller, obtaining the predicted classification vector is straightforward since these methods specifically set the stripe width to the resolution of the Hi-C matrix[3]; in the case of StripePy, we obtain the predicted classification vector by considering the coordinates of seed points as anchor point predictions; for Stripenn we consider the midpoint of each horizontal interval, since the method does not provide additional information to help picking a location.

- *Recognition* deals with the identification of rectangular regions[4] that host anchor points.

We decide to employ both formulations since neither of them alone is adequate for analysing the problem. Solely relying on recognition is insufficient to assess a set of tools because any partition of the Hi-C matrix can capture all anchor sites with no difficulty, albeit with a potentially excessive number of wide stripes. This limitation originates because it is not possible to obtain a ground truth notion of width or length. On the other hand, classification measures does not account that stripes are not unidimensional segments, but rather narrow rectangles.

The following "base" measures are adopted for the classification problem:

- *Sensitivity and Specificity* measure the ability of a method to correctly identify positives and negative cases, respectively. By *positives* we here mean bins that are flagged as anchor points, while *negatives* are bins that do not host any anchor. Sensitivity is also known as *True Positive Rate* (TPR) or *recall*, while specificity goes under the name of *True Negative Rate* (TNR).

- *Positive Predictive Value* (PPV), or *precision*, is the proportion of true positives to the total number of items classified as positives by the model.

Similarly, we introduce the following "base" recognition measures:

- *Anchor Hit Rate*, or AHR, computed as the fraction of anchor sites that lie in one of the candidate stripes. It is closely related to TPR, although we are here more permissive since we consider the estimated stripe width in its computation.

- *Fraction of Good Candidates*, or FGC, computed as the fraction predicted stripes hosting an anchor site. It resembles to PPV when accounting for the estimated stripe width.

It is worth noting that there is no metric corresponding to specificity, as we are only able to draw conclusions on candidate stripes (whether these are true or false positives) since we lack a notion of negative.

There are several metrics that are used to handle datasets with a severe skew in the class distribution. The following single-value metrics are adopted:

- *Balanced accuracy* and *Geometric Mean* are single-value metrics that combine sensitivity and specificity by computing their arithmetic or geometric mean. We denote them as bACC, and GM, respectively.

- *F1-score* and *Fowlkes–Mallows index*, here respectively denoted as F1c and FMc, are the harmonic and geometric means of precision and recall.

- *Jaccard Index* (JI), or Jaccard similarity coefficient, measures the similarity between true labels and predicted labels by taking the ratio between the number of true positives against the sum of correctly predicted positive samples and all incorrect predictions. This metric considers both false positives and false negatives in a balanced manner, excluding only the correctly predicted negative samples. It is particularly well-suited for detecting rare events, as it ensures that the evaluation is sensitive to the correct classification of infrequent positive instances without being dominated by numerous correct identifications of negative class instances.

In a similar manner, recognition measures are summarized by adapting F1-score and Fowlkes–Mallows index: in this adaptation, TPR is replaced by AHR and PPV is replaced by FGC. We denote them as F1r and FMr, respectively.

[3]These two methods return pairs of the form $\{i, i+r\}$, where r is the resolution and i is a genomic coordinate. We consider the bin of genomic coordinate i to host the anchor point.

[4]What stripes are, geometrically speaking

Benchmarking analysis

The following tools were installed for the comparative analyses: Chromosight version 1.6.3 [5], StripeCaller version 0.1.0 https://github.com/XiaoTaoWang/StripeCaller, Stripenn 1.1.65 [6].

Impact of the two filtering strategies

StripePy employs two filtering strategies. The first filtration takes place in Step 2 (see Methods), where local maximum points of the global pseudo-distribution are filtered by thresholding their topological persistence values. By applying this filtration to the global pseudo-distribution, we retain genomic bins with a stronger (uni-dimensional) linear pattern, as this process resemble the voting procedure from the Hough transform [3]. The effect of increasing thresholds on these local maximum points is illustrated in Extended Data Fig. 6A-C: in practice, requiring higher topological persistence prioritizes local maxima with greater height differences compared to their surrounding areas. Extended Data Fig. 6 D-E show the candidate stripes produced at the end of Step 3 (see Methods), where each seed site is enriched with estimates of stripe width and height. While acting on topological persistence already reduces the number of candidate stripes, it doesn't account for stripes being two-dimensional patterns. Therefore, a second filtering step is applied, which thresholds the relative change parameter – a score that compares the average signal inside and in the near proximity of a candidate stripe. The effect of a more stringent filtering on this parameter for the set of candidates of Extended Data Fig. 6E is displayed in Fig. G-I. As expected, the higher the threshold, the fewer candidates are returned.

