Viewing and Interpreting Chromosomal Interactions

Representation of Chromosome Interactions

bin / bin			i		
k			$A_{i,k}$		

$A_{i,k}$ is the number of interactions between the bin i and the bin k.

Interaction Data with 5 Bins

16	2	4	4	1
2	15	7	5	2
4	7	17	8	0
4	5	8	12	3
1	2	0	3	10

In practice, such matrices, say for human genome, have millions (or even billions) of entries for the entire genome.

Visualizing Interaction Matrices

Interaction matrices can be visualized using heatmaps. A heatmap, in general, is representation of data with colors. For us, in particular, it is a tile of nxn squares where the color of each square (pixel) is determined by the number of interactions. More interactions are typically displayed with darker colors.

Heatmaps in Matlab

Matlab has a built-in function to plot heatmaps:

HeatMap(Data, *)

Plots the heatmap and returns a heatmap object.

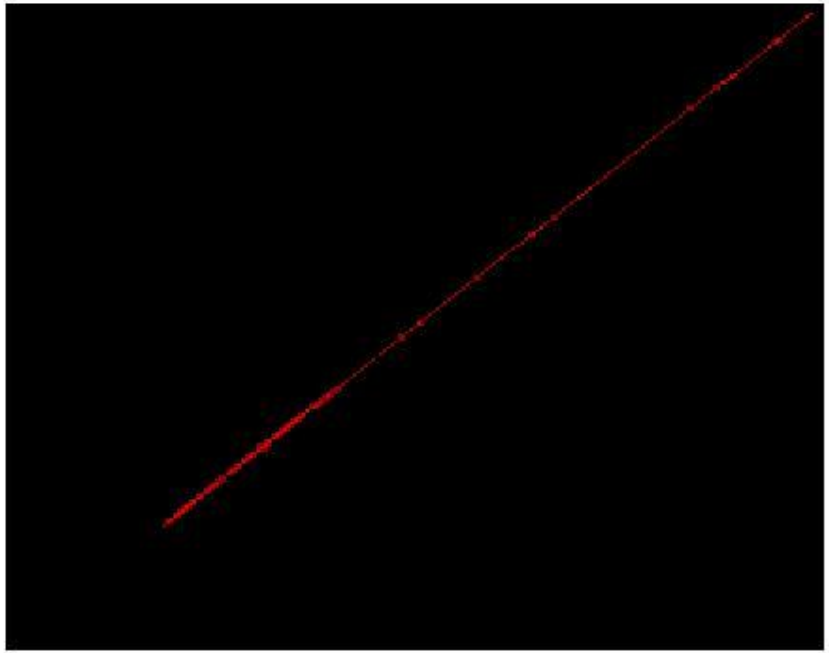
*Optional parameters

Data: Chromosome Interaction Matrix

Let’s import our data and try to visualize it.

```
matrix_file = '/Users/ozadamh/Documents/PSB_course/C-500000/C-500000_raw.txt';

raw_matrix = dlmread(matrix_file);

HeatMap(raw_matrix)
```

