DiNAMICDuo Vignette

Vonn Walter

2023-03-02

Introduction

Genomic instability is one of the hallmarks of cancer, and it can lead to a variety of structural alterations, including DNA copy number gains and losses. We will collectively refer to these changes as *DNA copy number alterations* or *CNAs*. Although underlying genomic instability produces CNAs that are randomly scattered throughout the genome, some CNAs provide a selective growth advantage and thus may be observed in the same genomic region across multiple independent samples. We refer to these CNAs as *recurrent*, and identifying recurrent CNAs is of interest to cancer researchers because they may harbor genes that contribute to the cancer phenotype, e.g., oncogenes or tumor suppressors for recurrent gains and losses, respectively. In contrast, genes in *sporadic* CNAs are less likely to have biological relevance because they are randomly scattered throughout the genome.

A number of statistical methods have been developed to identify recurrent CNAs in a single cohort, including GISTIC (Beroukim et al. (2007)), RUBIC (van Dyk et al. (2016)), and DiNAMIC (Walter et al. (2011)). Remarkably, to date no methodology has been developed to identify recurrent *DNA copy number differences* between two groups even though the corresponding analysis for gene expression data - identifying differentially expressed genes in two groups of samples - is a standard part of many gene expression profiling studies. This motivated our interest in developing DiNAMICDuo, which builds upon and extends the original DiNAMIC package while leveraging the theoretical studies of *cyclic shift testing* in Walter et al. (2015). Although DiNAMICDuo employs the same cyclic shift permutation scheme that was introduced in DiNAMIC, it utilizes custom Python scripts and the reticulate R package (Ushey et al. (2020)) to achieve pronounced gains in computational efficiency. DiNAMICDuo also includes R versions of the Python code for users who who prefer to run a "full R" version.

DiNAMICDuo can identify recurrent CNAs in a single cohort, as was done in the original DiNAMIC package, or identify recurrent copy number differences in two cohorts. Thus a DiNAMICDuo analysis starts with either a single copy number matrix X or two copy number matrices X and Y. Most DiNAMICDuo functions are designed to work with both X and Y, but they can be applied to X alone by setting Y = NULL. Examples are shown below.

We will assume that rows of X and Y are indexed by genes and columns are indexed by samples. Although X and Y do not need to have the same number of columns, most of the functions in DiNAMICDuo assume that the rows of X and Y are indexed by the same set of ordered genes. DiNAMICDuo's dataPrep function can be applied to ensure this is the case. The entries of X and Y are gene-level quantitative copy number values. It is assumed that these values represent log ratios of tumor vs. normal copy number. Thus 0 represents copy neutral, positive values correspond to copy number gains, and negative values correspond to copy number losses. If necessary, data should be pre-processed accordingly.

Users can apply DiNAMICDuo to their own data or data that has been downloaded from public repositories. Here we apply DiNAMICDuo to DNA copy number data from The Cancer Genome Atlas that was downloaded from the Broad Institute's Firehose GDAC (https://gdac.broadinstitute.org/). Because of the size restrictions on data sets included in R packages, it was not possible to utilize the entire set of genomewide copy number values. Instead, the data included with the package contains gene level copy number matrices *luadSubset* (3475 genes, 65 samples) and *luscSubset* (3480 genes, 60 samples) that were taken from the TCGA lung

adenocarcinoma (LUAD) and lung squamous cell carcinoma (LUSC) DNA copy number data sets, respectively. Samples were selected randomly from each cohort. A common set of 3455 genes was also randomly selected, then copy number values for a handful of genes were added to create *luadSubset* and *luscSubset* with non-identical sets of genes. As shown below, applying dataPrep restricts *luadSubset* and *luscSubset* to the common set of 3455 genes.

Using DiNAMICDuo

Python and NumPy

Both Python and the NumPy Python library must be installed if the cyclicShiftCol.py function will be called. Undoubtedly there is more than one way to do this, but the reticulate R package can be used to accomplish both tasks. From R, reticulate::install_miniconda() will install Miniconda, but users are advised that this downloads ~50Mb and takes time. Subsequently, reticulate::import("numpy", delay_load = TRUE) will install NumPy. The use of cyclicShiftCol.py provides increased computational efficiency. However, it is important to note that all DiNAMIC.Duo analyses can be performed in R, as shown below.

Loading the Data

We begin by loading the data. It is assumed that the copy number matrices X and Y have genes in the rows and samples in the columns. By design, the number of genes and samples in luadSubset and luscSubset are not the same.

```
#> Warning: package 'biomaRt' was built under R version 4.0.3
#> Warning: package 'dinamic' was built under R version 4.0.3
```

```
library(DiNAMIC.Duo)
#> Warning: package 'DiNAMIC.Duo' was built under R version 4.0.5
#> Is miniconda installed? (Yes/no/cancel)
data(DiNAMIC.Duo)
dim(luadSubset)
#> [1] 3475   65
dim(luscSubset)
#> [1] 3480   60
```

Data Preparation

As noted above, the same genes are not present in *luadSubset* and *luscSubset*. Because a comparison of two copy number matrices requires a common set of ordered genes, the dataPrep function was designed to create copy number matrices X and Y of the appropriate form. In addition, dataPrep utilizes biomaRt to create a data frame *posDT* that contains genomic position information for the genes that index the rows of X and Y. These objects are stored in a list whose components are named X, Y, and posDT. As shown in later sections of this vignette, *posDT* is used in downstream functions such as peeling, plotting, and results processing. The arguments of dataPrep are the initial copy number matrices X and Y, as well as the species of the organism that yielded the samples. Currently only human and mouse are supported. Set Y = NULL when analyzing a single copy number matrix X. For brevity, we write *pD* for preppedData, the output of dataPrep.

