An introduction to OSAT

Li Yan, Changxing Ma, Dan Wang, Qiang Hu, Maochun Qin, Jianmin Wang, Song Liu

`li.yan@roswellpark.org`

October 30, 2024

Contents

1 Overview

A sizable genomics study such as microarray often involves the use of multiple batches (groups) of experiment due to practical complication. The systematic, non-biological differences between batches are referred as batch effects. Batch effects are wide-spread occurrences in genomic studies, and it has been shown that noticeable variation between different batch runs can be a real concern, sometimes even larger than the biological differences[1–4]. Without sound

experiment design and statistical analysis methods to handle batch effects, misleading or even erroneous conclusions could be made [1,2,4]. This especially important issue is unfortunately often overlooked, partially due to the complexity and multiple steps involved in genomics studies.

To minimize batch effects, a careful experiment design should ensure the even distribution of biological groups and confounding factors across batches. A study where most tumor samples are collected from male and healthy ones from female will have difficulty ruling out the effect of gender. Likewise, it is equally problematic if one batch run contains most samples of a particular biological group. Therefore, in an ideal genomics design, the groups of the main interest, as well as important confounding variables should be balanced and replicated across the batches to form a Randomized Complete Block Design (RCBD) [5–7]. It makes the separation of the real biological effect of our interests and effects by other confounding factors statistically more powerful.

However, despite all best effort, it is often than not that the collected samples are not complying with the original ideal RCBD design. This is due to the fact that these studies are mostly observational or quasi-experimental since we usually do not have full control over sample availability. In clinical genomics study, samples may be rare, difficult or expensive to collect, irreplaceable, or even fail QC test before genomics profiling. As a result, experiments are often driven by the availability of the final collection of samples which is usually unbalanced and incomplete. The unbalance and incompleteness nature of sample availability in genomics study, without appropriate attention, could lead to drastic batch effects. Therefore, it is necessary to develop effective and handy tool to assign collected samples across batches in an appropriate way to minimize batch effects.

We developed a tool called OSAT (Optimal Sample Assignment Tool) to facilitate the allocation of collected samples to different batches. With minimum steps, it produces setup that optimizes the even distribution of samples in groups of biological interest into different batches, reducing the confounding or correlation between batches and the biological variables of interest. It can also optimize the even distribution of confounding factors across batches. Our tool can handle challenging instances where incomplete and unbalanced sample collections are involved as well as ideal balanced RCBD. OSAT provides several predefined batch layout for some of the most commonly used genomics platform. Written in a modularized style in the open source R environment, it provides the flexibility for users to define the batch layout of their own experiment platform, as well as optimization objective function for their specific needs, in sample-to-batch assignment to minimize batch effects.

Please see our paper at http://www.biomedcentral.com/1471-2164/13/689 for more information.

2   Demonstration

An example file is included for demonstration. It represent samples from a study where the primary interest is to investigate the expression differentiation in case versus control groups (variable `SampleType`).   `Race` and `AgeGrp` are clinically important variables that may have impact on final outcome. We consider them as confounding variables. A total of 576 samples are included in the study, with one sample per row in the example file.

We will use 6 Illumina 96-well HYP MultiBeadChip plates for this expression profiling experiment. Each plate will be processed at a time as a batch. This particular plate consists of 8 BeadChips, which in turn holds 12 wells in 6 rows and 2 columns. Each well holds an individual sample. We had predefined several commonly used batch layouts (e.g., chips or plates) to make the usage of OSAT straightforward and simple. See section 4.2.1 for a whole list of predefined chips and plates and more details. Our goal here is to assign samples with different characteristic as even as possible to each plate.

2.1   Example data

The data can be accessed by:

```
> library(OSAT)
> inPath <- system.file('extdata', package='OSAT')
> pheno <- read.table(file.path(inPath, 'samples.txt'),
+                     header=TRUE, sep="\t", colClasses="factor")
```

As shown in **Table 1**, the two groups in our primary interest `SampleType` are not balanced due to limitations in the sample collection process. To reduce possible batch effect across different plates, it is critical to make sure on each plate there are similar number of samples in each of the 2 stratas (Case/Control). Similarly, it is also preferred to

Table 1: Example Data

	ID	SampleType	Race	AgeGrp
X	1 : 1	Case :317	European:401	(0,30] : 75
X.1	10 : 1	Control:259	Hispanic:175	(30,40] :113
X.2	100 : 1			(40,50] :134
X.3	101 : 1			(50,60] :102
X.4	102 : 1			(60,100]:152
X.5	103 : 1			
X.6	(Other):570			

Table 2: Data distribution

	SampleType	Race	AgeGrp	Freq
1	Case	European	(0,30]	8
2	Control	European	(0,30]	58
3	Case	Hispanic	(0,30]	0
4	Control	Hispanic	(0,30]	9
5	Case	European	(30,40]	21
6	Control	European	(30,40]	54
7	Case	Hispanic	(30,40]	6
8	Control	Hispanic	(30,40]	32
9	Case	European	(40,50]	34
10	Control	European	(40,50]	52
11	Case	Hispanic	(40,50]	46
12	Control	Hispanic	(40,50]	2
13	Case	European	(50,60]	40
14	Control	European	(50,60]	44
15	Case	Hispanic	(50,60]	16
16	Control	Hispanic	(50,60]	2
17	Case	European	(60,100]	84
18	Control	European	(60,100]	6
19	Case	Hispanic	(60,100]	62
20	Control	Hispanic	(60,100]	0

have homogeneous makeup in each batch on confounding variables `Race/AgeGrp` which are not balanced as well. **Table 2** is the sample distribution when we included these three important variables considered here:

```
> with(pheno, as.data.frame(table(SampleType, Race, AgeGrp)))
```

In the following section, we showed how OSAT can be used to perform sample-to-batch assignment for this example data.

2.2 Working flow

First, sample pheno information and the variables considered (i.e., `SampleType, Race, AgeGrp`) can be captured by:

```
> gs <- setup.sample(pheno, optimal=c("SampleType", "Race", "AgeGrp"))
```

A recommendation for practice is to put the variable of primary interest as the first of the list.

Second, 6 Illumina 96-well HYP MultiBeadChip plates were needed for the 576 samples. Each plate was treated as a batch. We will refer this assembly of plates (i.e., batches) as container.

```
> gc <- setup.container(IlluminaBeadChip96Plate, 6, batch='plates')
```

we had predefined the layout of this particular plate in our package as `IlluminaBeadChip96Plate`. See section 4.2.1 for more details on predefined plates. If the number of sample is less than the number of available wells, the empty well(s) will be left randomly throughout the plates (section 5.1), or specified by user. Section 5.2 provides additional information on how to exclude certain wells from sample assignment.

Third, samples will be assigned into container using our default method which consists of a block randomization step followed by an step (this will take a few minutes):

```
> set.seed(123)      # to create reproducible result
> # demonstration only. nSim=5000 or more are commonly used.
> gSetup <- create.optimized.setup(sample=gs, container=gc, nSim=1000)
```

This is all one need to create an optimal sample-to-batch assignment using OSAT. All relevant output information is hold in the object `gSetup`. Details on methods implemented in OSAT can be found in the next section 3 and 4.3.

