# MCbiclust Tutorial

*Robert Bentham*

*23 September 2015*

## Contents

## About

MCbiclust is a R package for running massively correlating biclustering analysis on gene expression data. This tutorial aims to give a full guide on the installation and use of this package as demonstrated on data from the Cancer Cell Line Encyclopedia.

## Installation and loading

Raw code for the package is available at https://github.com/rbentham/MCbiclust. To load on your machine follow these steps:

1. Create a folder called MCbiclust containing all the files on the github.
2. On terminal/command line, go to the directory containing the MCbiclust folder.
3. Run command `R CMD build MCbiclust` on the terminal/command line. This builds the file "MCbiclust_1.0.0.tar.gz"
4. While running R the package can now be installed from source

```
install.packages(path_to_file, repos = NULL, type="source")
```

Where "path_to_file" on windows will be replaced by the path to the file ( e.g. for windows "C:\MCbiclust_1.0.0.tar.gz" or on linux/mac "/Users/bobbybentham/MCbiclust_1.0.0.tar.gz")

The package can now be loaded, with others that are necessary for the analysis

```
library(MCbiclust)
library(gplots)
library(ggplot2)
library(pander)
```

## Load example CCLE data with known mitochondrial genes

For this example analysis we will be seeking to find biclusters related to mitochondrial function in the cancer cell line encyclopedia. For this two datasets are needed, both of which are available on the MCbiclust package. The first in `CCLE_data` that contains the gene expression values found in the CCLE data set, the second, `Mitochondrial_genes`, is a list of mitochondrial genes that can be found from MitoCarta1.0.

```
data(CCLE_data)
data(Mitochondrial_genes)
```

It is a simple procedure to create a new matrix `CCLE.mito` only containing the mitochondrial genes. While there are 1023 known mitochondrial genes, not all of these are measured in `CCLE_data`.

```
mito.loc <- which(as.character(CCLE_data[,2]) %in% Mitochondrial_genes)
CCLE.mito <- CCLE_data[mito.loc,-c(1,2)]
row.names(CCLE.mito) <- CCLE_data[mito.loc,2]
```

## Find bicluster seed for first of these initial seeds

The first step in using MCbiclust is to find a subset of samples that have the most highly correlating genes in the chosen gene expression matrix. This is done by, calculating the associated correlation matrix and then calculating the absolute mean of the correlations, as a correlation score.

This is achieved with function `FindSeed`, the argument `gem` stands for gene expression matrix, `seed.size` indicates the size of the subset of samples that is sought. `iterations` indicates how many iterations of the algorithm to carry out before stopping, in general the higher the iterations the more optimal the solution in terms of maximising the strength of the correlation.

For reproducibility `set.seed` has been used to set R's pseudo-random number generator. It should also be noted that the for `gem` the data matrix can not contain all the genes, since `FindSeed` involves the calculation of correlation matrices which are not computationally efficient to compute if they involve greater than ~1000 genes.

```
set.seed(102)
CCLE.seed <- FindSeed(gem = CCLE.mito,
                      seed.size = 10,
                      iterations = 10000)
```

`FindSeed` has two additional options, `initial.seed` allows the user to specify the initial subsample to be tested, by default the initial sample subset is randomly chosen. The second is `full.detail` which by default is set to `FALSE`, when set to `TRUE` this returns a `data.frame` containing the initial subsample list and how it evolves over the iterations.

There is a function that can calculate the correlation score used in FindSeed, though in general you should not need to use it, unless you wish to manually check the chosen seed is an improvement on one that is randomly generated.

```
set.seed(103)
random.seed <- sample(seq(length = dim(CCLE.mito)[2]), 10)
CorScoreCalc(CCLE.mito,random.seed)
```

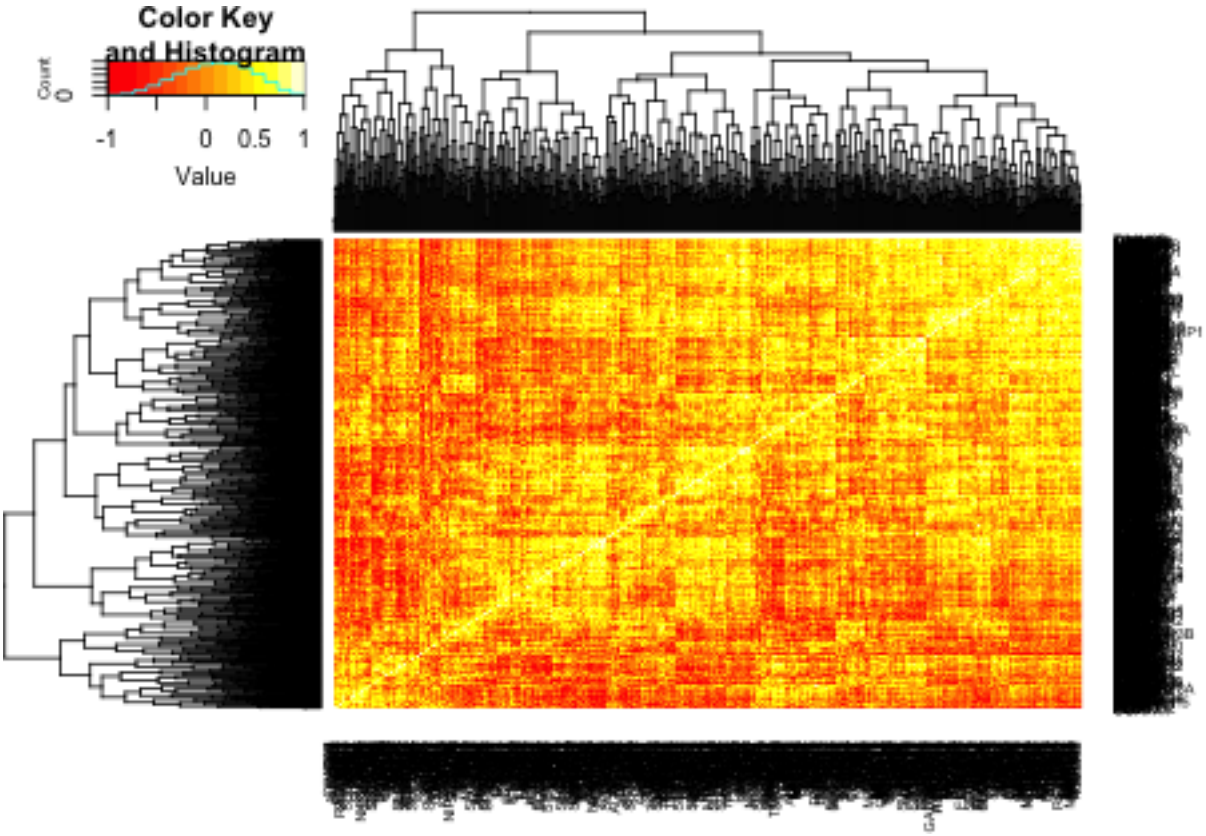
```
## [1] 0.3058819
```

```
CorScoreCalc(CCLE.mito,CCLE.seed)
```

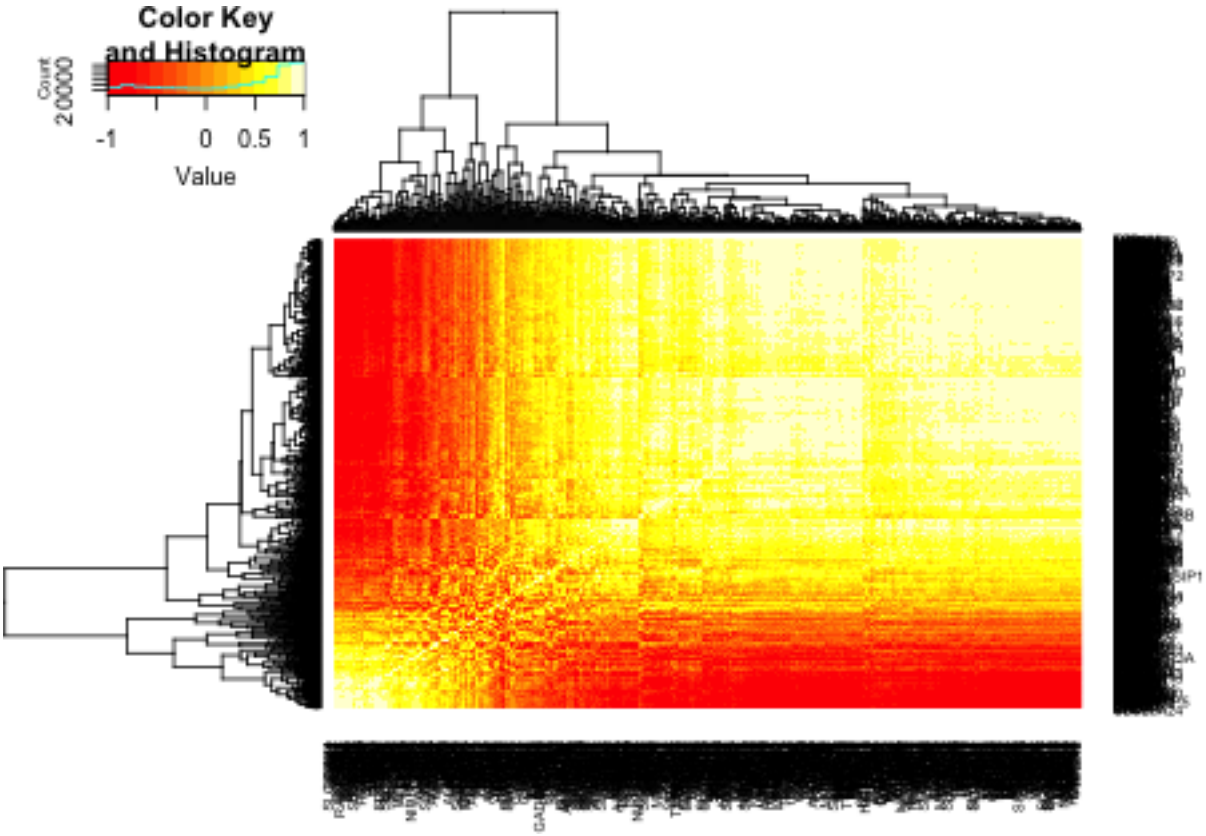
```
## [1] 0.6190576
```

The results of `FindSeed` can also be visualised by examining the associated correlation matrix, and viewing the result as a heatmap. Again it is easy to see the difference between the random subsample and the one outputted from FindSeed.

```
CCLE.random.cor <- cor(t(CCLE.mito[,random.seed]))
heatmap.2(CCLE.random.cor,trace = "none")
```




```
CCLE.mito.cor <- cor(t(CCLE.mito[,CCLE.seed]))
heatmap.2(CCLE.mito.cor,trace = "none")
```