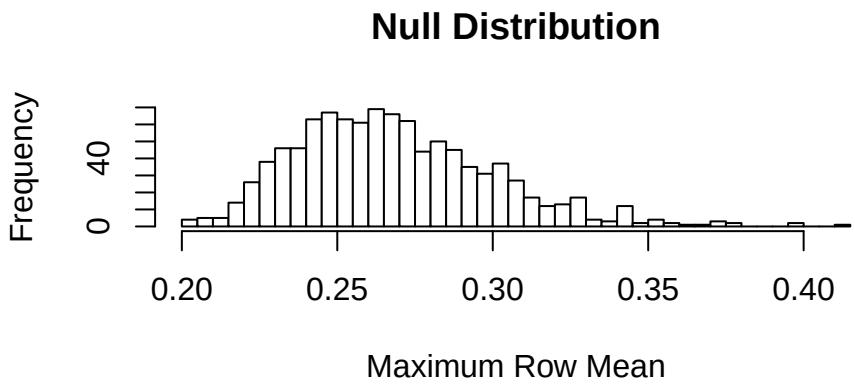
```
pD = dataPrep(X = luadSubset, Y = luscSubset, species = "human")
names(pD)
#> [1] "X"     "Y"     "posDT"
dim(pD[["X"]])
#> [1] 3455   65
```

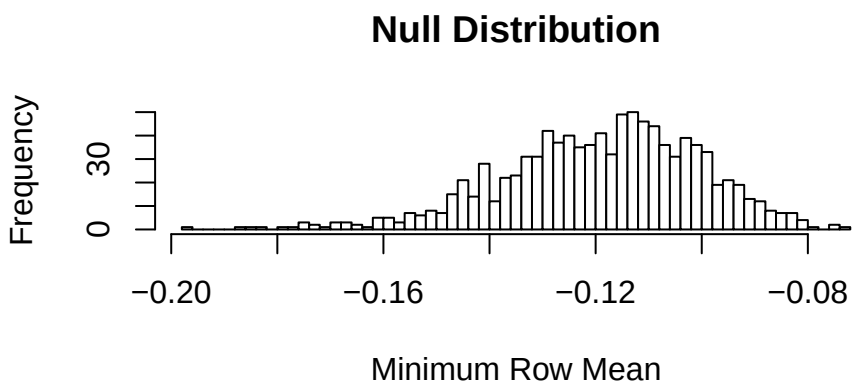
```
dim(pD[["Y"]])
#> [1] 3455   60
all(rownames(pD[["X"]]) == rownames(pD[["Y"]]))
#> [1] TRUE
head(pD[["posDT"]])
#>           chr   start     end mean_pos cytoband
#> OR4F29      1  450703  451697   451200   p36.33
#> AGRN        1 1020120 1056118  1038119   p36.33
#> TTLL10      1 1173880 1197936  1185908   p36.33
#> PUSL1       1 1308597 1311677  1310137   p36.33
#> RN7SL657P   1 1405460 1405752  1405606   p36.33
#> VWA1        1 1434861 1442882  1438871   p36.33
```

Cyclic Shift Permutation

Many methods for identifying recurrent CNAs utilize permutation-based null distributions to assess statistical significance. A detailed discussion of this topic is beyond the scope of this vignette. The original DiNAMIC manuscript (Walter et al. (2011)) introduced a *cyclic shift* permutation scheme that largely preserves the underlying correlation structure of the data. The cyclicNullR function employs an R-based version of the cyclic shift permutation scheme to create an empirical null distributions for either the maximum and minimum row means (when analyzing a single copy number matrix X) or the maximum and minimum difference of row means (when analyzing two copy number matrices X and Y). First we illustrate this function using the LUSC data in *pD*, which is stored in pD[[“Y”]].

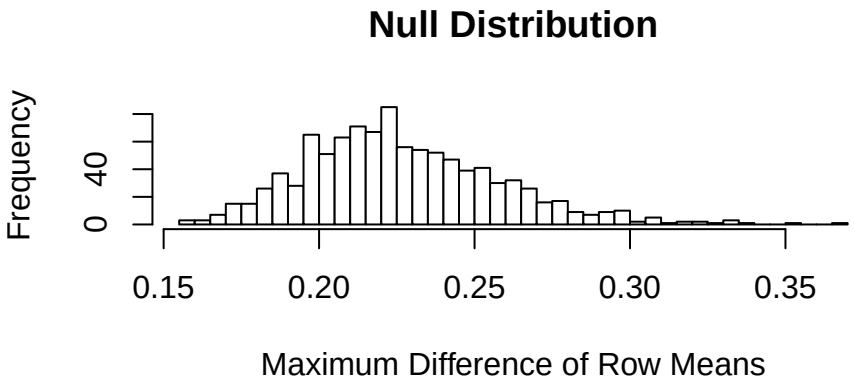
```
nullDist1 = cyclicNullR(X = pD[["Y"]], Y = NULL, numPerms = 1e3, randomSeed = 1)
par(mfrow = c(2, 1))
hist(nullDist1[,"maxNullDist"], breaks = 50, col = "white", main = "Null Distribution",
      xlab = "Maximum Row Mean")
hist(nullDist1[,"minNullDist"], breaks = 50, col = "white", main = "Null Distribution",
      xlab = "Minimum Row Mean")
```

