Design and Analysis of Experiments with Randomizr

Alexander Coppock

2015-11-23

`randomizr` is a small package for r that simplifies the design and analysis of randomized experiments. In particular, it makes the random assignment **procedure** transparent, flexible, and most importantly reproduceable. By the time that many experiments are written up and made public, the process by which some units recieved treatments is lost or imprecisely described. The `randomizr` package makes it easy for even the most forgetful of researchers to generate error-free, reproduceable random assignments.

A hazy understanding of the random assignment procedure leads to two main problems at the analysis stage. First, units may have different probabilities of assignment to treatment. Analyzing the data as though they have the same probabilities of assignment leads to biased estimates of the treatment effect. Second, units are sometimes assigned to treatment as a **cluster**. For example, all the students in a single classroom may be assigned to the same intervention together. If the analysis ignores the clustering in the assignments, estimates of uncertainty may be incorrect.

A Hypothetical Experiment

Throughout this vignette, we'll pretend we're conducting an experiment among the 592 individuals in the built-in `HairEyeColor` dataset. As we'll see, there are many ways to randomly assign subjects to treatments. We'll step through five common designs, each associated with one of the five `randomizr` functions: `simple_ra()`, `complete_ra()`, `block_ra()`, `cluster_ra()`, and `block_and_cluster_ra()`.

Setting up the experiment

We first need to transform the dataset, which has each row describe a **type** of subject, to a new dataset in which each row describes an individual subject.

```
# Load built-in dataset
data(HairEyeColor)
HairEyeColor <- data.frame(HairEyeColor)

# Transform so each row is a subject
# Columns describe subject's hair color, eye color, and gender
hec <- HairEyeColor[rep(1:nrow(HairEyeColor),
                        times = HairEyeColor$Freq), 1:3]

N <- nrow(hec)

# Fix the rownames
rownames(hec) <- 1:N
```

Typically, researchers know some basic information about their subjects before deploying treatment. For example, they usually know how many subjects there are in the experimental sample (N), and they usually know some basic demographic information about each subject.

Our new dataset has 592 subjects. We have three pretreatment covariates, `Hair`, `Eye`, and `Sex`, which describe the hair color, eye color, and gender of each subject.

We now need to create simulated *potential outcomes*. We'll call the untreated outcome `Y0` and we'll call the treated outcome `Y1`. Imagine that in the absence of any intervention, the outcome (`Y0`) is correlated with out pretreatment covariates. Imagine further that the effectiveness of the program varies according to these covariates, i.e., the difference between `Y1` and `Y0` is correlated with the pretreatment covariates.

If we were really running an experiment, we would only observe either `Y0` or `Y1` for each subject, but since we are simulating, we generate both. Our inferential target is the average treatment effect (ATE), which is defined as the average difference between `Y0` and `Y1`.

```
# Set a seed for reproducability
set.seed(343)

# Create untreated and treated outcomes for all subjects
hec <- within(hec,{
  Y0 <- rnorm(n = N,mean = (2*as.numeric(Hair) + -4*as.numeric(Eye) + -6*as.numeric(Sex)), sd = 5)
  Y1 <- Y0 + 6*as.numeric(Hair) + 4*as.numeric(Eye) + 2*as.numeric(Sex)
})

# Calculate true ATE
with(hec, mean(Y1 - Y0))
```

```
## [1] 25.26351
```

We are now ready to allocate treatment assignments to subjects. Let's start by contrasting simple and complete random assignment.

Simple Random Assignment

Simple random assignment assigns all subjects to treatment with an equal probability by flipping a (weighted) coin for each subject. The main trouble with simple random assignment is that the number of subjects assigned to treatment is itself a random number - depending on the random assignment, a different number of subjects might be assigned to each group.