Our First Heatmap

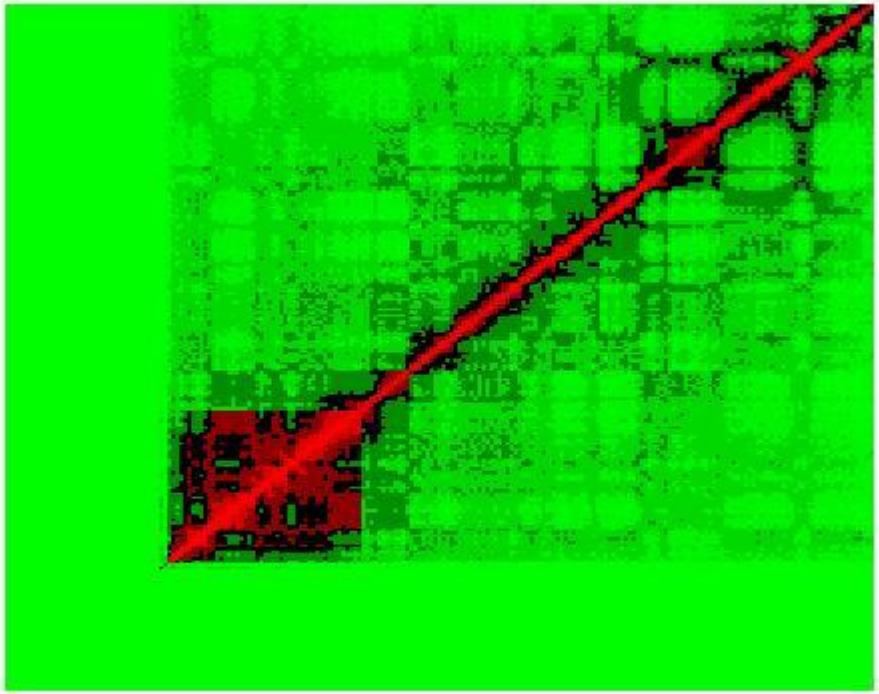


What is wrong with the interaction data?

Scaling! Remember how we scale RNA-Seq data in the bootcamp!

Let’s try to look at the data in log scale

HeatMap(log(raw_matrix))



Changing the colormap may help

Colormap

Colormap is a matrix where each row corresponds to a particular color by holding its RGB values. In order to reverse the colormap, we need to put the colormap matrix upside-down.

In-Class Exercise: Write a function whose input is a matrix and output is a matrix whose rows are the same as input in reverse order.

Example:

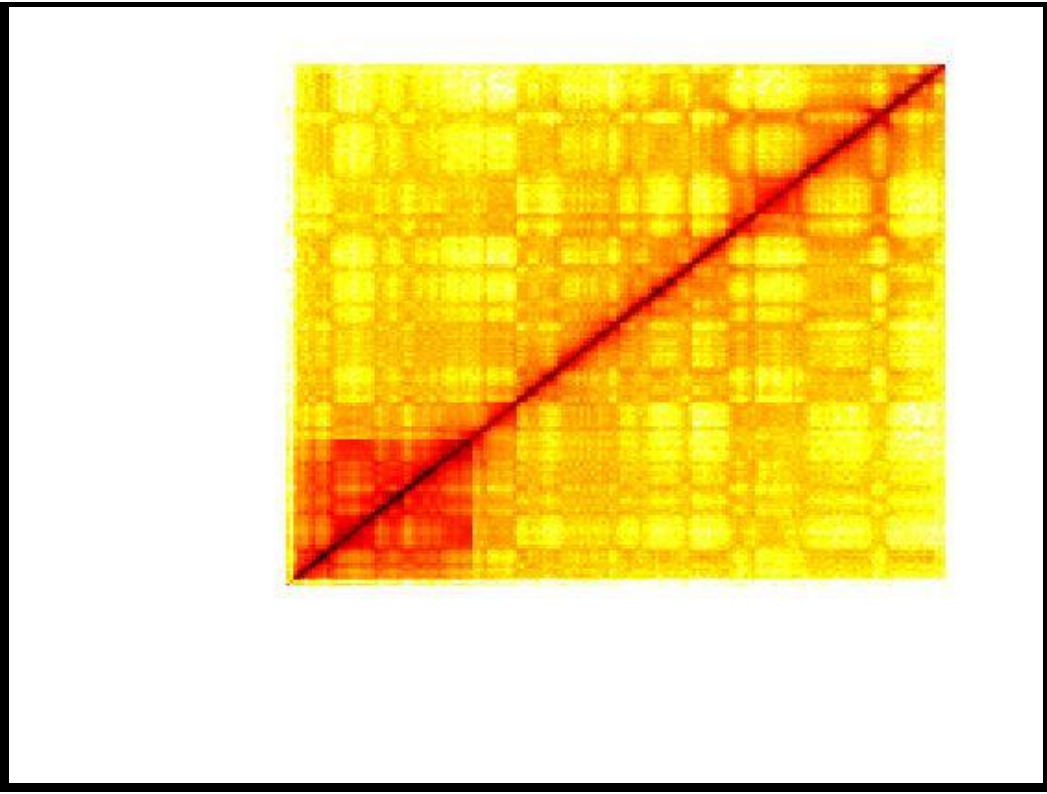
Input:

1	1	1
2	4	8
3	6	9

Output:

3	6	9
2	4	8
1	1	1

Reversed Colormap



Reverse Matrix Function

```
function[upsideDownMatrix] = makeUpsideDownMatrix(originalMatrix)
   [nrows, ncols] = size(originalMatrix);
   upsideDownMatrix = zeros(nrows, ncols);
   for i = 1:nrows
      upsideDownMatrix(i,1:ncols) = originalMatrix(nrows - i + 1,
1:ncols);
   end
end
```

A simpler way of doing the same thing

```
function[upsideDownMatrix] = makeUpsideDownMatrix(originalMatrix)
    upsideDownMatrix = originalMatrix(end:-1:1, :)
end
```

Too Many Zeros

The wide white space around the edges are coming from zeros. Let’s remove them from our matrix. But how can we know which rows / columns to remove?

```
row_sums = sum(raw_matrix);
```

Gives the sum or rows. Using sum on the matrix and on its transpose shows that the first 38 rows / columns are all 0’s so we can remove them.

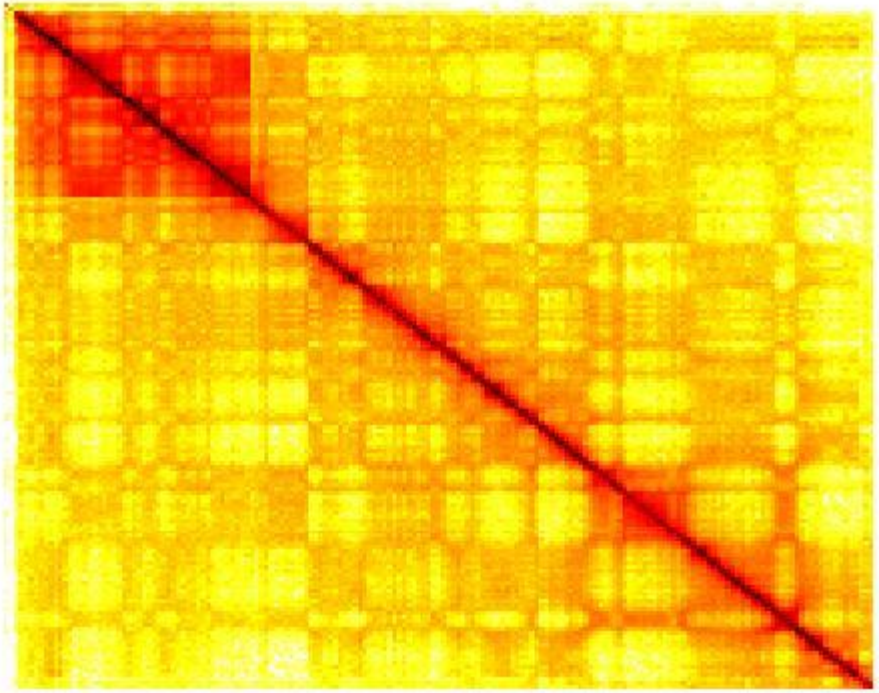
```
clipped_matrix = raw_matrix(39:215 ,39:215 );
```

```
HeatMap( log(clipped_matrix), 'Colormap', reversed_colormap
);
```

Flipping The Diagonal

Traditionally, the diagonal goes from top-left to bottom right. We can easily do that by turning the matrix upside down.

```
upside_down_matrix = makeUpsideDownMatrix(clipped_matrix);

HeatMap( log(upside_down_matrix), 'Colormap',
reversed_colormap );
```