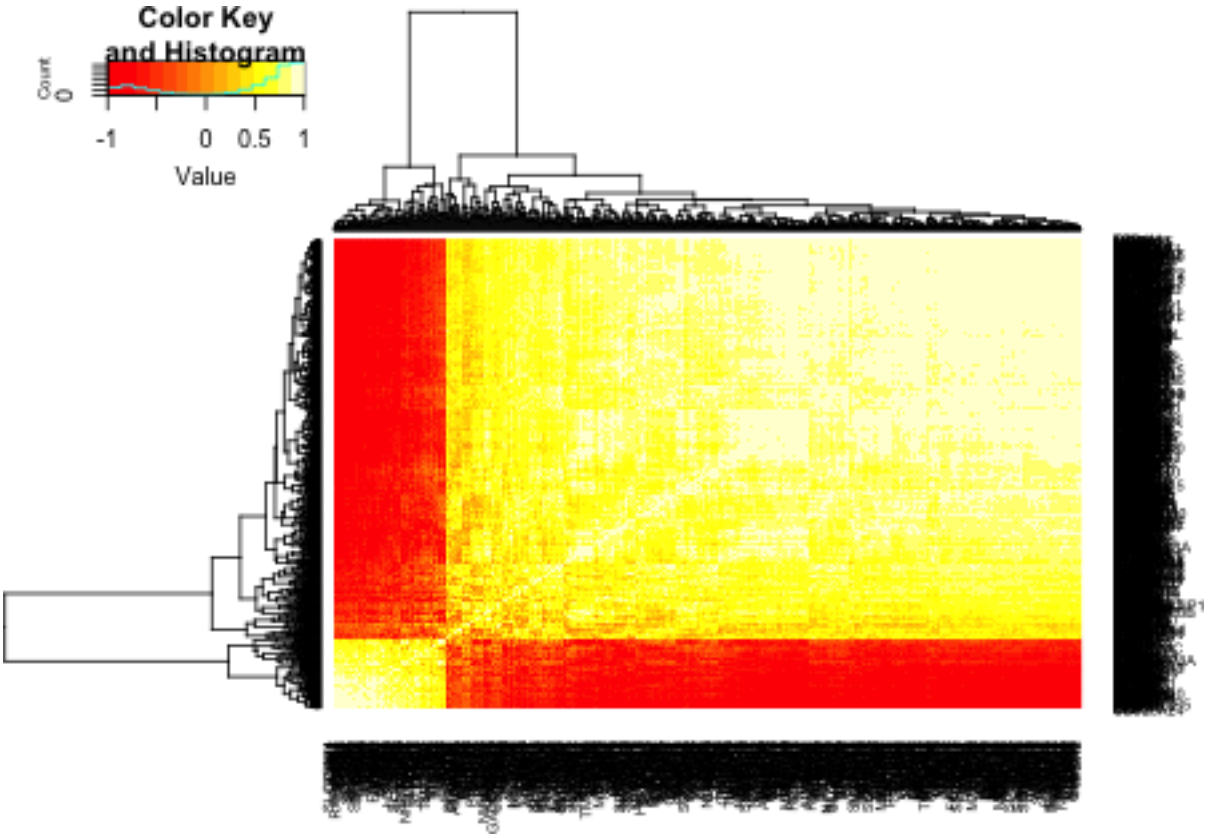



Note that when the genes are represented as the rows in a matrix, that matrix needs to be transposed before the calculation of the correlation matrix.

`heatmap.2` is a function from the `gplots` R package.

As can be clearly seen from the heat map, not all the mitochondrial genes are equally strongly correlated to each other. There is a function in `MCbiclust` which automatically selects those genes that are most strongly associated with the pattern. This function is `HclustGenesHiCor` and it works by using hierarchical clustering to select the genes into n different groups, and then discarding any of these groups that fails to have a correlation score greater than the correlation score from all the genes together.

```
CCLE.hicor.genes <- HclustGenesHiCor(CCLE.mito,CCLE.seed,cuts = 8)
CCLE.mito.cor2 <- cor(t(CCLE.mito[as.numeric(CCLE.hicor.genes),CCLE.seed]))
CCLE.heat <- heatmap.2(CCLE.mito.cor2,trace = "none")
```




There are two groups of genes, strongly correlated to themselves and anti-correlated to each other.

```
CCLE.groups <- list(labels(CCLE.heat$rowDendrogram[[1]]),
                    labels(CCLE.heat$rowDendrogram[[2]]))
```

## Calculate the correlation vector

As was seen in the last section, a distinct correlation pattern was found. However this was only examined for genes involved in mitochondrial function. Non-mitochondrial genes are likely also involved in this pattern and it is important to identify them.

All genes can be measured by how they match to this pattern in two steps. The first step is to summarise this pattern. This is done by finding a subset of genes which all strongly correlate with each other, and calculating their average expression value. The function `GeneVecFun` achieves this step. Similarly to `HclustGenesHiCor` the genes are clustered into groups using hierarchical clustering, but the best group is judged by the correlation score multiplied by the square root of the number of genes. This is done to bias against selecting a group of very small genes.

The second function `CalcCorVector` calculates the correlation vector by calculating the correlation of the average expression value found in the first step to every gene measured in the data set. This value is called the correlation vector.

```
CCLE.gene.vec <- GeneVecFun(CCLE.mito, CCLE.seed, 10)
CCLE.cor.vec <- CalcCorVector(gene.vec = CCLE.gene.vec,
                              gem = CCLE_data[,-c(1,2)][,CCLE.seed])
```

## Gene Set Enrichment

Using the calculated correlation vector, it is a relatively simple task to perform gene set enrichment. This can be done on any platform (e.g. DAVID, gprofiler, etc.) but MCbiclust comes with an inbuilt function for calculating GO enrichment values using the Mann-Whitney non-parametric test.

```
GSE.MW <- GOEnrichmentAnalysis(gene.names = as.character(CCLE_data[,2]),
                               gene.values = CCLE.cor.vec,
                               sig.rate = 0.05)
```

There are 1082 significant terms and the top 10 most significant can be viewed below:

| | GOID | TERM | num.genes |
|---|---|---|---|
| **1404** | GO:0006396 | RNA processing | 993 |
| **14993** | GO:0003723 | RNA binding | 1808 |
| **13638** | GO:0012505 | endomembrane system | 4489 |
| **13788** | GO:0030529 | ribonucleoprotein complex | 744 |
| **2489** | GO:0009653 | anatomical structure morphogenesis | 3188 |
| **14696** | GO:0098588 | bounding membrane of organelle | 2558 |
| **17948** | GO:0044822 | poly(A) RNA binding | 1170 |
| **3308** | GO:0016071 | mRNA metabolic process | 885 |
| **13965** | GO:0031981 | nuclear lumen | 2785 |
| **13496** | GO:0005794 | Golgi apparatus | 1552 |

Table 1: Table continues below

| | g.in.genelist | p.value | g.av.value |
|---|---|---|---|
| **1404** | 629 | 6.343e-84 | 0.7072 |
| **14993** | 1346 | 3.887e-70 | 0.5299 |
| **13638** | 3002 | 2.523e-64 | 0.02952 |
| **13788** | 528 | 5.976e-63 | 0.6651 |
| **2489** | 2184 | 4.844e-60 | 0.01128 |
| **14696** | 1891 | 1.365e-58 | -0.005724 |
| **17948** | 1019 | 1.763e-55 | 0.5275 |
| **3308** | 537 | 1.814e-54 | 0.6566 |
| **13965** | 1990 | 2.273e-53 | 0.4681 |
| **13496** | 1173 | 1.978e-52 | -0.07116 |

## Sample ordering

Already all the genes in the data set have had the correlation calculated to the pattern found. One more task that can be readily done is to order the samples according to the strength of correlation. Function `FindSeed` found the initial $n$ samples that had a very strong correlation with the gene set of interest, the $n+1$ sample is to be selected as that sample which best maintains the correlation strength, this process can be simply repeated until all or the desired number of samples are ordered.

`SampleSort` is the function in `MCbiclust` that completes this procedure, it has 4 main inputs, the first is the gene expression matrix with all the samples and the gene set of interest. `seed` is the initial subsample found with `FindSeed`. For increasing what can be a very slow computation, the code can be run on multiple cores, with the number of cores selected from the argument `num.cores` and instead of sorting the entire length, only the first `sort.length` samples need to be ordered.

```
CCLE.samp.sort <- SampleSort(CCLE.mito[as.numeric(CCLE.hicor.genes),],
                             seed = CCLE.seed,num.cores = 3,
                             sort.length = 100)
```

```
## Loading required package: compiler
```

```
## [1]   0.6499904 666.0000000  10.0000000
## [1]   0.6123552 296.0000000  20.0000000
## [1]   0.5692166 266.0000000  30.0000000
## [1]   0.5326192 903.0000000  40.0000000
## [1]   0.502641 863.000000  50.000000
## [1]   0.474983 729.000000  60.000000
## [1]  0.4495553 92.0000000 70.0000000
## [1]   0.4275554 829.0000000  80.0000000
## [1]   0.4088987 414.0000000  90.0000000
```

# PCA and ggplot2

Once the samples have been sorted it is possible to summarise the correlation pattern found using principal component analysis (PCA).

PCA is a method of dimensional reduction, and converts a data set to a new set of variables known as the principal components. These are designed to be completely uncorrelated or orthogonal to each other. In this way the principal components are new variables that capture the correlations between the old variables, and are in fact a linear combination of the old variables. The first principal component (PC1) is calculated as the one that explains the highest variance within the data, the second than is that which has the highest variance but is completely uncorrelated or orthogonal to the previous principal component. In this way additional principal components are calculated until all the variance in the data set is explained.

PC1 captures the highest variance within the data, so if PCA is run on the found bicluster with very strong correlations between the genes, PC1 will be a variable that summarises this correlation.