The `simple_ra()` function has one required argument `N`, the total number of subjects. If no other arguments are specified, `simple_ra()` assumes a two-group design and a 0.50 probability of assignment.

```
library(randomizr)
Z <- simple_ra(N = N)
table(Z)
```

```
## Z
##   0   1
## 301 291
```

To change the probability of assignment, specify the `prob` argument:

```
Z <- simple_ra(N = N, prob = 0.30)
table(Z)
```

```
## Z
##   0   1
## 402 190
```

If you specify `num_arms` without changing `prob_each`, `simple_ra()` will assume equal probabilities across all arms.

```
Z <- simple_ra(N = N, num_arms = 3)
table(Z)
```

```
## Z
##  T1  T2  T3
## 186 191 215
```

You can also just specify the probabilites of your multiple arms. The probabilities must sum to 1.

```
Z <- simple_ra(N = N, prob_each = c(.2, .2, .6))
table(Z)
```

```
## Z
##  T1  T2  T3
## 125 111 356
```

You can also name your treatment arms.

```
Z <- simple_ra(N = N, prob_each = c(.2, .2, .6),
               condition_names=c("control", "placebo", "treatment"))
table(Z)
```

```
## Z
##   control   placebo treatment
##       103       127       362
```

Complete Random Assignment

Complete random assignment is very similar to simple random assignment, except that the researcher can specify *exactly* how many units are assigned to each condition.

The syntax for `complete_ra()` is very similar to that of `simple_ra()`. The argument `m` is the number of units assigned to treatment in two-arm designs; it is analogous to `simple_ra()`'s `prob`. Similarly, the argument `m_each` is analogous to `prob_each`.

If you only specify `N`, `complete_ra()` assigns exactly half of the subjects to treatment.

```
Z <- complete_ra(N = N)
table(Z)
```

```
## Z
##   0   1
## 296 296
```

To change the number of units assigned, specify the `m` argument:

```
Z <- complete_ra(N = N, m=200)
table(Z)
```

```
## Z
##   0   1
## 392 200
```

If you specify multiple arms, `complete_ra()` will assign an equal (within rounding) number of units to treatment.

```
Z <- complete_ra(N = N, num_arms = 3)
table(Z)
```

```
## Z
##  T1  T2  T3
## 198 197 197
```

You can also specify exactly how many units should be assigned to each arm. The total of `m_each` must equal `N`.

```
Z <- complete_ra(N = N, m_each = c(100, 200, 292))
table(Z)
```

```
## Z
##  T1  T2  T3
## 100 200 292
```

You can also name your treatment arms.

```
Z <- complete_ra(N = N, m_each = c(100, 200, 292),
                 condition_names=c("control", "placebo", "treatment"))
table(Z)
```

```
## Z
##   control   placebo treatment
##       100       200       292
```

Simple and Complete Random Assignment Compared

When should you use `simple_ra()` versus `complete_ra()`? Basically, if the number of units is known beforehand, `complete_ra()` is always preferred, for two reasons: 1. Researchers can plan exactly how many treatments will be deployed. 2. The standard errors associated with complete random assignment are generally smaller, increasing experimental power. See this guide on EGAP for more on experimental power.

Since you need to know `N` beforehand in order to use `simple_ra()`, it may seem like a useless function. Sometimes, however, the random assignment isn't directly in the researcher's control. For example, when deploying a survey exeriment on a platform like Qualtrics, simple random assignment is the only possibility due to the inflexibility of the built-in random assignment tools. When reconstructing the random assignment for analysis after the experiment has been conducted, `simple_ra()` provides a convenient way to do so.

To demonstrate how `complete_ra()` is superior to `simple_ra()`, let's conduct a small simulation with our `HairEyeColor` dataset.

```r
sims <- 1000

# Set up empty vectors to collect results
simple_ests <- rep(NA, sims)
complete_ests <- rep(NA, sims)

# Loop through simulation 2000 times
for(i in 1:sims){
  hec <- within(hec,{

    # Conduct both kinds of random assignment
    Z_simple <- simple_ra(N = N)
    Z_complete <- complete_ra(N = N)

    # Reveal observed potential outcomes
    Y_simple <- Y1*Z_simple + Y0*(1-Z_simple)
    Y_complete <- Y1*Z_complete + Y0*(1-Z_complete)
    })

  # Estimate ATE under both models
  fit_simple <- lm(Y_simple ~ Z_simple, data=hec)
  fit_complete <- lm(Y_complete ~ Z_complete, data=hec)

  # Save the estimates
  simple_ests[i] <- coef(fit_simple)[2]
  complete_ests[i] <- coef(fit_complete)[2]
}
```

The standard error of an estimate is defined as the standard deviation of the sampling distribution of the estimator. When standard errors are estimated (i.e., by using the `summary()` command on a model fit), they are estimated using some approximation. This simulation allows us to measure the standard error directly, since the vectors `simple_ests` and `complete_ests` describe the sampling distribution of each design.

```r
sd(simple_ests)
```

```
## [1] 0.6245056
```

```r
sd(complete_ests)
```

```
## [1] 0.596656
```

In this simulation complete random assignment led to a 8.72% decrease in sampling variability. This decrease was obtained with a small design tweak that costs the researcher essentially nothing.