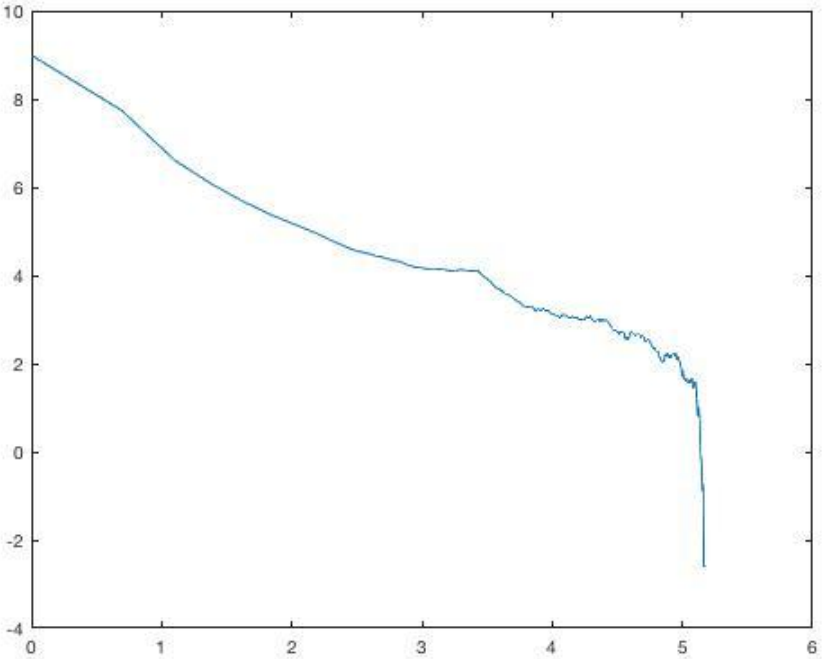


Distance Decay

In general, the number interactions go down as the distance between two bins increase. In other words, as we move away from the diagonal, the color gets lighter. A good way to analyze this is visualizing the average number of interactions with respect to distance.

```
function[diagonal_means] = getScalingPlotData(mymatrix)

[nrows, ncols] = size(mymatrix);
diagonal_means = zeros(1, nrows);

for i = 1:nrows
    this_sum = 0;
    for k = 1:(ncols-i) + 1
        this_number = mymatrix(i + (k-1) ,k);
        if(isnan(this_number))
            this_number = 0;
        end
        this_sum = this_sum + this_number;
    end
    this_average = this_sum / ( nrows - i + 1);
    diagonal_means(i) = this_average
end
end
```

```
scaling_data = getScalingPlotData(raw_matrix);

plot(scaling_data);

plot( log(1:nrows) , log(scaling_data) );
```