`PC1VecFun` is a function that calculates the PC1 values for all sorted samples. It takes three inputs: 1.`top.gem` is the gene expression matrix with only the most highly correlated genes but with all the sample data. 1. `seed.sort` is the sorting of the data samples found with function `SampleSort` 1. `n` is the number of samples used for initially calculating the weighting of PC1. If set to 10, the first 10 samples are used to calculate the weighting of PC1 and then the value of PC1 is calculated for all samples in the ordering.

```
top.mat <- CCLE.mito[as.numeric(CCLE.hicor.genes),]
pc1.vec <- PC1VecFun(top.gem = top.mat, seed.sort = CCLE.samp.sort, n = 10)
```

As an alternative to calculating PC1, the user may want to calculate the average expression value of certain gene sets. This gives a better idea of the type of regulation occurring in the correlation pattern, as an abstract notion of a principal component does not have to be understood.

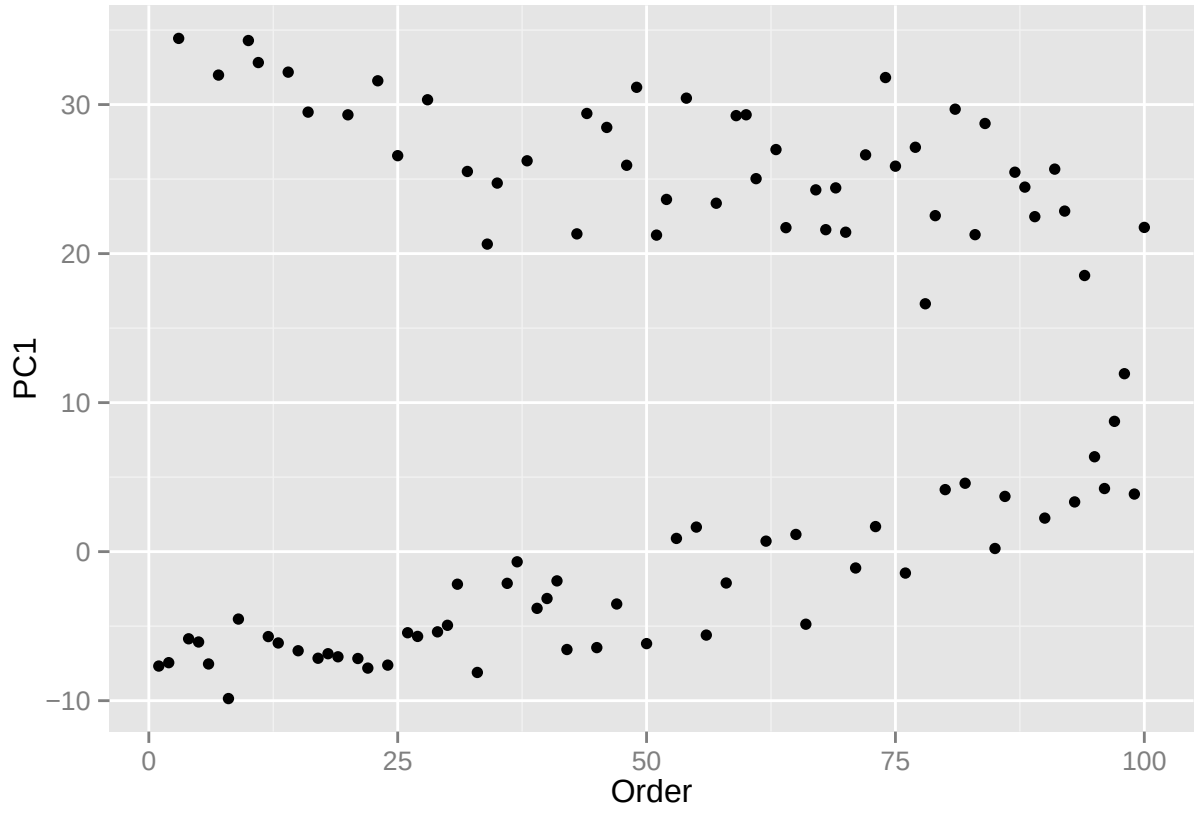
Either the two groups of mitochondrial genes strongly correlated to themselves and anti-correlated to each other could be used, or the top $n$ correlating and anti-correlating genes in the correlation vector, or those genes in the correlation vector above a particular threshold.

```
av.genes.group1 <- colMeans(CCLE.mito[CCLE.groups[[1]], CCLE.samp.sort])
av.genes.group2 <- colMeans(CCLE.mito[CCLE.groups[[2]], CCLE.samp.sort])

top.1000 <- order(CCLE.cor.vec, decreasing = T)[seq(1000)]
bottom.1000 <- order(CCLE.cor.vec, decreasing = F)[seq(1000)]

greater.0.9 <- which(CCLE.cor.vec > 0.9)
less.m0.9 <- which(CCLE.cor.vec < -0.9)

av.genes.top1 <- colMeans(CCLE_data[top.1000, CCLE.samp.sort])
av.genes.top2 <- colMeans(CCLE_data[bottom.1000, CCLE.samp.sort])

av.genes.thrhld1 <- colMeans(CCLE_data[greater.0.9, CCLE.samp.sort])
av.genes.thrhld2 <- colMeans(CCLE_data[less.m0.9, CCLE.samp.sort])
```

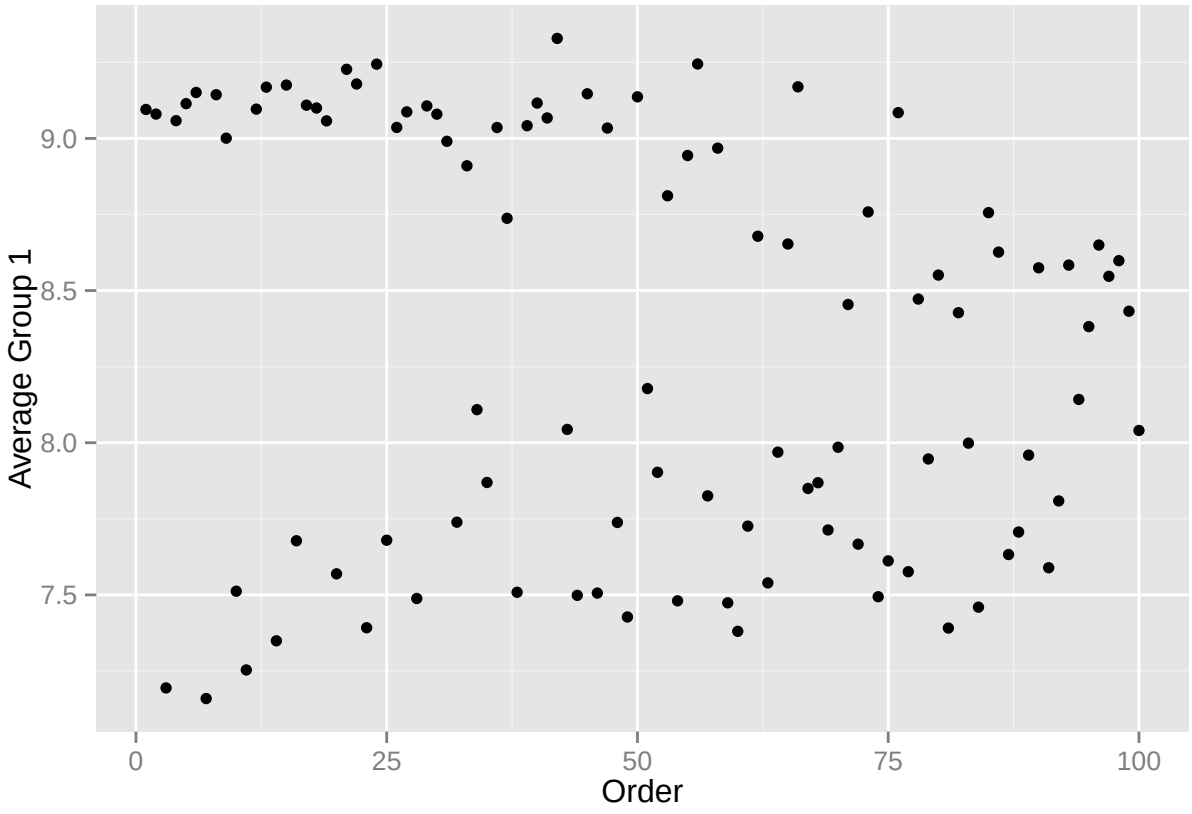
Once the samples have been ordered and PC1 and the average gene sets calculated it is a simple procedure to produce plots of these against the ordered samples.

This is done here using the `ggplot2` package, a plotting package for R by Hadley Wickam.

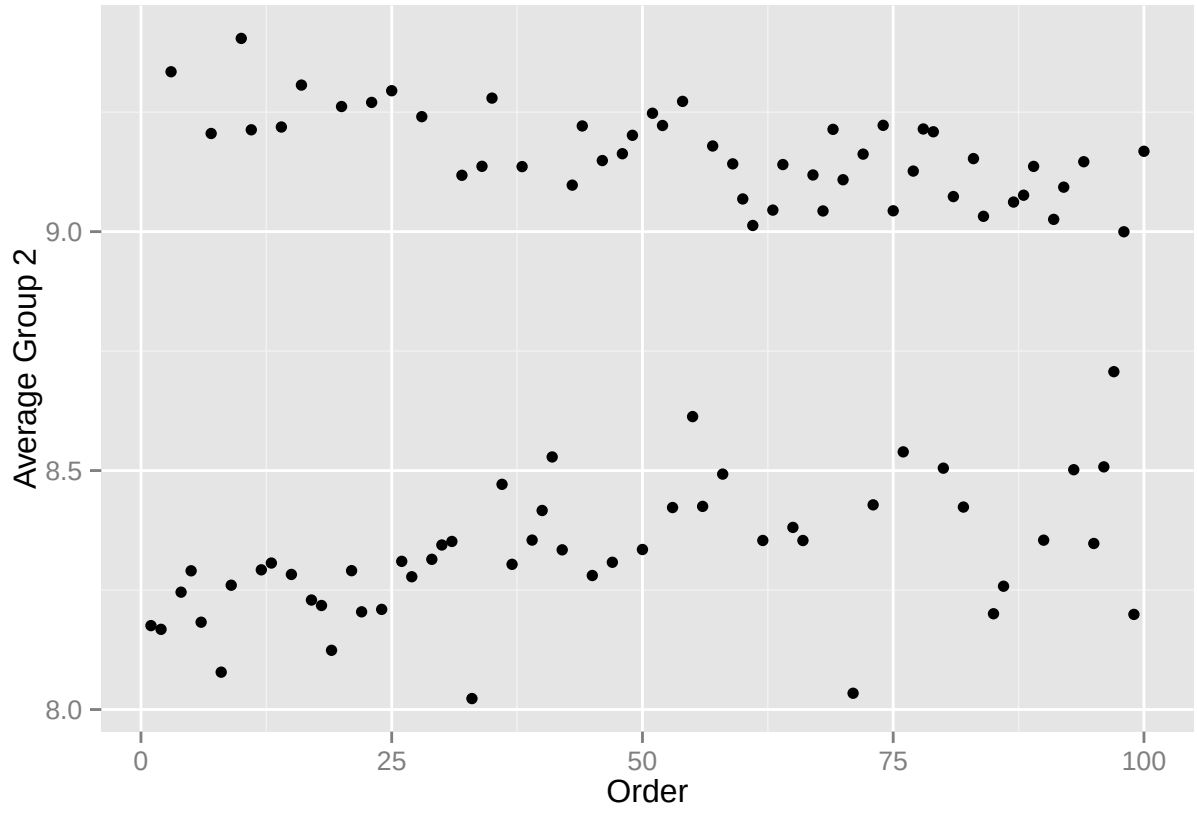
```
CCLE.df <- data.frame(CCLE.name = colnames(CCLE_data)[-c(1,2)][CCLE.samp.sort],
                      PC1 = pc1.vec,
                      Average.Group1 = av.genes.group1,
                      Average.Group2 = av.genes.group2,
                      Order = seq(length = length(pc1.vec)))

ggplot(CCLE.df, aes(Order,PC1)) +
  geom_point() + ylab("PC1")
```




```
ggplot(CCLE.df, aes(Order,Average.Group1)) +
  geom_point() + ylab("Average Group 1")
```




```
ggplot(CCLE.df, aes(Order,Average.Group2)) +
  geom_point() + ylab("Average Group 2")
```



