# Store, retrieve and plot Hi-C data with HiCBricks

Koustav Pal
IFOM

## Contents

## 1 Introduction

HiCBricks is a storage and retrieval library for high-resolution Hi-C data. This library attempts to encapsulate the entire HDF data structure and present accessor functions allowing users to access and manipulate Hi-C data without having to deal with the structural complexity associated to dealing with HDF files. Please note, HiCBricks is not meant to compete with other data formats. Instead it is a library to better facilitate access to Hi-C data stored in various formats. Therefore, it is not possible at this time to create a direct mirror of for example, mcool files. mcool files, for the uninitiated are a standard data format for the 4D nucleome project and better facilitates the storage of very large experiments within a single data file. HiCBricks, on the other hand stores the complete matrix, as opposed to the upper triangle in mcools files. As you will see in this vignette, it is a very simple task to import entire resolutions/normalizations from a standard 4DN mcool file to a Brick object. With HiCBricks, in this vignette you will;

1. Create a brick for a single chromosomes
2. do TAD calls on this brick files
3. keep the tad calls (or any ranges object) associated to the Brick file for future reference
4. and later on retrieve them for quick plotting and viewing with HiCBrick functions

## 2 Loading 4DN mcool files as Brick objects

In this exercise we will download a file from the 4DN data portal at `https://data.4dnucleome.org/`. For the purposes of this vignette we will use a randomly chosen H1-hESC Hi-C data using DpnII: `https://data.4dnucleome.org/files-processed/4DNFI7JNCNFB/@@download/4DNFI7JNCNFB.mcool`
You can download the file using curl. It contains normalised Hi-C data on H1-hESC cells using the DpnII enzyme.

```
> curl_download(url = "https://data.4dnucleome.org/files-
+     processed/4DNFI7JNCNFB/@@download/4DNFI7JNCNFB.mcool",
+     destfile = "H1-hESC-HiC-4DNFI7JNCNFB.mcool")
```

Note that there are a few normalisation weights available within the sample. We can check what normalisation weights are available using

Brick_list_mcool_normalisations() function. Please note, that this function does not list the normalisation weights available within the mcool file, but rather lists normalisation factors that HiCBricks accepts from the mcool files. Notice, the human readable names and the actual dataset names, what you are interested in are the human readable names, which makes the type of normalisation apparent. Such as for example, KR is an abbreviation for the Knight Ruitz matrix balancing algorithm introduced by Rao et al., 2014 [1].

```
> Brick_list_mcool_normalisations(names.only = TRUE)
```

This lists only the human-readable names. To see both the abbreviation and the base column name, will list all names including their linked columns in the HDF file.

```
> Brick_list_mcool_normalisations(names.only = FALSE)
```

The 4D nucleome project bins their data into several different resolutions, to check out the available resolutions, it is as simple as:

```
> mcoolName="H1-hESC-HiC-4DNFI7JNCNFB.mcool"
> Brick_list_mcool_resolutions(mcool = filename)
```

Once, you have viewed all of the listed resolutions, we can go ahead and create Brick bricks from these files. Please note, that users cannot load multiple resolutions in a single Brick file, this is because the aim of HiCBricks is not to create an alternative storage format to already very well designed pre-existing formats, but rather to create a straightforward, memory efficient storage and retrieval library for already existing Hi-C data formats in R.
Although it is possible to load all chromosomes in a single Brick object, this is not recommended because,

1. It is not possible for multiple processes to concurrently access the same HDF file. Therefore, it hinders parallelization.
2. The presence of multiple matrices in the same file, results in an increase in read and write speeds.

Instead when using very large matrices, users are encouraged to separate the matrices chromosome by chromosome into different brick objects. For the purposes of this vignette, we want to do TAD calls with the local score differentiator (LSD) module and finally plot these TAD calls. We can now go ahead and create a Brick corresponding for a single chromosome from the downloaded mcool file and import data relevant to the cis-interaction maps for that chromosome.

```
> Output.brick <- "H1-hESC-HiC-4DNFI7JNCNFB-10000-ICE-
+     normalised-chr1.brick"
> mcool <- mcoolName
> CreateBrick_from_mcool(Brick = Output.brick,
+     mcool = mcool,
+     binsize = 10000,
+     chrs = "chr1")
> Brick_load_data_from_mcool(Brick = Output.brick,
+     mcool = mcool,
+     chr1 = "chr1",
+     chr2 = "chr1",
+     binsize = 10000,
+     cooler.batch.size = 1000000,
+     matrix.chunk = 2000,
+     dont.look.for.chr2 = TRUE,
+     remove.prior = TRUE,
+     norm.factor = "Iterative-Correction")
```

The first function creates the basic Brick structure, whereas the second function loads data into the structure. Using the param 'chrs', users can limit the structure created to the relevant chromosomes or if left NULL, will create the structure for all chromosome pairs. Please note, that if the length of chrs is 2, 4 interaction maps will be created. Two for the 'cis' interaction maps and two for the 'trans' interaction maps.
Notice, that there are a few options allowing users to manipulate data read and write speeds. 'cooler.batch.size' determines the number of records read per iteration through an mcool file. 'matrix.chunk' determines the size of the matrix square that will be loaded per iteration through an mcool file. If you are loading 'cis' matrices, it is recommended to set the 'dont.look.for.chr2' parameter to TRUE, as the first read records for chr1 will always correspond to those originating from chr2. In cases of trans matrices, this option should be set to FALSE allowing the function to locate the first occurence of a chr1 vs chr2 interaction. 'remove.prior' defaults to FALSE and prevents users from loading datasets twice.

