Generating Unif(0,1) Numbers

Calvin Jouard and Josh Brown

2022-07-25

Code For Project

```
rm(list=ls())
library(magrittr)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x tidyr::extract()   masks magrittr::extract()
## x dplyr::filter()    masks stats::filter()
## x dplyr::lag()       masks stats::lag()
## x purrr::set_names() masks magrittr::set_names()
```

```
library(ggplot2)
library(stringr)
library(randtests)
library(spgs)
```

```
##
## Attaching package: 'spgs'

## The following object is masked from 'package:randtests':
##
##     rank.test
```

```
library(SyncRNG)
library(poolr)
library(microbenchmark)

#create data frame to write seed analysis to
df <- data.frame("SeedNum" = NA,
```

```r
                 "RNG_type" = NA,
                 "Runs Test (p-value)" = NA,
                 "Chi-Sqr. GOF (p-value)" = NA,
                 "Serial Correlation (stand. test statistic)" = NA)

df <- df[-1,]

### Functions providing analysis ###
# Serial/Autocorrelation test
act <- function(unif_vector) {

  n <- length(unif_vector)

  # configure vectors for serial correlation equation
  r_1 <- unif_vector[-1] # r+1 vector, everything but 1st entry
  r <- unif_vector[-n] # r vector, leaves off last entry

  # Calculate lag-1 correlation
  lag_1_corr <- 12/(n-1) * sum(r_1 * r) - 3

  # calculate variance of correlation distribution
  phat_var <- (13 * n - 19) / (n - 1)^2

  # calculate probability
  Z_corr <- lag_1_corr / sqrt(phat_var)

  # get standardized test statistic
  return (pnorm(Z_corr))
}

# Unifs/second timer
timer <- function(generator) {

  n <- 10000

  # run microbenchmark to time function
  result <- summary(microbenchmark(generator(), unit="s"))

  # extract mean time in 100 trialsa
  mean_seconds <- result$mean

  unifs_per <- n / mean_seconds

  return (unifs_per)
}

### RNG's and Seed analysis ###
# desert island generator function
desert_island <- function(n=10000, x_0=11){
  rns <- c()
  x_i <- NULL

  for (i in 1:n) {
    if (length(rns) == 0) {
      x_i <- (16807 * x_0) %% (2^31 - 1)

    } else {
      x_i <- (16807 * x_i) %% (2^31 - 1)
    }

    r_i <- x_i / (2^31 - 1)
    rns <- append(rns, r_i)
  }
  rns
}

##Desert Island Run
for (x in rep.int(1:1000, 1)) {
  di_nums <- desert_island(10000, x)
  RunsTests <- runs.test(di_nums)$p.value # returns p-value
  ChiSquaredGOF <- chisq.unif.test(di_nums, min.bin.size = 10)$p.value # results from chi-squared test
  SerialCorrelation <- act(di_nums) # returns p-value

  generator_stats <- c("SeedNum" = x,
                       "RNG_type" = "Desert Island",
                       "Runs Test (p-value)" = RunsTests,
                       "Chi-Sqr. GOF (p-value)" = ChiSquaredGOF,
                       "Serial Correlation (stand. test statistic)" = SerialCorrelation)

  df[nrow(df) + 1, ] <- generator_stats
}

RNGkind(kind = "Mersenne-Twister")
#Mersenne-Twister_RNG <- runif(10000)

##Mersenne-Twister RUN
for (x in rep.int(1:1000, 1)) {
  set.seed(x)
  Mersenne_Twister_RNG <- runif(10000)
  RunsTests <- runs.test(Mersenne_Twister_RNG)$p.value # returns p-value
  ChiSquaredGOF <- chisq.unif.test(Mersenne_Twister_RNG, min.bin.size = 10)$p.value # results from chi-
  SerialCorrelation <- act(Mersenne_Twister_RNG) # returns p-value

  generator_stats <- c("SeedNum" = x,
                       "RNG_type" = "Mersenne-Twister",
                       "Runs Test (p-value)" = RunsTests,
                       "Chi-Sqr. GOF (p-value)" = ChiSquaredGOF,
                       "Serial Correlation (stand. test statistic)" = SerialCorrelation)

  df[nrow(df) + 1, ] <- generator_stats
}

RNGkind(kind = "Wich")
#Wichmann_Hill_RNG <- runif(10000)

##Wichmann_Hill RUN
```

```r
for (x in rep.int(1:1000, 1)) {
  set.seed(x)
  Wichmann_Hill_RNG <- runif(10000)
  RunsTests <- runs.test(Wichmann_Hill_RNG)$p.value # returns p-value
  ChiSquaredGOF <- chisq.unif.test(Wichmann_Hill_RNG, min.bin.size = 10)$p.value # results from chi-squ
  SerialCorrelation <- act(Wichmann_Hill_RNG) # returns p-value

  generator_stats <- c("SeedNum" = x,