Default values for these thresholds are provided with the software to simplify the user's work and enable a fair comparison with alternative stripe callers. However, optimal results may require adjusting these parameters based on the specific contact map. It is often best to run the tool on a chromosome and fine-tune the parameter values on a case-by-case basis to achieve the most accurate results.

Pre-processing of interaction matrices released by ENCODE and the 4DNucleome

Matrices in .mcool format published by the 4DNucleome are generated using a version of Cooler that leads to data corruption when generating the .mcool files by iterative coarsening a .cool file [5]. Thus, prior to using dataset 4DNFI6HDY7WZ and 4DNFI9GMP2J8 [7, 8] for any data analysis or benchmark, we restored the corrupted files using `hictk fix-mcool` [9].

Next, interaction matrices in .hic format were converted to .mcool format using `hictk convert` [9].

Code availability

StripePy is released to the public as an open source, MIT-licensed Python application.
StripePy source code is hosted on GitHub at https://github.com/paulsengroup/StripePy and is archived on Zenodo at DOI:10.5281/zenodo.14536416 [10].
StripePy can be easily installed from source or PyPI using pip https://pypi.org/project/stripepy-hic.
Containerized versions of StripePy are regularly published on DockerHub at https://hub.docker.com/r/paulsengroup/stripepy.

Code used for the benchmarks and data analyses presented in this article are hosted on a separate GitHub repository at https://github.com/paulsengroup/2024-stripepy-paper. A copy of the code in this repository is archived on Zenodo at DOI:10.5281/zenodo.14536323 [11].

Data availability

This study re-analyzed a number of publicly available datasets released by the 4DNucleome [12, 13] and ENCODE [14–17].

[5] Cooler data corruption: https://hictk.readthedocs.io/en/stable/file_validation.html#cooler-index-corruption

Hi-C and Micro-C datasets:

- 4DNFI6HDY7WZ [7, 18] - H1-hESC (in situ Hi-C; DpnII)
- 4DNFI9GMP2J8 [8, 18] - H1-hESC (Micro-C)
- ENCFF216QQM [19] - GM12878 (in situ Hi-C; DpnII)
- ENCFF993FGR [20] - GM12878 (intact MNase Hi-C)

CTCF ChIP-Seq datasets:

- ENCFF692RPA [21] - H1-hESC
- ENCFF796WRU [22] - GM12878

The results presented throughout the manuscript have been deposited on Zenodo at DOI:10.5281/zenodo.14449731 [23].

The benchmark developed as part of this publication and used to contrast the stripe callers considered in this manuscript, StripeBench, has also been deposited on Zenodo at DOI:10.5281/zenodo.14448329 [24].

Acknowledgements

We thank the 4DNucleome Network and the lab of Job Dekker for contributing several of the Hi-C and Micro-C datasets used as part of this study.

We thank the ENCODE Consortium and the labs of Erez Aiden and Michael Snyder for contributing with ENCODE data used as part of this study.

Furthermore, we would like to acknowledge Mr. Bendik Berg for his contributions in packaging StripePy and configuring the framework for unit testing.

Author contributions

Conceptualization: AR, RR, JP; algorithm design: AR; software development: AR, RR; benchmarking & data analysis: AR; writing-review, editing & final approval: AR, RR, JP; supervision: JP; funding acquisition: JP.

Funding

This work was supported by the Norwegian Research Council projects 324137 and 343102.