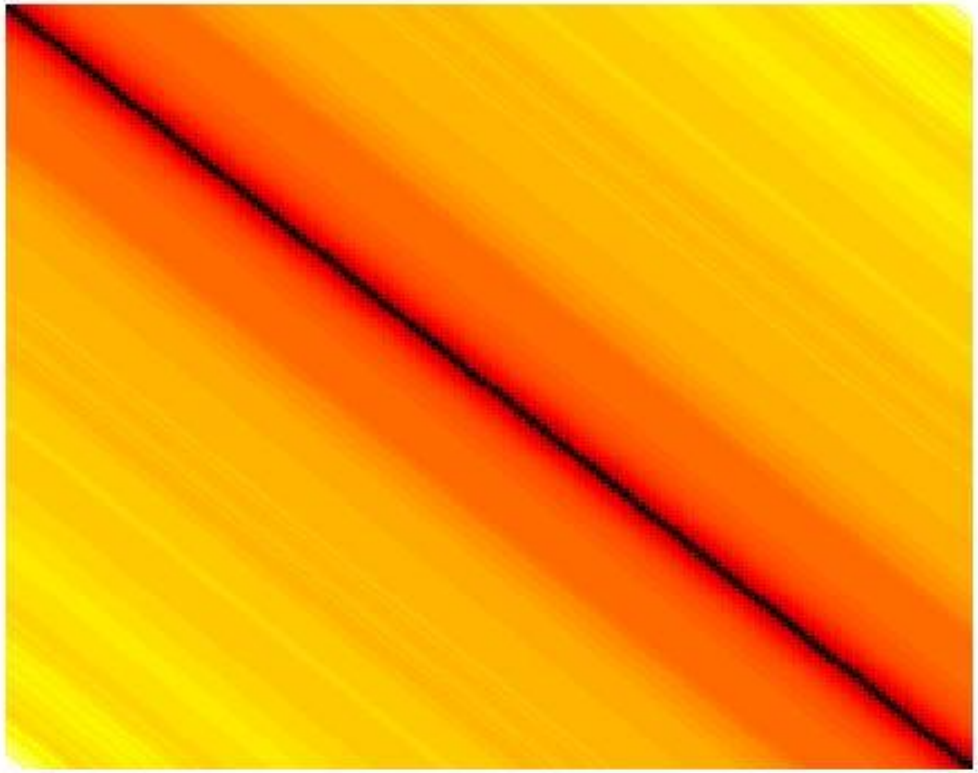


Scaling Data Heatmap

Let’s generate the heatmap that gives us the interactions if every bin behaved as expected with respect to distance.

Exercise : Write the function **convertScalingDataToMatrix**

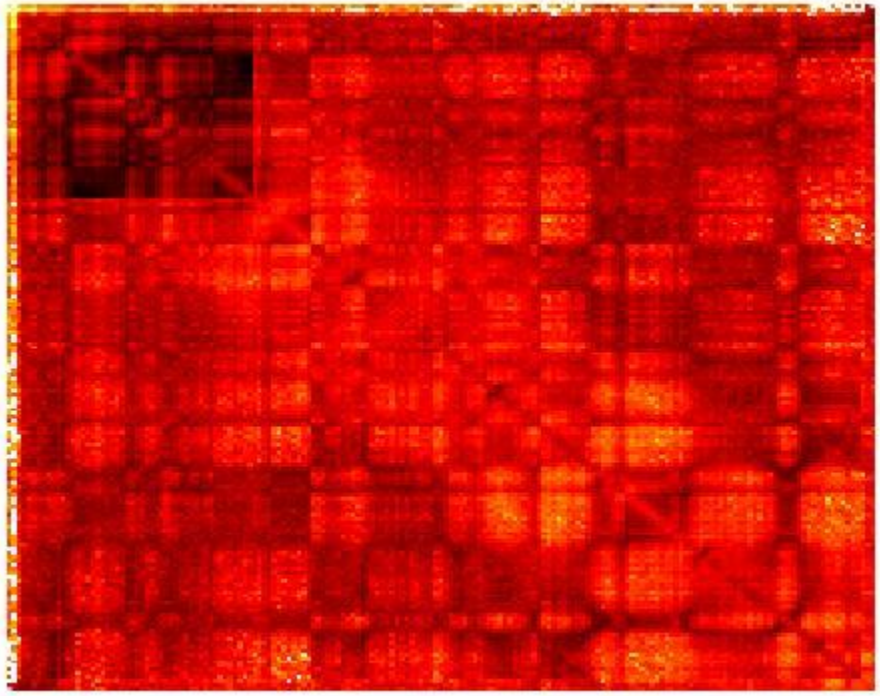
```
function[scaling_matrix] =
convertScalingDataToMatrix(scaling_data)
    [nrows, ncols] = size(scaling_data);
    scaling_matrix = zeros(ncols, ncols);
    for i = 1:ncols
        for k = 1:(ncols - i + 1)
            scaling_matrix(i + (k-1) ,k) =
scaling_data(i);
            scaling_matrix(k, i + (k-1)) =
scaling_data(i);
        end
    end
end
```



Observed / Expected

Let’s compare each pixel to the expected value with respect to the distance.