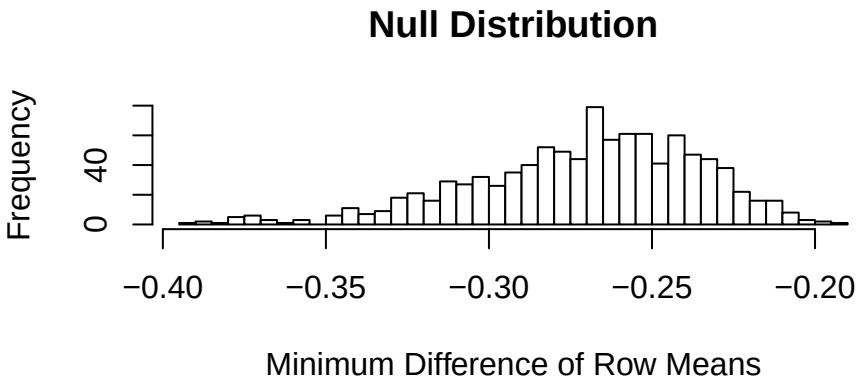




Next we consider differences of row means for two copy number matrices $X =$ pD[[“X]] (LUAD) and $Y =$ pD[[“Y]] (LUSC) using the same function. Later in this section we provide an example to show how to use create the corresponding null distributions using DiNAMICDuo's Python code.

```
nullDist2 = cyclicNullR(X = pD[["X"]], Y = pD[["Y"]], numPerms = 1e3, randomSeed = 1)
par(mfrow = c(2, 1))
hist(nullDist2[,"maxNullDist"], breaks = 50, col = "white", main = "Null Distribution",
      xlab = "Maximum Difference of Row Means")
hist(nullDist2[,"minNullDist"], breaks = 50, col = "white", main = "Null Distribution",
      xlab = "Minimum Difference of Row Means")
```



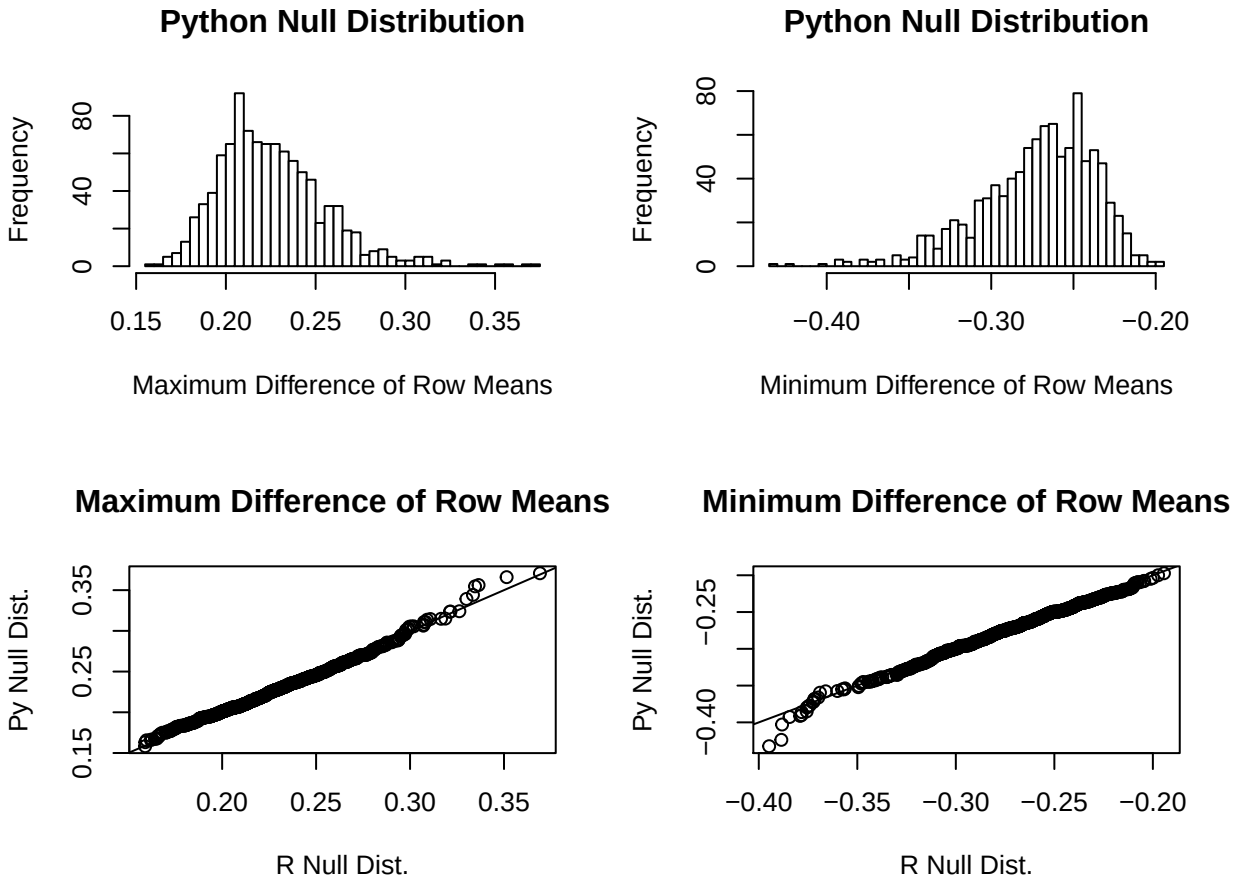


As is the case with many other permutation-based approaches, applying cyclic shift permutations can be computationally intensive, especially for large copy number matrices X and Y. For example, it takes approximately ten seconds to compute the null distribution for the maximum and minimum column mean of X = pD[["Y"]], the LUSC data (3455 genes, 60 samples), using 1000 random cyclic shifts. Users should expect computation time to scale with both the dimensions of the data and the number of cyclic shifts. Fortunately, this process only needs to be performed once in a given analysis.

As noted above, DiNAMICDuo includes Python code that provides a more computationally efficient way to generate the empirical null distributions when compared to the R-based approach. Here we show how to specify the path for the source_python function in reticulate R package as well as how to call DiNAMICDuo's cyclicNullPython function.

```
path = paste(system.file(package = "DiNAMIC.Duo"), "python", "cyclicShiftCol.py", sep = "/")
reticulate::source_python(path)
pyOutputList = cyclicNullPython(x = pD[["X"]], y = pD[["Y"]], numPerms = 1e3, randomSeed = 1234)
pyOutput = cbind(pyOutputList[[1]], pyOutputList[[2]])
colnames(pyOutput) = c("maxNullDist", "minNullDist")
head(pyOutput)
#>      maxNullDist minNullDist
#> [1,]   0.2027987  -0.2908256
#> [2,]   0.1973192  -0.2336962
#> [3,]   0.1822962  -0.2232654
#> [4,]   0.2046013  -0.2342282
#> [5,]   0.2274936  -0.2322974
#> [6,]   0.2428513  -0.2366538
```

Histograms and qqplots show that both the R- and Python-based null distributions are highly concordant.