The sample assignment can be output to CSV by:

```
> write.csv(get.experiment.setup(gSetup),
+           file="gSetup.csv", row.names=FALSE)
```

the output CSV file is sorted by the original row order from the sample list.

After the sample-to-batch assignment, a MSA-4 robotic loader can be used to load the samples onto the MultiBead-Chip plate. If we are going to use a robotic loader, such as MSA 4 robotic loader, we can map our design to the loader and export the final lineup to a CSV file,

```
> out <- map.to.MSA(gSetup, MSA4.plate)
> write.csv(out, "MSAsetup.csv", row.names = FALSE)
```

the CSV file is sorted by the order used in the MSA robotic loader:

```
> head(out)
```

	MSA_plate	MSA_ID	plates	wellID	ID	SampleType	Race	AgeGrp	unusedWell
1	1	A01	1	1	162	Case	Hispanic	(60,100]	FALSE
2	1	B01	1	2	214	Case	European	(50,60]	FALSE
3	1	C01	1	3	475	Control	European	(0,30]	FALSE
4	1	D01	1	4	565	Control	Hispanic	(0,30]	FALSE
5	1	E01	1	5	72	Case	Hispanic	(50,60]	FALSE
6	1	F01	1	6	446	Control	European	(0,30]	FALSE

2.3   Summary of Results

A quick look at the sample distribution across the batches in our design created by OSAT:

```
> QC(gSetup)
```

One can perform Pearson's Chi-squared test to examine the association between batches and each of the variables considered. The results indicate that all variables considered are all highly uncorrelated with batches (`p-value >` `0.99`). See section 2.3 for more details.

Sample distribution by plates can also be visualized (**Figure 1**). It shows that samples with different characteristics were distributed across batches with only small variations. The small variation is largely due to the trade off in block randomizing multiple variables. The last plot is the index of optimization steps versus value of the objective function. The blue diamond indicate the starting point, and the red diamond mark the final optimal setup. It is clear that final setup is more optimal than the starting setup.

If blocking the primary variable (i.e., `SampleType`) is most important and the optimization of other variables considered are less important, a different algorithm can be used. See section 3.2 for an example and more details about this alternative method implemented in OSAT.

```
          Var X-squared df   p.value
1 SampleType 0.5401995  5 0.9905773
2       Race 0.1395369  5 0.9996319
3     AgeGrp 0.7971741 20 1.0000000
```
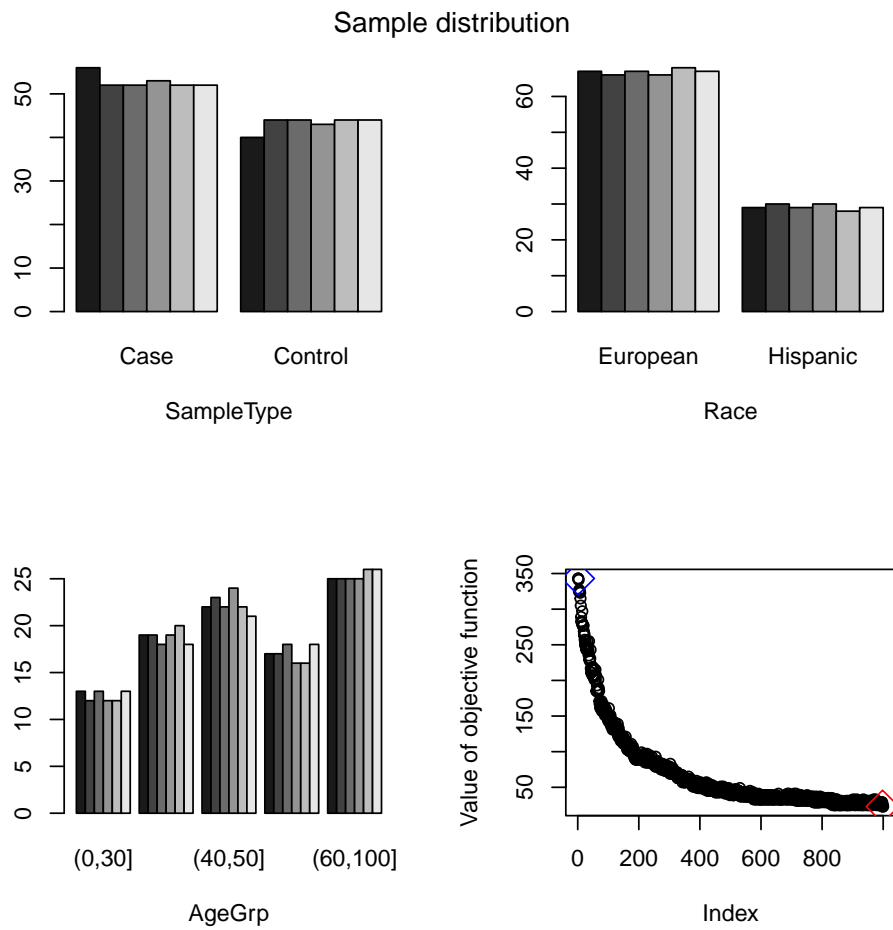

Figure 1: Number of samples per plate. Sample assignment using default algorithm.

3 Methodology review

From the perspective of experiment designs, blocking is the placement of experimental units in groups (blocks) that are similar to one another. Typically, a blocking factor is a source of variability that is not of primary interest but may affect the measured result. An example of a blocking factor might be the sex of patients; by blocking on sex, this source of variability is controlled for, thus leading to greater accuracy. For example, gender of patients could be treated as confounding variable when the primary interest is differential gene expression between healthy and tumor samples. By blocking on gender, this source of variability is controlled for, thus leading to greater accuracy of estimating differential gene expression.

Block randomization is an effective approach in controlling variability from nuisance variables and its interaction with variables of our primary interest. In such a design, samples with same characteristics are selected randomly to each block. One widely used design is randomized complete block design (RCBD) which requires that the sample characteristics in each block are as similar as possible[5]. In RCBD experiment, when the sample size of in each experiment groups are balanced (i.e., having same number of samples), the control of variability and interaction from nuisance variables can be achieve easily and ANOVA can be used for data analysis. In most cases, RCBD are more precise than the Complete Random Design (CRD) for which no blocking is performed[6,8].

Batch effect is one type of variability that is that is not of primary interest but ubiquitous in sizable genomic experiments. It had shown that, without appropriate control of batch effects, misleading or even erroneous conclusions could be made[1,2,4]. In order to minimize batch effects, the groups of the main interest, as well as important confounding variables should be balanced and replicated across the batches to form a RCBD in an ideal genomics design.

However, due to the practical complications described in the Overview section, genomics experiments are often driven by the availability of the final collection of samples which is usually unbalanced and incomplete. The unbalance and incompleteness nature of sample availability thus calls for the development of effective tool to assign collected samples across batches in an appropriate way to minimize batch effects at the genomic experiment stage.

In this work, we focused on the development of such a tool. Our tool is developed to optimize the even distribution of samples in groups of biological interest into different run batches, as well as the even distribution of confounding factors across batches. Without loss of generality, throughout the documents we refer variables of primary interest and the confounding variables selected to be blocked in the initial design as blocking variables, and the unique combination of levels in these variables as blocking strata or simply strata. Our goal is to minimize the differences in each stratum between batches through a block randomization step followed by an effective optimization step. Details will be discussed in the following sections.

