

EST-24107: Simulación

Profesor: Alfredo Garbuno Iñigo — Otoño, 2023 — Números aleatorios.

Objetivo: En esta sección veremos conceptos básicos del curso. Tanto para el trabajo computacional que realizaremos y la entrega de tareas, como las nociones básicas de generación de números aleatorios en una computadora.

Lectura recomendada: Capítulo 3 de [2]. Capítulo 2 de [1].

1. Agenda

Veremos un poco sobre el ambiente de trabajo. Veremos :

1. nociones básicas `Git` y `GitHub`;
2. familiaridad con `R`.
3. generación de variables aleatorias.

2. Control de versiones

Los *softwares* de control de versiones nos permiten llevar un registro y administración de cambios en archivos. Usualmente para proyectos de programación.

Ayudan a trabajar colaborativamente en ambientes de equipos de trabajo.

Aunque no exploraremos *todo* lo que se puede hacer con `Git` y `GitHub` lo usaremos para llevar un control del desarrollo y de entrega de tareas. Usaremos los principios mas básicos.

3. R statistical programming language

`R` es un lenguaje de programación orientado a cómputo estadístico y generación de gráficos estadísticos. Está escrito para interactuar por medio de ejecución de *scripts* (archivos de texto con instrucciones) o la consola interactiva. Ver Figura 1.

Es usual utilizar un ambiente de desarrollo para programar e interactuar con el lenguaje. Para `R` el mas común es `Rstudio` el cual tiene además algunas extensiones útiles para el desarrollo de análisis estadístico. Ver Figura 2.

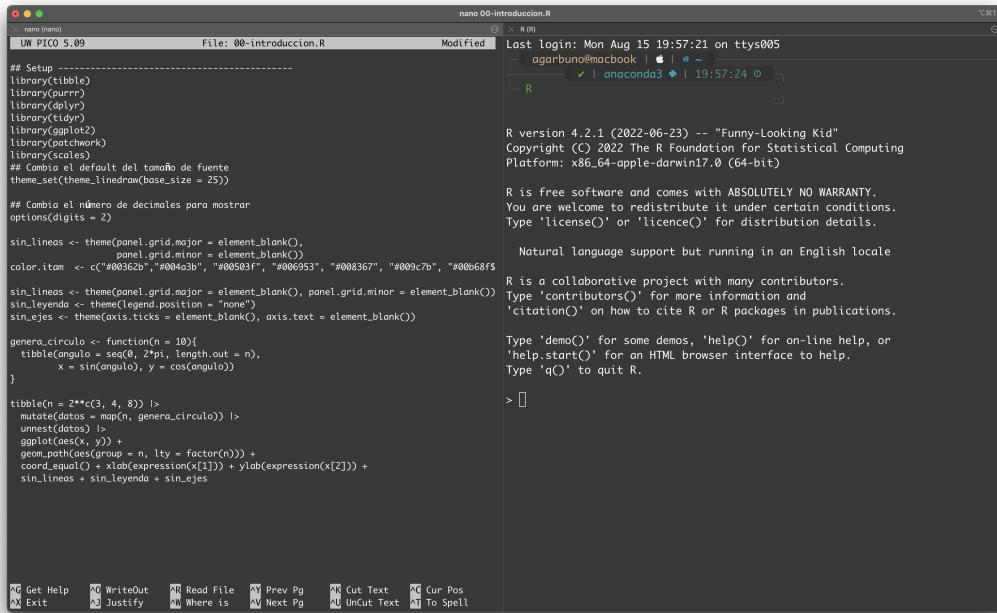
`Visual Studio Code` es una alternativa multi-lenguaje para desarrollar proyectos de análisis estadístico en `R`. Ver Figura 3.

Y habemos los que nos *conformamos* con un buen editor de texto como ambiente de desarrollo. Ver Figura 4.

4. Números aleatorios

Es posible, que cuando pensamos en generar números aleatorios, idealizamos con lanzar una moneda, un dado, una baraja o una rueda giratoria estilo *Jeopardy!*.