Peeling

The original DiNAMIC algorithm employs a *peeling* algorithm that allows users to identify multiple copy number peaks across the genome in a single copy number matrix X. In brief, suppose recurrent copy number gains produce a peak at marker k. The peeling algorithm identifies a genomic region consisting of markers around k that contribute to the peak. It then *peels* these markers by modifying select entries in X in order to remove the peak at k so the next peak can be identified. Negative peaks corresponding to recurrent copy number losses are handled similarly. By iteratively applying the peeling algorithm, multiple peaks across the genome can be identified.

The peelingOne function in DiNAMICDuo largely implements DiNAMIC's original peeling algorithm. The arguments are a copy number matrix X, the marker k that is to be peeled, the data frame *posDT* produced by dataPrep, and an optional argument *threshold* that can limit the size of the peeled region around k. The output is a two-component list whose entries are X, the peeled version of the input matrix X (the same notation is used intentionally), and *interval*, the endpoints of the peeled interval around k. By construction, the peeled interval cannot extend beyond the chromosome arm that contains k.

We begin by illustrating the use of peelingOne with X = pD[["Y"]], the LUSC data. The largest genomewide mean copy number value occurs at *SOX2*, a known oncogene in chr3q that is frequently amplified in LUSC. The peeled region consists of a substantial portion of the q arm of chr3. As shown in Figure 1, pronounced copy number gains are observed across chr3q. Thus this finding is not surprising.

```
k1 = which.max(rowMeans(pD[["Y"]]))
k1
#> SOX2
#>  794
```

```
pD[["posDT"]][k1,]
#>      chr     start       end  mean_pos cytoband
#> SOX2   3 181711925 181714436 181713180   q26.33
luscPeelOutput = peelingOne(X = pD[["Y"]], posDT = pD[["posDT"]], k = k1, threshold = NULL)
dim(luscPeelOutput[["X"]])
#> [1] 3455   60
luscPeelOutput[["interval"]]
#> [1] "93873051"  "197956610"
```

The peelingTwo function is a modified version of peelingOne that allows users to identify positive and negative copy number differences between two copy number matrices X and Y. The arguments and output of peelingTwo are similar to peelingOne, only now there are two copy number matrices X and Y.

We illustrate the use of peelingTwo by examining positive copy number differences between X = pD[["X"]] and Y = pD[["Y"]], i.e., markers that exhibit larger mean copy number values in LUAD than LUSC. The order of subtraction matters in this setting because positive copy number differences could arise from (i) gains in LUAD that are not observed in LUSC, (ii) gains in LUAD that are more pronounced than gains in LUSC, or (iii) copy number *losses* that are observed in LUSC but not LUAD. Undoubtedly there are other possibilities as well. Here we observe that the largest positive copy number difference occurs at *MBIP* in chr14q13.3, which is approximately 250kb from *NKX2-1*, a gene that frequently exhibits focal copy number gain in LUAD.

```
k2 = which.max(rowMeans(pD[["X"]]) - rowMeans(pD[["Y"]]))
k2
#> MBIP
#> 2360
pD[["posDT"]][k2,]
#>      chr    start      end mean_pos cytoband
#> MBIP  14 36298564 36320637 36309600    q13.3
luadLuscPeelOutput = peelingTwo(X = pD[["X"]],
Y = pD[["Y"]], posDT = pD[["posDT"]], k = k2,
threshold = NULL)
dim(luadLuscPeelOutput[["X"]])
#> [1] 3455   65
dim(luadLuscPeelOutput[["Y"]])
#> [1] 3455   60
luadLuscPeelOutput[["interval"]]
#> [1] "19975444" "92935285"
```

Plots

The genomePlot function allows users to produce genomewide copy number plots of the mean gene-level copy number values for a copy number matrix X or two copy number matrices X and Y. The primary argument for genomePlot is inputList, a three-component list that can be produced by dataPrep. If Y = NULL, only the mean copy number values in X are plotted; otherwise both copy number means for X and Y are plotted, as are the copy number means for X - Y. There are numerous additional arguments that can be used to tailor the plots, some of which are illustrated below. Please see the help menu or the User's Manual for more information.

We begin by showing the mean copy number values for LUSC in Figure 1. The recurrent arm level gains and losses of 3q and 3p, respectively, are visible, as are focal gains of 11q13 and focal losses of 9p21.

```
genomePlot(inputList = list(X = pD[["Y"]], Y = NULL,
posDT = pD[["posDT"]]), lineColorVec = c("black"),
```

```
ylimLow = -0.5, ylimHigh = 1.6, xaxisLabel = "Chromosome",
yaxisLabel = "Copy Number Mean")
```

