Systematic load balancing block randomization for arbitrary individual assignment probabilities

Macartan

September 3, 2017

Description of a randomization function that handles heterogeneous probabilities and awkward integer blocks sizes. The function is a lot slower than `randomizr` for routine tasks but can handle less routine tasks that `randomizr` cannot.

Two illustrations of functionality made possible with this function.

1: Jack and Jill have a race. Jill is faster than Jack and has a higher probability of winning. You want to simulate a distribution of wins. This is a situation where probabilites are heterogeneous and in which there is a target number of units to be selected. This problem is neither simple nor complete, as understood by `randomizr`.

2: You have 2 districts with 3 villages each. You want to assign 3 villages to treatment, blocking by district, and with equal probabilities for all units. This randomization requires an allocation both across and within blocks whereas `randomizr` only allocates within blocks. More generally, the issues here is that the target number to be assigned in a given block is not an integer.

These problems can both occur in a given problem and indeed you would expect them to whenever there are generic probabilities and blocks. They are not convoluted examples and it would be nice to have functionality that can handle them.

1 The function

The basic function works by doing systematic sampling over a random (but block preserving) order.

```r
.prob_ra <- function(p = .5,
                     b = NULL,
                     n = NULL,
                     tol = 10){
  # Housekeeping

  if(is.null(n)) {if(!is.null(b)) n <- length(b)
                  if( is.null(b) & length(p)>1)  n <- length(p)}
  if(length(p) == 1) p <- rep(p, n)
  if(is.null(b))     b <- rep(1, n)
  p   <- round(p, tol)
  m   <- ceiling(sum(p))

  if(m == 0) return(rep(0, length(p)))

  # Figure out if we have to deal with a random total
  tag <- m > floor(sum(p))

  if(tag){
    p <- c(p, ceiling(sum(p)) - sum(p))
    n <- n+1
    b <- c(b, ".dummy")
    }

  base <- p - p%%1
  p <- p - base

  # randomly order blocks then reorder within blocks
  b_names   <- unique(b)
  k         <- length(b_names)
  seq1      <- rep(NA, length(b))
  b_shuffle <- sample(1:k)
  for(j in 1:k) seq1[b==b_names[j]] <- b_shuffle[j]
  seq2      <- rank(seq1 + runif(n))
  p[seq2]   <- p

  # Now  do systematic assignment
  s   <- (cumsum(p) +m*runif(1))%%m
  e   <- s - floor(s)
  out <- 1*(e < c(e[n], e[-n]))
  out <- out[seq2]
  out <- out + base
  if(tag) out <- out[-n]
  return(out)
}
```

The more general function applies this for each treatment:

```
prob_ra <- function(p = .5,
                    b = NULL,
                    n = NULL){

if(is.null(ncol(p))) {Z <- .prob_ra(p, b, n)
} else {
Z <- matrix(NA, nrow(p), ncol(p))
Z[,1] <- .prob_ra(p[,1],b,n)

for(j in 2:ncol(p)){
  q <- p[,j]
  q[apply(Z, 1, sum, na.rm = TRUE)==1] <- 0
  q <- q/(1-apply(as.matrix(p[,1:(j-1)]), 1, sum, na.rm = TRUE))
  q[is.nan(q)] <- 0
  Z[,j] <- .prob_ra(as.vector(q), b, n)
  }
Z <-Z%*%matrix(1:ncol(p),ncol(p))
}
Z}
```

2 Illustration: One arm

With random data:

```
s    <- 100
p    <- runif(s)
b    <- sample(1:5, s, replace = TRUE, prob = 1:5)
sims <- 10000
runs <- replicate(sims, prob_ra(p, b))
```

2.1 Total selected is as tight as possible

There should only be a unit difference between the totals assigned in any set of runs:

```
table(apply(runs, 2, sum))
```

```
##
##   48   49
## 1959 8041
```