3.1 Objective

To create sample assignment with homogeneous batches, we need to measure the variation of sample characteristics between batches. We define objective function based on principle of least square method.

By combining the levels in variables of interest, we can create a unified variable. Assuming there are s levels in the combined variable with S_j samples in each strata, $j = 1 \ldots s$. Total sample size $N = \sum_{j=1}^{s} S_j$. In the case of balanced design, we have equal sample size in each strata: $S_1 = \ldots = S_s = S$.

Assuming we have m batches with $B_i, i = 1 \ldots m$ wells in each batch. In an ideal balanced randomized complete block design experiment, each batch includes same number of available wells, $B_1 = \ldots = B_m = B$, and equal number of samples from each sample strata. In most real cases however, batches may have different number of wells due to various reasons (such as wells reserved for QC), but the differences are usually not significant.

The average (expected) number of sample from each strata to each batches is denoted as E_{ij}. Under most scenarios, the expected sample number would likely not be an integer. We split it to its integer part and fractal part as

$$E_{ij} = \frac{B_i S_j}{\sum_i B_i} = [E_{ij}] + \delta_{ij}. \tag{1}$$

where $[E_{ij}]$ is the integer part of the expected number and δ_{ij} is the fractal part. In the case of equal batch size, it reduces to $E_{ij} = \frac{S_j}{m}$. When we have RCBD, all δ_{ij} are zero.

For an actual sample assignment

$$
\begin{array}{c}
\begin{array}{cccc} S_1 & S_2 & \ldots & S_s \end{array} \\
\begin{array}{c} B_1 \\ B_2 \\ \vdots \\ B_m \end{array}
\begin{pmatrix}
n_{11} & & \ldots & n_{1s} \\
& n_{22} & \ldots & n_{2s} \\
\vdots & \vdots & \ddots & \vdots \\
& & \ldots & n_{ms}
\end{pmatrix}
\end{array}
$$

our goal is to minimize the difference between expected sample size E_{ij} and the actual sample size n_{ij}.

3.1.1 Objective function

In this package, we define the optimal design as a sample assignment setup that minimizes our objective function. By default, our objective function is based on the sum of squares of the difference between expected and actual sample size in each strata from the combined variables considered in assignment[9]. Assuming there are o levels in the combined optimization variable strata with O_l samples in each strata, $l = 1 \ldots o, \sum_l O_l = N$, then the expected number of sample in each optimization variable strata is $E_{il}^* = \frac{B_i O_l}{\sum_i B_i}$ and we define our objective function as

$$
V = \sum_{il} (n_{il} - E_{il}^*)^2 \tag{2}
$$

where n_{il} is the number of sample in each optimization strata from a sample assignment. We chose sample setup with minimum value of V as our optimal sample assignment.

Another objective function is a weighed version of the Sum of Squares method discussed above. Instead of treat each variable equally in the objective function, we can give them different weight to reflect the importance of the particular variable. The weight usually comes from empirical evidence. Relative weight for each optimization strata can be easily calculated based on the weight given to each variable. Assuming the weight are $w_l, l = 1 \ldots o$, then our objective function will be

$$
V = \sum_{il} (w_l(n_{il} - E_{il}^*))^2 \tag{3}
$$

Other objective functions can be used to optimize the sample-to-batch assignment. Our package is constructed to easily adopt user-defined objective function. See section 4.3.1 for more details.

3.2   Algorithms

The current version of OSAT provided two algorithms for creation of optimal sample assignment across the batches. Both algorithms are composed of a block randomization step and an optimization step.

The default algorithm (implemented in function *optimal.shuffle*) sought to first block all variables considered (i.e., list of variables of primary interests and all confounding variables provided to the optimal argument) to generate a single initial assignment setup, then identify the optimal one which minimize the objective functions (i.e., the one with most homogeneous cross-batch strata distribution) through shuffling the initial setup.

The block randomization step is to create an initial setup of randomized sample assignment based on combined strata of all blocking variables considered. In this step, we sampling i sets of samples from each sample strata S_j with size $[E_{ij}]$. Similarly select j sets of wells from each B_i batches with size of $[E_{ij}]$, then the two selections are linked together by the ij subgroup, randomized in each of them.

In a balanced design this will place all samples to proper batch. Otherwise we assign the rest of samples $r_j = S_j - \sum_i [E_{ij}]$ sample to the available wells in each block $w_i = B_i - \sum_j [E_{ij}]$. The probability of a sample in r_j from strata S_j assigned to a well from block B_i is proportional to the fractal part of the expected sample size δ_{ij}.

The goal of block randomization step is to create an initial setup of random sample assignment conformed to the blocking requirement, i.e. the samples from each stratum are evenly distributed to each batch. In the case of a RCBD, each batch will have equal number of samples with same characteristic and there is no need for further optimization. In other instances when the collection of samples is unbalanced and/or incomplete, an optimization step is needed to create a more optimal setup of sample assignment.

The optimization step aims to identify an optimal setup of sample assignments from multiple candidates, which are generated through resampling the initial setup obtained in the block randomization step. Specifically, after initial

setup is created, we randomly select k samples from different batches and shuffle them between batches to create a new sample assignment. $k = 2$ by default but could be any number up to half of the sample size. Value of the objective function is calculated on the new setup and compared to that of the original one. If the value is smaller then the new assignment replaces the previous one. This procedure will continue until we reach a preset number of attempts (usually in the tens of thousands).

Because this step only shuffles limited number of samples, computational speed is very fast, and can easily generate tens of thousands modified setup from initial setup. In addition, because it follows the decreasing of values from objective function, the method will converge relatively rapidly. One potential concern is that this method may not be able to reach global minimum. This concern will largely be alleviated when a significant number of shuffling is performed in real work.

The alternative algorithm (implemented in function *optimal.blcok*) sought to first block specified variables (e.g., list of variables of primary interests and certain confounding variables specified in the strata argument) to generate a pool of assignment setups, then select the optimal one which minimize the objective functions based on all variables considered (i.e., list of variables provided to the optimal argument, including those variables which are not included in the block randomization step).

More specifically, multiple (typically thousands of or more) sample assignment setups are first generated by procedure described in the block randomization step above, based only on the list of specified blocking variable(s). Then, the optimal setup is chosen by selecting the setup of sample assignment (from the pool generated in block randomization step) which minimizes the value of the objective function based on all variables considered. This algorithm will guarantee the identification of a setup that is conformed to the blocking requirement for the list of specified blocking variables, while attempt to minimize the variations between batches regarding the other variables considered. In theory it could reach global minimum for the optimization. On the other hand, due to the complexity involved in generating multiple candidate assignments through block randomization, this computational speed of this method is relatively slow. In addition, as the assignment of each setup is generated independently, the convergence to optimal might be slow.

We applied both algorithms to our demon exemplary data and compared their results in the next section. More details of the implementations of these two methods are described in section 4.3.