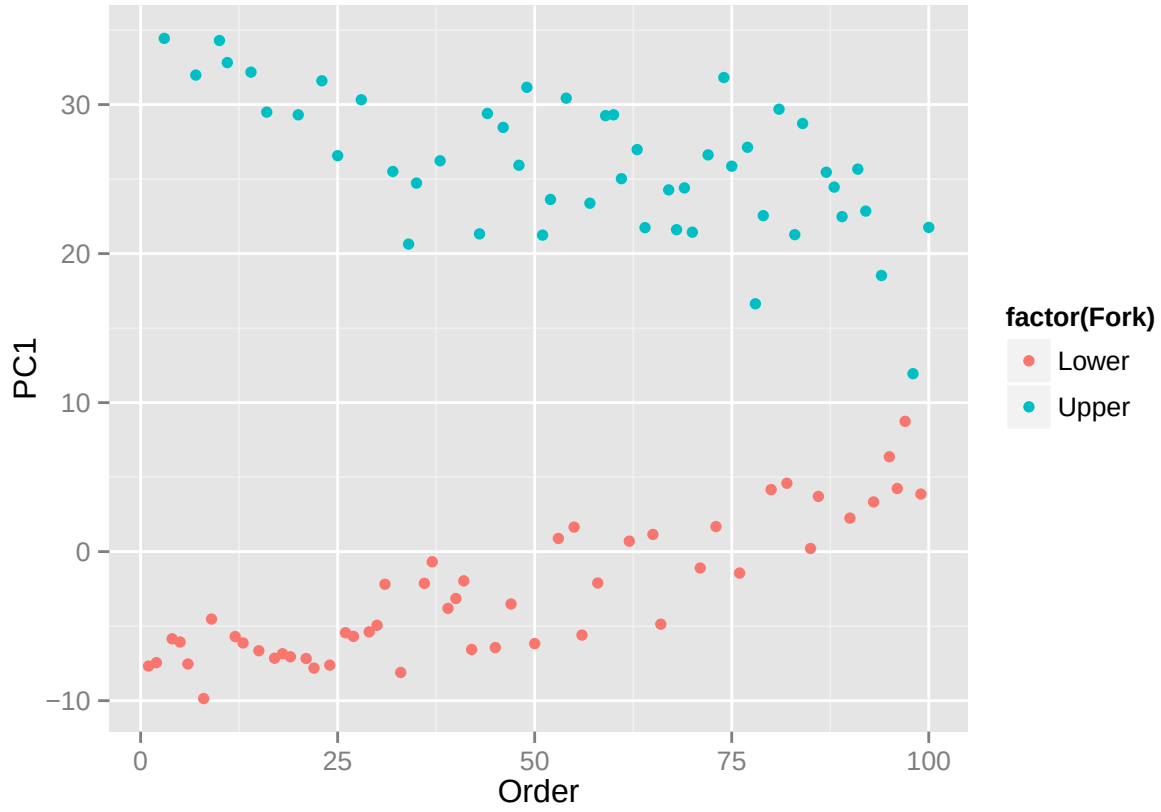

It is natural to classify the samples into different groups based on the value of PC1 or the average expression of the gene sets above. With a strong correlation pattern bicluster plotting PC1 against the sorted pattern results in a 'fork' like pattern as seen above. Those samples on the upper fork hence receive the label "Upper Fork" samples while those on the lower fork are known as "Lower Fork" samples.

Given a numeric vector of PC1 values, the samples can be classified as "Upper" or "Lower" using the `ForkClassifier` function. This function has two inputs, the PC1 numeric vector and how many of the first samples do you wish to classify.

```
fork.class <- ForkClassifier(pc1.vec,100)
fork.status <- rep("Normal",length(pc1.vec))
fork.status[fork.class$Upper] <- "Upper"
fork.status[fork.class$Lower] <- "Lower"
```

This fork information can now be added to the plot.

```
CCLE.df <- data.frame(CCLE.name = colnames(CCLE_data)[-c(1,2)][CCLE.samp.sort],
                      PC1 = pc1.vec,
                      Order = seq(length = length(pc1.vec)),
                      Fork = fork.status)

ggplot(CCLE.df, aes(Order,PC1)) +
  geom_point(aes(colour=factor(Fork))) + ylab("PC1")
```




This is still however not very enlightening, and in the next sections additional information will be added to these plots.

## Comparing results with sample/copynumber data

This section will deal with two addition data sets both of which are available in the MCbiclust package.

1. CCLE sample information, a data set containing information for every sample in the data set, including gender of the patient the cell line was derived from, as well as the primary site it came from.
2. A data set of the log ratio copynumber for samples in the CCLE data set. Using this we data we can check if there are any regions of significantly different copy number between the two found groups.

This section is meant as an example of the type of analysis that can be done with additional data set. Each new data set may have different additional data available with it and may be in formats that need some extra work to become compatible with the results from the `MCbiclust` analysis.

### CCLE sample data

This data set is available within the `MCbiclust` package.

```
data(CCLE_samples)
```

The first step is to compare the column names of both data sets and to make sure we are dealing with the same correctly labeled samples.

```
CCLE.samples.names <- as.character(CCLE_samples[,1])
CCLE.data.names <- colnames(CCLE_data)[-c(1,2)]
CCLE.samples.names[seq(10)]
```

```
##  [1] "1321N1_CENTRAL_NERVOUS_SYSTEM" "143B_BONE"
##  [3] "22RV1_PROSTATE"                "2313287_STOMACH"
##  [5] "253JBV_URINARY_TRACT"          "253J_URINARY_TRACT"
##  [7] "42MGBA_CENTRAL_NERVOUS_SYSTEM" "5637_URINARY_TRACT"
##  [9] "59M_OVARY"                     "639V_URINARY_TRACT"
```

```
CCLE.data.names[seq(10)]
```

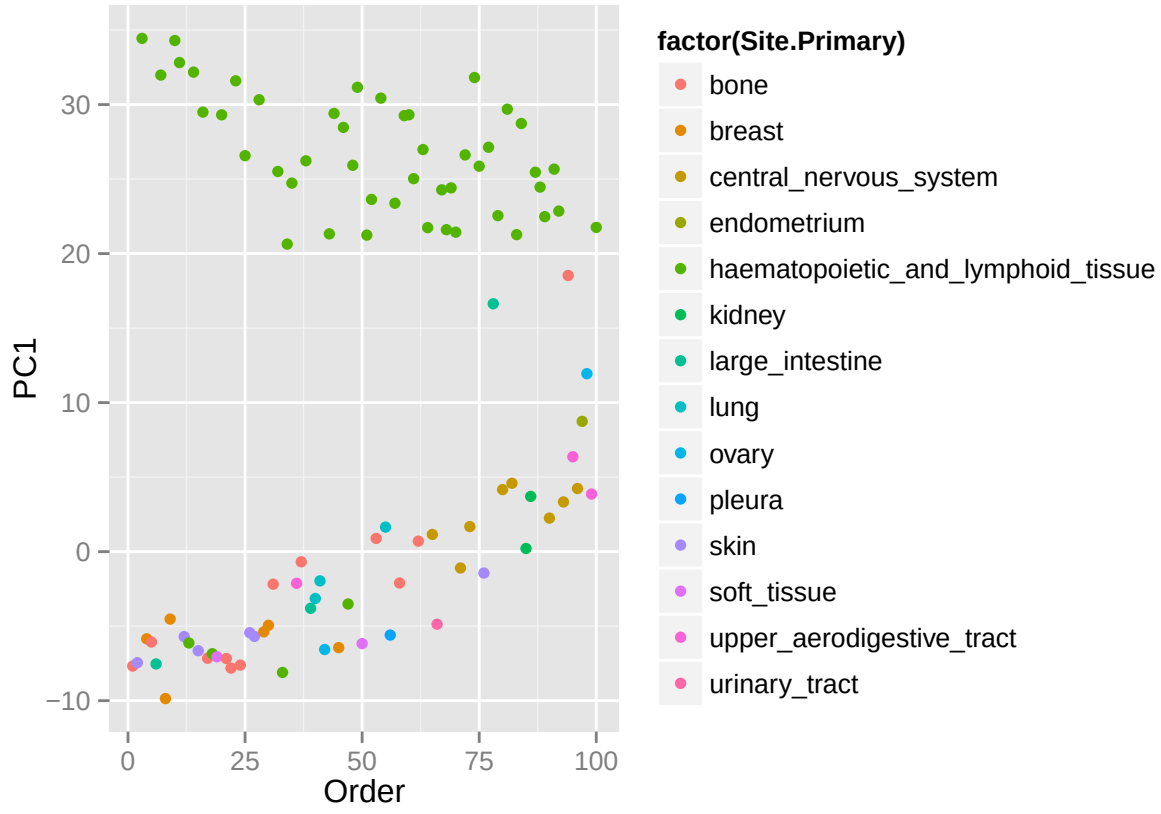
```
##  [1] "X1321N1_CENTRAL_NERVOUS_SYSTEM"
##  [2] "X143B_BONE"
##  [3] "X22RV1_PROSTATE"
##  [4] "X2313287_STOMACH"
##  [5] "X42MGBA_CENTRAL_NERVOUS_SYSTEM"
##  [6] "X5637_URINARY_TRACT"
##  [7] "X59M_OVARY"
##  [8] "X639V_URINARY_TRACT"
##  [9] "X647V_URINARY_TRACT"
## [10] "X697_HAEMATOPOIETIC_AND_LYMPHOID_TISSUE"
```

In this case some samples have an additional "X" not present in some CCLE_samples data so it is necessary to add it for consistency.