## 3 Loading other datasets as Brick objects

Currently, HiCBricks only supports data import from 2D matrices and mcool files with more to come. If you have a 2D matrix, you can load it like:

```
> library("HiCBricks")
> Bintable.path <- system.file("extdata",
+     "Bintable_40kb.txt",
+     package = "HiCBricks")
> Chromosomes <- "chr19"
> Output.Filename <- "test.hdf"
> Path_to_cached_file <- CreateBrick(ChromNames = Chromosomes,
+     BinTable = Bintable.path,
+     bin.delim = " ",
+     Output.Filename = Output.Filename,
+     exec = "cat",
+     remove.existing = TRUE)
> Test.mat <- matrix(NA,nrow = 800, ncol = 800)
> Row <- row(Test.mat)
> Col <- col(Test.mat)
> Dist <- Col - Row
> Matrix.file <- "Test_matrix.txt"
> write.table(x = Dist,
+     file = Matrix.file,
+     sep = " ",
+     quote = FALSE,
+     row.names = FALSE,
+     col.names = FALSE)
> Brick.file <- Path_to_cached_file
> Brick_load_matrix(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19",
+     matrix.file = Matrix.file,
+     delim = " ",
+     exec = "cat",
+     remove.prior = TRUE)
```

```
[1] TRUE
```

If you have a very large 2D cis matrix, you can load data till a certain distance:

```
> Brick_load_cis_matrix_till_distance(Brick = Brick.file,
+     chr = "chr19",
+     matrix.file = Matrix.file,
+     delim = " ",
+     distance = 100,
+     remove.prior = TRUE)
```

```
[1] TRUE
```

## 4 Accessing ranges objects in a brick store

HiCBricks implementes different fetch/get methods. Users can list attributes of a Brick object and they can fetch the same objects. Users have access to Ranges objects and matrix subset operations.

```
> Brick.file <- system.file("extdata",
+     "test.hdf",
+     package = "HiCBricks")
> Brick_list_rangekeys(Brick.file)
```

```
[1] "Bintable"    "test_ranges"
```

This lists the available rangekeys in the Brick file. Alternatively, tf you are interested in only the bintable associated to the Hi-C experiment, you can:

```
> Brick_get_bintable(Brick.file)
```

```
GRanges object with 1479 ranges and 0 metadata columns:
                          seqnames            ranges strand
                             <Rle>         <IRanges>  <Rle>
            chr19:1:40000    chr19           1-40000      *
        chr19:40001:80000    chr19       40001-80000      *
       chr19:80001:120000    chr19      80001-120000      *
      chr19:120001:160000    chr19     120001-160000      *
      chr19:160001:200000    chr19     160001-200000      *
                      ...      ...               ...    ...
  chr19:58960001:59000000    chr19 58960001-59000000      *
  chr19:59000001:59040000    chr19 59000001-59040000      *
  chr19:59040001:59080000    chr19 59040001-59080000      *
  chr19:59080001:59120000    chr19 59080001-59120000      *
  chr19:59120001:59160000    chr19 59120001-59160000      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Otherwise, you can retrieve the object using Brick_get_ranges the method called by Brick_get_bintable.

```
> Brick_get_ranges(Brick = Brick.file,
+     rangekey = "Bintable")
```

```
GRanges object with 1479 ranges and 0 metadata columns:
                          seqnames            ranges strand
                             <Rle>         <IRanges>  <Rle>
            chr19:1:40000    chr19           1-40000      *
```

```
        chr19:40001:80000    chr19       40001-80000      *
       chr19:80001:120000    chr19      80001-120000      *
      chr19:120001:160000    chr19     120001-160000      *
      chr19:160001:200000    chr19     160001-200000      *
                      ...      ...               ...    ...
  chr19:58960001:59000000    chr19 58960001-59000000      *
  chr19:59000001:59040000    chr19 59000001-59040000      *
  chr19:59040001:59080000    chr19 59040001-59080000      *
  chr19:59080001:59120000    chr19 59080001-59120000      *
  chr19:59120001:59160000    chr19 59120001-59160000      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