3.2.1 Comparison of algorithms

To demonstrate the difference of the two algorithms, we create a new optimal setup using the second method described above (this is slower than default method and will take a few minutes):

```
> gs2 <- setup.sample(pheno,  strata=c("SampleType"),
+           optimal=c("SampleType", "Race", "AgeGrp") )
> set.seed(123)    # to create reproducible result
> # demonstration only. nSim=5000 or more are commonly used.
> gSetup2 <- create.optimized.setup("optimal.block",
+           sample=gs2, container=gc, nSim=1000)

> QC(gSetup2)
```

In this setup, we treat `SampleType` as our only blocking variable in the block randomization step. The other two variables, together with `SampleType`, will be optimized in the optimization step. Comparing to the result using default algorithm in section 2.3 which block all three variables in block randomization step, Pearson's Chisq test shows `p-value` increase on our blocking variable `SampleType` and decrease on other two variables not included in the block randomization step, reflecting the trade-off in prioritizing the blocking on variable of our primary interest. **Figure 2** shows the blocking on our primary interest `SampleType` is almost perfect, with small variation only due to the inherent limitation of the starting data (i.e., unbalanced sample collection). On the other hand, the uniformity of the other two variables not included in block randomization step had been scarified (compared with **Figure 1**). The last plot is the index of generated setups versus value of the objective function. The blue diamond indicate the first generated setup, and the red diamond mark the final selected setup. It is clear that the optimization step picked up the most optimal one from the pool of geneerated setups.

```
          Var  X-squared df   p.value
1 SampleType 0.03507789  5 0.9999879
2       Race 4.57188457  5 0.4703235
3     AgeGrp 6.04463944 20 0.9988359
```
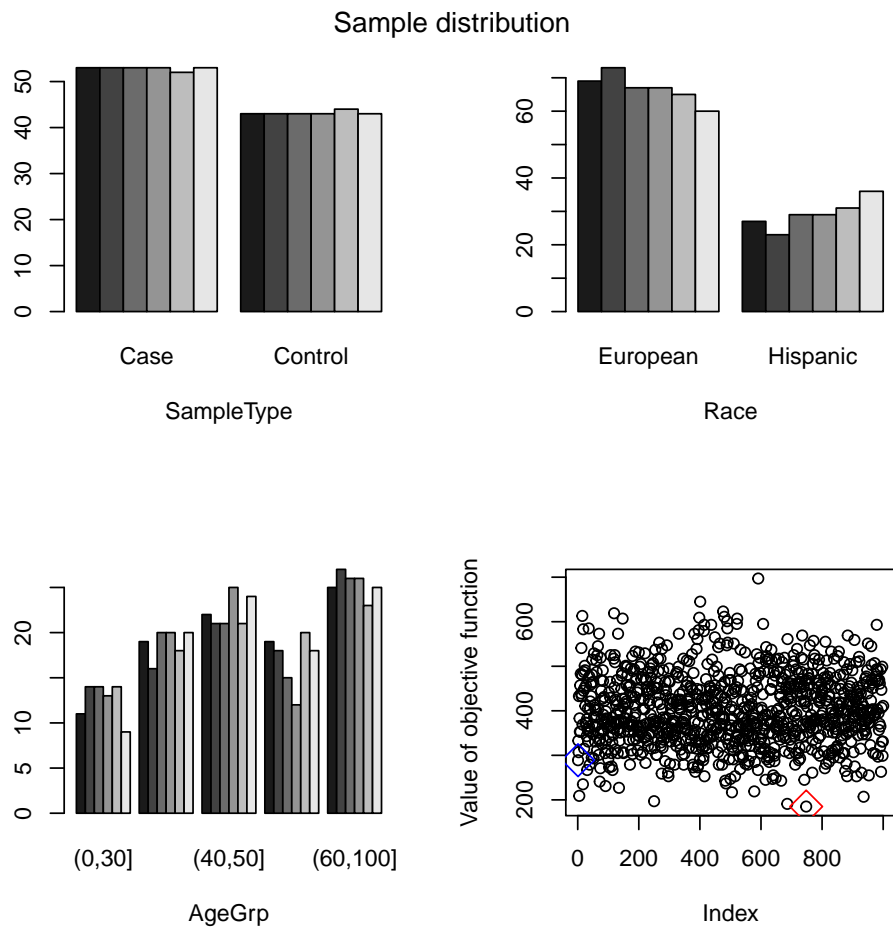

Figure 2: Number of samples per plate. Sample assignment use `optimal.block` method.

3.3 Potential problems with completely random assignment

Simply performing complete randomizations could lead to undesired sample-to-batch assignment, especially for unbalanced and/or incomplete sample sets. In fact, there is substantial chance that any one of the variables in the sample will be statistically dependent on batches if complete randomization is carried out. For example, simulation and theory shows that, if only one random variable is involved and we randomly assign samples to different batches, there will be about 5% chance that statistical dependency exists between batch and the variable (data not shown). With increasing number of variables involved, the overall chance that any one of the variables show a statistically significant difference between batches will increase drastically. Simulation shows that, when simple randomization method is used with three variables, there is about 14% chance that at least one of the variables is statistically depending on batches (data not shown). This phenomenal is well known and linked directly to the multi-testing problem. Sometimes the choice of randomization seed can produce unacceptable results. For example, if we randomly assign our samples to the container, the innocent choice of a seed 397 in our R session will show:

```
> set.seed(397)         # an unfortunate choice
> c1 <- getLayout(gc)   # available wells
> c1 <- c1[order(runif(nrow(c1))),] # shuffle randomly
> randomSetup <- cbind(pheno, c1[1:nrow(pheno), ])
>       # create a sample assignment

> multi.barplot(randomSetup, grpVar='plates',
+       varList=c("SampleType", "Race", "AgeGrp"),
+       main="A bad random case")


> multi.chisq.test(randomSetup, grpVar='plates',
+       varList=c("SampleType", "Race", "AgeGrp"))

$method
[1] "Pearson's Chi-squared test"

$data.name
[1] "randomSetup"

$stat
        Var X-squared df      p.value
1 SampleType  13.25243  5 0.021124664
2       Race  14.22455  5 0.014244218
3      AgeGrp  39.75020 20 0.005371387
```

Pearson's Chisq test indicate all three variables are statistically dependent on batches with `p-values < 0.05`. **Figure 3** shows the sample distribution in each plate based on this assignment.

4 Classes and Methods

We present brief descriptions for some of the classes and methods defined in the package. For more details, please refer to the manual.