Block Random Assignment

Block random assignment (sometimes known as stratified random assignment) is a powerful tool when used well. In this design, subjects are sorted into blocks (strata) according to their pre-treatment covariates, and then complete random assignment is conducted within each block. For example, a researcher might block on gender, assigning exactly half of the men and exactly half of the women to treatment.

Why block? The first reason is to signal to future readers that treatment effect heterogeneity may be of interest: is the treatment effect different for men versus women? Of course, such heterogeneity could be explored if complete random assignment had been used, but blocking on a covariate defends a researcher (somewhat) against claims of data dredging. The second reason is to increase precision. If the blocking variables are predicitive of the outcome (i.e., they are correlated with the outcome), then blocking may help to decrease sampling variability. It's important, however, not to overstate these advantages. The gains from a blocked design can often be realized through covariate adjustment alone.

Blocking can also produce complications for estimation. Blocking can produce different probabilities of assignment for different subjects. This complication is typically addressed in one of two ways: "controlling for blocks" in a regression context, or inverse probabilitity weights (IPW), in which units are weighted by the inverse of the probability that the unit is in the condition that it is in.

The only required argument to `block_ra()` is `block_var`, which is a vector of length `N` that describes which block a unit belongs to. `block_var` can be a factor, character, or numeric variable. If no other arguments are specified, `block_ra()` assigns an approximately equal proportion of each block to treatment.

```
Z <- block_ra(block_var = hec$Hair)
table(Z, hec$Hair)
```

```
##
## Z   Black Brown Red Blond
##   0    54   143  36    64
##   1    54   143  35    63
```

For multiple treatment arms, use the `num_arms` argument, with or without the `condition_names` argument

```
Z <- block_ra(block_var = hec$Hair, num_arms=3)
table(Z, hec$Hair)
```

```
##
## Z    Black Brown Red Blond
##   T1    36    95  24    42
##   T2    36    95  23    42
##   T3    36    96  24    43
```

```
Z <- block_ra(block_var = hec$Hair, condition_names=c("Control", "Placebo", "Treatment"))
table(Z, hec$Hair)
```

```
##
## Z           Black Brown Red Blond
##   Control      36    95  24    42
##   Placebo      36    95  24    42
##   Treatment    36    96  23    43
```

`block_ra()` provides a number of ways to adjust the number of subjects assigned to each conditions. The `prob_each` argument describes what proportion of each block should be assigned to treatment arm. Note of course, that `block_ra()` still uses complete random assignment within each block; the appropriate number of units to assign to treatment within each block is automatically determined.

```
Z <- block_ra(block_var = hec$Hair,prob_each = c(.3, .7))
table(Z, hec$Hair)
```

```
##
## Z   Black Brown Red Blond
##   0    33    85  22    38
##   1    75   201  49    89
```

For finer control, use the `block_m` argument, which takes a matrix with as many rows as there are blocks, and as many columns as there are treatment conditions. Remember that the rows are in the same order as `sort(unique(block_var))`, a command that is good to run before constructing a `block_m` matrix.

```
sort(unique(hec$Hair))
```

```
## [1] Black Brown Red   Blond
## Levels: Black Brown Red Blond
```

```
block_m <- rbind(c(78, 30),
                 c(186, 100),
                 c(51, 20),
                 c(87,40))

block_m
```

```
##      [,1] [,2]
## [1,]   78   30
## [2,]  186  100
## [3,]   51   20
## [4,]   87   40
```

```
Z <- block_ra(block_var = hec$Hair, block_m = block_m)
table(Z, hec$Hair)
```

```
##
## Z   Black Brown Red Blond
##   0    78   186  51    87
##   1    30   100  20    40
```