2.2 Total selected *in each block* is also as tight as possible

Should be only max 1 unit between min and max

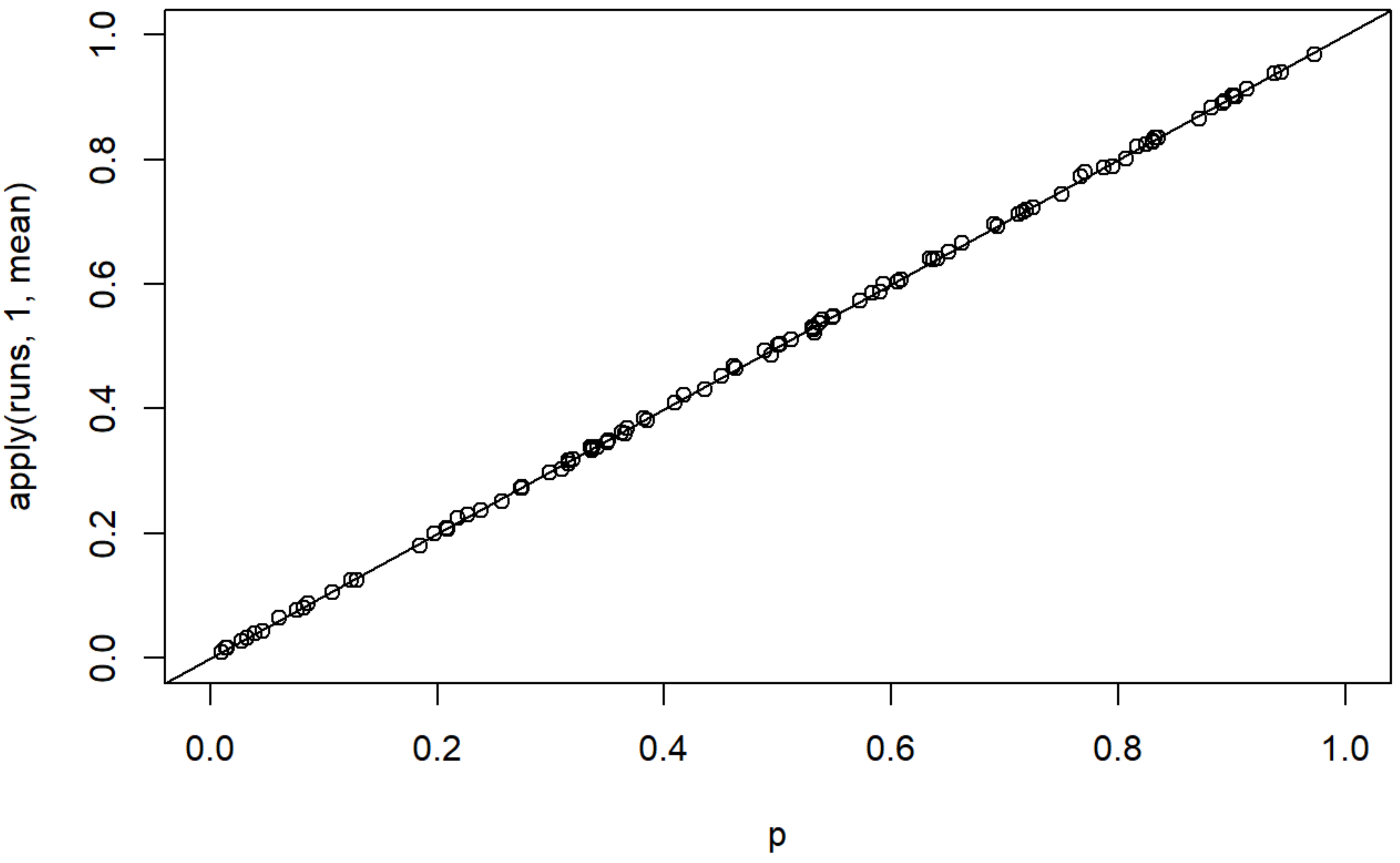
```
bin_dist <- apply(runs, 2, function(j) table(b, j)[,2])
table_check <- t(rbind(apply(bin_dist, 1, function(j)  c(mean(j), min(j), max(j)))))
colnames(table_check) <- c("sim_p", "min", "max")
```

```
kable(round(cbind(size = table(b), true_p = aggregate(p, by = list(b), FUN = sum)[,2]
, table_check), 2))
```

size	true_p	sim_p	min	max
10	4.66	4.66	4	5
11	5.33	5.34	5	6
22	9.59	9.58	9	10
21	10.43	10.43	10	11
36	18.80	18.79	18	19

2.3 True assignment probabilities are respected at the lowest level

```
plot(p, apply(runs, 1, mean), xlim = c(0,1), ylim = c(0,1))
abline(0,1)
```



3 Illustration: Multiple arms

The function can also be used sequentially for multiple treatment. In this case it implements the based treatment in a *hierarchical* manner, which preserves individual probabilities, but prioritizes balancing by order.