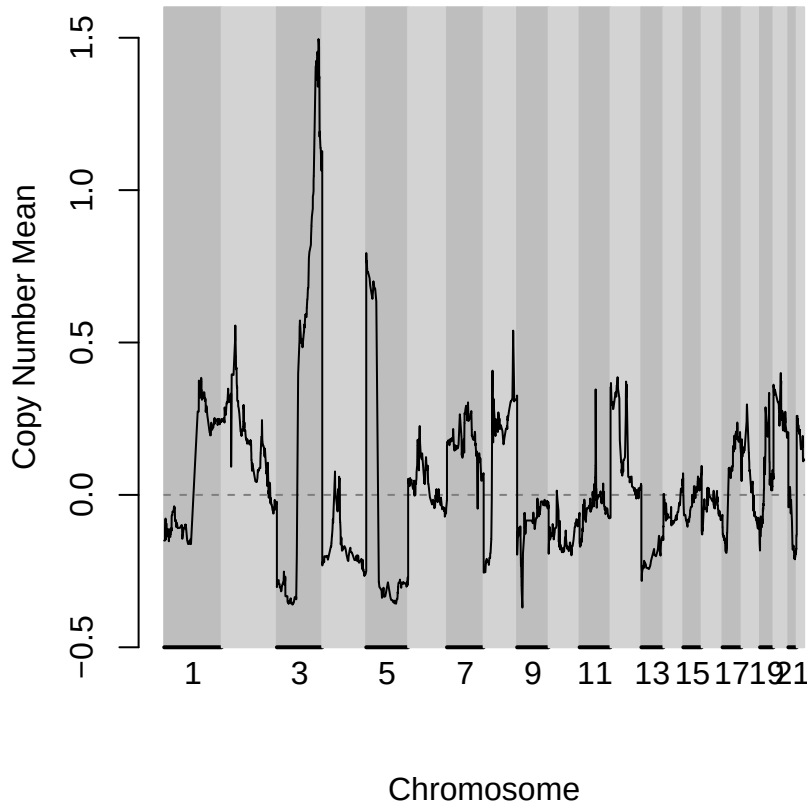


Figure 1: Figure 1

For the purposes of illustration, in Figure 2 we show a genomewide copy number plot for a peeled version of the LUSC data. Note that the peak in chr3q is no longer present, but the copy number values elsewhere in the genome are unchanged. In brief, a multiplicative scaling factor is applied to to the entries in X that contribute to the peak at k, and after peeling the mean copy number value at k is equal to the overall mean copy number value in X. Details can be found in the original DiNAMIC manuscript.

```
genomePlot(inputList = list(X = luscPeelOutput[["X"]], Y = NULL,
posDT = pD[["posDT"]]), lineColorVec = c("black"),
ylimLow = -0.5, ylimHigh = 1.6, xaxisLabel = "Chromosome",
yaxisLabel = "Copy Number Mean")
```

We show copy number means for X = LUAD, Y = LUSC, and X - Y = LUAD - LUSC in Figure 3. Here the copy number values for X - Y are displayed in black, and we clearly see the positive copy number peak at chr14q13.3 that was described above. Note that copy number gains in chr3q are much more pronounced in LUSC (blue) than LUAD (red). Thus the copy number means for X - Y in that region of the genome are negative.

```
genomePlot(inputList = pD, ylimLow = -1.6, ylimHigh = 1.6,
showLegend = T, legendText = c("LUAD", "LUSC", "LUAD-LUSC"),
legendXQuantile = 0.4, legendYCoord = 1.5,
xaxisLabel = "Chromosome", yaxisLabel = "Copy Number Mean")
```

By design, the plots produced by genomePlot allow users to visualize copy number gains and losses across the genome. These figures may be supplemented with plots produced by genomeChrPlot, which makes similar

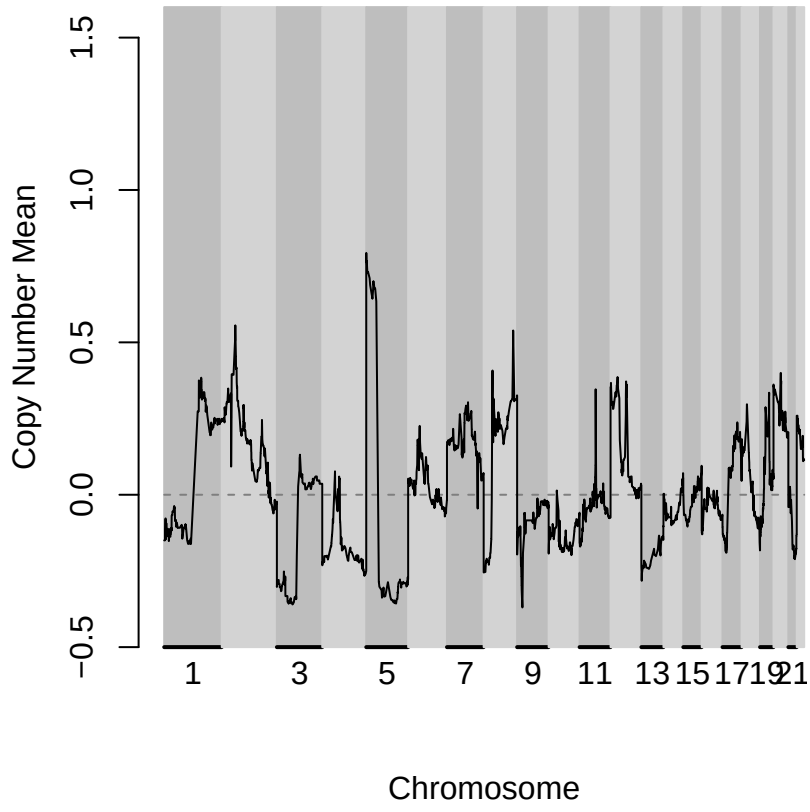


Figure 2: Figure 2

plots at the chromosome level. Separate plots are produced for each chromosome. All of the arguments of genomePlot are used by genomeChrPlot, and there is an additional argument plottingChrs that specifies the chromosomes to be plotted.

```
genomeChrPlot(inputList = pD, plottingChrs = c(3),
ylimLow = -1.6, ylimHigh = 1.6,
showLegend = T, legendText = c("LUAD", "LUSC", "LUAD-LUSC"),
legendXQuantile = 0.05, legendYCoord = 1.5,
xaxisLabel = "Chromosome", yaxisLabel = "Copy Number Mean")
```

Iterated Peeling

As seen in the plots above, tumor cohorts often exhibit multiple positive and negative peaks across the genome corresponding to multiple recurrent gains and losses, respectively. The peelingOneIterate function iteratively calls the peelingOne function in order to identify multiple peaks and peeled regions across the genome. Although peelingOneIterate can identify peaks and peeled regions without assessing their statistical significance, p-values will be computed if a null distribution is provided. Gains and losses should be analyzed separately.