                       "RNG_type" = "Wichmann-Hill",
                       "Runs Test (p-value)" = RunsTests,
                       "Chi-Sqr. GOF (p-value)" = ChiSquaredGOF,
                       "Serial Correlation (stand. test statistic)" = SerialCorrelation)

  df[nrow(df) + 1, ] <- generator_stats
}

#TAUSWORTHE RUN
tausworthe <- function(num_unifs=10000, seed=12) {

  rng <- SyncRNG(seed=seed) #initialize generator object

  rns <- c() #initialize list that will hold unif numbers

  for (i in 1:num_unifs) {
    rns <- append(rns, rng$rand())
  }
  rns # return list of unifs
}

for (x in rep.int(1:1000, 1)) {
  Tausworthe_RNG <- tausworthe(seed = x)
  RunsTests <- runs.test(Tausworthe_RNG)$p.value # returns p-value
  ChiSquaredGOF <- chisq.unif.test(Tausworthe_RNG, min.bin.size = 10)$p.value # results from chi-square
  SerialCorrelation <- act(Tausworthe_RNG) # returns p-value

  generator_stats <- c("SeedNum" = x,
                       "RNG_type" = "Tausworthe LSFR",
                       "Runs Test (p-value)" = RunsTests,
                       "Chi-Sqr. GOF (p-value)" = ChiSquaredGOF,
                       "Serial Correlation (stand. test statistic)" = SerialCorrelation)

  df[nrow(df) + 1, ] <- generator_stats
}

# RANDU generator function and RUN
randu <- function(n=10000, x_0=11){
  rns <- c()
  x_i <- NULL

  for (i in 1:n) {
    if (length(rns) == 0) {
      x_i <- (65539 * x_0) %% 2^31
    } else {
      x_i <- (65539 * x_i) %% 2^31
    }

    r_i <- x_i / 2^31
    rns <- append(rns, r_i)
  }
  rns
}

for (x in rep.int(1:1000, 1)) {
  RANDU_RNG <- randu(x_0 = x)
  RunsTests <- runs.test(RANDU_RNG)$p.value # returns p-value
  ChiSquaredGOF <- chisq.unif.test(RANDU_RNG, min.bin.size = 10)$p.value # results from chi-squared tes
  SerialCorrelation <- act(RANDU_RNG) # returns p-value

  generator_stats <- c("SeedNum" = x,
                       "RNG_type" = "RANDU",
                       "Runs Test (p-value)" = as.numeric(RunsTests),
                       "Chi-Sqr. GOF (p-value)" = as.numeric(ChiSquaredGOF),
                       "Serial Correlation (stand. test statistic)" = as.numeric(SerialCorrelation))

  df[nrow(df) + 1, ] <- generator_stats
}

df_withmutate <- df

df_withmutate %<>% mutate("Runs_Test_Uniformity" = case_when(Runs.Test..p.value. <= .05 ~ "Reject", TRUI
                          "Chi_Squared_Uniformity" = case_when(Chi.Sqr..GOF..p.value. <= .05 ~ "Reject"
                          "Serial_Correlation_Uniformity" = case_when(Serial.Correlation..stand..test.s

df_withmutate$Runs.Test..p.value. <- as.numeric(df_withmutate$Runs.Test..p.value.)
df_withmutate$Chi.Sqr..GOF..p.value. <- as.numeric(df_withmutate$Chi.Sqr..GOF..p.value.)
df_withmutate$Serial.Correlation..stand..test.statistic. <- as.numeric(df_withmutate$Serial.Correlation


### Aggregate data and generate dataframe with stats per generator ###
overall_stats <- data.frame(Test = c("Runs Test (p-value)",
                                     "Chi-Squared Test (p-value)",
                                     "Serial Correlation (p-value)"))

for (generator in unique(df$RNG_type)) {
  my_cols <- df[df$RNG_type == generator,]

  runs_p <- round(fisher(as.numeric(my_cols$Runs.Test..p.value.))$p, 3)

  chisq_p <- round(fisher(as.numeric(my_cols$Chi.Sqr..GOF..p.value.))$p, 3)

  serial_p <- round(fisher(as.numeric(my_cols$Serial.Correlation..stand..test.statistic.))$p, 3)

  agg_gen_stats <- data.frame(gen_name = c(runs_p, chisq_p, serial_p))

  overall_stats <- cbind(overall_stats, agg_gen_stats)
}

colnames(overall_stats) <- append(c("Test"), unique(df$RNG_type))

# run microbenchmark to time mersenn-twister
RNGkind(kind = "Mersenne-Twister")
mt_unifs_per <- round(10000 / summary(microbenchmark(runif(10000), unit="s"))$mean)

# run microbenchmark to time wichman-hill
RNGkind(kind = "Wich")
wh_unifs_per <- round(10000 / summary(microbenchmark(runif(10000), unit="s"))$mean)

# time the rest of the generators
di_unifs_per <- round(timer(desert_island))
tw_unifs_per <- round(timer(tausworthe))
ru_unifs_per <- round(timer(randu))
```