4.1 Sample

The sample information from data frame x, blocking variables, and other variables considered in the optimization step can be integrated by creating a class `gSample` object using the construction function:

```
setup.sample(x, optimal, strata)
```

If pheno data is a phenoData slot in an `ExpressionSet` object x then the construction function should be called by:
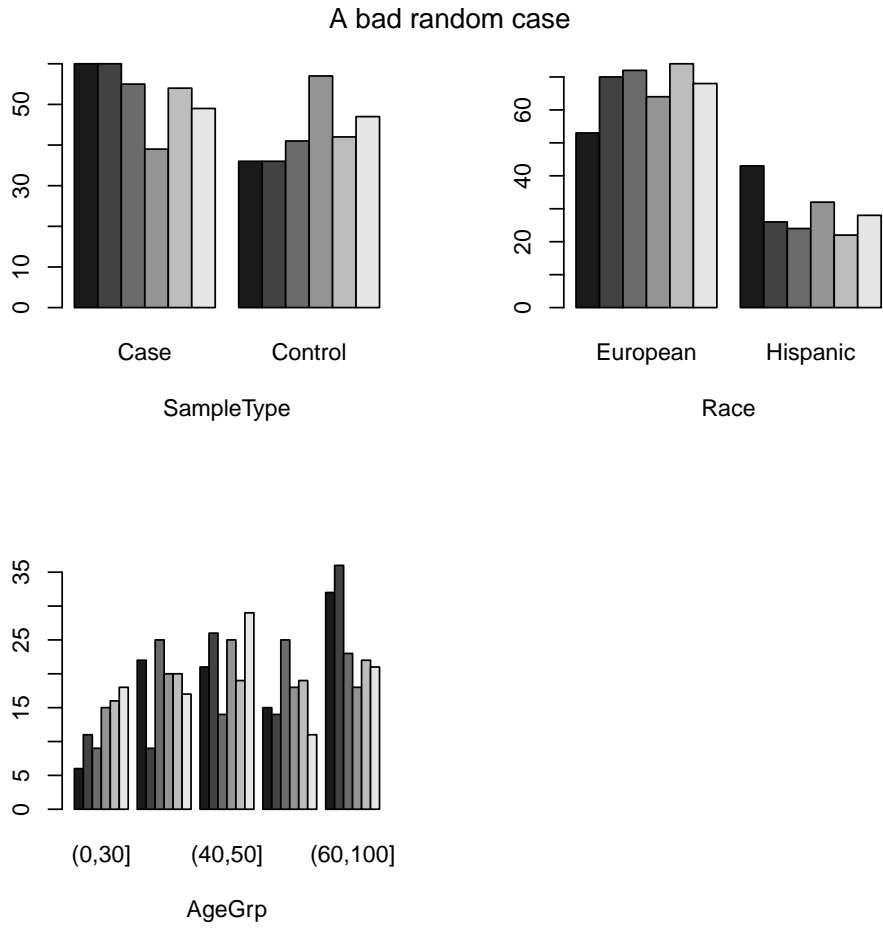
Figure 3: Number of samples per plate. A bad complete random assignment.

```
setup.sample(pData(x), optimal, strata)
```

We can inspect the data regarding to variables considered and resulting strata by typing its name. Using our example data,

```
> gs <- setup.sample(pheno, optimal=c("SampleType", "Race", "AgeGrp"),
+             strata=c("SampleType"))
> gs

An object of class "gSample"
The raw data are
  ID SampleType    Race     AgeGrp
1  1       Case Hispanic   (60,100]
2  2       Case Hispanic   (60,100]
3  3       Case European   (60,100]
4  4       Case European    (50,60]
5  5       Case European    (50,60]
6  6       Case European     (0,30]


 ...
      ID SampleType    Race  AgeGrp
571 571    Control European (40,50]
572 572    Control Hispanic (30,40]
573 573    Control European (30,40]
574 574    Control Hispanic (30,40]
575 575    Control European (40,50]
576 576    Control European  (0,30]


 Blocking strata in the data:

  SampleType Freq sFactor
1       Case  317       1
2    Control  259       2


 Optimization strata in the data

   SampleType    Race   AgeGrp Freq oFactor
1        Case European   (0,30]    8       1
2     Control European   (0,30]   58       2
3        Case Hispanic   (0,30]    0       3
4     Control Hispanic   (0,30]    9       4
5        Case European  (30,40]   21       5
6     Control European  (30,40]   54       6
7        Case Hispanic  (30,40]    6       7
8     Control Hispanic  (30,40]   32       8
9        Case European  (40,50]   34       9
10    Control European  (40,50]   52      10
11       Case Hispanic  (40,50]   46      11
12    Control Hispanic  (40,50]    2      12
13       Case European  (50,60]   40      13
14    Control European  (50,60]   44      14
15       Case Hispanic  (50,60]   16      15
16    Control Hispanic  (50,60]    2      16
17       Case European (60,100]   84      17
18    Control European (60,100]    6      18
```

```
19       Case Hispanic (60,100]   62       19
20    Control Hispanic (60,100]    0       20
```

in addition to variables from the original sample dataset, two new factors are created: `sFactor` and `oFactor` are factors represent the combined strata created based on the combination of variables considered, respectively. Some getter functions exist for a `gSample` object; refer to manual for more details.

4.2 Container

In high-throughput genomics experiment the specimens are usually placed into assembly of plates that have fixed number of chips and wells. All information related to the experimental instrument layout is kept in a class `gContainer` object. We need to know the number of plates used, the plate's layout, the layout of chips on each plate, and the layout of the wells on each chip. The `gContainer` object holds all these information. It also hold our decision on which level of blocks (plates or chips) the batch effect need to be considered. For large scale experiments (several hundreds of samples or more) we usually concern batch effect between plates. Occasionally, for smaller sample size, we may chose 'chips' level in the construction of the container object. In addition, if for any reason certain wells in the assembly are not available for sample placement (for example reserved for negative control), a list of excluded wells on these plates can be specified. The object can be constructed by its constructor:

```
setup.container(plate, n, batch='plate', exclude=NULL, ...)
```

In example, we used 6 Illumina 96-well HYP MultiBeadChip plates. This particular plate consists of 8 BeadChips each, which in turn holds 12 wells in 6 rows and 2 columns. We are trying to control the plate level batch effect,

```
> gc <- setup.container(IlluminaBeadChip96Plate, 6,
+          batch='plates')
> gc

An object of class "gContainer"

It consists of 6 IlluminaBeadChip96Plate plates.
The block level is set at plates level.
  plates cFactor Freq
1      1       1   96
2      2       2   96
3      3       3   96
4      4       4   96
5      5       5   96
6      6       6   96
The container layout is
@data$container
  plates chipRows chipColumns chips rows columns wells chipID rowID wellID
1      1        1           1     1    1       1     1      1     1      1
2      1        1           1     1    2       1     2      1     1      2
3      1        1           1     1    3       1     3      1     1      3
4      1        1           1     1    4       1     4      1     1      4
5      1        1           1     1    5       1     5      1     1      5
6      1        1           1     1    6       1     6      1     1      6
  cFactor
1       1
2       1
3       1
4       1
5       1
6       1
```

```
...
    plates chipRows chipColumns chips rows columns wells chipID rowID wellID
571      6        2           4     8    1       2     7     48    286    571
572      6        2           4     8    2       2     8     48    286    572
573      6        2           4     8    3       2     9     48    286    573
574      6        2           4     8    4       2    10     48    286    574
575      6        2           4     8    5       2    11     48    286    575
576      6        2           4     8    6       2    12     48    286    576
    cFactor
571       6
572       6
573       6
574       6
575       6
576       6
```

Relevant information can be reviewed by simply typing the name of the container:

a new variable `cFactor` is create to represent the batches in the container. Variable `wellID` identify all wells in the container. Each well will hold an individual sample.

A number of commonly used types of batch layout (e.g., plates and/or chips by Illumina Inc.) had been predefined in the package for the convenience of users, which makes the creation of a container object fairly easy in many cases. Please see next section 4.2.1 for a whole list of predefined batch layout. It is also very straightforward to define the batch layout of your own genomics platform. See section 4.2.1 and manual for more details.

The exclusion list is simply a data frame that indicated which wells in the plate assembly need to be excluded from block randomization and and optimization for any reasons. See additional case study in section 5.2 for explanation and example.