To illustrate peelingOneIterate, we identify peaks and peeled regions corresponding to recurrent gains and losses in $X =$ pD[["Y"]], the LUSC data. The top five results are shown below, and these correspond to positive and negative peaks in Figure 1.

```
luscGainPeel = peelingOneIterate(X = pD[["Y"]], posDT = pD[["posDT"]], gain = TRUE, nullDist = nullDist
luscGainPeel
#>   chr peelStart  peelStop   peakLoc peakVal peakPVal
```

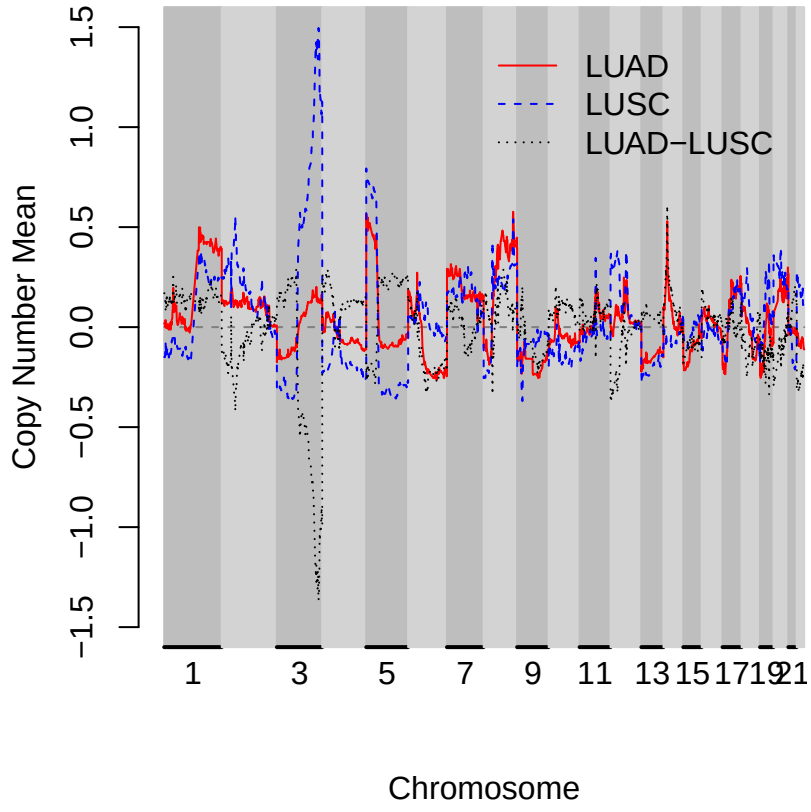


Figure 3: Figure 3

```
#> 1   3  93873051 197956610 181711925   1.496    0.001
#> 2   5    271621  43707405    271621  0.7933    0.001
#> 3   2   3379675  88861563  61888724  0.5558    0.001
#> 4   8  51319577 145066685 127794526   0.539    0.001
#> 5   8  37736601  42896275  38728186  0.4076    0.002
luscLossPeel = peelingOneIterate(X = pD[["Y"]], posDT = pD[["posDT"]], gain = FALSE, nullDist = nullDis
luscLossPeel
#>   chr peelStart  peelStop   peakLoc peakVal peakPVal
#> 1   9   1980290  41199261  24542952 -0.3697    0.001
#> 2   3   2088933  88149885  69168782   -0.36    0.001
#> 3   5  54455699 181159285 129097688 -0.3569    0.001
#> 4  13  18538880 113928991  19958670 -0.2822    0.001
#> 5   4  52051304 190072506 182169293 -0.2662    0.001
```

Next we used peelingTwoIterate to identify peaks and peeled regions corresponding to recurrent copy number differences between X = LUAD and Y = LUSC seen in Figure 3.

```
luadLuscPosPeel = peelingTwoIterate(X = pD[["X"]], Y = pD[["Y"]], posDT = pD[["posDT"]], gain = TRUE, nu
luadLuscPosPeel
#>   chr peelStart  peelStop   peakLoc peakVal peakPVal
#> 1  14  19975444  92935285  36298564  0.6091    0.001
#> 2   3   2088933  88149885  84958981  0.2898    0.039
#> 3   8  51319577 145066685  86047109  0.2883    0.039
#> 4   4     53286  48780322  24520192  0.2876     0.04
#> 5   5  54455699 181159285 113022099  0.2689    0.094
```