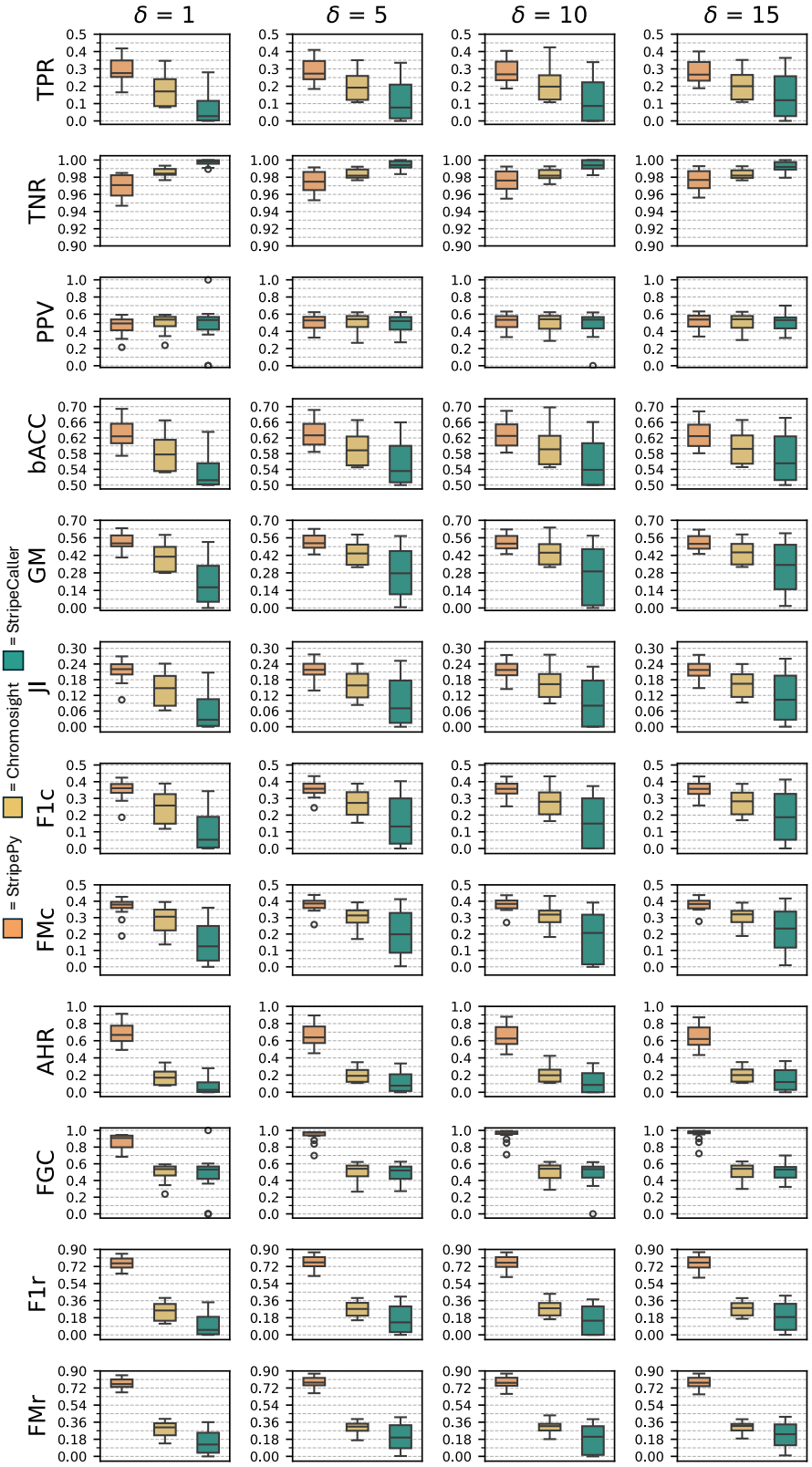
Competing interests

The authors declare no competing interests.