You can also subset the retrieved ranges by the chromosome of interest.

```
> Brick_get_ranges(Brick = Brick.file,
+     rangekey = "Bintable",
+     chr = "chr19")
```

```
GRanges object with 1479 ranges and 0 metadata columns:
                          seqnames            ranges strand
                             <Rle>         <IRanges>  <Rle>
            chr19:1:40000    chr19           1-40000      *
        chr19:40001:80000    chr19       40001-80000      *
       chr19:80001:120000    chr19      80001-120000      *
      chr19:120001:160000    chr19     120001-160000      *
      chr19:160001:200000    chr19     160001-200000      *
                      ...      ...               ...    ...
  chr19:58960001:59000000    chr19 58960001-59000000      *
  chr19:59000001:59040000    chr19 59000001-59040000      *
  chr19:59040001:59080000    chr19 59040001-59080000      *
  chr19:59080001:59120000    chr19 59080001-59120000      *
  chr19:59120001:59160000    chr19 59120001-59160000      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

## 4.1 Identifying matrix row/col using ranges operations

Users may sometimes find it useful to identify the corresponding matrix row/col for a particular coordinate. Then it it as simple as:

```
> testRun=Brick_return_region_position(Brick = Brick.file,
+     region = "chr19:5000000:10000000")
> head(testRun)
```

```
[1] 126 127 128 129 130 131
```

This does a 'within' overlap operation and returns the corresponding coordinates. Therefore, sometimes when the region of interest is smaller than the ranges corresponding to the particular matrix region of interest, this function may fail. To have more fine-grain control, users may choose to use Brick_fetch_range_index which is called by Brick_return_region_position.

```
> testRun=Brick_fetch_range_index(Brick = Brick.file,
+     chr = "chr19",
+     start = 5000000,
+     end = 10000000)
```

This function will return a GRanges object containing one row for each element in the provided [chr,start,end] vectors, with a 'NumericList' column 'Indexes' corresponding to the overlapping row/col coordinate of that matrix.

## 5 Accessing matrices in a brick store

There are three ways to subset matrices.

- By distance
- Selecting sub-matrices
- Selecting rows or columns

### 5.1 Retrieving points separated by a certain distance

It is possible to get the interactions between genomic loci separated by a certain distance.

```
> Values <- Brick_get_values_by_distance(Brick = Brick.file,
+     chr = "chr19",
+     distance = 4)
```

Users can also choose to transform it during retrieval

```
> Failsafe_median_log10 <- function(x){
+     x[is.na(x) | is.nan(x) | is.infinite(x)] <- 0
+     return(median(log10(x+1)))
+ }
> Brick_get_values_by_distance(Brick = Brick.file,
+     chr = "chr19",
+     distance = 4,
+     FUN = Failsafe_median_log10)
```

```
[1] 1.470637
```

They can even subset the values by a certain region of interest

```
> Failsafe_median_log10 <- function(x){
+     x[is.na(x) | is.nan(x) | is.infinite(x)] <- 0
+     return(median(log10(x+1)))
+ }
> Brick_get_values_by_distance(Brick = Brick.file,
+     chr = "chr19",
+     distance = 4,
+     constrain.region = "chr19:1:5000000",
+     FUN = Failsafe_median_log10)
```

```
[1] 1.664331
```

## 5.2 Retrieving subsets of a matrix

HiCBricks, implements word-alike retrieval of sub-matrices. That means, you can retrieve data using coordinate in coordinate like fashion.

```
> Sub.matrix <- Brick_get_matrix_within_coords(Brick = Brick.file,
+     x.coords="chr19:5000001:10000000",
+     force = TRUE,
+     y.coords = "chr19:5000001:10000000")
```

This is the same as:

```
> x.axis <- 5000000/40000
> y.axis <- 10000000/40000
> Sub.matrix <- Brick_get_matrix(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19",
+     x.vector = c(x.axis:y.axis),
+     y.vector = c(x.axis:y.axis))
```

Notice, that this selection is not the same as the previous one and it has one more row and column. This is because the region of interest spans from 5000001:10000000, which starts from the 'x.axis + 1' and not from 'x.axis'.
Finally, it is also possible to fetch entire rows and columns. Users can do so either with names, which correspond to names of the matrix rows/cols from the bintable. If these are names, it is required to specify 'by = "ranges" '.

```
> Coordinate <- c("chr19:1:40000","chr19:40001:80000")
> Brick.file <- system.file("extdata",
+     "test.hdf",
+     package = "HiCBricks")
> Test_Run <- Brick_fetch_row_vector(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19",
+     by = "ranges",
+     vector = Coordinate,
+     regions = c("chr19:1:1000000", "chr19:40001:2000000"))
```

Users can also choose to fetch data 'by = "positions" '.