Abstract

For Project Topic #17, we decided on selecting 5 UNIF generators to determine if we could find good/bad generators. We selected some ones Dr. Goldman described as bad and some ones Dr. Goldman described as good. We look into the Desert Island generator, a Tausworthe generator, the RANDU generator, the Mersenne-Twister generator, and the Wichmann-Hill generator. To test for independence and identically distributed uniforms from these RNGs, we run a χ^2 goodness-of-fit test, the von Neumann test for independence, and a runs tests. We will also compare the models based on run-time efficiency for speed of RNG generation of each RNG.

Background and Description of Problem

The main area of this problem derives from two places; not all random number generators are good and not all random number generators are equally good. You want your random number generator to be quick, show independence, and show a good fit/spread of numbers. We'll talk below about the process of picking the RNGs, some tidbits about what we learned about them in class, and then we'll discuss our workflow for building out the RNGs we needed to build.

Main Findings

From our Week 6 Homework, we chose the 5 RNGs that were talked about in class, plus some of the built in RNGs that fall in R. We coded up the Desert Island & RANDU generators, which were shared as bad RNGs, and found built in packages for the Tausworthe, Mersenne-Twister, and Wichmann-Hill RNGs. Both the Mersenne-Twister and Wichmann-Hill generators were not discussed in class, but were products of the native `runif` function which we discovered while looking at certain way to code this in R. Due to them being local in R and not having to build functions, the "pre-built" functions do run quicker.

Our process followed this general outline:

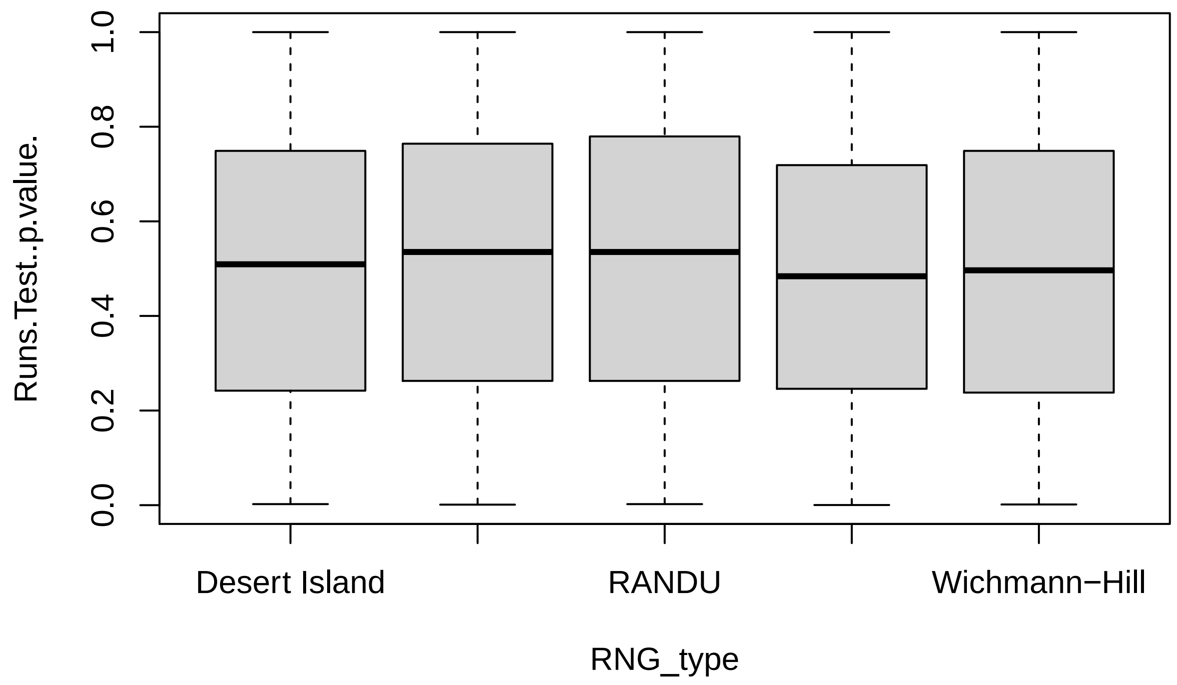
1. We tested each generator with 1,000 different seeds (ranging from 1 to 1000) to try different seeds per generator.

2. For each seed, we collected results from the Runs Test, Chi-Squared Goodness-of-Fit test and Serial Correlation test, tests to determine fit and independence.
3. We then extracted a combined p-value for each test via the Fisher method, using the native `fisher()` function to look at the p-values as a whole rather than just averaging them.
4. The run-time efficiency for each generator was then tested using a random seed. Each generator was forced to produce 10,000 uniform numbers 100 times via the `microbenchmark` package and function.

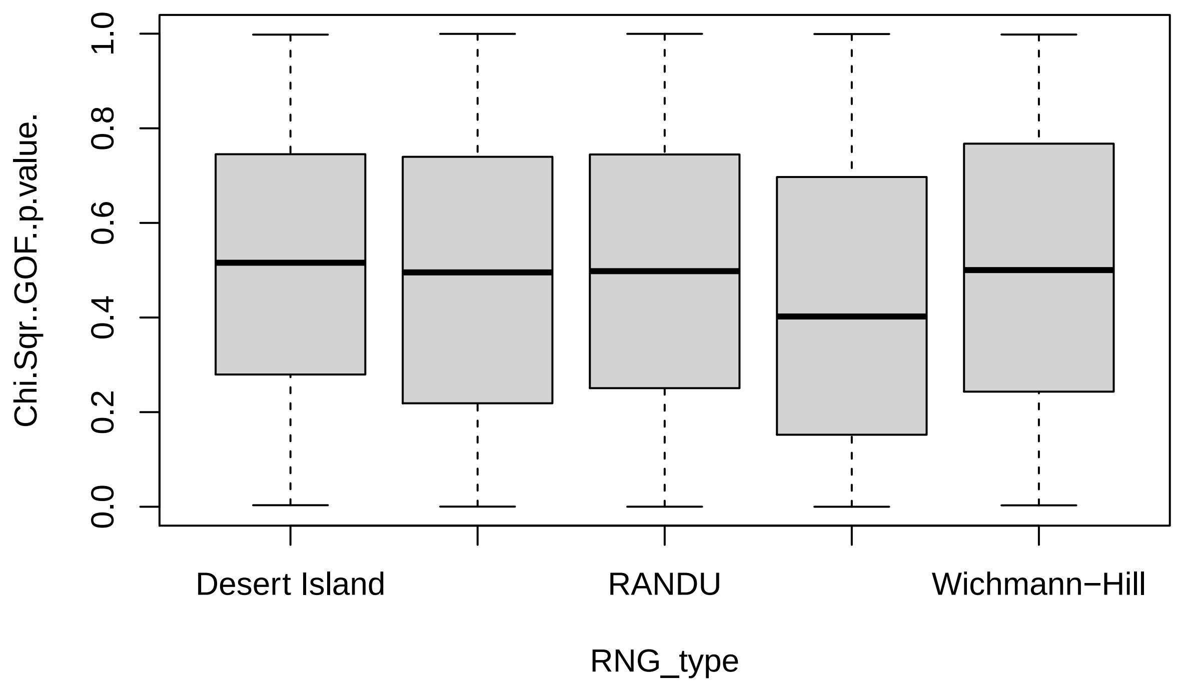
An α level of 0.05 was used for all tests. The null hypothesis for the Runs Test and Chi-Squared test would be rejected if $p \leq 0.05$, while the null hypothesis of independence for Serial correlation was rejected only if $|p| \leq 0.025$