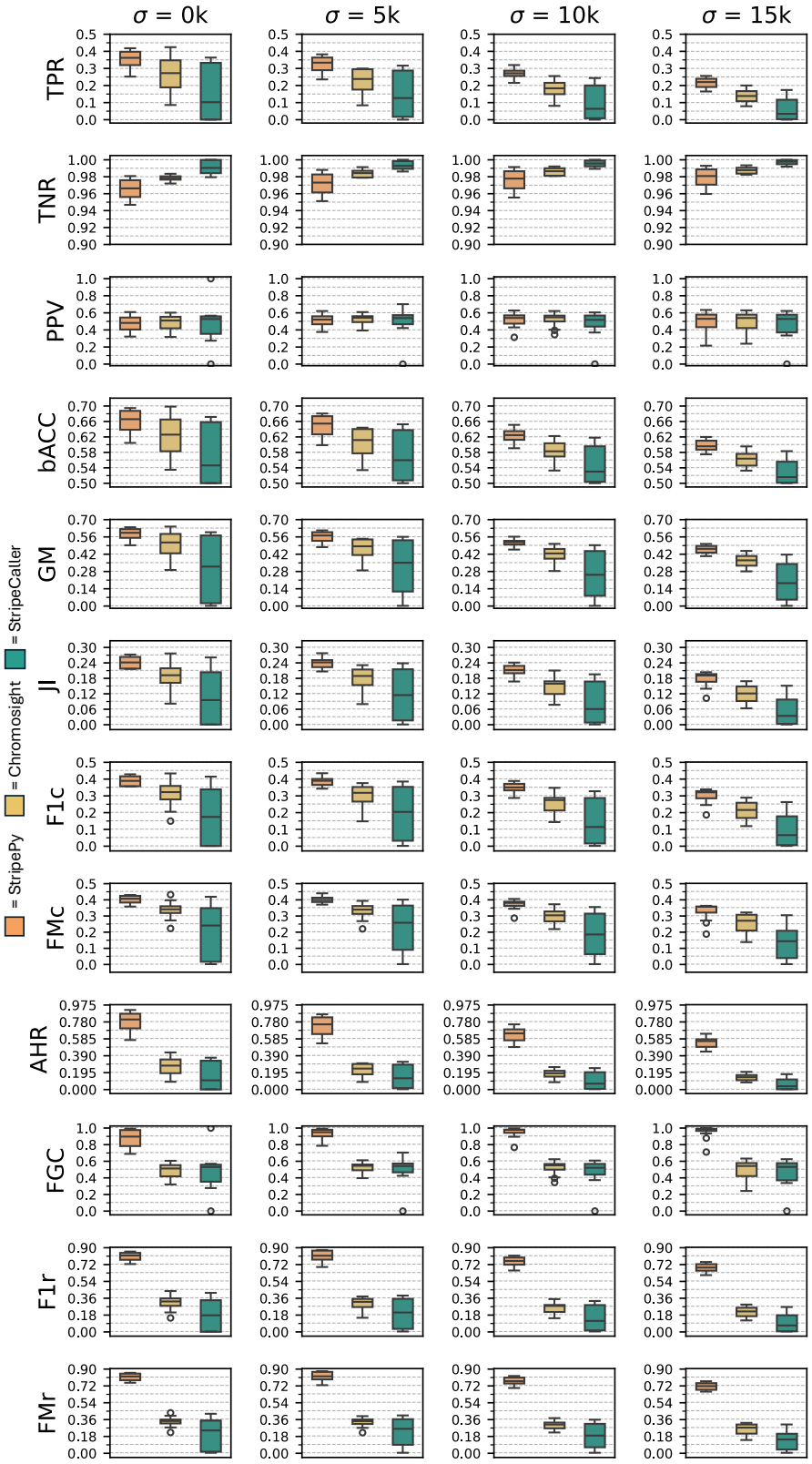
References

1. Raffo, A. & Biasotti, S. Weighted quasi-interpolant spline approximations: Properties and applications. *Numerical Algorithms* **87,** 819–847 (2021).
2. Raffo, A. & Biasotti, S. Weighted Quasi-Interpolant Spline Approximations of Planar Curvilinear Profiles in Digital Images. *Mathematics* **9** (2021).