4 Números aleatorios



The screenshot shows two windows side-by-side. The left window is a terminal session for R version 4.2.1, displaying the standard welcome message and help information. The right window is a terminal session for nano (a text editor), showing a script named '00-introduccion.R'. The script contains R code for generating a plot of sine waves. The code includes various library imports, theme settings, and a main function 'genera_circulo' that generates a plot of sine waves over a unit circle.

```

## Setup
library(cgrid)
library(courr)
library(dplyr)
library(tidyverse)
library(ggplot2)
library(patchwork)
library(scales)
## Cambia el default del tamaño de fuente
theme_set(theme_linedraw(base_size = 25))

## Cambia el número de decimales para mostrar
options(digits = 2)

sin_lineas <- theme(panel.grid.major = element_blank(),
                     panel.grid.minor = element_blank())
color_item <- c("#003620", "#0043b", "#006953", "#008367", "#009c7b", "#0068f5")

sin_lineas <- theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank())
sin_leyenda <- theme(legend.position = "none")
sin_ejes <- theme(axis.ticks = element_blank(), axis.text = element_blank())

genera_circulo <- function(n = 10){
  tibble(angulo = seq(0, 2*pi, length.out = n),
        x = sin(angulo), y = cos(angulo))
}

tibble(f = 2*pi*(0, 4, 8)) %>
  mutate(datos = map(f, genera_circulo)) %>
  unnest(datos) %>
  ggplot(aes(x, y)) +
  geom_path(data=nes$group = n, lty = factor(n))) +
  coord_equal() + xlab(expression(x[1])) + ylab(expression(x[2])) +
  sin_lineas + sin_leyenda + sin_ejes
}

tibble(f = 2*pi*(0, 4, 8)) %>
  mutate(datos = map(f, genera_circulo)) %>
  unnest(datos) %>
  ggplot(aes(x, y)) +
  geom_path(data=nes$group = n, lty = factor(n))) +
  coord_equal() + xlab(expression(x[1])) + ylab(expression(x[2])) +
  sin_lineas + sin_leyenda + sin_ejes
}

```

FIGURA 1. Dos ventanas, un editor de texto y una consola de R.

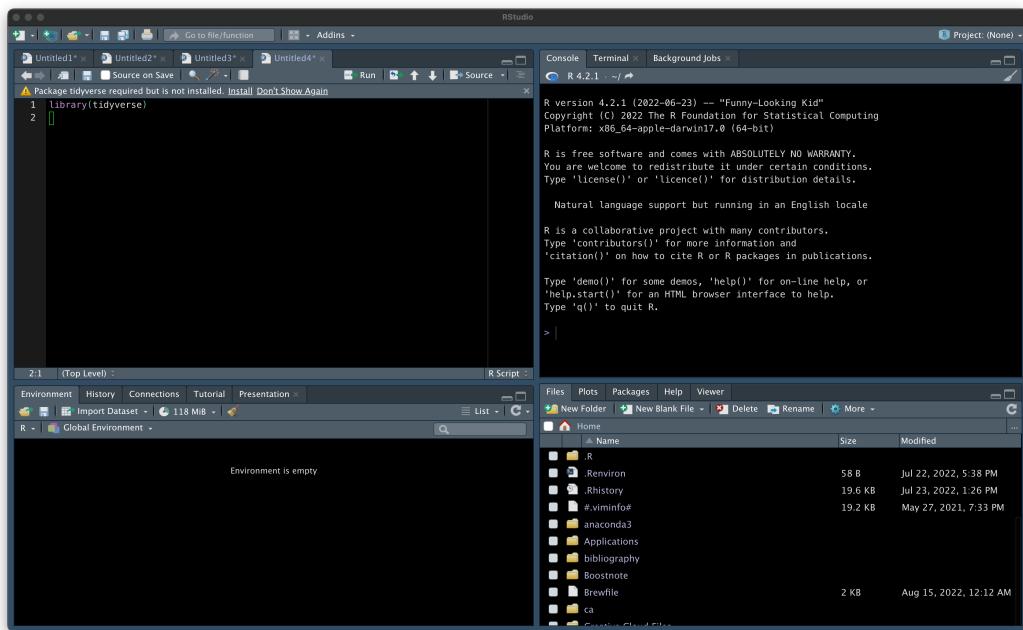


FIGURA 2. Un ambiente de desarrollo, Rstudio.