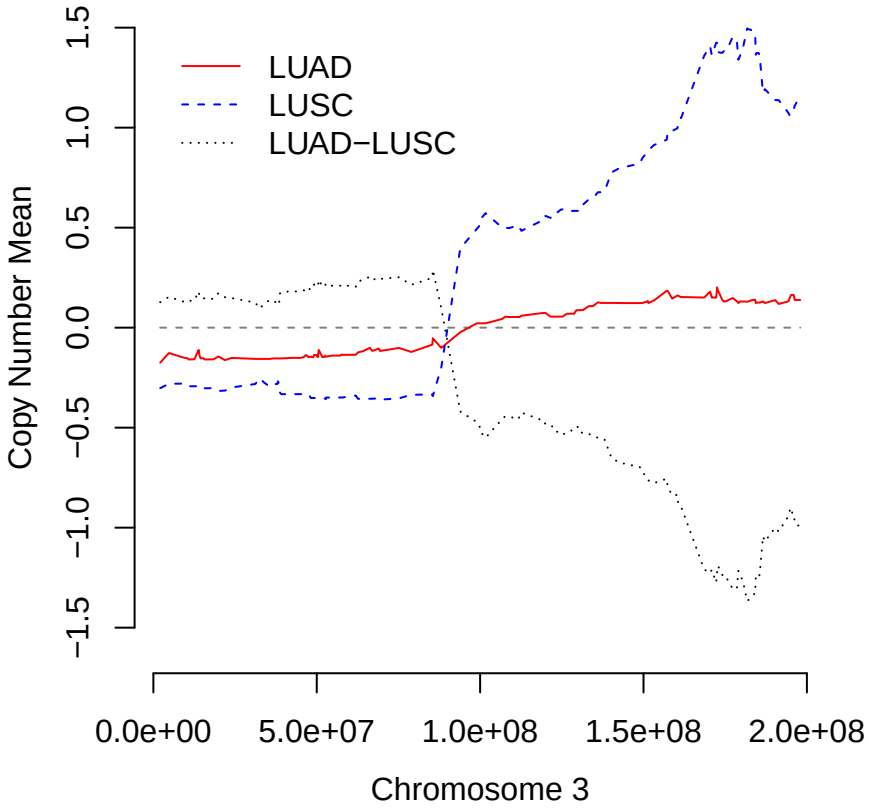


Figure 4: Figure 4

```
luadLuscNegPeel = peelingTwoIterate(X = pD[["X"]], Y = pD[["Y"]], posDT = pD[["posDT"]], gain = FALSE, r
luadLuscNegPeel
#>   chr peelStart  peelStop   peakLoc peakVal peakPVal
#> 1   3  93873051 197956610 181711925  -1.366    0.001
#> 2   2  46293667  88861563  61888724 -0.4208    0.001
#> 3  12     66767  32646050   1791963 -0.3748     0.01
#> 4   8  35235475  42896275  38728186 -0.3373    0.046
#> 5  19  29001670  58558954  39825350 -0.3367    0.046
```

Processing the Peeling Results

After iteratively applying either peelingOne or peelingTwo, users can apply the resultsProcess function to create summary output files similar to those produced by GISTIC. These files contain the following information for each peak k: the genomic position of k, the p-value of the mean copy number value at k based on the empirical null distribution computed using cyclic shift testing, the boundaries of the peeled region around k, and the genes in the peeled region (in alphabetical order). Depending on the size of the peeled region, the number of genes can be large. Below we illustrate resultsProcess by producing subsets of the gain and loss output for X = LUSC.

```
luscGainResults = resultsProcess(peel.results = luscGainPeel, posDT = pD[["posDT"]])
luscGainResults[1:20,]
#>           [,1]         [,2]        [,3]        [,4]         [,5]
#> chr       "3"          "5"         "2"         "8"          "8"
#> peelStart "93873051"   "271621"    "3379675"   "51319577"   "37736601"
#> peelStop  "197956610"  "43707405"  "88861563"  "145066685"  "42896275"
#> peakLoc   "181711925"  "271621"    "61888724"  "127794526"  "38728186"
#> peakVal   "1.496"      "0.7933"    "0.5558"    "0.539"      "0.4076"
#> peakPVal  "0.001"      "0.001"     "0.001"     "0.001"      "0.002"
#> G1        "ABHD10"     "ADAMTS12"  "ADD2"      "ADHFE1"     "ADAM3A"
#> G2        "ABTB1"      "ANKH"      "AFTPH"     "ANKRD46"    "ADAM9"
#> G3        "ACTL6A"     "BRIX1"     "APOB"      "ASPH"       "ERLIN2"
#> G4        "ADPRH"      "CDH6"      "ASPRV1"    "C8orf33"    "GINS4"
#> G5        "B3GNT5"     "CDH9"      "ATL2"      "CPNE3"      "MIR4469"
#> G6        "BBX"        "CEP72"     "B3GNT2"    "CPQ"        "PLAT"
#> G7        "CCDC50"     "GDNF"      "C2orf42"   "CRH"        "PLEKHA2"
#> G8        "CCDC80"     "IL7R"      "C2orf73"   "CSPP1"      "RN7SL709P"
#> G9        "CD80"       "IRX1"      "CCT7"      "CYP7B1"     "STAR"
#> G10       "CEP63"      "IRX2"      "COMMD1"    "DCSTAMP"    "TACC1"
#> G11       "CHST2"      "LIFR"      "CYP1B1"    "EBAG9"      NA
#> G12       "CLDN1"      "MTMR12"    "CYP26B1"   "EXOSC4"     NA
#> G13       "CLDN18"     "MTRR"      "CYS1"      "FAM91A1"    NA
#> G14       "CLSTN2"     "NNT"       "DNAJC27"   "GDAP1"      NA
luscLossResults = resultsProcess(peel.results = luscLossPeel, posDT = pD[["posDT"]])
luscLossResults[1:20,]
#>           [,1]         [,2]        [,3]         [,4]         [,5]
#> chr       "9"          "3"         "5"          "13"         "4"
#> peelStart "1980290"    "2088933"   "54455699"   "18538880"   "52051304"
#> peelStop  "41199261"   "88149885"  "181159285"  "113928991"  "190072506"
#> peakLoc   "24542952"   "69168782"  "129097688"  "19958670"   "182169293"
#> peakVal   "-0.3697"    "-0.36"     "-0.3569"    "-0.2822"    "-0.2662"
#> peakPVal  "0.001"      "0.001"     "0.001"      "0.001"      "0.001"
#> G1        "ACO1"       "ABHD14B"   "ABLIM3"     "ADPRHL1"    "ABCG2"
#> G2        "ARID3C"     "ARHGEF3"   "ACSL6"      "ALG5"       "ADH1C"
```

```
#> G3        "C9orf131" "ARPP21"   "ADAMTS6"   "ANKRD10"   "ADH4"
#> G4        "CBWD6"    "BSN"      "ANXA6"     "CCDC70"    "AFF1"
#> G5        "CCDC107"  "BTD"      "ARL14EPL"  "CDK8"      "ALB"
#> G6        "CCDC171"  "CADM2"    "ARSI"      "CDX2"      "ANKRD50"
#> G7        "CCL19"    "CADPS"    "ATP6AP1L"  "CENPJ"     "ANTXR2"
#> G8        "CCL27"    "CAMP"     "B4GALT7"   "COL4A1"    "ARL9"
#> G9        "CER1"     "CAND2"    "C5orf58"   "CPB2"      "ARSJ"
#> G10       "CNTFR"    "CCDC174"  "CAMK2A"    "CUL4A"     "ASB5"
#> G11       "CREB3"    "CCDC66"   "CARTPT"    "DHRS12"    "BBS12"
#> G12       "DCAF10"   "CCDC71"   "CCDC125"   "DOCK9"     "BTC"
#> G13       "DCTN3"    "CCK"      "CCDC69"    "DZIP1"     "CASP3"
#> G14       "DNAI1"    "CDC25A"   "CCNB1"     "EBPL"      "CBR4"
```