3.1 Multiple Treatments Illustration

```
s  <- 100
b  <- sample(1:5, s, replace = TRUE, prob = 1:5)
p1 <- runif(s)
p2 <- runif(s)*(1-p1)
p <- cbind(p1,p2)
```

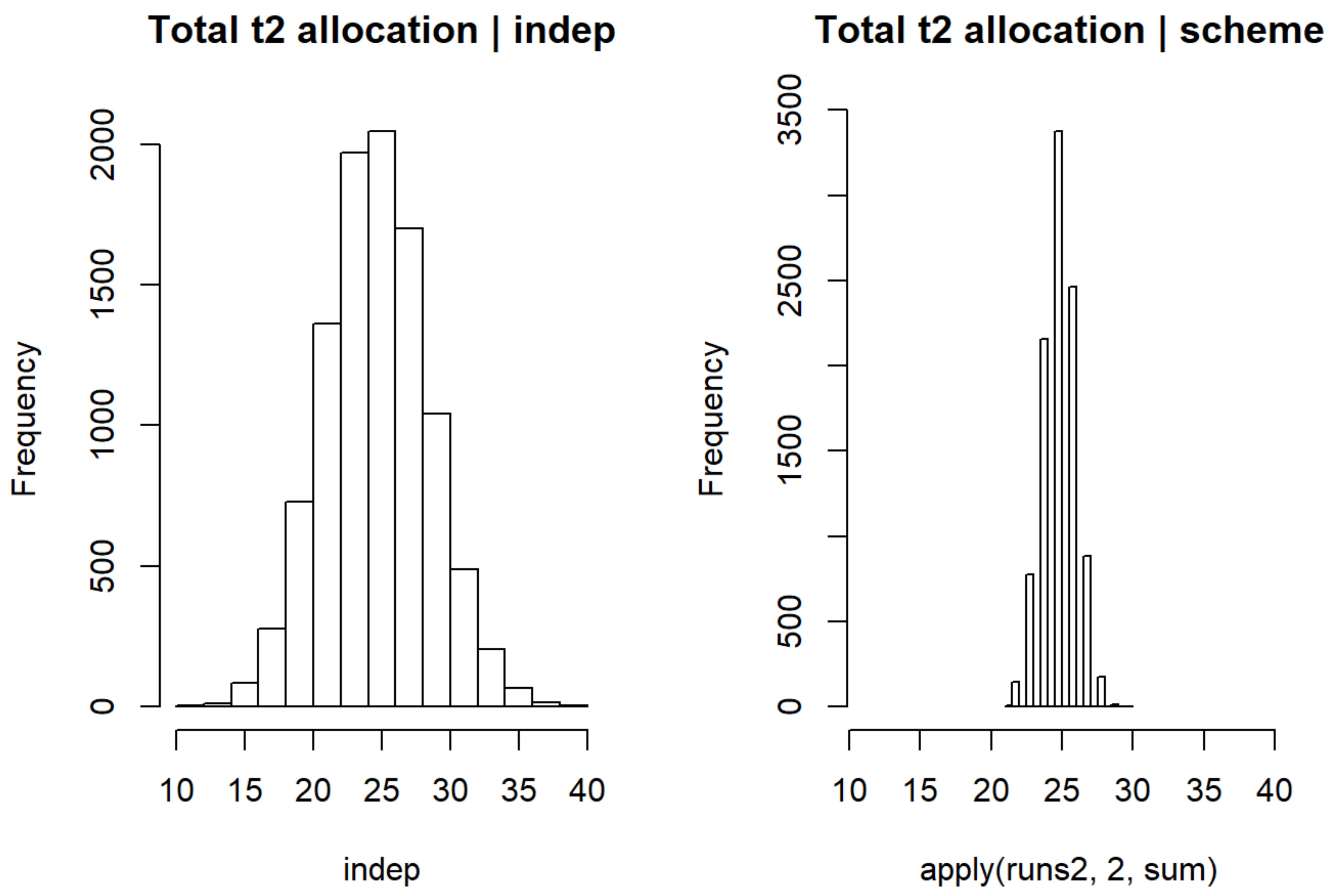
Note that t2 will be systematic, like t1, *given* t1, but not unconditionally systematic

We do two step allocation: first allocate t1 optimally and then given this allocation we allocate t2. We do this many times to check that the probability of assignments are all correct for p2.

```
runs2 <- replicate(sims, 1*(as.vector(prob_ra(p))==2))
```

TRhe result is much tighter than independent, but not as tight as possible as possible

```
par(mfrow=c(1,2))
indep <- replicate(sims,  sum(rbinom(length(p2), 1, p2)))
hist(indep, main = "Total t2 allocation | indep")
hist(apply(runs2, 2, sum), main = "Total t2 allocation | scheme", xlim = range(indep)
)
```



(Aside: would be useful to compare with distribution given random independent block targets.)