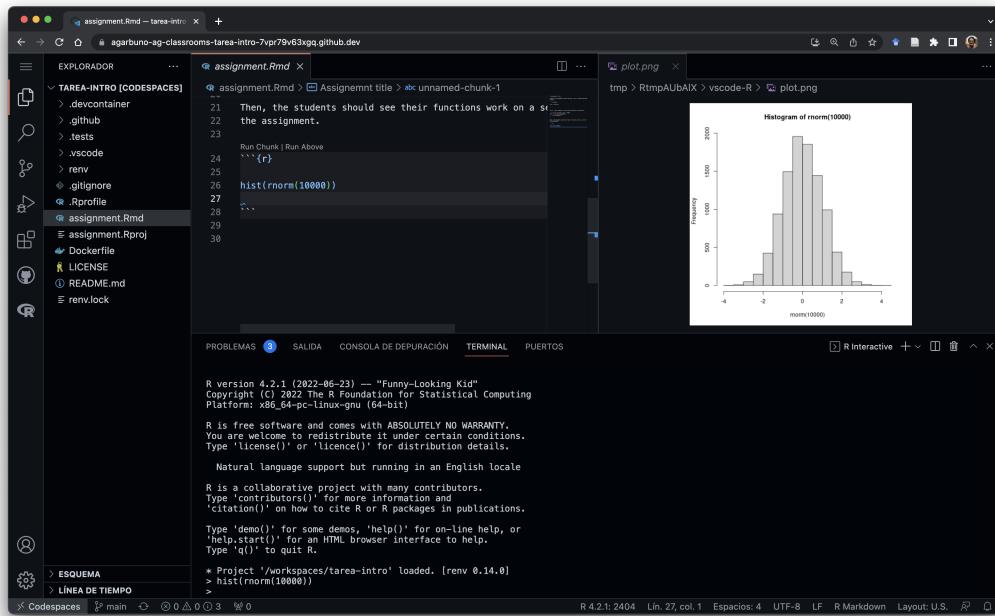


FIGURA 3. Un ambiente de desarrollo general, *Visual Code Studio*. En la imagen se muestra una sesión en un explorador de internet.

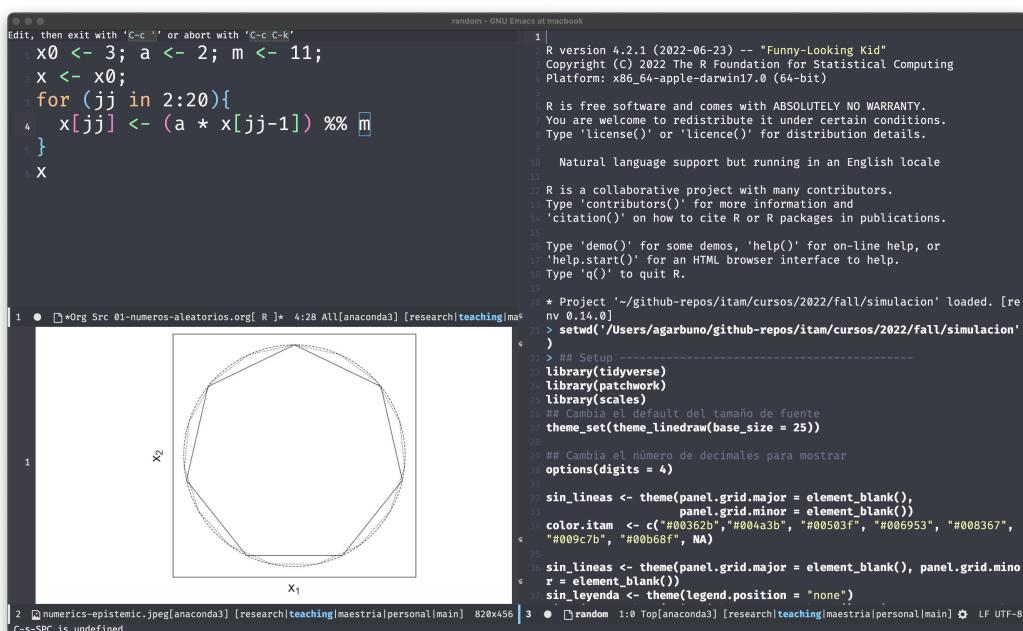
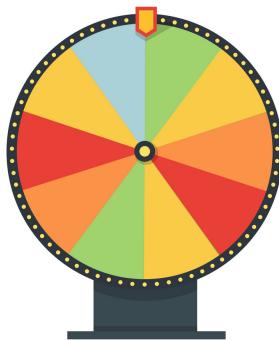


FIGURA 4. Ambiente de desarrollo basado en *Emacs*.