Exercise: Generate the Observed / Expected Matrix and plot it.

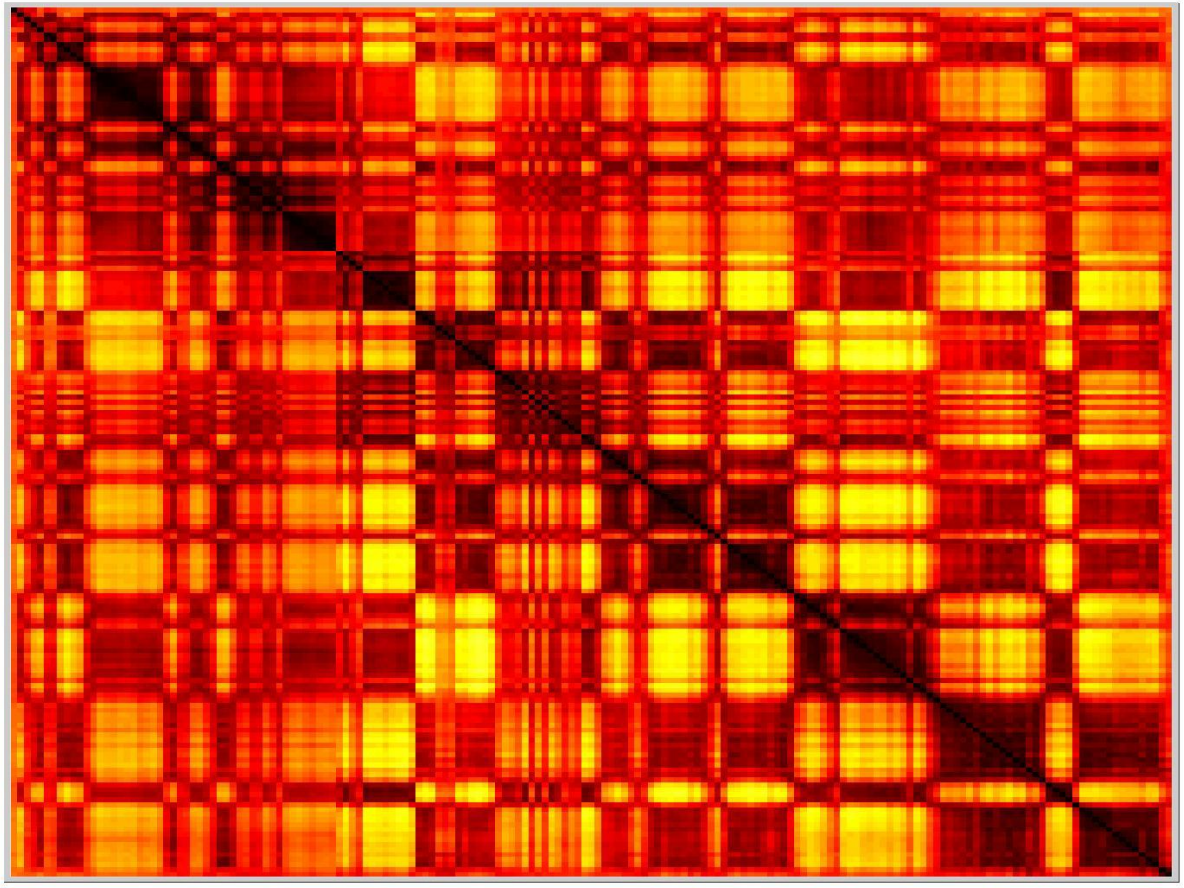


Correlation Heatmap

In the observed / expected heatmap, we can look at the correlation of rows and columns and plot it in a heatmap. The resulting heatmap is defined by elements a_{ij} where a_{ij} is the pearson correlation of the ith row and jth column in the observed / expected map.