In the example above, the different blocks have different probabilities of assignment to treatment. In this case, people with Black hair have a $30/108 = 27.8\%$ chance of being treated, those with Brown hair have $100/286 = 35.0\%$ change, etc. Left unadressed, this discrepancy could bias treatment effects. We can see this directly with the `declare_ra()` function.

```
declaration <- declare_ra(block_var = hec$Hair, block_m = block_m)

# show the probability that each unit is assigned to each condition
head(declaration$probabilities_matrix)
```

```
##         prob_0    prob_1
## [1,] 0.7222222 0.2777778
## [2,] 0.7222222 0.2777778
## [3,] 0.7222222 0.2777778
## [4,] 0.7222222 0.2777778
## [5,] 0.7222222 0.2777778
## [6,] 0.7222222 0.2777778
```

```
# Show that the probability of treatment is different within block
table(hec$Hair, round(declaration$probabilities_matrix[,2], 3))
```

```
##
##         0.278 0.282 0.315 0.35
##   Black   108     0     0    0
##   Brown     0     0     0  286
##   Red       0    71     0    0
##   Blond     0     0   127    0
```

There are commong two ways to address this problem: LSDV (Least-Squares Dummy Variable, also known as "control for blocks") or IPW (Inverse-probability weights).

The following code snippet shows how to use either the LSDV approach or the IPW approach. A note for scrupulous readers: the estimands of these two approaches are subtly different from one another. The LSDV approach estimates the average **block-level** treatment effect. The IPW approach estimates the average **individual-level** treatment effect. They can be different. Since the average block-level treatment effect is not what most people have in mind when thinking about causal effects, analysts using this approach should present both. The `condition_probs()` function used to calculate the probabilities of assignment is explained below.

```
hec <- within(hec,{
  Z_blocked <- block_ra(block_var = hec$Hair,block_m=block_m, condition_names = c(0, 1))
  Y_blocked <- Y1*(Z_blocked) + Y0*(1-Z_blocked)
  cond_prob <- obtain_condition_probabilities(declaration, Z_blocked)
  IPW_weights <- 1/(cond_prob)
})

fit_LSDV <- lm(Y_blocked ~ Z_blocked + Hair, data=hec)
fit_IPW <- lm(Y_blocked ~ Z_blocked, weights=IPW_weights, data=hec)

summary(fit_LSDV)
```

```
##
## Call:
## lm(formula = Y_blocked ~ Z_blocked + Hair, data = hec)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -22.1086  -4.2795   0.3799   4.6342  18.6040
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -16.2037     0.6855 -23.639  < 2e-16 ***
## Z_blocked    24.9493     0.6090  40.965  < 2e-16 ***
## HairBrown     2.3447     0.7809   3.003  0.00279 **
## HairRed       5.8248     1.0547   5.523 5.02e-08 ***
## HairBlond     9.3325     0.9039  10.325  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.903 on 587 degrees of freedom
## Multiple R-squared:  0.7555, Adjusted R-squared:  0.7539
## F-statistic: 453.5 on 4 and 587 DF,  p-value: < 2.2e-16
```

```r
summary(fit_IPW)
```

```
##
## Call:
## lm(formula = Y_blocked ~ Z_blocked, data = hec, weights = IPW_weights)
##
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -41.038  -6.768   0.114   6.872  41.074
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -12.3758     0.4691  -26.38   <2e-16 ***
## Z_blocked    24.9021     0.6634   37.54   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.41 on 590 degrees of freedom
## Multiple R-squared:  0.7048, Adjusted R-squared:  0.7043
## F-statistic:  1409 on 1 and 590 DF,  p-value: < 2.2e-16
```

How to create blocks? In the `HairEyeColor` dataset, we could make blocks for each unique combination of hair color, eye color, and sex using `dplyr`'s `id` function:

```r
suppressMessages(library(dplyr))
block_id <- id(hec[,c("Hair", "Eye", "Sex")])
block_var <- paste0("block_", sprintf("%02d", block_id))
table(block_var)
```

```
## block_var
## block_01 block_02 block_03 block_04 block_05 block_06 block_07 block_08
##       32       36       11        9       10        5        3        2
## block_09 block_10 block_11 block_12 block_13 block_14 block_15 block_16
##       53       66       50       34       25       29       15       14
## block_17 block_18 block_19 block_20 block_21 block_22 block_23 block_24
##       10       16       10        7        7        7        7        7
## block_25 block_26 block_27 block_28 block_29 block_30 block_31 block_32
##        3        4       30       64        5        5        8        8
```