VectorStock® VectorStock.com/12245422

En nuestra computadora, los **números pseudo-aleatorios** son secuencias generadas de manera determinista de manera que *parecen* ser variables aleatorias uniformes independientes. Es decir, parecen ser

$$x_i \stackrel{\text{iid}}{\sim} U(0, 1). \quad (1)$$

El procedimiento mas común es utilizar una semilla x_0 y calcular recursivamente valores x_n con $n \geq 1$ por medio de

$$x_n = ax_{n-1} \pmod{m}, \quad (2)$$

donde a y m son enteros positivos.

Métodos que utilizan procedimientos similares se llaman **generadores congruenciales lineales**.

Nota que x_n es un valor entre $0, 1, \dots, m - 1$. Llamamos a la cantidad x_n/m un número pseudo-aleatorio uniforme. Esto nos da un valor en el intervalo $(0, 1)$.

Las constantes a y m se escogen de tal forma que:

1. Para cualquier punto inicial, la secuencia *parezca* ser un secuencia de números aleatorios uniformes.
2. Para cualquier punto inicial, el tiempo estimado para ver una repetición sea muy largo.
3. Se pueda calcular la secuencia eficientemente.

La constante m está asociada al periodo de la secuencia y depende del valor de a para garantizar que se alcanza (**periodo completo**). Por ejemplo, podemos utilizar

$$x_n = 3x_{n-1} \pmod{5}, \quad (3)$$

para generar la secuencia a partir de $x_0 = 3$.

```

1 x0 <- 3; a <- 3; m <- 5;
2 x <- x0;
3 for (jj in 2:10){
4   x[jj] <- (a * x[jj-1]) %% m
5 }
6 x

```

```

1 [1] 3 4 2 1 3 4 2 1 3 4

```

Si cambiamos los valores podemos conseguir un periodo mas largo y por lo tanto un mayor colección de números aleatorios.

```

1 x0 ← 3; a ← 2; m ← 11;
2 x ← x0;
3 for (jj in 2:20){
4   x[jj] ← (a * x[jj-1]) %% m
5 }
6 x

```

```

1 [1] 3 6 1 2 4 8 5 10 9 7 3 6 1 2 4 8 5 10 9 7

```

```

1 u ← x[1:(m-1)] / m
2 u

```

```

1 [1] 0.27273 0.54545 0.09091 0.18182 0.36364 0.72727 0.45455 0.90909 0.81818
     0.63636

```

Usualmente m se escoge como un **número primo** de longitud igual al máximo número representable en una computadora.

Por ejemplo, en una máquina de 32-bits se ha visto que $m = 2^{31} - 1$ y $a = 7^5 = 16,807$ funcionan bien.

Esta elección nos permite generar una gran densidad en el intervalo $(0, 1)$. ¿Por qué?

4.1. Aleatorios en lenguajes de programación

Los lenguajes de programación tienen funciones para generar números aleatorios. Por ejemplo, en **Matlab** el enfoque es cómputo numérico por lo tanto el generador de aleatorios uniformes es la opción estándar.

```

1 rand
1 0.875566919023124

```

El lenguaje de **python** es multi-propósito. Por lo tanto, no es una opción *natural* y se llaman módulos especializados para generar números aleatorios. El módulo para generar números aleatorios tiene cierto nivel de compatibilidad con otros lenguajes.

```

1 import numpy as np
2 np.random.random()
1 0.9820617713830841

```

Por último, **R** es un lenguaje que se originó en la comunidad estadística. Por lo tanto, la generación de números aleatorios requiere de la distribución de interés.

```
1 runif(1)
```

```
1 [1] 0.8014
```