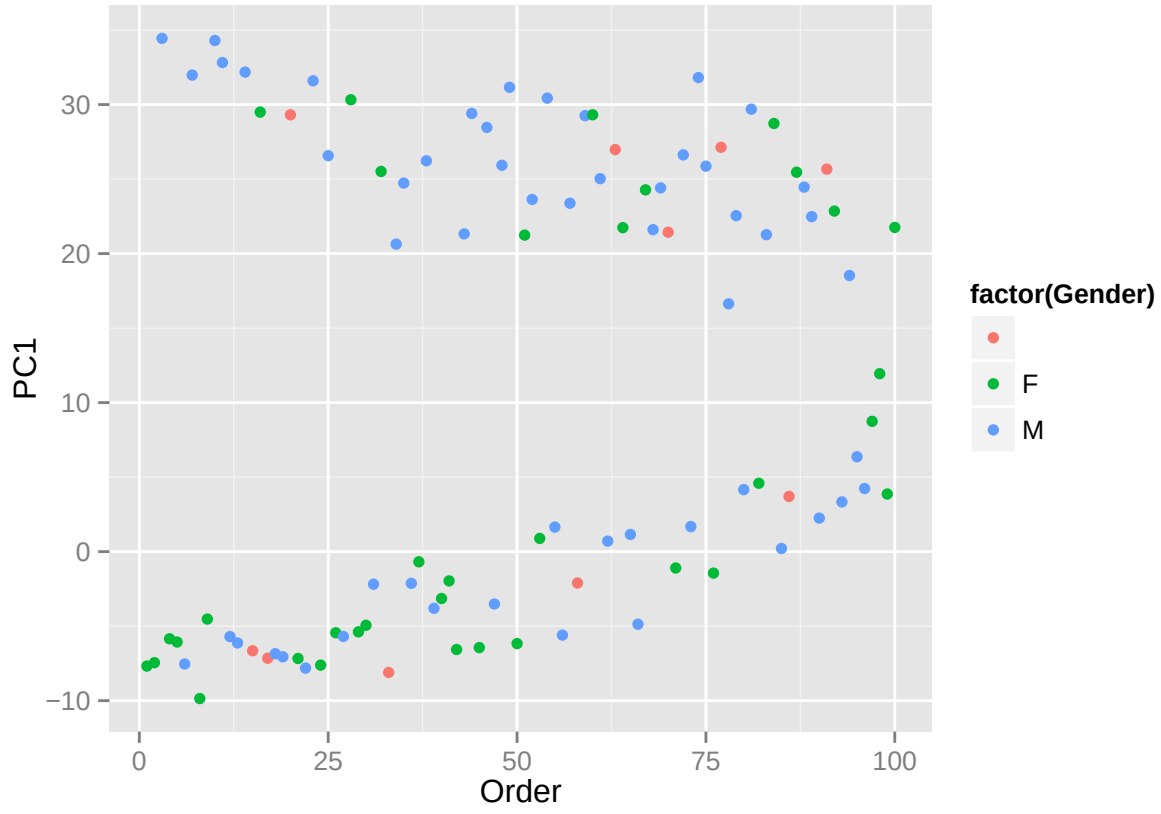
```
CCLE.samples.names[c(1:15)] <- paste("X",CCLE.samples.names[c(1:15)], sep="")
CCLE_samples$CCLE.name <- CCLE.samples.names
```

Using the `dplyr` library, it is possible to join this new data set to the one we made for plotting the values of PC1 in the previous section. This can be easily done as both datasets share a column - the name of the samples. Once this is done, it is again simple to produce additional plots.

```
library(dplyr)
CCLE.df.samples <- inner_join(CCLE.df,CCLE_samples,by="CCLE.name")

ggplot(CCLE.df.samples, aes(Order,PC1)) +
  geom_point(aes(colour=factor(Site.Primary))) + ylab("PC1")
```




```
ggplot(CCLE.df.samples, aes(Order,PC1)) +
  geom_point(aes(colour=factor(Gender))) + ylab("PC1")
```




Since in this case the data is categorical, it can be tested for significance using Pearson's chi squared test.

```
library(MASS)

# create contingency tables
ctable.site <- table(CCLE.df.samples$Fork, CCLE.df.samples$Site.Primary,
                     exclude = c("autonomic_ganglia", "biliary_tract",
                                 "liver","oesophagus","pancreas",
                                 "prostate","salivary_gland","small_intestine",
                                 "stomach","thyroid"))
ctable.gender <- table(CCLE.df.samples$Fork, CCLE.df.samples$Gender,
                       exclude = "U")

chisq.test(ctable.site)
```

```
## Warning in chisq.test(ctable.site): Chi-squared approximation may be
## incorrect
```

```
##
##  Pearson's Chi-squared test
##
## data:  ctable.site
## X-squared = 76.9374, df = 13, p-value = 4.141e-11
```

```
chisq.test(ctable.gender)
```

```
## Warning in chisq.test(ctable.gender): Chi-squared approximation may be
## incorrect
```

```
##
##  Pearson's Chi-squared test
##
## data:  ctable.gender
## X-squared = 4.8918, df = 2, p-value = 0.08665
```

As was easily apparent from examining the plots, the primary site the cell line is derived from is highly significant, while gender is not.

Next instead of categorical data, numerical data will be examined.

## Copynumber data

This data set is bigger and more complicated than the sample information, containing the copynumber alterations (CNA) across the entire genome. The aim of this analysis will be look for significant differences between the two groups identified as "Upper" and "Lower" fork samples.

```
data(CCLE_copy)
```

First, the data set must be made compatible with the gene expression data set. To do this, any genes which has a Log-ratio copynumber NA, is first removed.

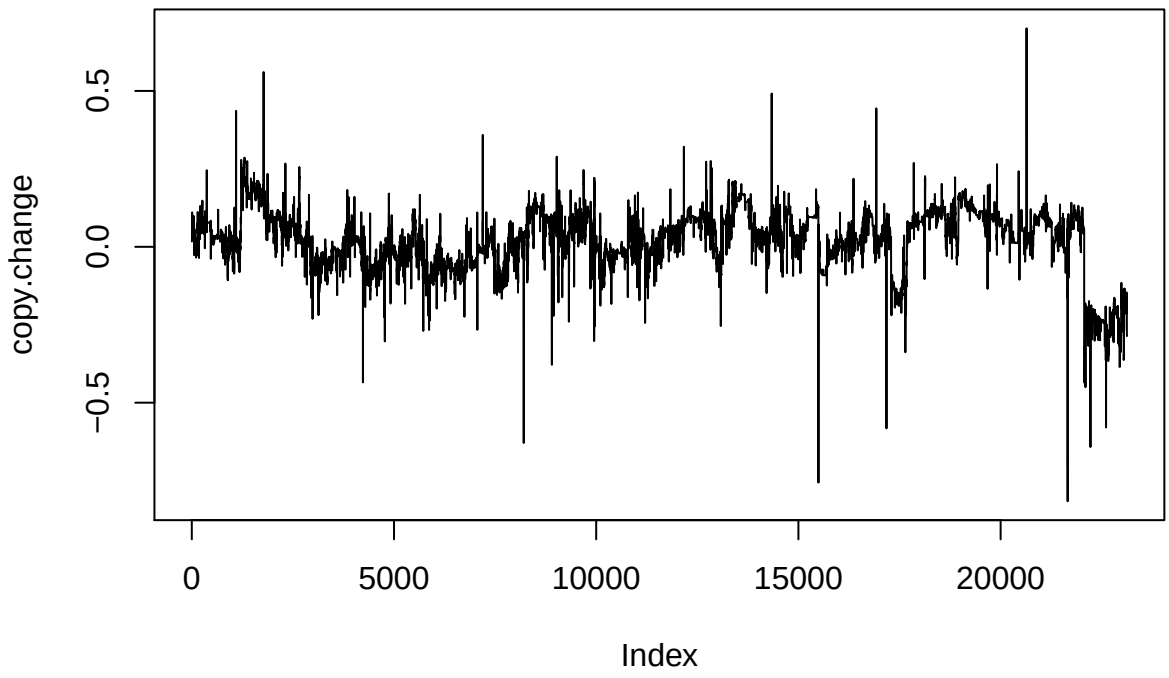
```
CCLE_copy <- CCLE_copy[-which(is.na(CCLE_copy[,5]) == T),]
```

In the copynumber data set there are some addition samples not present in the gene expression data set and vice versa. A map must be made to identify the same samples across the two data set.

```
CCLE.copy.names <- colnames(CCLE_copy)[-c(1:4)]

a2 <- which(CCLE.copy.names %in% CCLE.data.names)
a3 <- seq(length = length(CCLE.copy.names))[-a2]
```

In this case the samples are actually in the same order, but `CCLE_copy` has additional samples which are those measured in `a3`. From here the upper and lower fork samples can be selected and the average copynumber calculated, as well as the copy number change which can then be plotted.

```
a4 <- which(CCLE.df$Fork == "Upper")
a5 <- which(CCLE.df$Fork == "Lower")

av.copy.upper <- rowMeans(CCLE_copy[,-c(1:4)][,-(a3)][,CCLE.samp.sort[a4]])
av.copy.lower <- rowMeans(CCLE_copy[,-c(1:4)][,-(a3)][,CCLE.samp.sort[a5]])

copy.change <- av.copy.upper - av.copy.lower
plot(copy.change,type = "l")
```




The next step is to find in which regions of the genome is this copy change significant. This can most easily be done with using a permutation test.