4.2.1 Predefined batch layout

Function `predefined()` shows a list of predefined object that represent some of commonly used plates and chips. Simply typing the name in R shows the layout and other information of each predefined chips and plates.

IlluminaBeadChip is a chip with 2 columns and 6 rows, a total of 12 wells:

```
> IlluminaBeadChip

An object of class "BeadChip"
Illumina Bead Chip have 6 rows and 2 columns.
It has 6 rows and 2 columns.
The layout is
@layout
   rows columns wells
1     1       1     1
2     2       1     2
3     3       1     3
4     4       1     4
5     5       1     5
6     6       1     6
7     1       2     7
8     2       2     8
9     3       2     9
10    4       2    10
11    5       2    11
12    6       2    12
```

GenotypingChip is a chip with 1 columns and 12 rows.

IlluminaBeadChip24Plate, IlluminaBeadChip48Plate, IlluminaBeadChip96Plate are plates that hold 2, 4, 8 IlluminaBeadChip chips and have 24, 48, 96 wells, respectively.

4.2.2 Define your own batch layout

As long as the physical layout is know for a plate or chip, it is straight forward to define your own object in this package. For example, if you have a chip that has 12 wells arranged by a 2 columns and 6 rows pattern, and the index of the wells are filled columns by columns, then we can create (and have a look at) a new chip object simply:

```
> myChip <- new("BeadChip", nRows=6, nColumns=2, byrow=FALSE,
+       comment="Illumina Bead Chip have 6 rows and 2 columns.")
> myChip

An object of class "BeadChip"
Illumina Bead Chip have 6 rows and 2 columns.
It has 6 rows and 2 columns.
The layout is
@layout
   rows columns wells
1     1       1     1
2     2       1     2
3     3       1     3
4     4       1     4
5     5       1     5
6     6       1     6
7     1       2     7
8     2       2     8
9     3       2     9
10    4       2    10
11    5       2    11
12    6       2    12
```

The `byrow` parameter is same as that in function `matrix()`. The chip happens to be identical to the IlluminaBeadChip defined in the package:

```
> identical(myChip, IlluminaBeadChip)

[1] TRUE
```

Similarly, we can define plate simply based on the chips it contains. Here we create a new plate that consistent 8 `myChip` we just defined:

```
> myPlate <- new("BeadPlate", chip=IlluminaBeadChip,
+       nRows=2L, nColumns=4L,
+       comment="Illumina BeadChip Plate with 8 chips and 96 wells")
> myPlate

An object of class "BeadPlate"
Illumina BeadChip Plate with 8 chips and 96 wells
It has 2 rows and 4 columns of IlluminaBeadChip chips.
The layout is
@layout
  chipRows chipColumns chips rows columns wells
1        1           1     1    1       1     1
2        1           1     1    2       1     2
```

```
3        1          1    1    3        1    3
4        1          1    1    4        1    4
5        1          1    1    5        1    5
6        1          1    1    6        1    6

...
   chipRows chipColumns chips rows columns wells
91        2          4    8    1        2    7
92        2          4    8    2        2    8
93        2          4    8    3        2    9
94        2          4    8    4        2    10
95        2          4    8    5        2    11
96        2          4    8    6        2    12
```

```
> identical(myPlate, IlluminaBeadChip96Plate)
```

```
[1] TRUE
```

it happens to be identical to the `IlluminaBeadChip96Plate` plate defined in the package.

4.3 Sample assignment

Sample assignment and all relevant information are stored in a class `gExperimentSetup` object. If only block randomization is needed, we can simply create a experiment assignment by

```
> gSetup0 <- create.experiment.setup(gs, gc)
> myAssignment <- get.experiment.setup(gSetup0)
```

However in most cases we optimize our setup based on our choice of blocking variables and other variables considered. This can be done by calling function

```
create.optimized.setup(fun, sample, container, nSim, ...)
```

where `fun` is the name of a user defined objective function. Parameter `fun` can be omitted, then the default optimization function `optimal.shuffle` is called. `sample` and `container` are objects from `gSample` and `gContainer` classes, respectively. The function will return a class `gExperimentSetup` object which holds all releveant information.

The default optimization function `optimal.shuffle(sample, container, nSim, k=2, ...)` will first create initial setup of assignment as candidate through blocking all variables considered. Then it will randomly select and shuffle `k` (default k=2) samples between batches to create a new setup. It will calculate the objective function in the new setup and have its result compared to that of the current candidate. The candidate setup will be replaced if the value of objective function in new setup is smaller. The process will repeat up to `nSim` times.

Alternatively, we could use function `optimal.block(sample, container, nSim)`, which is to create multiple setup of sample assignment based only on specified blocking variables, and select the optimal setup based on all variables considered (including the variables not included in block randomization step).

```
gSetup1 <- create.optimized.setup(fun="optimal.shuffle",
      sample=gs, container=gc, nSim=1000)
gSetup2 <- create.optimized.setup(fun="optimal.block",
      sample=gs, container=gc, nSim=1000)
```

Same functionality can be invoked by apply the optimizing objective function to any `gExperimentSetup` object directly:

```
gSetup1 <- optimal.shuffle(gSetup0, nSim=1000)
gSetup2 <- optimal.block(gSetup0, nSim=1000)
```

This syntax make it easy to try algorithms and compare the results. More details on create your own optimizing objective function see below.

4.3.1 Define your own optimizing objective function

Our package allows user to define optimization objective function themselves. It is straightforward for user to define their own optimization objective function, following a few parameter requirements for the package to work. In particular, the objective function parameters should include a `gExperimentSetup` object, and number of loops used in the form of

```
myFun <- function(x, nSim, ...)
```

It can include additional parameters for its own in . . . parameter list.

Also, the function should return a list that including following elements:

```
return(list(setup=, link=, average=, dev=))
```

where elements `setup` is a `gExperimentSetup` object, `link` is a data frame present a link between the sample and container, `average` is the expected number of samples from each blocking strata in each of the optimization blocks, `dev` a vector of values created by the actual objective function.

In the current version of OSAT, we have defined two different algorithms. They will generally generate similar assignment plan.

4.4 QC, summary, and output

To see a summary of the experimental design, simply type the name of the object we created:

```
gSetup
```

most relevant information will be shown.

The final design can be retrieved by

```
gSetup.data <- get.experiment.setup(gSetup)
```

which creates a data frame holding linked data of sample variables and container variables, sorted by the initial order from the sample list.

The distribution of the sample by batches can be visualized graphically using function `plot(gSetup, ...)`

```
plot(gSetup, main="Summary")
```

which will create a barplot of sample counts in each batch for every variable considered in design.

We can simply use function `QC()` to get both plot and Pearson's Chi-squared test of independence between batches and variables involved:

```
QC(gSetup)
```

To output the design layout in CSV format,

```
write.csv(get.experiment.setup(gSetup), file="gSetup.csv",
          row.names=FALSE)
```

4.4.1 Map to robotic loader

The sample plates are usually loaded into machine through a robotic loader, for example MRA-4 loader. The layout of the loader plate can be seen in the included documents. After the assignment to the plates are finished, we can map the bead chip plate to the MRA-4 plate, and have it exported to a CSV file by

```
out <- map.to.MSA(gSetup)
write.csv(out, "MSAsetup.csv", row.names = FALSE)
```

5 Additional Case studies

In this section we present a few more case studies to further demonstrate the usage of OSAT package.