```
> Coordinate <- c(1,2)
> Brick.file <- system.file("extdata",
+     "test.hdf",
+     package = "HiCBricks")
> Test_Run <- Brick_fetch_row_vector(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19",
+     by = "position",
+     vector = Coordinate,
+     regions = c("chr19:1:1000000", "chr19:40001:2000000"))
```

If regions is provided, it will subset the corresponding row/col by the specified region. *regions* must be in coordinate format as shown below.

## 5.3 Accessing matrix metadata columns

There are several metrics which are computed at the time of matrix load. Principally,

- *bin.coverage* quantifies the proportion of non-zero rows/cols
- *row.sums* quantifies the total signal value of any row
- *sparsity* quantifies the proportion of non-zero values at a certain distance from the diagonal

Sparsity is only quantified if a matrix is defined as sparse during matrix load.

Users can check the names of the various matrix metadata columns.

```
> Brick_list_matrix_mcols()
```

```
       bin.cov        row.sums          sparse
"bin.coverage"      "row.sums"      "sparsity"
```

And then fetch one such metadata column

```
> Brick.file <- system.file("extdata",
+     "test.hdf",
+     package = "HiCBricks")
> testRun <- Brick_get_matrix_mcols(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19",
+     what = "row.sums")
>
```

# 6 Matrix utility functions

There are several utility functions that a user may take advantage of to do various checks.

## 6.1 Check if a matrix has been loaded into the Brick store

```
> Brick_matrix_isdone(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1] TRUE
```

## 6.2 Check if a matrix was defined as a sparse matrix

```
> Brick_matrix_issparse(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1] FALSE
```

## 6.3 Check the maximum distance until which a matrix was loaded

```
> Brick_matrix_maxdist(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1] 1479
```

## 6.4 Check if a matrix was defined

```
> Brick_matrix_minmax(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1]   0.0000 674.4584
```

## 6.5 Check the minimum and maximum values of a matrix

```
> Brick_matrix_minmax(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1]   0.0000 674.4584
```

### 6.6 Get the matrix dimensions, irrespective of the maxdist value

```
> Brick_matrix_dimensions(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1] 1479 1479
```

### 6.7 Get the filename of a loaded matrix

```
> Brick_matrix_filename(Brick = Brick.file,
+     chr1 = "chr19",
+     chr2 = "chr19")
```

```
[1] IMR90_RepA_ICEd_40000_chr19.mat.gz
Levels: IMR90_RepA_ICEd_40000_chr19.mat.gz
```

## 7 Call Topologically Associated Domains with Local score differentiator (LSD)

Local score differentiator (LSD) is a TAD calling procedure based on the directionality index introduced by Dixon et al., 2012 [2]. It partitions the genome into segments based on the local directionality index distribution. We first introduced this procedure with our study Pal et al., 2018 [?]. This has been adapted to work with HiCBricks to show how different analysis procedures can take advantage of the HiCBricks accessor functions.

```
> Brick.file <- system.file("extdata",
+     "test.hdf",
+     package = "HiCBricks")
> Chromosome <- "chr19"
> di_window <- 10
> lookup_window <- 30
> TAD_ranges <- Brick_local_score_differentiator(Brick = Brick.file,
+     chrs = Chromosome,
+     di.window = di_window,
+     lookup.window = lookup_window,
+     strict = TRUE,
+     fill.gaps=TRUE,
+     chunk.size = 500)
```

'lookup.window' value corresponds to the local window used to subset the directionality index distribution. Setting 'strict' to TRUE, adds another additional filter wherein the directionality index is required to be less than or greater than 0 at potential change points identifying a domain boundary. LSD works by identifying domain starts and ends, if a particular domain start was not identified, but the adjacent domain end was identified, 'fill.gaps' if set to TRUE, will infer the adjacent bin from the adjacent domain end as a domain start bin and create a domain. Any domains identified by 'fill.gaps' are annotated under the 'level' column in the resulting GRanges object with the value 2. 'chunk.size' corresponds to the size of the square to retrieve and process per iteration.
These TAD calls can also be stored along with the Brick file.

```
> Name <- paste("LSD",
+     di_window,
+     lookup_window,
+     Chromosome,sep = "_")
> Brick_add_ranges(Brick = Path_to_cached_file,
+     ranges = TAD_ranges,
+     rangekey = Name)
```

```
[1] TRUE
```

## 8 Fetching associated ranges from a Brick file

Since a brick store is an on-disk database, it is possible to fetch ranges objects associated to the brick store. Users can first list the available ranges objects. Once, they have identified the available rangekeys, users can fetch the required rangekeys.