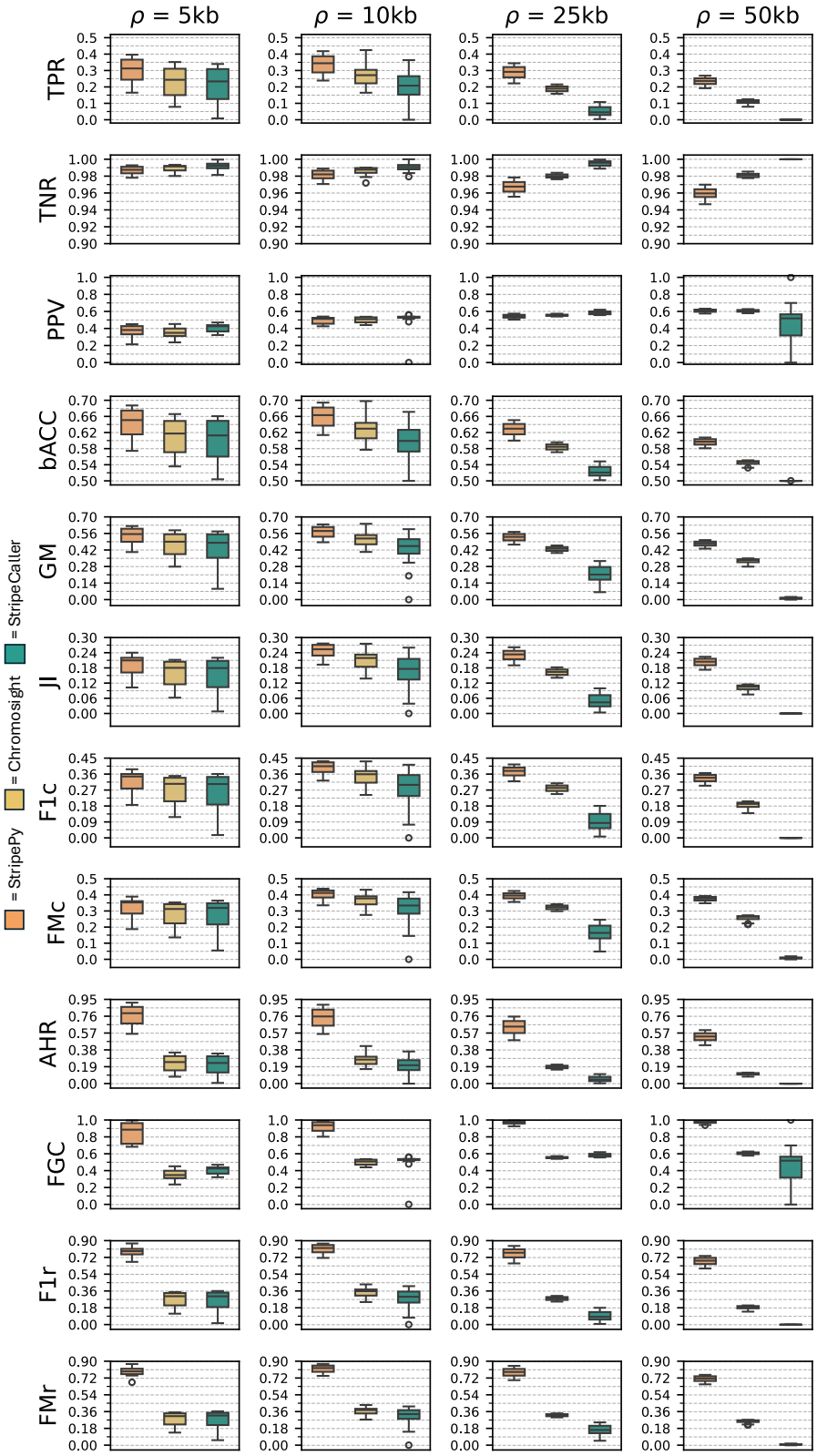
3. Hough, P. V. C. *Method and means for recognizing complex patterns* US Patent 3069654. Dec. 1962.
4. Mukhopadhyay, P. & Chaudhuri, B. B. A survey of Hough Transform. *Pattern Recognition* **48,** 993–1010 (2015).
5. Matthey-Doret, C. *et al.* Computer vision for pattern detection in chromosome contact maps. *Nature Communications* **11,** 5795–5795 (2020).
6. Sora Yoon, Aditi Chandra & Vahedi, G. Stripenn detects architectural stripes from chromatin conformation data using computer vision. *Nature Communications* **13** (2022).
7. *4DNFI6HDY7WZ.mcool; 4DN Data Portal — identifiers.org* https://identifiers.org/4dn:4DNFI6HDY7WZ. [Accessed 06-12-2024].
8. *4DNFI9GMP2J8.mcool; 4DN Data Portal — identifiers.org* https://identifiers.org/4dn:4DNFI9GMP2J8. [Accessed 06-12-2024].
9. Rossini, R. & Paulsen, J. hictk: blazing fast toolkit to work with .hic and .cool files. *Bioinformatics* **40,** btae408 (June 2024).
10. Rossini, R., Raffo, A. & Berg, B. *paulsengroup/StripePy: v0.0.2* version v0.0.2. Dec. 2024. https://doi.org/10.5281/zenodo.14536416.
11. Raffo, A. & Rossini, R. *paulsengroup/2024-stripepy-paper: v1.0.0* version v1.0.0. Dec. 2024. https://doi.org/10.5281/zenodo.14536323.
12. Dekker, J. *et al.* The 4D nucleome project. *Nature* **549,** 219–226. http://dx.doi.org/10.1038/nature23884 (Sept. 2017).
13. Reiff, S. B. *et al.* The 4D Nucleome Data Portal as a resource for searching and visualizing curated nucleomics data. *Nature Communications* **13** (May 2022).
14. An integrated encyclopedia of DNA elements in the human genome. *Nature* **489,** 57–74 (Sept. 2012).
15. Luo, Y. *et al.* New developments on the Encyclopedia of DNA Elements (ENCODE) data portal. *Nucleic Acids Research* **48,** D882–D889 (Nov. 2019).
16. Kagda, M. S. *et al. Data navigation on the ENCODE portal* 2023. https://arxiv.org/abs/2305.00006.
17. Hitz, B. C. *et al.* The ENCODE Uniform Analysis Pipelines. http://dx.doi.org/10.1101/2023.04.04.535623 (Apr. 2023).
18. Krietenstein, N. *et al.* Ultrastructural Details of Mammalian Chromosome Architecture. *Molecular Cell* **78,** 554–565.e7 (May 2020).
19. *ENCFF216QQM; ENCODE — identifiers.org* https://identifiers.org/encode:ENCFF216QQM. [Accessed 06-12-2024].
20. *ENCFF993FGR; ENCODE — identifiers.org* https://identifiers.org/encode:ENCFF993FGR. [Accessed 06-12-2024].
21. *ENCFF692RPA; ENCODE — identifiers.org* https://identifiers.org/encode:ENCFF692RPA. [Accessed 06-12-2024].
22. *ENCFF796WRU; ENCODE — identifiers.org* https://identifiers.org/encode:ENCFF796WRU. [Accessed 06-12-2024].
23. Raffo, A. *Data analysis results for StripePy manuscript* version v1.0.0 (Zenodo, Dec. 2024). https://doi.org/10.5281/zenodo.14449731.
24. Raffo, A. *StripeBench: a benchmark to compare architectural stripe callers* version v1.0.1 (Zenodo, Dec. 2024). https://doi.org/10.5281/zenodo.14535653.