5.1 Incomplete batches

Since the number of wells available on a batch (e.g., plate/chip) is fixed, often experimenter cannot fill up all of them due to sample unavailability. In this scenario it is recommended that we distribute the unused wells randomly across all batches. This is done by default in our program without any additional instruction.

For example, assuming there are 10 samples not available for some reason in our sample list, the following create setup using the same container as before with the rest of 566 samples:

```
> # for demonstration, assume 10 bad samples
> badSample <- sample(576, 10, replace=FALSE)
> # create sample object using available samples
> gs3 <- setup.sample(pheno[-badSample, ],
+       optimal=c("SampleType", "Race", "AgeGrp"),
+       strata=c("SampleType") )
> # use the same container setup as before
> # demonstration only. nSim=5000 or more are commonly used.
> gSetup3 <- create.optimized.setup(sample=gs3,
+         container=gc, nSim=1000)
```

To fill batches sequentially and have the empty wells left at the end of last plate is discouraged. This practice will create a batch that is immediately different from the others, making statistical analysis more complex and less powerful. However, if there are strong reasons to do so, we can exclude the wells at the end of last plate for sample assignment, using method described in the following section.

5.2 Excluded well positions

If for any reason we need to reserve certain wells for other usage, we can exclude them from the sample assignment process. For this one can create a data frame to mark these excluded wells. Any wells in the container can be identified by its location identified by three variable `"plates"`, `"chips"`, `"wells"`. Therefore the data frame for the excluded wells should have these three columns.

For example, if we will use the first well of the first chip on each plate to hold QC samples, these wells will not be available for sample placement. We have 6 plates in our example so the following will reserve the 6 wells from sample assignment:

```
> excludedWells <- data.frame(plates=1:6, chips=rep(1,6),
+                   wells=rep(1,6))
```

Our program can let you exclude multiple wells at the same position of plate/chip. For example, the following data frame will exclude the first well on each chips regardless how many plates we have:

```
> ex2 <- data.frame(wells=1)
```

In our example we have 48 chips so 48 wells will be excluded using this data frame for exclusion. We can pass our exclusion data frame to the container setup function using this command:

```
> gc3 <- setup.container(IlluminaBeadChip96Plate, 6,
+           batch='plates', exclude=excludedWells)
```

5.3 Blocking and optimization on chip level

In certain circumstances, batch effects between chips maybe considered. This task can be accomplished by simply indicate the level of block in the container construction, for example

```
> cnt <- setup.container(IlluminaBeadChip96Plate, 2, batch='chips')
```

will create a container with 16 chips. Blocking and optimization on chip level will be used in following sample assignment.

5.4 Paired samples

OSAT can be used to handle experiment design with paired examples.

Assuming each individual listed in our `samples.txt` file had 2 specimens. One is collected prior certain treatment and the other after. We would like to keep the two specimens on the same chip, to reduce the batch effect on the treatment effect. Other requirements are similar to those in the section 2.1.

To accomplish this new task, one chip can only hold specimens from 6 individuals. We would first assign specimen pairs onto chips, then shuffle them within each chip randomly to further eliminate potential location bias within chips. To do this, we will create a mock chip that only has 6 rows and 1 column instead of 2 columns. Each row on this mock chip will be assign to one individual. Plate and container are created based on this new chip. After assignment, we will expand the chips to 2 columns.

First of all the pairs are assigned into rows of the mock chip (noticed now we need 12 plates):

```
> # create mock chip. each row represent one individual
> newChip <- new("BeadChip",  nRows=6, nColumns=1, byrow=FALSE,
+     comment="mock chip")
> # a mock plate based on above chip, same physical layout
> newPlate <- new("BeadPlate", chip=newChip,
+     nRows=2L, nColumns=4L,
+     comment="mock plate")
> # create containers based on above mock chip/plate
> gcNew <- setup.container(newPlate, 12, batch="plates")
> # assign pairs into locations on the mock chip
> # this will take some time
> set.seed(12345)
> # demonstration only. nSim=5000 or more are commonly used.
> gPaired <- create.optimized.setup("optimal.block", sample=gs,
+           container=gcNew, nSim=1000)
```

The above steps create sample setup that place paired sample to rows of each real BeadChip, the following steps expand the chips to real Illumina BeadChip. First assign specimens into first column on each real chip:

```
> set.seed(456)
> out1 <- get.experiment.setup(gPaired)
> out1$Replica <- FALSE
> idx <- sample(nrow(out1), ceiling(nrow(out1)/2), replace=FALSE)
>      # randomly decided if the first specimen is placed in column 1
> out1[idx, "Replica"] <- TRUE
```

above we randomly selected half of before-treatment specimens and half of after-treatment specimens into first the column. The rest specimens will be placed into the second column:

```
> out2 <- out1
> out2$columns <- 2          # specimen placed in the second column
> out2$wells <- out2$wells+6 # correct well number
> out2$Replica <- !out1$Replica        # indicate second specimen
```

19

```
> out3 <- rbind(out1, out2)   # all specimens
> # sort to order on plates/chips/rows
> idx1 <- with(out3, order(plates, chips,  rows, columns, wells))
> out3 <- out3[idx1,]   # sort to order on plates/chips/wells
```

This procedure will place paired specimens on the same row of the same chip. If do not want to keep pairs of specimens on the same row, we can shuffle one more time:

```
> ## shuffle within chip
> set.seed(789)
> idx2 <- with(out3, order(plates, chips, runif(nrow(out3))))
> out4 <- cbind(out3[, 1:8], out3[idx2, 9:11], Replica=out3[, 12])
> # sort to order on plates/chips/wells
> idx3 <- with(out4, order(plates,  chips, wells))
> out4 <- out4[idx3,]
```

Check the assignment:

```
> # SampleType and replica distribution by plate
> ftable(xtabs( ~plates +SampleType+ Replica, out3))
```

The final assignment quality can be checked by Pearson Chisq test and visualized by barplot:

```
> multi.barplot(out3, grpVar='plates', varList=c("SampleType", "Replica",
+               "Race", "AgeGrp"), main="paired sample")


> multi.chisq.test(out3, grpVar='plates', varList=c("SampleType", "Replica",
+               "Race", "AgeGrp"))

$method
[1] "Pearson's Chi-squared test"

$data.name
[1] "out3"

$stat
        Var   X-squared df    p.value
1 SampleType  0.4910905 11 0.9999988
2    Replica  0.0000000 11 1.0000000
3       Race  5.3024011 11 0.9156416
4     AgeGrp 29.2638486 44 0.9570941
```

The Pearson's test and Figure 4 show perfect `Replica` balance between plates due to the fact all paired specimens are placed on the same chip.