```
> Brick_list_rangekeys(Brick = Path_to_cached_file)
```

```
[1] "Bintable"        "LSD_10_30_chr19"
```

```
> TAD_ranges <- Brick_get_ranges(Brick = Path_to_cached_file,
+     rangekey = Name)
```

## 9 Creating pretty heatmaps using HiCBricks

Using HiCBricks functions, users can plot pretty Hi-C heatmaps. In the following we list the most basic commands required to generate a heatmap.

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal.pdf",
+     Bricks = Brick.file,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     palette = "Reds",
+     width = 10,
+     height = 11,
+     return.object=TRUE)
```

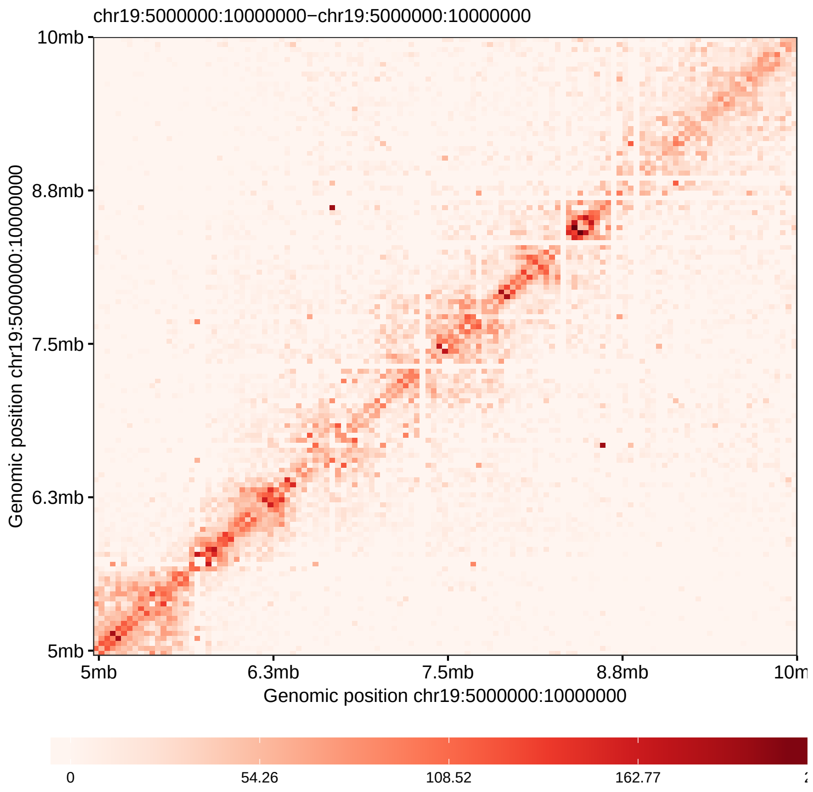

Figure 1: Simple heatmap generated with HiCBricks.

Notice the 'palette' argument. It requires the user to provide a palette name from either the 'RColorBrewer' or 'viridis' colour palettes. It is not possible at this time to provide user defined colour palettes.
Since we are directly plotting the Hi-C signal, the colours may seem a bit muted (Figure 1). We should go ahead and change that with a $log_{10}$ transformation, which will squeeze the signal distribution and make it pretty (Figure 2).

```
> Failsafe_log10 <- function(x){
+     x[is.na(x) | is.nan(x) | is.infinite(x)] <- 0
+     return(log10(x+1))
+ }
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal2.pdf",
+     Bricks = Brick.file,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     FUN = Failsafe_log10,
+     legend.title = "Log10 Hi-C signal",
+     palette = "Reds",
+     width = 10,
+     height = 11,
+     return.object=TRUE)
```

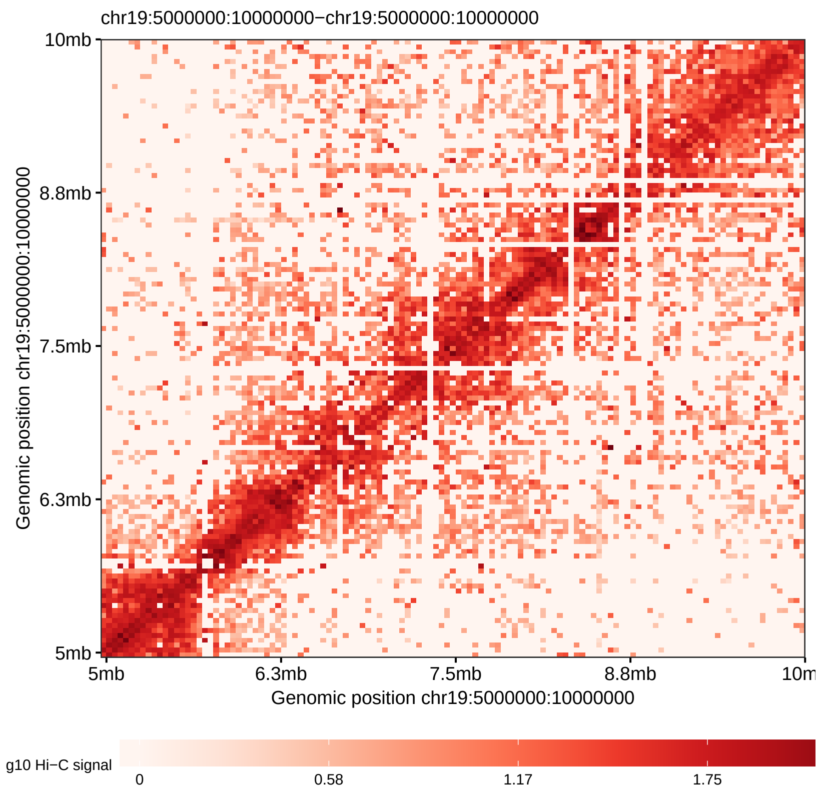

Figure 2: Heatmap using $log_{10}$ transformation for signal distribution.