3.2 Similarly total selected in each bin is tight but not as tight as possible

Ideally max 1 unit between min and max

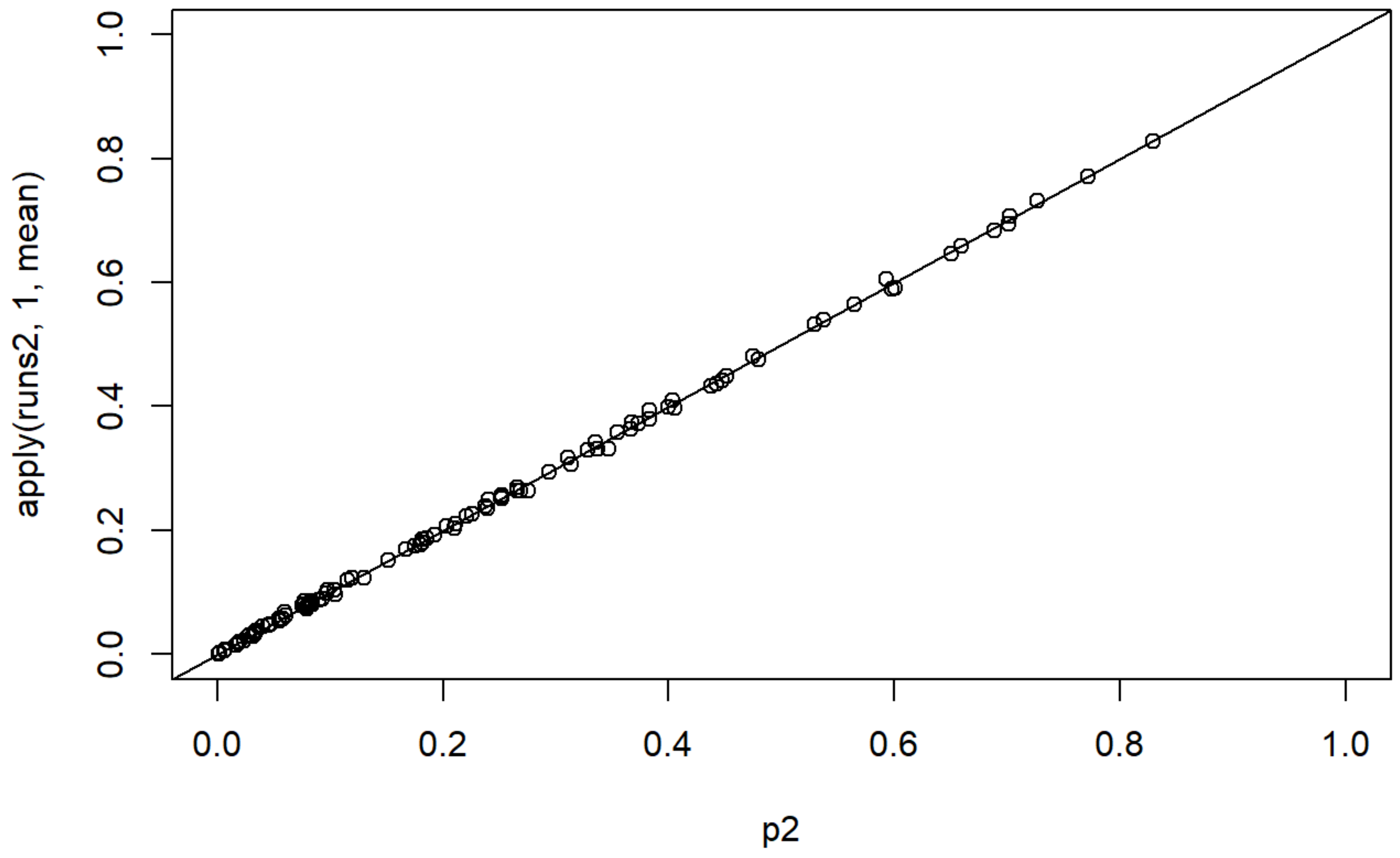
```
bin_dist <- apply(runs2, 2, function(j) table(b, j)[,2])
table_check <- t(rbind(apply(bin_dist, 1, function(j)  c(mean(j), min(j), max(j)))))
colnames(table_check) <- c("sim_p", "min", "max")

kable(round(cbind(size = table(b), true_p = aggregate(p2, by = list(b), FUN = sum)[,2
], table_check), 2))
```

size	true_p	sim_p	min	max
5	1.13	1.14	0	5
10	1.71	1.72	0	7
20	7.07	7.04	1	14
27	7.53	7.55	2	14
38	7.63	7.62	2	15