An alternative is to use the `blockTools` package, which constructs matched pairs, trios, quartets, etc. from pretreatment covariates.

```r
library(blockTools)

# BlockTools requires that all variables be numeric
numeric_mat <- model.matrix(~Hair+Eye+Sex, data=hec)[,-1]

# BlockTools also requres an id variable
df_forBT <- data.frame(id_var = 1:nrow(numeric_mat), numeric_mat)

# Conducting the actual blocking: let's make trios
out <- block(df_forBT, n.tr = 3, id.vars = "id_var",
```

```
            block.vars = colnames(df_forBT)[-1])

# Extact the block_ids
hec$block_id <- createBlockIDs(out, df_forBT, id.var = "id_var")

# Conduct actual random assignment with randomizr
Z_blocked <- block_ra(block_var = hec$block_id, num_arms = 3)
head(table(hec$block_id, Z_blocked))
```

```
##    Z_blocked
##     T1 T2 T3
##   1  1  1  1
##   2  1  1  1
##   3  1  1  1
##   4  1  1  1
##   5  1  1  1
##   6  1  1  1
```

A note for `blockTools` users: that package also has an assignment function. My preference is to extract the blocking variable, then conduct the assignment with `block_ra()`, so that fewer steps are required to reconstruct the random assignment or generate new random assignments for a randomization inference procedure.

Clustered assignment

Clustered assignment is unfortunate. If you can avoid assigning subjects to treatments by cluster, you should. Sometimes, clustered assignment is unavoidable. Some common situations include:

1. Housemates in households: whole households are assigned to treatment or control
2. Students in classrooms: whole classrooms are assigned to treatment or control
3. Residents in towns or villages: whole communities are assigned to treatment or control

Clustered assignment decreases the effective sample size of an experiment. In the extreme case when outcomes are perfectly correlated with clusters, the experiment has an effective sample size equal to the number of clusters. When outcomes are perfectly uncorrelated with clusters, the effective sample size is equal to the number of subjects. Almost all cluster-assigned experiments fall somewhere in the middle of these two extremes.

The only required argument for the `cluster_ra()` function is the `clust_var` argument, which is a vector of length `N` that indicates which cluster each subject belongs to. Let's pretend that for some reason, we have to assign treatments according to the unique combinations of hair color, eye color, and gender. For this, we'll use `dplyr`'s `id()` function again.

```
clust_id <- id(hec[,c("Hair", "Eye", "Sex")])
clust_var <- paste0("clust_", sprintf("%02d", clust_id))
hec$clust_var <- clust_var

Z_clust <- cluster_ra(clust_var = clust_var)
head(table(clust_var, Z_clust))
```

```
##           Z_clust
## clust_var   0  1
##   clust_01 32  0
##   clust_02 36  0
##   clust_03  0 11
##   clust_04  0  9
##   clust_05  0 10
##   clust_06  0  5
```

This shows that each cluster is either assigned to treatment or control. No two units within the same cluster are assigned to different conditions.

As with all functions in `randomizr`, you can specify multiple treatment arms in a variety of ways:

```
Z_clust <- cluster_ra(clust_var=clust_var, num_arms=3)
head(table(clust_var, Z_clust))
```

```
##           Z_clust
## clust_var  T1 T2 T3
##   clust_01  0  0 32
##   clust_02 36  0  0
##   clust_03 11  0  0
##   clust_04  0  0  9
##   clust_05  0  0 10
##   clust_06  5  0  0
```

... or using `condition_names`

```
Z_clust <- cluster_ra(clust_var=clust_var,
                      condition_names=c("Control", "Placebo", "Treatment"))
head(table(clust_var, Z_clust))
```

```
##           Z_clust
## clust_var  Control Placebo Treatment
##   clust_01      32       0         0
##   clust_02       0       0        36
##   clust_03       0       0        11
##   clust_04       0       0         9
##   clust_05       0      10         0
##   clust_06       0       5         0
```

... or using `m_each`, which describes how many clusters should be assigned to each condition. `m_each` must sum to the number of clusters.

```
Z_clust <- cluster_ra(clust_var=clust_var, m_each=c(5, 15, 12))
head(table(clust_var, Z_clust))
```

```
##           Z_clust
## clust_var  T1 T2 T3
##   clust_01  0 32  0
##   clust_02 36  0  0
##   clust_03  0 11  0
##   clust_04  0  9  0
##   clust_05  0  0 10
##   clust_06  0  0  5
```

Blocked and clustered assignment

The power of clustered experiments can sometimes be improved through blocking. In this scenario, whole clusters are members of a particular block – imagine villages nested within discrete regions, or classrooms nested within discrete schools.