The permutation test here works by randomly re-labeling the samples into new groups of the same size of the Upper and Lower fork groups. These new groups are then used to calculate a new copynumber change vector. This process is repeated 100 times, the p-values are then calculated from the resulting distribution of random copy-number changes and then undergone multiple hypothesis adjustment.

```
max.change2 <- list()
for(i in seq(100)){
  set.seed(i)
  rand.g1 <- sample(seq(100),length(a4))
  rand.g2 <- seq(100)[-rand.g1]
  rand.copy1 <- rowMeans(CCLE_copy[,-c(1:4)][,-(a3)][,CCLE.samp.sort[rand.g1]])
  rand.copy2 <- rowMeans(CCLE_copy[,-c(1:4)][,-(a3)][,CCLE.samp.sort[rand.g2]])

  max.change2[[i]] <- (abs(rand.copy1 - rand.copy2))
}

max.change2all <- unlist(max.change2)
all.pvalue <- seq(length = length(copy.change))
for(i in 1:length(copy.change)){
  all.pvalue[i] <- length(which(max.change2all > abs(copy.change[i])))/2312400
}

all.pvalue.adj <- p.adjust(all.pvalue)
sig.copy.change <- which(all.pvalue.adj < 0.05)
```

```
sig.copy.df <- cbind(CCLE_copy[sig.copy.change,c(1:4)],
                     Average_copy_g1=av.copy.upper[sig.copy.change],
                     Average_copy_g2=av.copy.lower[sig.copy.change],
                     copy_change=copy.change[sig.copy.change],
                     pvalue_adj = all.pvalue.adj[sig.copy.change])
```

| | geneName | NumChr | txStart | txEnd |
|---|---|---|---|---|
| **21658** | IGLL5 | 22 | 21559959 | 21568013 |

Table 3: Table continues below

| | Average_copy_g1 | Average_copy_g2 | copy_change |
|---|---|---|---|
| **21658** | -0.4653 | 0.3508 | -0.8161 |

Table 4: Table continues below

| | pvalue_adj |
|---|---|
| **21658** | 0.05 |

## Dealing with multiple runs

`MCbiclust` is a stochastic method so for best results it needs to be run multiple times, in practice this means using high-performance computing the run the algorithm on a computer cluster which will be dealt with in a later section. Here however the task of dealing with the results will be looked at. The algorithm will be run 100 times with only 500 iterations each. Typically more iterations are required, but for this demonstration it will be sufficient.

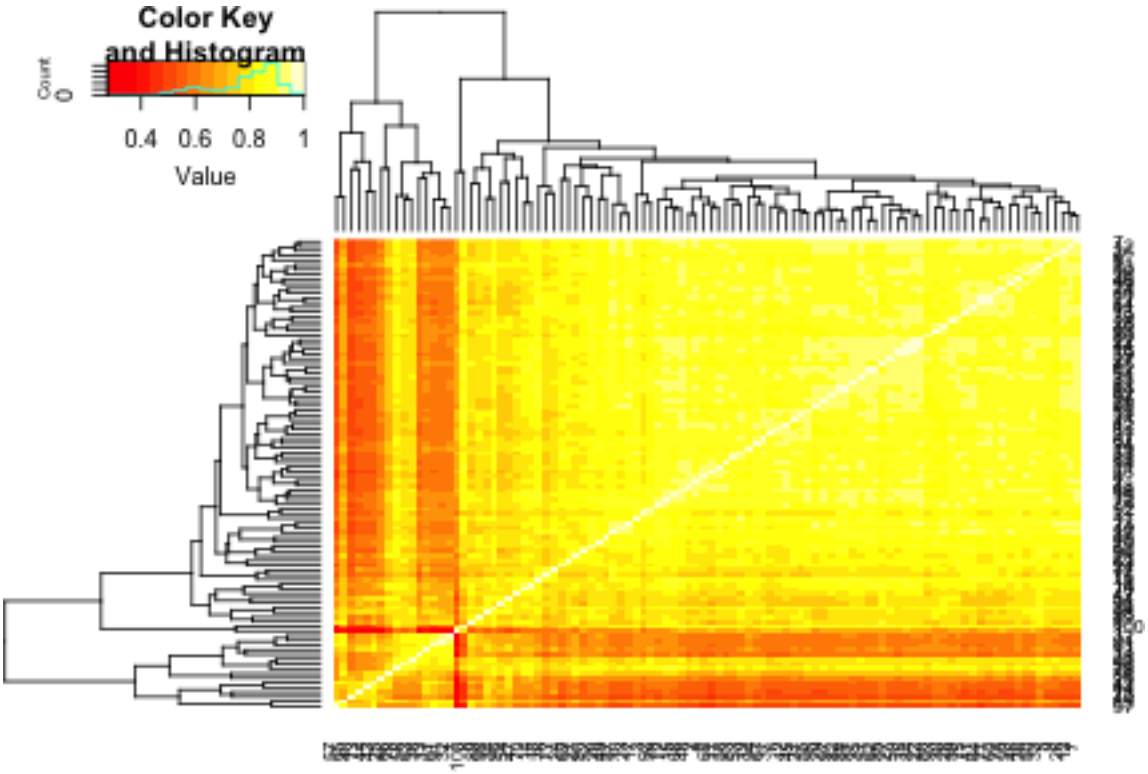
```
CCLE.multi.seed <- list()
initial.seed1 <- list()

for(i in seq(100)){
  set.seed(i)
  initial.seed1[[i]] <- sample(seq(length = dim(CCLE.mito)[2]),10)
  CCLE.multi.seed[[i]] <- FindSeed(gem = CCLE.mito,
                                   seed.size = 10,
                                   iterations = 500,
                                   initial.seed = initial.seed1[[i]])
}
```

The associated correlation vector must also be calculated for each run and these correlation vectors can be put into a matrix.

```
CCLE.cor.vec.multi <- list()

for(i in seq(100)){
  CCLE.gene.vec <- GeneVecFun(CCLE.mito, CCLE.multi.seed[[i]], 10)
  CCLE.cor.vec.multi[[i]] <- CalcCorVector(gene.vec = CCLE.gene.vec,
                                           gem = CCLE_data[,-c(1,2)][,CCLE.multi.seed[[i]]])
}

multi.run.cor.vec.mat <- matrix(0,length(CCLE.cor.vec.multi[[1]]),length(CCLE.cor.vec.multi))
for(i in 1:100){
  multi.run.cor.vec.mat[,i] <- CCLE.cor.vec.multi[[i]]
}
rm(CCLE.cor.vec.multi)
```

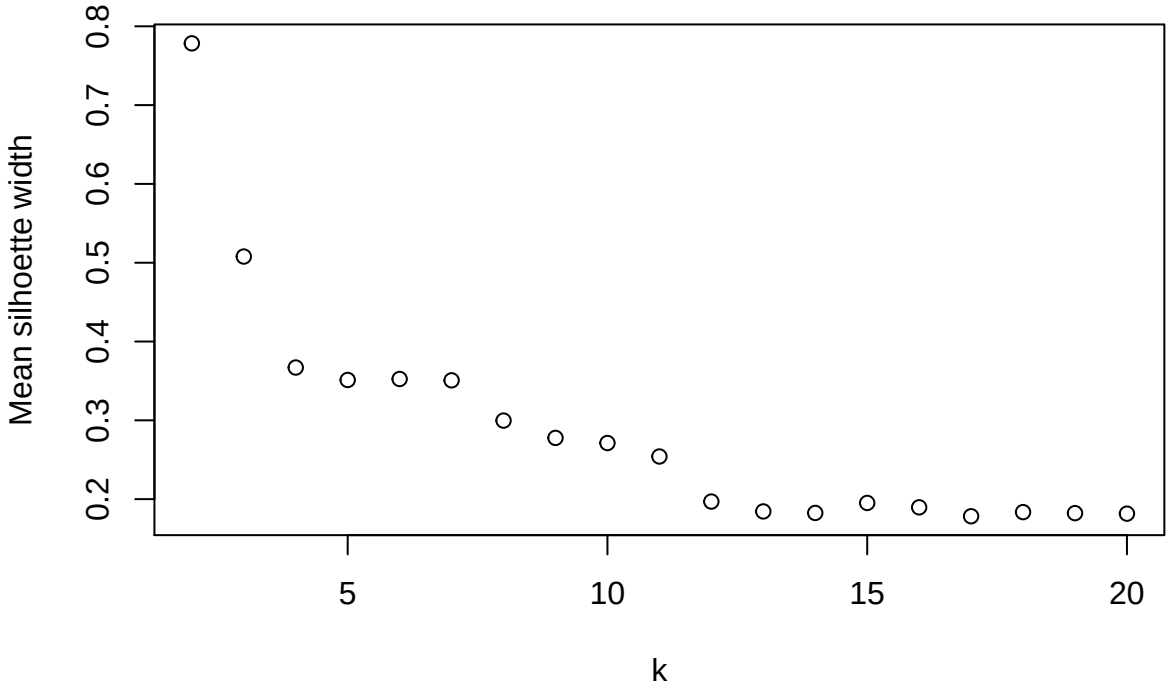
A correlation matrix can be formed from the correlation vectors, and in this way they can be viewed as a heatmap.

```
routput.corvec.matrix.cor.heat <- heatmap.2(abs(cor((multi.run.cor.vec.mat))),trace="none",
                                            distfun = function(c){as.dist(1 - abs(c))})
```