4.2. Transformación de uniformes

Es natural considerar la generación de números aleatorios en el intervalo $[a, b]$:

```
1 runif(100, min = 7, max = 10)
```

```
1 [1] 9.286 9.048 8.911 9.859 7.916 7.920 9.439 9.622 8.246 8.244 7.902 7.878
2 [13] 7.183 8.307 9.108 8.722 9.362 8.579 7.768 7.476 7.094 7.044 9.368 9.498
3 [25] 8.931 8.322 8.872 9.668 7.170 8.253 8.487 9.462 9.806 8.967 9.830 8.391
4 [37] 8.056 7.042 7.899 8.021 9.580 8.690 9.086 7.447 9.765 7.560 9.530 9.146
5 [49] 9.000 7.353 8.293 9.743 8.181 7.550 8.636 9.807 7.536 7.317 9.299 7.965
6 [61] 9.591 7.963 8.752 9.045 9.170 9.009 7.347 7.335 9.810 7.066 7.398 9.235
7 [73] 9.664 8.769 9.089 9.171 7.732 8.858 7.685 7.202 9.073 9.853 9.047 9.387
8 [85] 8.046 8.176 7.220 8.025 9.501 8.183 9.928 8.396 8.588 8.326 8.094 7.745
9 [97] 7.649 9.783 8.565 9.753
```

Pregunta. ¿Cuál es la relación que existe entre $X \sim U(0, 1)$ y $Y \sim U(a, b)$?

4.3. Reproducibilidad

Hemos establecido que la generación de números *pseudo-aleatorios* es un procedimiento determinista. Si sabemos la semilla que generó la secuencia y el algoritmo que la genera, podemos generar dos secuencias idénticas. Por lo tanto, variables aleatorias completamente dependientes.

```
1 runif(5)
2 runif(5)
```

```
1 [1] 0.1432 0.5805 0.8893 0.4105 0.1474
2 [1] 0.7946 0.5606 0.1151 0.3609 0.7199
```

```
1 set.seed(108); runif(5)
2 set.seed(108); runif(5)
```

```
1 [1] 0.4551 0.4040 0.3513 0.6643 0.4635
2 [1] 0.4551 0.4040 0.3513 0.6643 0.4635
```

4.4. Aleatoriedad o pseudo-aleatoriedad

Consideremos una secuencia generada X_1, \dots, X_n . Entonces el conocimiento de X_n entonces no debería de dar información sobre X_{n+1} si no conocemos el generador.

La pseudo-aleatoriedad de nuestra secuencia es limitada. Pues dos muestras (X_1, \dots, X_n) y (Y_1, \dots, Y_n) que sean producidas por el mismo algoritmo no son independientes, ni idénticamente distribuidas o comparables en algún sentido probabilístico.

La validez de un generador se basa en una secuencia X_1, \dots, X_n con $n \rightarrow \infty$. No en una colección infinita de réplicas con longitud fija.

La distribución de esta colección de tuplas depende únicamente de la distribución de las semillas iniciales. Ver Capítulo 2 de [1].

Definición 4.1 (Generador pseudo-aleatorio). Decimos que un algoritmo es un **generador de números uniformes pseudo-aleatorios** si para algún valor inicial u_0 y la aplicación de una transformación $D : \mathbb{R} \rightarrow \mathbb{R}$ produce una secuencia

$$u_n = D^n(u_0) = \underbrace{(D \circ \dots \circ D)}_{n \text{ veces}}(u_0), \quad (4)$$

de valores en el intervalo $(0, 1)$. Además, el comportamiento de los valores (u_1, \dots, u_n) se comportan como si fueran una muestra iid de variables uniformes (V_1, \dots, V_n) .

4.5. Comportamiento uniforme

Para validar que el generador de pseudo-aleatorios es válido tendremos que comparar las muestras generadas contra una distribución de probabilidad uniforme. ¿Qué podemos hacer?

4.6. Ideas

Podemos comparar contra la distribución teórica que estamos generando. Esto es para tratar de garantizar estadísticamente que nuestra muestra se *ve* como una realización de números aleatorios uniformes.