Listed below are each of the individual RNGs with comments of analysis.

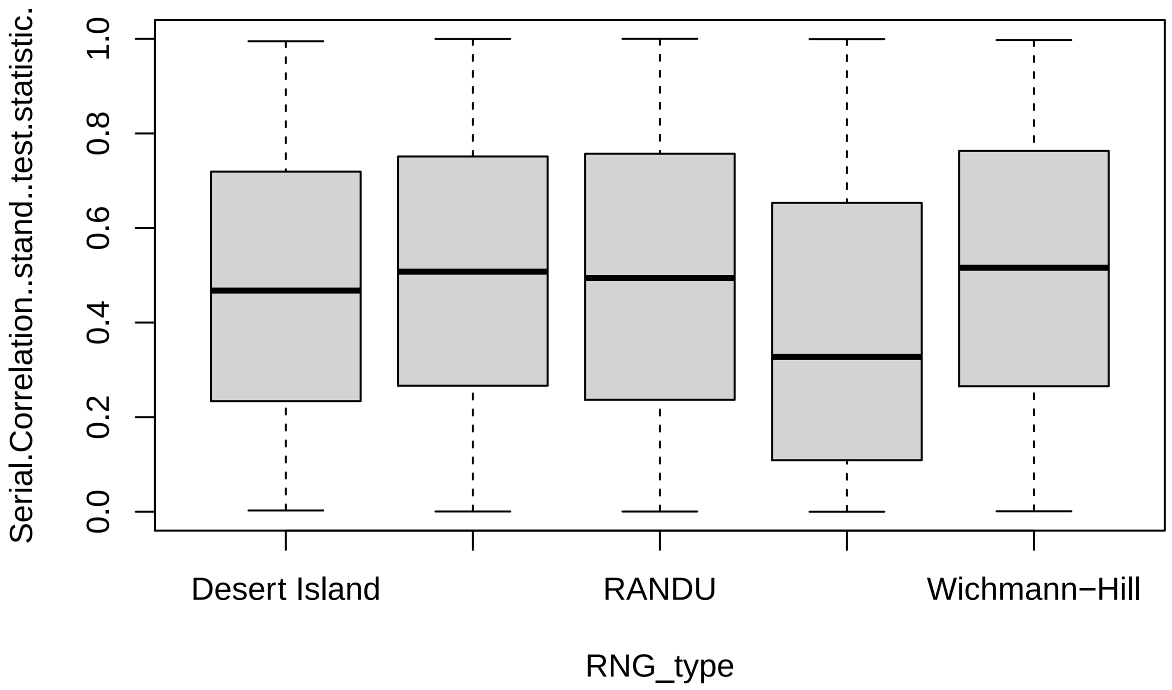
```r
boxplot(Runs.Test..p.value. ~ RNG_type, data = df_withmutate)
```



```r
boxplot(Chi.Sqr..GOF..p.value. ~ RNG_type, data = df_withmutate)
```



```r
boxplot(Serial.Correlation..stand..test.statistic. ~ RNG_type, data = df_withmutate)
```



```r
overall_stats[nrow(overall_stats) + 1,] <- c("Nums generated per sec.",
                                             di_unifs_per, mt_unifs_per,
                                             wh_unifs_per,
                                             tw_unifs_per,
                                             ru_unifs_per)

print(overall_stats)
```

```
##                             Test Desert Island Mersenne-Twister Wichmann-Hill
## 1             Runs Test (p-value)         0.491            0.915         0.221
## 2      Chi-Squared Test (p-value)         0.933            0.062         0.637
## 3 Serial Correlation (p-value)            0.106            0.857         0.683
## 4         Nums generated per sec.         72824         34485969      32665327
##   Tausworthe LSFR RANDU
## 1           0.169 0.959
## 2               0 0.526
## 3               0 0.259
## 4           15818 59256
```