Because peelingOneIterate and peelingTwoIterate produce the same output format, the resultsProcess function can be applied when analyzing differences between two copy number matrices.

```
luadLuscPosResults = resultsProcess(peel.results = luadLuscPosPeel, posDT = pD[["posDT"]])
luadLuscPosResults[1:20,]
#>           [,1]        [,2]       [,3]        [,4]       [,5]
#> chr       "14"        "3"        "8"         "4"        "5"
#> peelStart "19975444"  "2088933"  "51319577"  "53286"    "54455699"
#> peelStop  "92935285"  "88149885" "145066685" "48780322" "181159285"
#> peakLoc   "36298564"  "84958981" "86047109"  "24520192" "113022099"
#> peakVal   "0.6091"    "0.2898"   "0.2883"    "0.2876"   "0.2689"
#> peakPVal  "0.001"     "0.039"    "0.039"     "0.04"     "0.094"
#> G1        "ABHD12B"   "ABHD14B"  "ADHFE1"    "ABCA11P"  "ABLIM3"
#> G2        "ACTR10"    "ARHGEF3"  "ANKRD46"   "ADD1"     "ACSL6"
#> G3        "ACYP1"     "ARPP21"   "ASPH"      "AFAP1"    "ADAMTS6"
#> G4        "BMP4"      "BSN"      "C8orf33"   "C4orf19"  "ANXA6"
#> G5        "C14orf119" "BTD"      "CPNE3"     "CPZ"      "ARL14EPL"
#> G6        "C14orf28"  "CADM2"    "CPQ"       "FGFBP2"   "ARSI"
#> G7        "C14orf39"  "CADPS"    "CRH"       "FRYL"     "ATP6AP1L"
#> G8        "CEP128"    "CAMP"     "CSPP1"     "GABRA2"   "B4GALT7"
#> G9        "CHGA"      "CAND2"    "CYP7B1"    "GABRB1"   "C5orf58"
#> G10       "CTSG"      "CCDC174"  "DCSTAMP"   "GABRG1"   "CAMK2A"
#> G11       "DDHD1"     "CCDC66"   "EBAG9"     "HS3ST1"   "CARTPT"
#> G12       "DHRS4"     "CCDC71"   "EXOSC4"    "KCTD8"    "CCDC125"
#> G13       "EAPP"      "CCK"      "FAM91A1"   "KLB"      "CCDC69"
#> G14       "EMC9"      "CDC25A"   "GDAP1"     "LAP3"     "CCNB1"
luadLuscNegResults = resultsProcess(peel.results = luadLuscNegPeel, posDT = pD[["posDT"]])
luadLuscNegResults[1:20,]
#>           [,1]        [,2]       [,3]        [,4]       [,5]
#> chr       "3"         "2"        "12"        "8"        "19"
#> peelStart "93873051"  "46293667" "66767"     "35235475" "29001670"
#> peelStop  "197956610" "88861563" "32646050"  "42896275" "58558954"
#> peakLoc   "181711925" "61888724" "1791963"   "38728186" "39825350"
#> peakVal   "-1.366"    "-0.4208"  "-0.3748"   "-0.3373"  "-0.3367"
#> peakPVal  "0.001"     "0.001"    "0.01"      "0.046"    "0.046"
#> G1        "ABHD10"    "ADD2"     "C12orf71"  "ADAM3A"   "ALDH16A1"
#> G2        "ABTB1"     "AFTPH"    "CACNA2D4"  "ADAM9"    "ALKBH6"
#> G3        "ACTL6A"    "ASPRV1"   "CLEC12B"   "ERLIN2"   "APLP1"
#> G4        "ADPRH"     "B3GNT2"   "DCP1B"     "GINS4"    "BCL2L12"
#> G5        "B3GNT5"    "C2orf42"  "DDX11"     "MIR4469"  "BLVRB"
#> G6        "BBX"       "C2orf73"  "FGD4"      "PLAT"     "C19orf47"
```

```
#> G7        "CCDC50"    "CCT7"      "FGFR1OP2" "PLEKHA2"    "C19orf48"
#> G8        "CCDC80"    "COMMD1"    "FKBP4"    "RN7SL709P" "C19orf81"
#> G9        "CD80"      "CYP26B1"   "FOXJ2"    "STAR"      "C5AR2"
#> G10       "CEP63"     "DOK1"      "GOLT1B"   "TACC1"     "CA11"
#> G11       "CHST2"     "EGR4"      "ING4"     "UNC5D"     "CACNG7"
#> G12       "CLDN1"     "EMX1"      "IQSEC3"   NA          "CALM3"
#> G13       "CLDN18"    "EPAS1"     "KCNA6"    NA          "CCDC9"
#> G14       "CLSTN2"    "ETAA1"     "LDHB"     NA          "CD79A"
```

References

Beroukim et al. 2007. *Assessing the Significance of Chromosomal Aberrations in Cancer: Methodology and Application to Glioma. Proc. Nat. Acad. Sci.*

Ushey et al. 2020. *Reticulate: Interface to Python.* https://CRAN.R-project.org/package=reticulate.

van Dyk et al. 2016. *RUBIC Identifies Driver Genes by Detecting Recurrent DNA Copy Number Breaks. Nature Comm.* https://doi.org/10.1038/ncomms12159.

Walter et al. 2011. *DiNAMIC: A Method to Identify Recurrent DNA Copy Number Aberrations in Tumors. Bioinformatics.*

———. 2015. *Consistent Testing for Recurrent Genomic Aberrations. Biometrika.*