3.3 But again the true probabilities preserved at unit level (and so also at block levels)

```
plot(p2, apply(runs2, 1, mean), xlim = c(0,1), ylim = c(0,1))
abline(0,1)
```



4 Harder designs

Multiple treatments and arms can be difficult sometimes.

“Neat” designs with no integer issues are handled easily:

```
b <- rep(1:4, each = 4)
p <- matrix(.25, 16, 4)
z <- prob_ra(b=b, p = p)
table(b, z)
```

```
##     z
## b   1 2 3 4
##   1 1 1 1 1
##   2 1 1 1 1
##   3 1 1 1 1
##   4 1 1 1 1
```

A hard example with reasonable solution:

```
p <- rbind(c(2/3, 1/3, 0),
           c( 0,  2/3, 1/3),
           c( 1/3, 0,  2/3))
p <- rbind(p,p,p,p)
b <- rep(1:2, each = 6)
z <- prob_ra(b=b, p = p)
table(z, b)
```

```
##    b
## z   1 2
##   1 2 2
##   2 2 3
##   3 2 1
```

another hard one:

```
p <- t(replicate(12, c(.5, .25, .25)))
b <- rep(1:4, each = 3)
z <- prob_ra(b=b, p = p)
table(z, b)
```

```
##    b
## z   1 2 3 4
##   1 2 1 1 2
##   2 1 1 1 0
##   3 0 1 1 1
```

A harder case with suboptimal results and where ordering matters

Here is a hard case with no blocks but multiple treatments. The issue is that optimality depends on the ordering of the blocks.

Consider this:

```
sims <- 10000
p <- t(matrix(c(.15,.65,.2, .47, .48, .05), 3,2))
p
```

```
##      [,1] [,2] [,3]
## [1,] 0.15 0.65 0.20
## [2,] 0.47 0.48 0.05
```

```
runs <- sapply(1:sims, function(j) prob_ra(p = p))
round(sapply(1:3, function(j) apply(runs==j, 1, sum)), 2)/sims
```

```
##        [,1]   [,2]   [,3]
## [1,] 0.1442 0.6553 0.2005
## [2,] 0.4728 0.4778 0.0494
```

Assignment probabilities are hard. But ideally there would be at least one unit in treatment 2 in each draw, but sometimes none here....

```
set.seed(17)
prob_ra(p = p)
```

```
##      [,1]
## [1,]    3
## [2,]    1
```

Report the number in T2 in each draw:

```
share_in_t2 <- table(apply(runs==2, 2, sum))/sims
share_in_t2
```

```
##
##      0      1      2
## 0.1228 0.6213 0.2559
```

This has too much diversity as seen here by the set of cases in which no unit is assigned to T2. Though it still produces the correct allocations on average:

```
c(expectation = sum(p[,2]), average = (share_in_t2%*%as.numeric(names(share_in_t2))))
```

```
## expectation     average
##      1.1300      1.1331
```

Compare with this:

```
p <- p[, c(2,1,3)]
p
```

```
##      [,1] [,2] [,3]
## [1,] 0.65 0.15 0.20
## [2,] 0.48 0.47 0.05
```

```
runs2 <- sapply(1:sims, function(j) prob_ra(p = p))
round(sapply(1:3, function(j) apply(runs2==j, 1, mean)), 2)
```

```
##      [,1] [,2] [,3]
## [1,] 0.64 0.15 0.21
## [2,] 0.49 0.47 0.05
```

```
share_in_t1 <- table(apply(runs2==1, 2, sum))/sims
share_in_t1
```

```
##
##      1      2
## 0.8704 0.1296
```

```
c(expectation = sum(p[,1]), average = (share_in_t1%*%as.numeric(names(share_in_t1))))
```

```
## expectation     average
##      1.1300      1.1296
```

Note that in the first case the treatment is alternatively given to 1 or no units; in the second case it is given to 1 or 2 units.