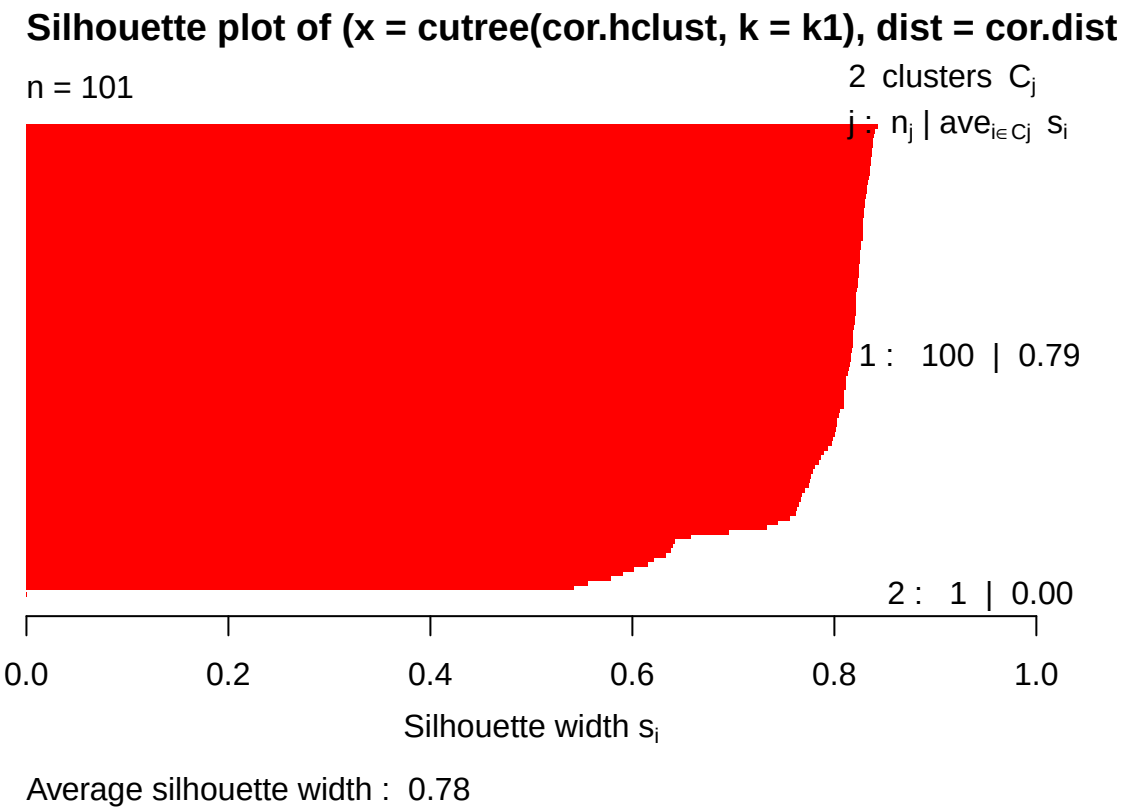



It needs to be known how many distinct patterns have been found, this is done with clustering and particular silhouette coefficients to judge what number of clusters is optimum within the data. Function `SilhouetteClustGroups` achieves this and uses hierarchical clustering to split the patterns into clusters, for comparison a randomly generated correlation vector is also added to allow for the possibility that all patterns found are best grouped into a single cluster.

```
multi.clust.groups <- SilhouetteClustGroups(cor.vec.mat = multi.run.cor.vec.mat,
                                            max.clusters = 20,
                                            plots = T)
```




```
## NULL
```

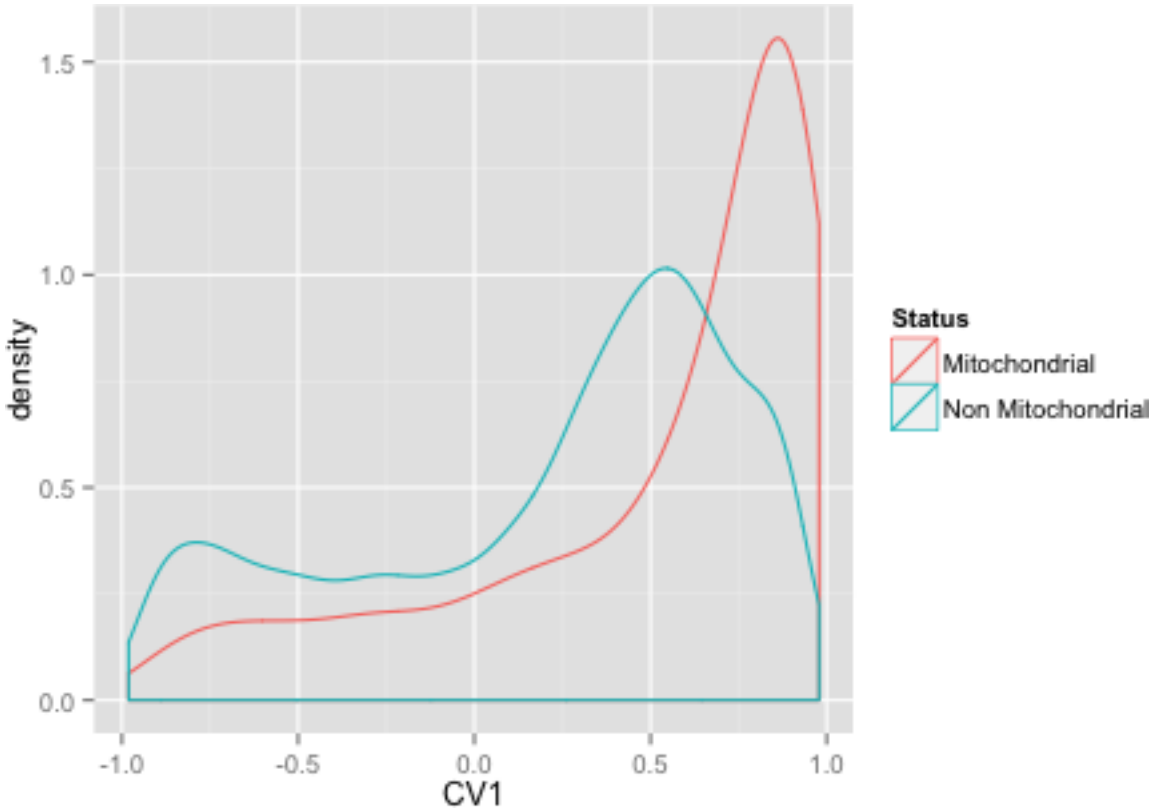



```
## NULL
```

Here only one cluster was found, and we can visualise this pattern (and any additional others found) with the function `CVPlot`, which highlights a chosen gene set, in this case the mitochondrial genes.

```
microarray.genes <- as.character(CCLE_data[,2])
average.corvec <- lapply(X = multi.clust.groups,
                         FUN = function(x) rowMeans(multi.run.cor.vec.mat[,x]))

CVPlot(cv.df = as.data.frame(average.corvec),
        geneset.loc = mito.loc,
        geneset.name = "Mitochondrial",
        alpha1 = 0.1)
```

```
##
## Attaching package: 'GGally'
##
## The following object is masked from 'package:dplyr':
##
##     nasa
```




As before can also calculate the gene set enrichment.

```
corvec.gsea <- lapply(X = average.corvec,
                      FUN = function(x) GOEnrichmentAnalysis(gene.names = microarray.genes,
                                                             gene.values = x,
                                                             sig.rate = 0.05))
```

Instead of using `SampleSort` a special sorting function for this case is used, `MultiSampleSort`. This works very similarly to `SampleSort` but selects the top genes in the average correlation vector and selects the initial seed with the highest correlation score corresponding to those genes.

```
CCLE.samp.multi.sort <-  MultiSampleSort(CCLE_data[,-c(1,2)], average.corvec, 750,
                            multi.clust.groups, initial.seed1,2,50)
```

```
## Loading required package: compiler
```

```
## [1]   0.8057651 905.0000000  10.0000000
## [1]   0.845124 286.000000  20.000000
## [1]   0.8573721 279.0000000  30.0000000
## [1]   0.8581973 729.0000000  40.0000000
```

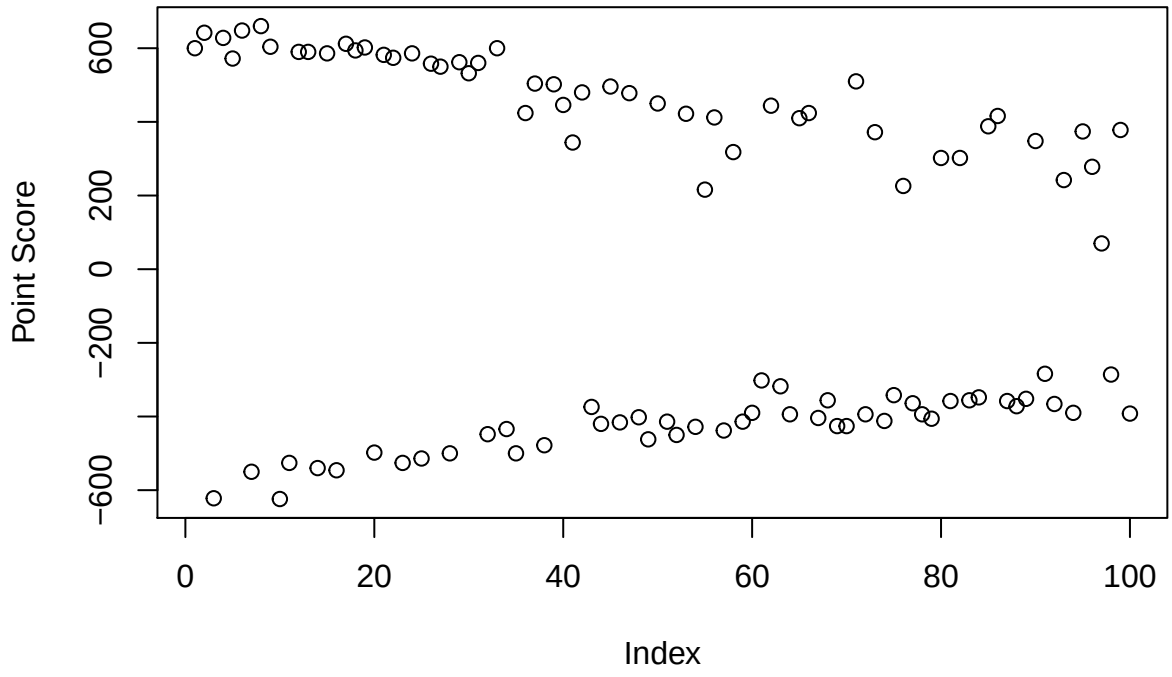
## Alternative methods based on known gene regulation groups

There are two alternative methods in the `MCbiclust` package, both to be used in scenarios when the regulation groups are known.

The first is the Point Scoring algorithm, `PointScore`, this scores the samples based on the gene expression in two given groups, $G1$ and $G2$. In any sample, for each gene in $G1$ if the expression is greater than the average expression for that gene, $+1$ is added to the score if it is below the average $-1$ is added. For genes in $G2$ the opposite is the case, with $+1$ being added if the gene is below the average and $-1$ being added if it is above average.