6   Discussion

Genomics experiments are often driven by the availability of the final collection of samples which is usually unbalanced and incomplete. Without appropriate attention, the unbalance and incompleteness nature of sample availability might lead to drastic batch effects. We developed a handy tool called OSAT (Optimal Sample Assignment Tool) to facilitate the allocation of collected samples to different batches in genomics study. With a block randomization step followed by an optimization step, it produces setup that optimizes the even distribution of samples in groups of biological interest into different batches, reducing the confounding or correlation between batches and the biological variables of interest. It can also optimize the homogeneous distribution of confounding factors across batches. While motivated to handle
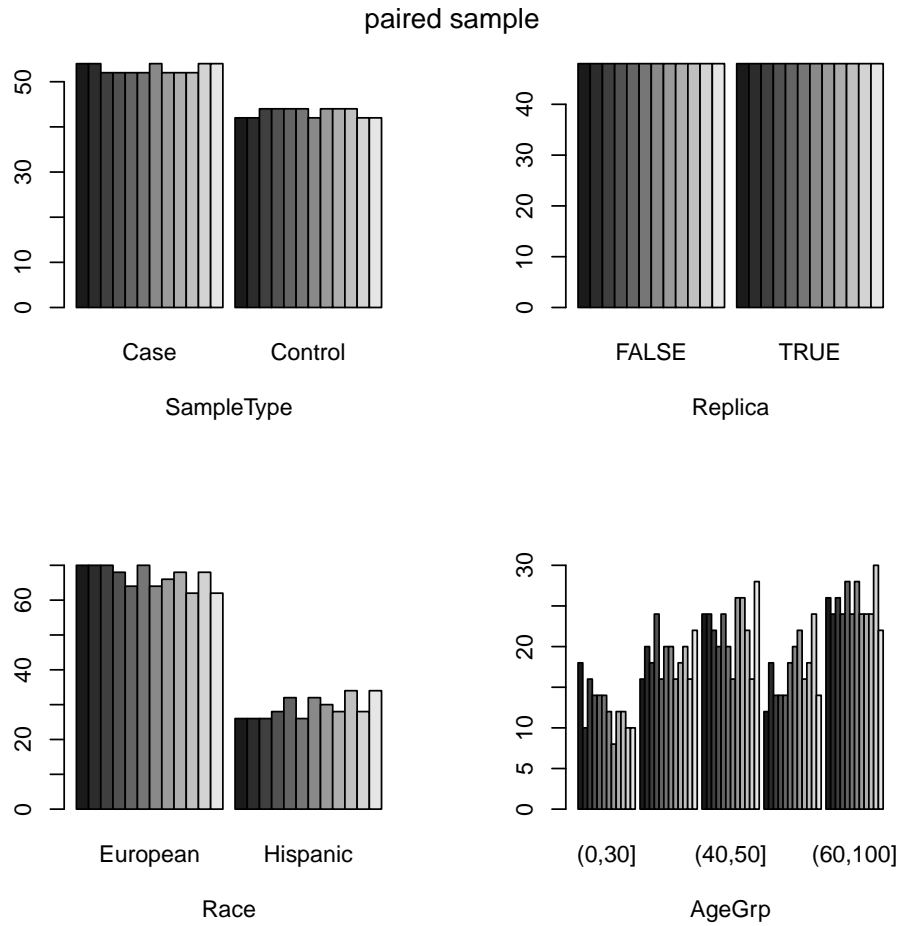
Figure 4: Number of samples per plate. Paired specimens are placed on the same chip. Sample assignment use `optimal.block` method.

challenging instances where incomplete and unbalanced sample collections are involved, OSAT can also handle ideal balanced RCBD.

As demonstrated in this document (section 3.3), complete randomization, often used in the sample assignment step of experiment practice, may produce undesirable setup showing batch dependence. OSAT package is designed to avoid such scenario, by introducing a simple pipeline to create sample assignment that minimizes association between sample characteristic and batches. OSAT provides predefined batch layout for some of the most commonly used genomics platform. Written in a modularized style in the open source R environment, it provides the flexibility for users to define the batch layout of their own experiment platform, as well as optimization objective function for their specific needs, in sample-to-batch assignment to minimize batch effects.

We should also mention that although the impact of batch effect on genomics study might be minimized through proper design and sample assignment, it may not be completely eliminated. Even with perfect design and best effort in all stages of experiment including sample-to-batch assignment, it is impossible to define or control all potential batch effects. Many statistical methods had been developed to estimate and reduce the impact of batch effect at the data analysis stage (i.e, after the experiment part is done)[10–12]. It is recommended that analytic methods handling batch effects are employed in all stages of a genomics study, from experiment design to data analysis.

li.yan@@roswellpark.org

7   Session information

```
> sessionInfo()

R version 4.4.1 (2024-06-14)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.1 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.26.so;  LAPACK version

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: Etc/UTC
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] xtable_1.8-4 OSAT_1.55.0

loaded via a namespace (and not attached):
[1] compiler_4.4.1  tools_4.4.1     maketools_1.3.1  buildtools_1.0.0
[5] knitr_1.48      xfun_0.48       sys_3.4.3
```

References

[1] Christophe G Lambert and Laura J Black. Learning from our GWAS mistakes: from experimental design to scientific method. *Biostatistics (Oxford, England)*, 13(2):195–203, April 2012. PMID: 22285994.

[2] Keith A Baggerly, Kevin R Coombes, and E. Shannon Neeley. Run batch effects potentially compromise the usefulness of genomic signatures for ovarian cancer. *Journal of Clinical Oncology*, 26(7):1186–1187, March 2008.

[3] Andreas Scherer. *Batch effects and noise in microarray experiments: sources and solutions*. John Wiley and Sons, December 2009.

[4] Jeffrey T. Leek, Robert B. Scharpf, Héctor Corrada Bravo, David Simcha, Benjamin Langmead, W. Evan Johnson, Donald Geman, Keith Baggerly, and Rafael A. Irizarry. Tackling the widespread and critical impact of batch effects in high-throughput data. *Nat Rev Genet*, 11(10):733–739, October 2010.

[5] L. Murray. In *Randomized Complete Block Designs*. John Wiley & Sons, Ltd, July 2005.

[6] Douglas C. Montgomery. *Design and Analysis of Experiments*. Wiley, 7 edition, July 2008.

[7] Kai-Tai Fang and Chang-Xing Ma. *Uniform and Orthogonal Designs*. Science Press,Beijing, September 2001.

[8] C. F. Jeff Wu and Michael S. Hamada. *Experiments: Planning, Analysis, and Optimization*. Wiley, 2nd edition, August 2009.

[9] Chang-Xing Ma, Kai-Tai Fang, and Erkki Liski. A new approach in constructing orthogonal and nearly orthogonal arrays. *Metrika*, 50(3):255–268, 2000.

[10] Jeffrey T. Leek, W. Evan Johnson, Hilary S. Parker, Andrew E. Jaffe, and John D. Storey. The sva package for removing batch effects and other unwanted variation in high-throughput experiments. *Bioinformatics*, January 2012.

[11] Chao Chen, Kay Grennan, Judith Badner, Dandan Zhang, Elliot Gershon, Li Jin, and Chunyu Liu. Removing batch effects in analysis of expression microarray data: An evaluation of six batch adjustment methods. *PLoS ONE*, 6(2):e17238, February 2011.

[12] Hanwen Huang, Xiaosun Lu, Yufeng Liu, Perry Haaland, and J. S Marron. R/DWD: Distance-Weighted discrimination for classification, visualization and batch adjustment. *Bioinformatics*, 28(8):1182–1183, April 2012.