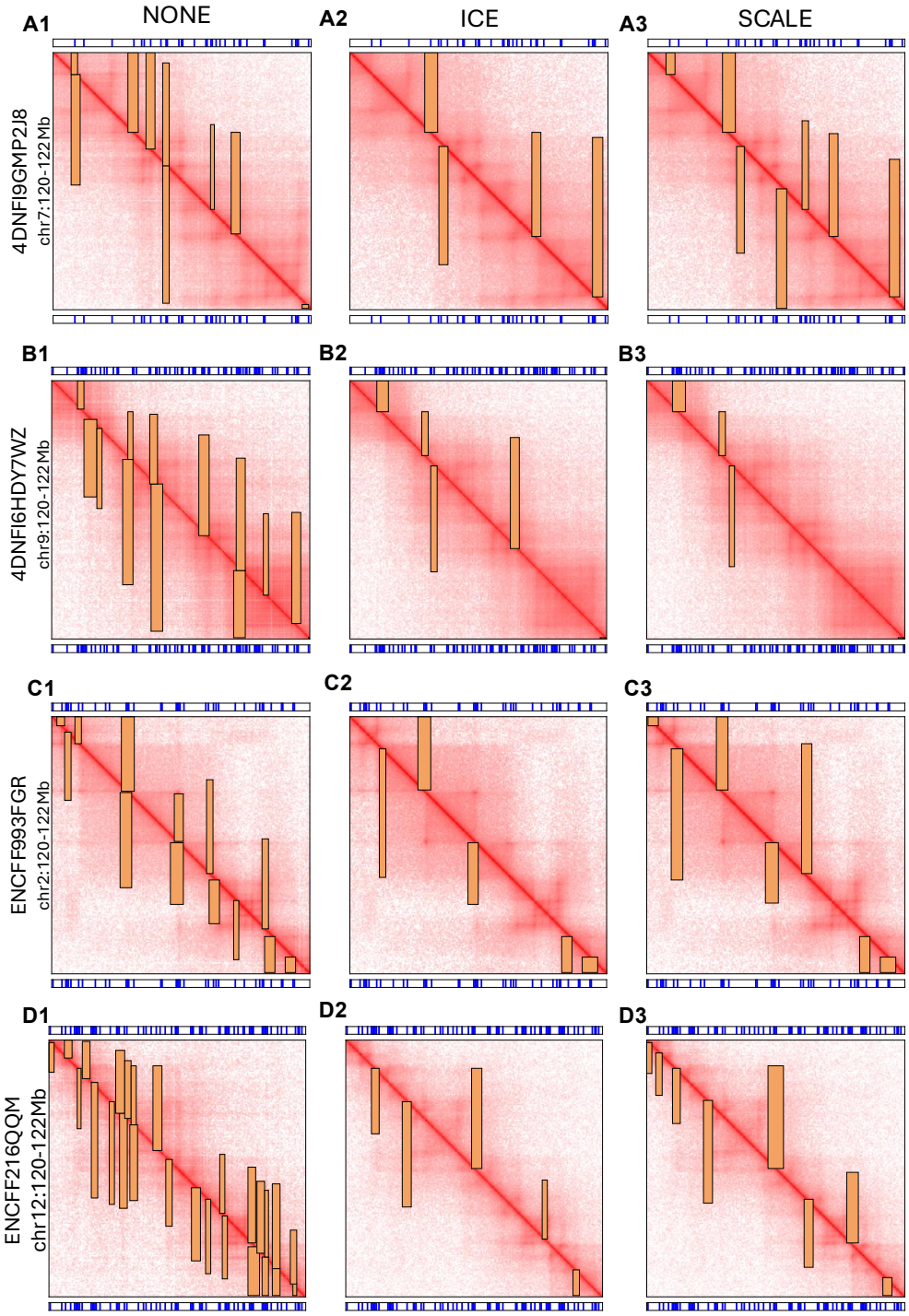
Extended Data Fig. 1. | Change in contact density for the StripeBench's benchmark. Boxplots of the classification (TPR, TNR, PPV, bACC, GM, JI, F1c, and FMc) and recognition (AHR, FGC, F1r, and FMr) measures by change in contact density.