In this way each sample is assigned a score, and it can be shown to have the same "fork" pattern as found in PC1.

```
CCLE.point.score <- PointScoreCalc(gem = CCLE.mito[as.numeric(CCLE.hicor.genes), CCLE.samp.sort],
                                   gloc1 = CCLE.groups[[1]],gloc2 = CCLE.groups[[2]])

plot(CCLE.point.score, ylab = "Point Score", xlab = "Index")
```



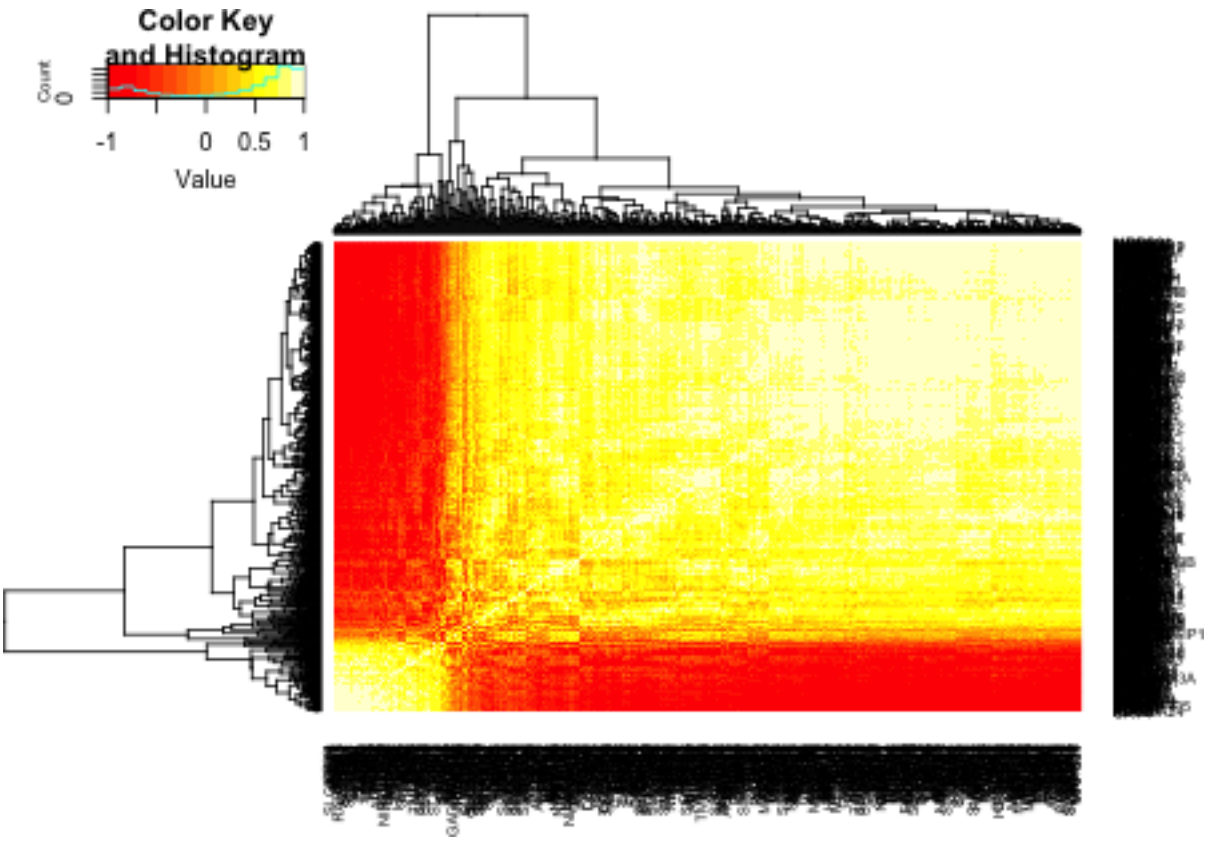

The second method is an alternative version of the `FindSeed` algorithm, that seeks a known pattern of two gene groups that are strongly correlated with each other but anti correlated with the other.

```
CCLE.seed.groups <- FindSeedGroups(gem = CCLE.mito,seed.size = 10,iterations = 10000,
                                   group1.loc = CCLE.groups[[1]], group2.loc = CCLE.groups[[2]])
```

```
CorScoreCalc(CCLE.mito[unlist(CCLE.groups),],CCLE.seed.groups)
```

```
## [1] 0.6891849
```

```
heatmap.2(cor(t(CCLE.mito[unlist(CCLE.groups),CCLE.seed.groups])),trace = "none")
```




## Running on a HPC (example for Legion)

What follows is an example script to be run on a HPC designed for the UCL Legion system. Note this script is not designed to run as is written, and will need to be adapted to suit the needs of the user. Two scripts are needed, a batch script and an R script. The Batch script will repeatedly run the RScript with different inputs (different random seeds) the results of which are saved to a sub-directory in Scratch.

```
########################################
# Legion_Example_Batch_Script.sh
########################################

#!/bin/bash -l
# Batch script to run an array job on Legion with the upgraded
# software stack under SGE.
# 1. Force bash
#$ -S /bin/bash
# 2. Request ten minutes of wallclock time (format hours:minutes:seconds).
#$ -l h_rt=0:27:0
# 3. Request 1 gigabyte of RAM.
#$ -l mem=1G
# 4. Request 10 gigabyte of TMPDIR space (default is 10 GB)
#$ -l tmpfs=10G
# 5. Set up the job array.  In this instance we have requested 1000 tasks
```

```
# numbered 1 to 1000.
#$ -t 1-1000
# 6. Set the name of the job.
#$ -N [JobName]
# 7. Select the project that this job will run under.
# Find <your_project_id> by running the command "groups"
#$ -P [ProjectName]
# 8. Set the working directory to somewhere in your scratch space.  This is
# a necessary step with the upgraded software stack as compute nodes cannot
# write to $HOME.
# Replace "<your_UCL_id>" with your UCL user ID :)
#$ -wd /home/[username]/Scratch/R_output

module unload compilers/intel/11.1/072
module unload mpi/qlogic/1.2.7/intel
module unload mkl/10.2.5/035


module load compilers/gnu/4.6.3
module load curl/7.21.3/gnu.4.6.3
module load atlas/3.8.3/gnu.4.6.3
module load gsl/1.15/gnu.4.6.3
module load fftw/3.3.1/double/gnu.4.6.3
module load hdf/5-1.8.7/gnu.4.6.3
module load netcdf/4.2.1.1/gnu.4.6.3
module load root/5.34.09/gnu.4.6.3
module load java/1.6.0_32
module load texlive/2012

module load r/3.0.1-atlas/gnu.4.6.3

Rscript /home/[username]/UCL_Legion_Example_Script.R $SGE_TASK_ID
```

```
########################################
# UCL_Legion_Example_Script.R
########################################

# load necessary arguments
args <- commandArgs(TRUE)
.libpaths("/path/to/R/librarys")

require(MCBiclust)

# load gene expression data to analyse
# data includes:
# 1. Gene expression matrix, (gem)
# 2. Any gene set locations of interest, e.g. mitochondrial (mito.loc)
# 3. (optional) a list of pre-generated initial seeds
load("/path/to/gene/expression/data")

# set random seed
set.seed(as.numeric(args[1]))
```

```
# can select a known gene set
gem.mito <- gem[mito.loc,]

# or alternatively can select a random gene set, e.g
# gem.random <- gem[sample(seq(length = dim(gem)[1]),1000),]

# Run the FindSeed algorithm
gem.seed <- FindSeed(gem = gem.mito, seed.size = 10,
                     iterations = 10000,
                     initial.seed = random.seed.list[as.numeric(args[1])])

# Calculate the correlation vector
gem.gene.vec <- GeneVecFun(gem.mito, gem.seed,splits = 20)
gem.cor.vec <- CalcCorVector(gene.vec = gem.gene.vec,
                             gem = gem[,gem.seed])

# Calculate the gene set enrichment

gem.gene.names <- rownames(gem)

GSE.MW <- GOEnrichmentAnalysis(gene.names = gem.gene.names,
                               gene.values = gem.cor.vec,
                               sig.rate = 0.05)

# Save data to files
write.table(data.frame(Seed=gem.seed),file=paste("Seed",args[1],"gem_list.txt",sep="_"))
write.table(data.frame(CV=gem.cor.vec),file=paste("Seed",args[1],"cor_vec.txt",sep="_"))
write.table(GSE.MW,file=paste("Seed",args[1],"GSE_MW.txt",sep="_"))
```

## Initial seed generation for HPC

If running on a HPC need multiple initial seeds, possible to generate them such that they are unlikely to contain many identical elements. With minimizing the overlap of initial seeds, a wider range of patterns should be seen.

This is achieved in the function `SeedGenerator` and aims to find initial seeds that satisfy Poisson-disc sampling properties. Poisson-disc sampling is often used in problems related to vision, this form of sampling selects random points which are tightly packed together but no closer than a specified minimum distance. In biology, the photoreceptor cells placement in the eye is an example of Poisson-disc sampling, since the method has few wasted photoreceptors, and the irregular placement means that vision is not susceptible to aliasing, where sampling of light causes different signals to become identical, or there is a distortion caused from the sampling. This is a major problem in computer vision for example when Moire patterns can occur.

The problem of finding initial seeds occurs in a multi-dimensional space and a combination of Bridson and Mitchell's best candidate algorithm that selects initial seeds with Poisson-disc sampling properties was used to form the algorithm in `SeedGenerator`.

```
seed.numbers <- 1000
random.seed.list <- SeedGenerator(seed.size = 10,
                                  numbers = seed.numbers,
                                  sample.length = dim(CCLE.mito)[2],
                                  break.num = 1,
                                  attempts = 100)
```

Note, for a long list of seeds takes awhile to generate.

It is easy to demonstrate the effectiveness of this seed generation vs randomly generated seeds.

a) Make list of every possible pair of the 1000 generated seeds

```
combinations <- lapply(apply(combn(seed.numbers,2),2,list),unlist)
```

b) Function to compare the length of intersection

```
len.inter <- function(x,y) length(intersect(x,y))
```

c) Generate seeds purely randomly

```
set.seed(101)
random.seed.matrix <- t(replicate(1000, sample(c(1:dim(CCLE.mito)[2]),10)))
random.seed.list2 <- lapply(apply(random.seed.matrix, 1, list),unlist)
```

d) Calculate intersections for both methods

```
intersections <- unlist(lapply(combinations,function(x) do.call(len.inter,random.seed.list[x])))
intersections2 <- unlist(lapply(combinations,function(x) do.call(len.inter,random.seed.list2[x])))

max(intersections)
```

```
## [1] 2
```

```
sum(intersections)
```

```
## [1] 49173
```

```
max(intersections2)
```

```
## [1] 3
```

```
sum(intersections2)
```

```
## [1] 51801
```

```
sum(intersections2) - sum(intersections)
```

```
## [1] 2628
```

```
rm(combinations)
```