Notice, how we created a new function for $log_{10}$ transformation. This function and others like it can be provided with the argument 'FUN'. This is an already much more dense heatmap.
Sometimes, the Hi-C signal distribution is biased by extreme values which tends

to blow up the entire distribution in a heatmap plot. We can pull in these values to create a more uniform and prettier picture, with the 'value.cap' argument.
'value.cap' takes as input a value ranging from 0,1 identifying the quantile at which the threshold will be applied. Also note, how the presence of this argument triggers presence of the greater than or less than sign (Figure 3).
Sometimes, it is desirable to plot the heatmap as a rotated heatmap.
But this looks ugly (Figure 4). To fix it we need to modify the 'width' and 'height' as the rotated plots are broader than they are taller (Figure 5).
We can now also plot the TADs on these plots.

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal3.pdf",
+     Bricks = Brick.file,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     FUN = Failsafe_log10,
+     value.cap = 0.99,
+     legend.title = "Log10 Hi-C signal",
+     palette = "Reds",
+     width = 10,
+     height = 11,
+     return.object=TRUE)
```

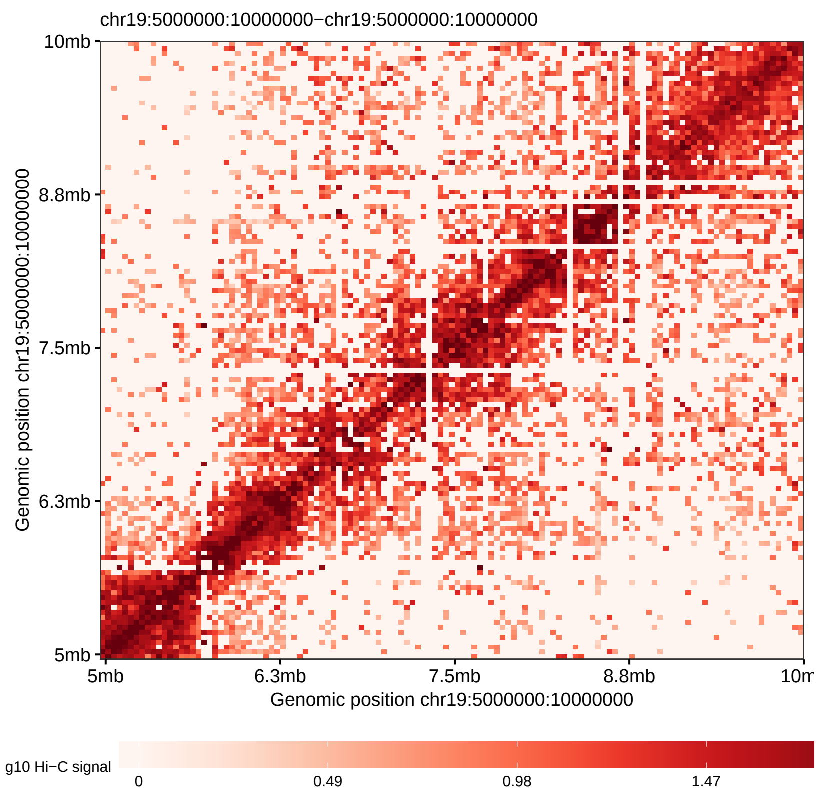

Figure 3: Heatmap using *value.cap* argument

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal4.pdf",
+     Bricks = Brick.file,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     FUN = Failsafe_log10,
+     value.cap = 0.99,
+     legend.title = "Log10 Hi-C signal",
+     palette = "Reds",
+     width = 10,
+     height = 11,
+     rotate = TRUE,
+     return.object=TRUE)
```