Desert Island Analysis

With the Desert Island generator, we can comfortably accept the null hypothesis of independence when it comes to the runs test, given the p-value of 0.49. The same can be said for the Chi-Squared test–we can safely accept the null hypothesis of uniformity. We've also conducted an additional independence test via the

Serial Correlation test, which measures independence between contiguous observations. We can also accept independence regarding this test, since probability is greater than our α value divided by 2 (0.025). That said, the serial correlation p-value is much less strong. In terms of the output per second, this generator comes in at 4th among the five we tested. It's ahead of Tausworthe, but pales in comparison to the native R generators in the Mersenne-Twister and Wichmann-Hill.

Tausworthe Analysis

The Tausworthe generator is easily the 'worst' of the bunch. It did pass the runs test for independence, but rounding p-values to 3 decimal places meant that our aggregate p-value from the Chi-Squared and Serial Correlation tests went to 0. Given these results, we reject the null hypotheses of independence and uniformity. To top it off, the Tausworthe implementation used generated the least amount of random uniform numbers per second, with ~13,000.

RANDU Analysis

Aggregate p-values for tests applied to RANDU indicated uniformity and independence ie. a failure to reject the null hypotheses. The implementation used produced the 3rd most uniform numbers per second, behind only the native R generators Mersenne-Twister and Wichmann-Hill. This performance is obviously a bit misleading though, given that when we plot contiguous triplets of Tausworthe-generated numbers we end up with 15 hyperplanes. This pattern suggests a lack of randomness to this generator. The discrepancy between this fact and our results calls into question the efficacy of the above tests. Had time permitted, similar graphical and visual analysis in detecting hyperplanes would have been useful in analyzing all generators.

Mersenne-Twister Analysis

The Mersenne-Twister RNG passed all 3 of our tests. The Chi-Squared test aggregate variable was very low and close to Reject Null Hypothesis level, but it fails to reject. With some more seed analysis, we could probably look more into this test to see if it is actually a good fit. The number of random numbers generated was the best and showed very strong values for no correlation and Runs Test. A fun fact is the Mersenne-Twister is the default RNG if you use 'runif' in R! The Mersenne-Twister has a cycle length of $2^{19937} - 1$.

Wichmann-Hill Analysis

The Wichmann-Hill RNG also passed all 3 of our tests and showed a very strong case for being very fast, much like the built-in counterpart RNG, the Mersenne-Twister RNG. Listed in the details of the Help section for ?Random, the Wichmann-Hill generator has a cycle length of 6.9536e12 which is seen as a very good cycle length! Looking at the boxplots we created, Wichmann appears to have better values across the GOF and Independence tests than the Mersenne-Twister RNG The Wichmann-Hill RNG is easy to switch to as well in R, only needing to type RNGkind(kind = "Wich") before calling for a random number generator.

Conclusion

Overall, we enjoyed this project to learn about how to derive random numbers from code and which generators to choose. We see the built-in packages in R are fast and test soundly as good RNGs, which is what we would suggest to anyone looking to code in R that needs to use RNGs. If we had more time, we definitely would try to code some harder generators and do some more seed analysis, as there could be some bias in our analysis from selecting just the first 1000 seeds.