As an example, let's group our clusters into blocks by size:

```
cluster_level_df <-
  hec %>%
  group_by(clust_var) %>%
  summarize(cluster_size = n()) %>%
  arrange(cluster_size) %>%
  mutate(block_var = paste0("block_", sprintf("%02d",rep(1:16, each=2))))

hec <- left_join(hec, cluster_level_df)
```

```
## Joining by: "clust_var"
```

```
# Extract the cluster and block variables
clust_var <- hec$clust_var
block_var <- hec$block_var

Z <- block_and_cluster_ra(clust_var = clust_var, block_var = block_var)
head(table(clust_var, Z))
```

```
##           Z
## clust_var   0  1
##   clust_01  0 32
##   clust_02  0 36
##   clust_03  0 11
##   clust_04  0  9
##   clust_05 10  0
##   clust_06  0  5
```

```
head(table(block_var, Z))
```

```
##           Z
## block_var  0 1
##   block_01 3 2
##   block_02 3 4
##   block_03 5 5
##   block_04 5 7
##   block_05 7 7
##   block_06 7 7
```

Calculating probabilities of assignment

All five random assignment functions in `randomizr` assign units to treatment with known (if sometimes complicated) probabilities. The `declare_ra()` and `obtain_condition_probabilities()` functions calculate these probabilities according to the parameters of your experimental design.

Let's take a look at the block random assignment we used before.

```
block_m <- cbind(c(78, 186, 51, 87),c(30, 100, 20, 40))
Z <- block_ra(block_var = hec$Hair,block_m=block_m)
table(Z, hec$Hair)
```

```
##
## Z   Black Brown Red Blond
##   0    78   186  51    87
##   1    30   100  20    40
```

In order to calculate the probabilities of assignment, we call the `declare_ra()` function with the same exact arguments as we used for the `block_ra()` call. The `declaration` object contains a matrix of probabilities of assignment:

```
declaration <- declare_ra(block_var = hec$Hair,block_m=block_m)
prob_mat <- declaration$probabilities_matrix

head(prob_mat)
```

```
##         prob_0    prob_1
## [1,] 0.7222222 0.2777778
## [2,] 0.7222222 0.2777778
## [3,] 0.7222222 0.2777778
## [4,] 0.7222222 0.2777778
## [5,] 0.7222222 0.2777778
## [6,] 0.7222222 0.2777778
```

The `prob_mat` objects has `N` rows and as many columns as there are treatment conditions, in this case 2.

In order to use inverse-probability weights, we need to know the probability of each unit being in the **condition that it is in**. For each unit, we need to pick the appropriate probability. This bookkeeping is handled automatically by the `obtain_condition_probabilities()` function.

```
cond_prob <- obtain_condition_probabilities(declaration, Z)
table(cond_prob, Z)
```

```
##                    Z
## cond_prob             0   1
##   0.277777777777778   0  30
##   0.28169014084507    0  20
##   0.31496062992126    0  40
##   0.34965034965035    0 100
##   0.65034965034965  186   0
##   0.68503937007874   87   0
##   0.71830985915493   51   0
##   0.722222222222222  78   0
```

Best practices

Random assignment procedure = Random assignment function

Random assignment procedures are often described as a series of steps that are manually carried out be the researcher. In order to make this procedure reproducible, these steps need to be translated into a **function** that returns a different random assignment each time it is called.

For example, consider the following procedure for randomly allocating school vouchers.

1. Every eligible student's names is put on a list
2. Each name is assigned a random number
3. Balls with the numbers associated with all students are put in an urn.
4. Then the urn is "shuffled"
5. Students names are drawn one by one from the urn until all slots are given out.
6. If one sibling in a family wins, all other siblings automatically win too.

If we write such a procedure into a function, it might look like this:

```
# 400 families have 1 child in the lottery, 100 families have 2
family_id <- c(sprintf("%03d", 1:500), sprintf("%03d", 1:100))

school_ra <- function(m){
  N <- length(family_id)
  random_number <- sample(1:N, replace=FALSE)
  Z <- rep(0, N)
  i <- 1
  while(sum(Z) <m){
    Z[family_id==family_id[random_number[i]]] <- 1
    i <- i + 1
  }
  return(Z)
}

Z <- school_ra(200)
table(Z)
```

```
## Z
##   0   1
## 400 200
```

This assignment procedure is complicated by the sibling rule, which has two effects: first, students are cluster-assigned by family, and second, the probability of assignment varies student to student. Obviously, families who have two children in the lottery have a higher probability of winning the lottery because they effectively have two "tickets." There may be better ways of running this assignment procedure (for example, with `cluster_ra()`), but the purpose of this example is to show how complicated *real-world* procedures can be written up in a simple function. With this function, the random assignment procedure can be reproduced exactly, the complicated probabilities of assignment can be caluculated, and the analysis is greatly simplified.