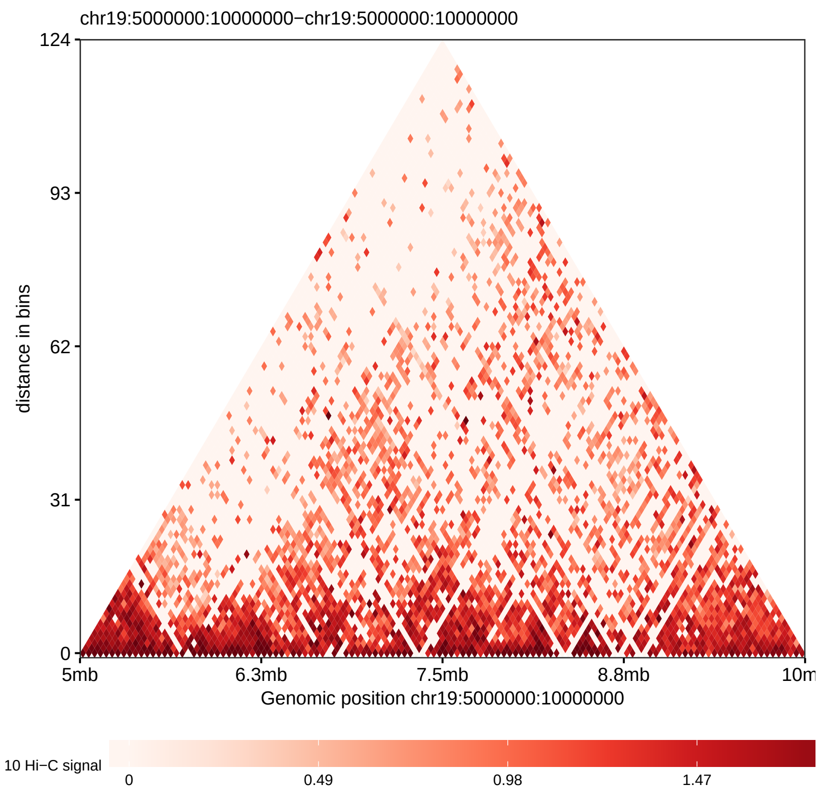

Figure 4: Rotated heatmap created by HiCBricks.

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal5.pdf",
+     Bricks = Brick.file,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     FUN = Failsafe_log10,
+     value.cap = 0.99,
+     legend.title = "Log10 Hi-C signal",
+     palette = "cividis",
+     width = 15,
+     height = 5,
+     rotate = TRUE,
+     return.object=TRUE)
```

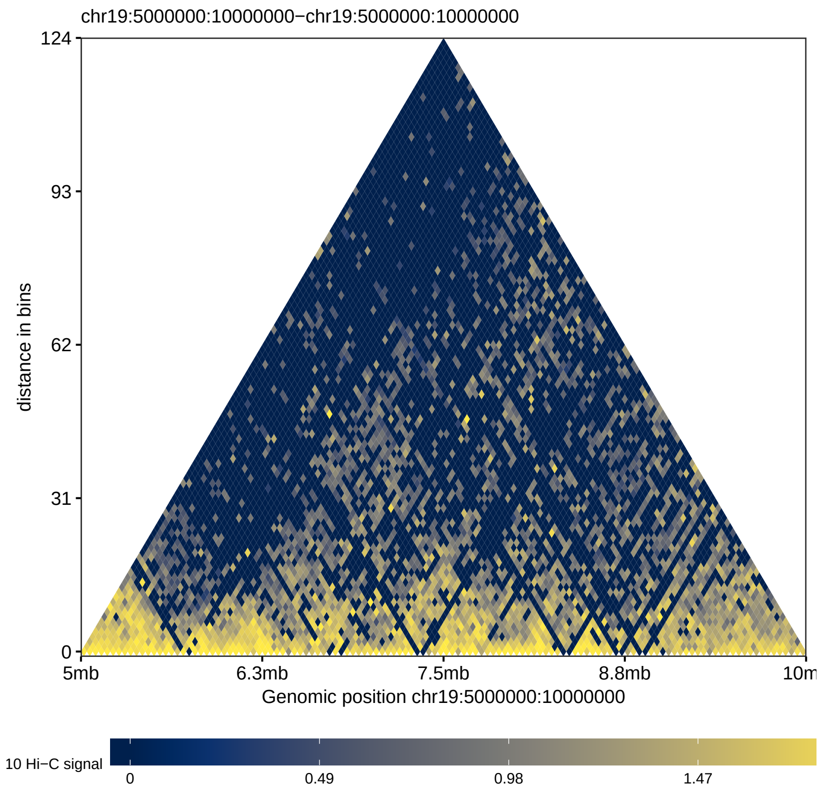

Figure 5: Rotated Hi-C matrix with corrected plot-sizes and using a different color palette.

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal6.pdf",
+     Bricks = Brick.file,
+     tad.ranges = TAD_ranges,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     colours = "#E0CA3C",
+     FUN = Failsafe_log10,
+     value.cap = 0.99,
+     legend.title = "Log10 Hi-C signal",
+     palette = "Reds",
+     width = 10,
+     height = 11,
+     return.object=TRUE)
```