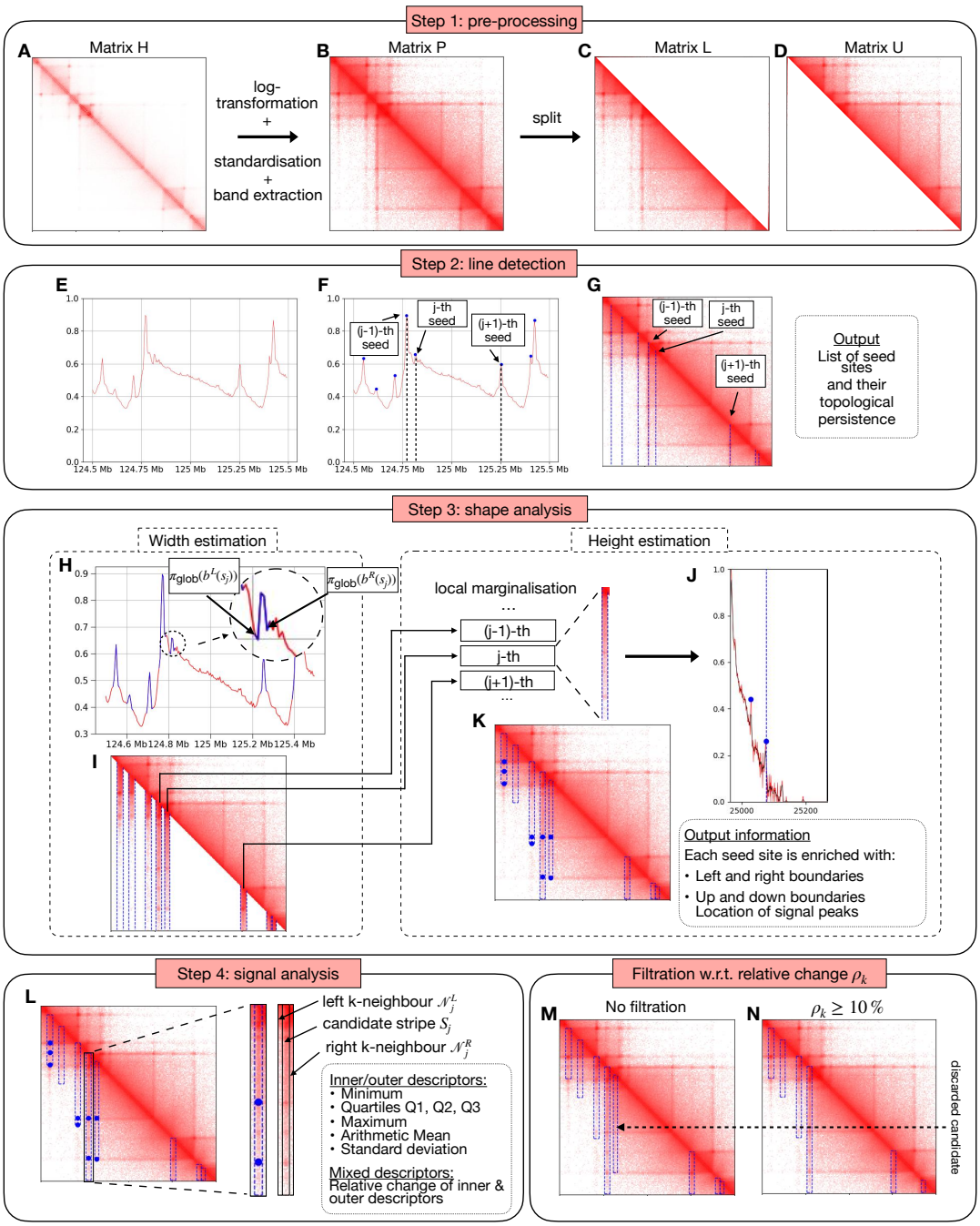
Extended Data Fig. 2. | Change in noise level for the StripeBench's benchmark. Boxplots of the classification (TPR, TNR, PPV, bACC, GM, JI, F1c, and FMc) and recognition (AHR, FGC, F1r, and FMr) measures by change in noise level.