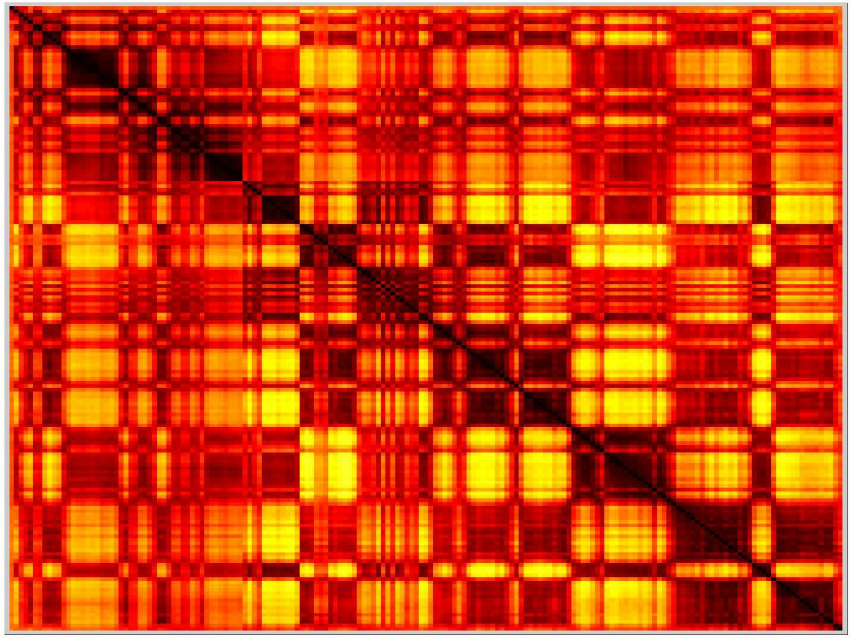
Luckily, we have the **corr()** function in matlab that does exactly this computation.

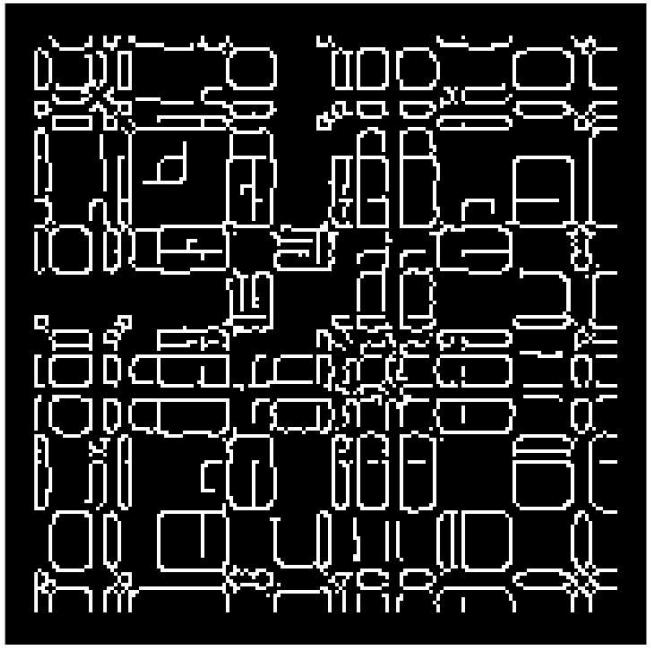
```
correlation_matrix_pre = corr(observed_over_expected);
correlation_matrix = makeUpsideDownMatrix(correlation_matrix_pre);
HeatMap(log(correlation_matrix_pre),  'Colormap', reversed_colormap);
```



Finding Compartments Using Edge Detection

```
imshow(correlation_matrix_pre);

edge_image = edge(correlation_matrix_pre, 'canny');

imshow(edge_image_pre);
```





The edge image is a binary matrix where 0 represents black and 1 represents white color.

A quick and dirty way of finding compartment boundaries is summing up the columns and plotting this profile.

```
edge_sums = sum(transpose(edge_image));

plot(edge_sums);
```

Peaks give compartment boundaries