So interestingly smart ordering can solve the problem; perhaps this can be partly built into the method.

5 Illustration of simple applications

The function is general enough to do normal blocks an clusters (with a wrapper that supplies cluster information) though it's probably a lot slower than `randomizr` functions for standard designs.

5.1 Just n provided

```
prob_ra(n = 4)
```

```
## [1] 0 1 0 1
```

5.2 Just blocks b provided

Here a matched pair

```
prob_ra(b=rep(1:5, each = 3))
```

```
##  [1] 0 1 0 1 0 1 1 0 0 0 0 1 1 1 0
```

5.3 Just probability vector p provided

```
prob_ra(p = c(.4, .6))
```

```
## [1] 0 1
```

5.4 Treatment probabilities do not have to sum to 1

Here probabilities for two treatments are provided and so there is an implicit residual category. The residual is assigned label 0.

```
p <- matrix(c(.25,.35,.4, .47, .48, .05), 3,2)
p
```

```
##      [,1] [,2]
## [1,] 0.25 0.47
## [2,] 0.35 0.48
## [3,] 0.40 0.05
```

```
set.seed(2)
prob_ra(p = p)
```

```
##      [,1]
## [1,]    2
## [2,]    0
## [3,]    1
```

5.5 Probabilities that *exceed* 1

One can think of the probability vector as reporting the expected number of units rather than the probability. In this case the values can exceed 1. This is useful for example if one simply wanted to do the allocation across blocks and do the within block allocation is a second stage.

```
prob_ra(p =  c(1.2, .8, 2.1, .9))
```

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

```
round(apply(replicate(2000, prob_ra(p =  c(1.2, .8, 2.1, .9))), 1, mean),2)
```

```
## [1] 1.22 0.79 2.10 0.90
```

6 Thoughts on integration with `randomizr`

6.1 Function to select

One could combine with randomizr by employing a general function that either determines whether balancing should be used for optimality or lets the user decide. Here is an example (not very general, just for the 50% probability assignment default).

```
randomize <- function(b, between_block = NULL){
  if(is.null(between_block)) {between_block <- sum(table(b)%%2)>0; print(paste("betwe
en_block set to", between_block))}
  if(between_block)  out <- prob_ra(b = b)
  if(!between_block) out <- block_ra(block_var = b)
  out
}
```

Three applications:

```
set.seed(1)
randomize(b=rep(1:2, each = 3), between_block = NULL)
```

```
## [1] "between_block set to TRUE"
```

```
## [1] 0 1 1 1 0 0
```

```
set.seed(1)
randomize(b=rep(1:2, each = 3), between_block = TRUE)
```

```
## [1] 0 1 1 1 0 0
```

```
set.seed(1)
randomize(b=rep(1:2, each = 3), between_block = FALSE)
```

```
## [1] 1 1 0 1 1 0
```

6.2 Speed:

This just does time on a single instance of a big many block problem.

```
record <- function(f){
 start.time <- Sys.time()
 f
 Sys.time()  - start.time
}
```

6.2.1 Easy case — defaults to `randomizr`

```
b <- rep(1:50000, each = 4)
easy <- c(
  null = record(randomize(b=b, between_block = NULL)),
  prob_ra = record(randomize(b=b, between_block = TRUE)),
  randomizr_A = record(randomize(b=b, between_block = FALSE)),
  randomizr_B = record(block_ra(block_var = b))
 )
```

```
## [1] "between_block set to FALSE"
```

6.2.2 Hard case — defaults to `prob_ra`

```
b <- rep(1:50000, each = 3)
hard <- c(
  null = record(randomize(b=b, between_block = NULL)),
  prob_ra = record(randomize(b=b, between_block = TRUE)),
  randomizr_A = record(randomize(b=b, between_block = FALSE)),
  randomizr_B = record(block_ra(block_var = b))
 )
```

```
## [1] "between_block set to TRUE"
```

```
kable(cbind(easy, hard))
```

	easy	hard
null	1.349885	25.072606
prob_ra	35.158895	23.158884
randomizr_A	1.134067	2.324931
randomizr_B	1.068285	2.323533

So `randomizr` is a *lot* faster and is rightly selected for the problems that don't require balancing (easy above). For hard problems there is a speed / efficiency trade off which users can decide on. The wrapper doesn't slow things down. A tricky feature for the speed/efficiency tradeoff is that the speed gains are very important for the simulations where they are done many times but not so important for an actual assignment decision.