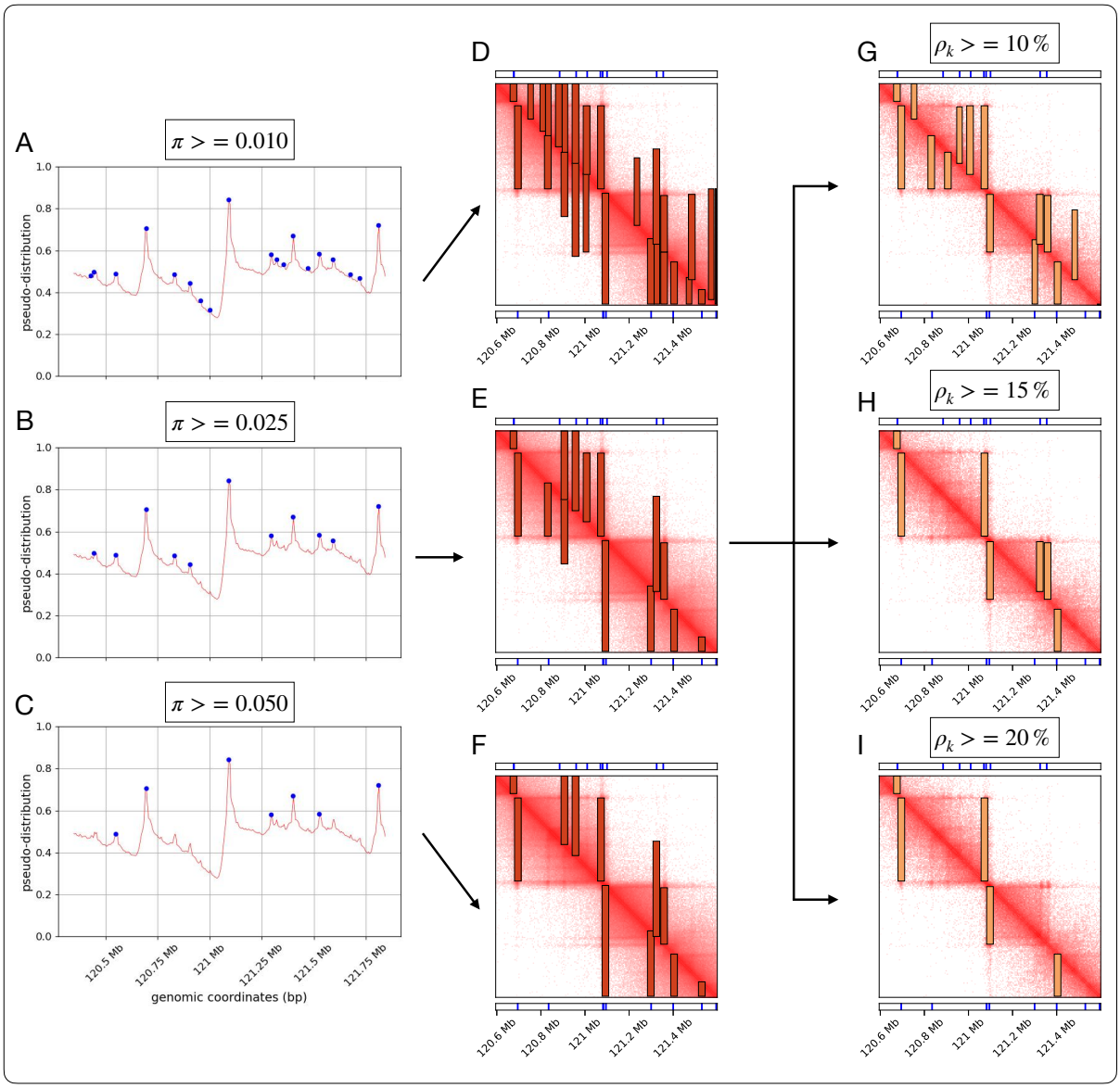
Extended Data Fig. 3. | Change in resolution for the StripeBench's benchmark. Boxplots of the classification (TPR, TNR, PPV, bACC, GM, JI, F1c, and FMc) and recognition (AHR, FGC, F1r, and FMr) measures by change in resolution.



Extended Data Fig. 4. | Effects of normalization. The plots represent four regions from as many contact maps. The upper and lower bands point at bin hosting CTCF sites, obtained from ChIP-Seq data. The images of the contact maps have superimposed the predictions produced by StripePy when applied to three conditions: no normalization (NONE), ICE and SCALE. For each map, the region is repeated three times to improve clarity: from left to right, columns refer to NONE, ICE, and SCALE.



Extended Data Fig. 5. | Extended overview of StripePy. Step 1: **A** region of an input contact map as it is, **B** after global enhancement via log-transformation, and **C-D** final split into lower- and upper-triangular parts; the remaining steps focused on on the lower-triangular part, to keep the plots readable. Step 2: **E** constrained pseudo-distribution obtained by marginalizing L; **F** constrained pseudo-distribution with superimposed persistent maximum points, a.k.a. seeds; **G** pre-processed Hi-C matrix with superimposed vertical lines corresponding to persistent seeds. Step 3: **H** study of neighbourhoods of seed sites allow to introduce notions of left and right boundaries, which can be then used to slice the pre-processed Hi-C matrix **I**; **J** local pseudo-distributions (original: red curve, smoothed: black curve) allow to introduce up/down boundaries (dashed blue line), as well as to detect the location signal peaks (blue dots); **K** overlap rectangles corresponding to candidate stripes, together with location of signal peaks, to the pre-processed Hi-C matrix. Step 4: **L** shows a region neighbouring a candidate stripe and list the sets of inner/outer/mixed descriptors. The initial set of candidate stripes, represented in **M**, is reduced by thresholding one of these descriptors, namely the relative change parameter **N**.



Extended Data Fig. 6. | StripePy's persistence vs. relative change filterings. Topological persistence rules out local maximum points (**A**, **B**, and **C**) in terms of the strength of the linear pattern that each bin exhibit. Three thresholds therefore produce as many collections of candidate stripes (**D**, **E**, and **F**) at the end of Step 3. In Step 4, thresholding relative changes further restricts the number of predicted stripes, this time on the basis of a metric that takes into account a well-defined rectangular region (**G**, **H**, and **I**).