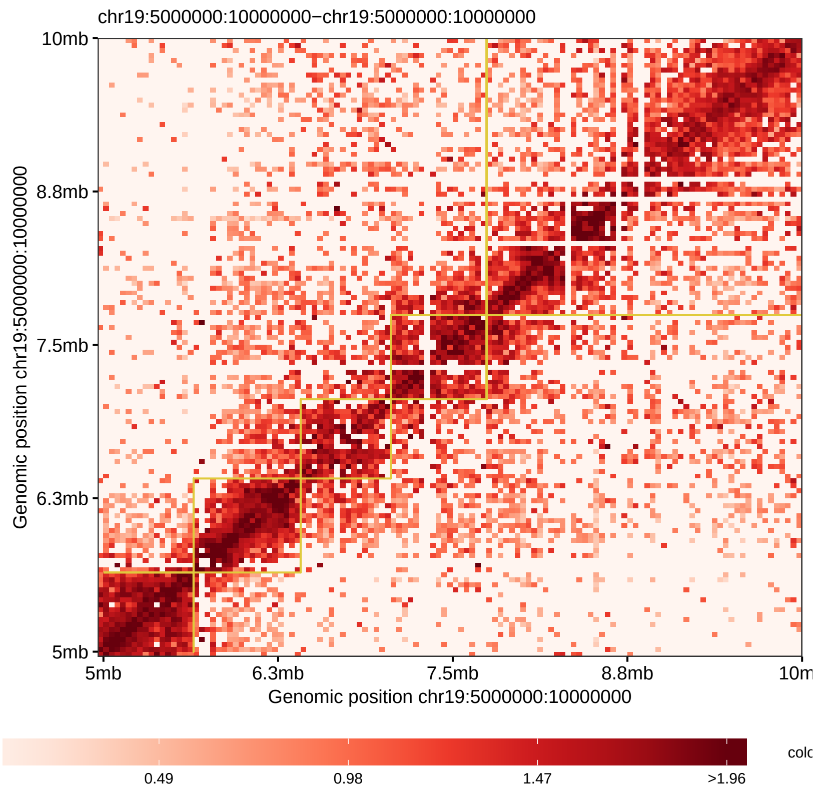

Figure 6: Hi-C matrix with TADs.

```
> Brick_vizart_plot_heatmap(File = "chr19-5MB-10MB-normal.pdf",
+     Bricks = Brick.file,
+     tad.ranges = TAD_ranges,
+     x.coords = "chr19:5000000:10000000",
+     y.coords = "chr19:5000000:10000000",
+     colours = "red",
+     FUN = Failsafe_log10,
+     value.cap = 0.99,
+     legend.title = "Log10 Hi-C signal",
+     palette = "cividis",
+     width = 15,
+     height = 9,
+     cut.corners = TRUE,
+     rotate = TRUE,
+     return.object=TRUE)
```

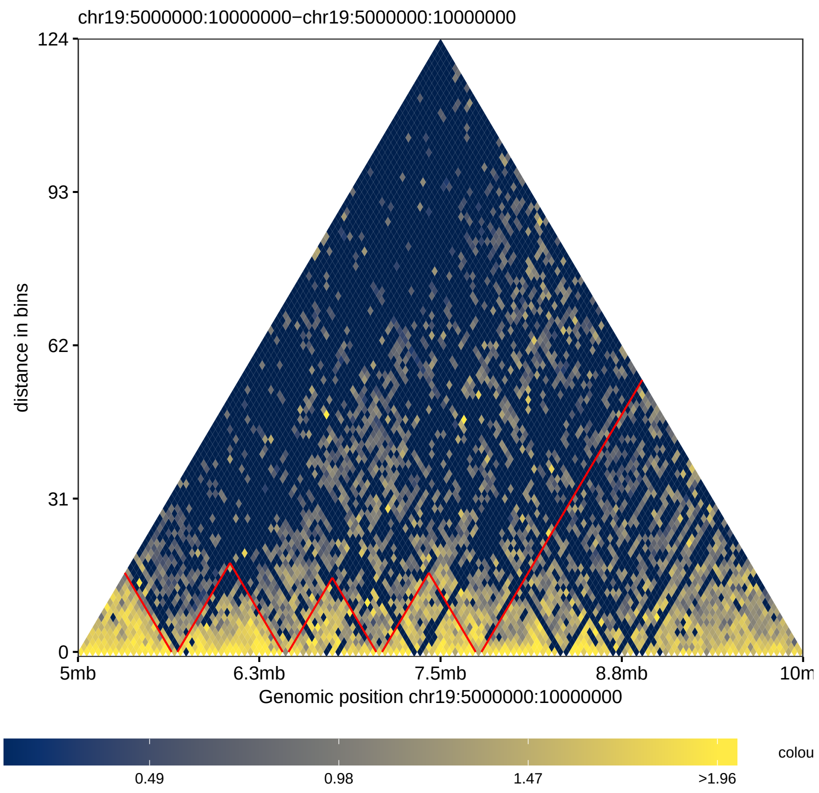

Figure 7: Rotated Hi-C matrix with TADs and a different color palette.

## References

[1] Rao SS, Huntley MH, Durand NC, Stamenova EK, Bochkov ID, Robinson JT, Sanborn AL, Machol I, Omer AD, Lander ES and Aiden EL A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. Cell, 2014

[2] Dixon JR, Selvaraj S, Yue F, Kim A, Li Y, Shen Y, Hu M, Liu JS and Ren B Topological domains in mammalian genomes identified by analysis of chromatin interactions. Nature 2012