In native R, the user sets the seed for random number generation (RNG) with set.seed(). Random number generators exist in C and C++ too; these need their own seeds, which are not obviously settable by set.seed(). Good news! It can be done.

pacman::p\_load(inline, purrr)

**rbernoulli**

Base R (or technically the stats package) provides no rbernoulli(). It’s a pretty gaping hole in the pantheon of rbeta(), rbinom(), rcauchy, rchisq(), rexp(), rf(), rgamma(), etc. Thankfully, Hadley Wickham noticed this and gave us purrr::rbernoulli().

set.seed(1)

rbernoulli(5, 0.7)

#> [1] FALSE TRUE TRUE TRUE FALSE

set.seed(1)

rbernoulli(5, 0.7)

#> [1] FALSE TRUE TRUE TRUE FALSE

So it seems like Hadley managed to get set.seed() to work with rbernoulli(). How did he do this? Let’s take a closer look at purrr::rbernoulli().

purrr::rbernoulli

#> function (n, p = 0.5)

#> {

#> stats::runif(n) > (1 - p)

#> }

#>

#>

Ah, it seems Hadley just wrapped runif(); hence, because set.seed() works with runif(), it works with his implementation of purrr::rbernoulli().

**C++ RNG**

The C++ standard library provides the header file, which includes Bernoulli RNG. Let’s give that a whirl.

cpp\_rbernoulli <- rcpp(c(n = "integer", p = "numeric", seed = "integer"), '

int n\_ = as(n), seed\_ = as(seed);

double p\_ = as(p);

std::default\_random\_engine generator(seed\_);

std::bernoulli\_distribution distribution(p\_);

IntegerVector out(n\_);

for (std::size\_t i = 0; i != n\_; ++i) {

out[i] = distribution(generator);

}

return out;

', includes = "#include ")

cpp\_rbernoulli(6, 0.7, seed = 1)

#> [1] 1 0 1 1 0 1

cpp\_rbernoulli(6, 0.7, seed = 1)

#> [1] 1 0 1 1 0 1

cpp\_rbernoulli(6, 0.7, seed = 2)

#> [1] 1 0 1 0 1 1

OK, so now we have cpp\_rbernoulli() which is working, but the user has to pass the seed as an argument of the function, there’s no option to use set.seed().

**get\_seed()**

If only there was a get\_seed() function that we could use. Well, here it is!

get\_seed <- function() {

sample.int(.Machine$integer.max, 1)

}

This gets a positive number in the unsigned 32-bit integer range (which is always a safe bet for a seed) and it is completely determined by set.seed(). Therefore, it’s fine to use as a seed itself. Let’s take a look.

set.seed(1)

replicate(6, get\_seed())

#> [1] 570175513 799129990 1230193230 1950361378 433108649 1929277158

set.seed(1)

replicate(6, get\_seed())

#> [1] 570175513 799129990 1230193230 1950361378 433108649 1929277158

set.seed(2)

replicate(6, get\_seed())

#> [1] 397031630 1508336757 1231208929 360888751 2026879546 2026097046

set.seed(2)

replicate(6, get\_seed())

#> [1] 397031630 1508336757 1231208929 360888751 2026879546 2026097046

So as we can see, setting a seed via set.seed() determines the seeds that subsequently come out of get\_seed(), so all is well with the world. get\_seed() can now be used to create a version of cpp\_rbernoulli() which uses set.seed(). For the sake of inflating my own ego, I’ll name this version after myself.

rorybernoulli <- function(n, p) {

cpp\_rbernoulli(n, p, get\_seed())

}

Let’s check that it’s in working order.

set.seed(1)

rorybernoulli(6, 0.7)

#> [1] 1 1 0 0 1 0

set.seed(1)

rorybernoulli(6, 0.7)

#> [1] 1 1 0 0 1 0

set.seed(2)

rorybernoulli(6, 0.7)

#> [1] 0 1 1 1 1 1

set.seed(2)

rorybernoulli(6, 0.7)

#> [1] 0 1 1 1 1 1

Everything is awesome.

**Benchmarking**

Lastly, let’s compare the two Bernoulli RNGs that we have now by asking them both to give us a million Bernoulli random numbers with p = 0.5.

bench::mark(purrr::rbernoulli(1e6, p = 0.5),

rorybernoulli(1e6, p = 0.5),

check = FALSE)

#> # A tibble: 2 x 10

#> expression min mean median max `itr/sec` mem\_alloc n\_gc n\_itr

#>

#> 1 purrr::rb… 26.9ms 29.45ms 28.57ms 32.7ms 34.0 11.45MB 4 13

#> 2 roryberno… 8.55ms 9.61ms 9.38ms 12.7ms 104. 3.82MB 4 46

#> # ... with 1 more variable: total\_time

Wow, rorybernoulli() is three times faster! I wasn’t expecting that. Perhaps it’s because there’s a quicker way of generating a Bernoulli random number than by going through a uniform random number (as purrr::rbernoulli() does). It’s also three times as efficient with memory, probably related to the time speedup. The point of me writing this post was to share this get\_seed() thing with people so that the can use set.seed() with Rcpp and the like; purrr::rbernoulli() was just a cool example of a non-base RNG that popped into my head. Maybe I should submit a pull request to purrr!