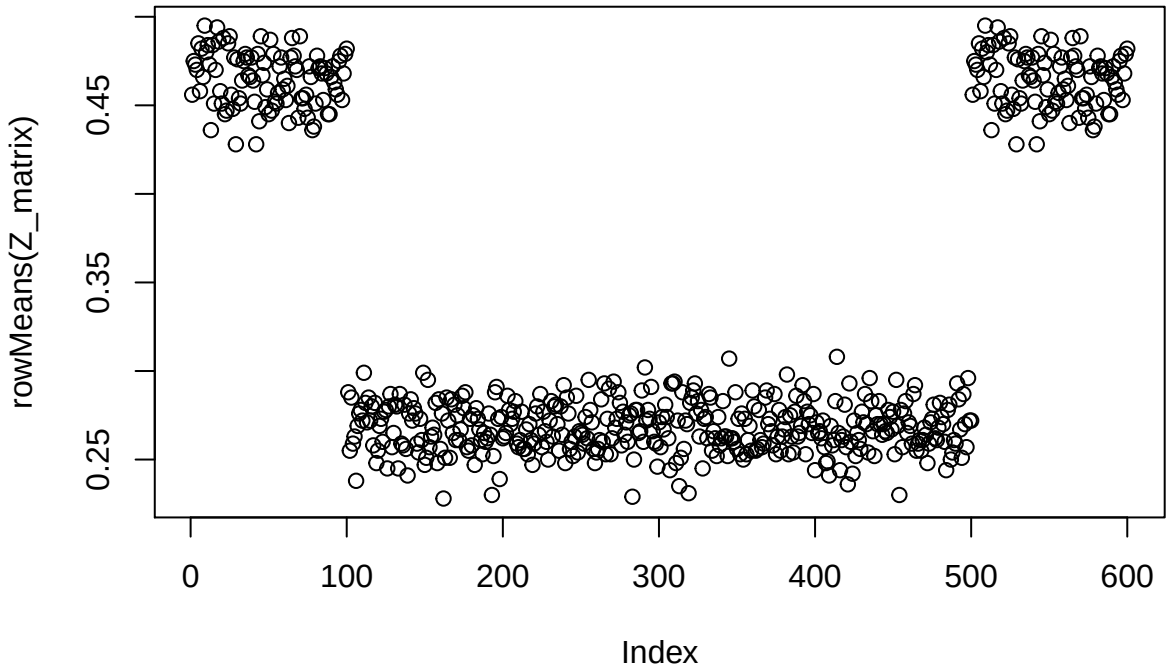
Check probabilities of assignment directly

For many designs, the probability of assignment to treatment can be calculated analytically. For example, in a completely randomized design with 200 units, 60 of which are assigned to treatment, the probability is exactly 0.30 for all units. However, in more complicated designs (such as the schools example described above), analytic probabilities are difficult to calculate. In such a situation, an easy way to obtain the probabilities of assignment is through simulation.

1. Call your random assignment function an approximately infinite number of times (about 10,000 for most purposes).

2. Count how often each unit is assigned to each treatment arm.

```
Z_matrix <- replicate(1000, school_ra(200))
plot(rowMeans(Z_matrix))
```



This plot shows that the students who have a sibling in the lottery have a higher probability of assignment. The more simulations, the more precise the estimate of the probability of assignment.

Save your random assignment

Whenever you conduct a random assignment for use in an experiment, save it! At a minimum, the random assignment should be saved with an id variable in a csv.

```
hec <- within(hec,{
  Z_blocked <- complete_ra(N = N, m_each = c(100, 200, 292),
                condition_names=c("control", "placebo", "treatment"))
  id_var <- 1:nrow(hec)
})
write.csv(hec[,c("id_var", "Z_blocked")], file="MyRandomAssignment.csv")
```