5. Prueba de Kolmogorov-Smirnov

Para comparar una muestra de números aleatorios podemos utilizar la prueba Kolmogorov-Smirnov (KS). La idea es sencilla: contrastar la **función de acumulación empírica** (la estimada en nuestra muestra) contra la **función de acumulación de una uniforme** (la distribución teórica).

Definición 5.1 (Función de acumulación de una variable uniforme). Decimos $x \sim \text{Uniforme}(a, b)$ si su función de acumulación es:

$$\mathbb{P}_x(x) = \mathbb{P}(X \leq x) = 1\{x \in (a, b)\} \times \frac{b - x}{b - a}. \quad (5)$$

Definición 5.2 (Función de acumulación empírica). Dada una muestra aleatoria X_1, \dots, X_n

de variables con función de distribución \mathbb{P} , definimos

$$\hat{\mathbb{P}}_n(x) = \frac{\text{muestras menores o iguales a } x}{n}. \quad (6)$$

Observación. La función de acumulación empírica (EDF) la podemos definir a través de los **estadísticos de orden**.

Definición 5.3 (Estadísticos orden). Dada una muestra aleatoria X_1, \dots, X_n los estadísticos de orden se definen como el reordenamiento $X_{(1)} \leq \dots \leq X_{(n)}$, donde

$$X_{(1)} = \min\{X_1, \dots, X_n\}, \quad \dots \quad X_{(n)} = \max\{X_1, \dots, X_n\}. \quad (7)$$

Teorema 5.4 (CDF de estadísticos de orden). Sea $\hat{\mathbb{P}}_n$ la función de acumulación empírica para una muestra aleatoria de X_1, \dots, X_n de \mathbb{P} . Entonces:

$$\text{Prob}\left\{\hat{\mathbb{P}}_n = \frac{k}{n}\right\} = \binom{n}{k} \mathbb{P}(x)^k (1 - \mathbb{P}(x))^{n-k}. \quad (8)$$

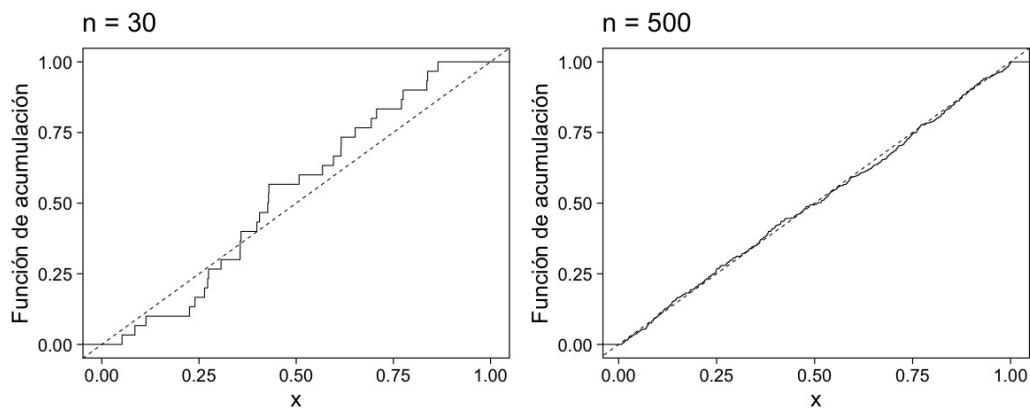
Proposición 5.5. El estimador $\hat{\mathbb{P}}_n$ es un **estimador insesgado puntual** y por el **teorema del límite central** tenemos que

$$\hat{\mathbb{P}}_n(x) \sim N\left(\mathbb{P}(x), \frac{\mathbb{P}_n(x)(1 - \mathbb{P}_n(x))}{n}\right). \quad (9)$$

Teorema 5.6 (Glivenko-Cantelli). El estimador $\hat{\mathbb{P}}_n(x)$ converge a $\mathbb{P}(x)$ de manera uniforme casi seguramente. Es decir,

$$\text{Prob}\left(\lim_{n \rightarrow \infty} \sup_{x \in \mathbb{R}} |\hat{\mathbb{P}}_n(x) - \mathbb{P}_n(x)| = 0\right) = 1. \quad (10)$$

5.1. Comparación



Por lo tanto, si medimos la **distancia máxima** entre la función de acumulación empírica y la teórica en el largo plazo la diferencia será 0.

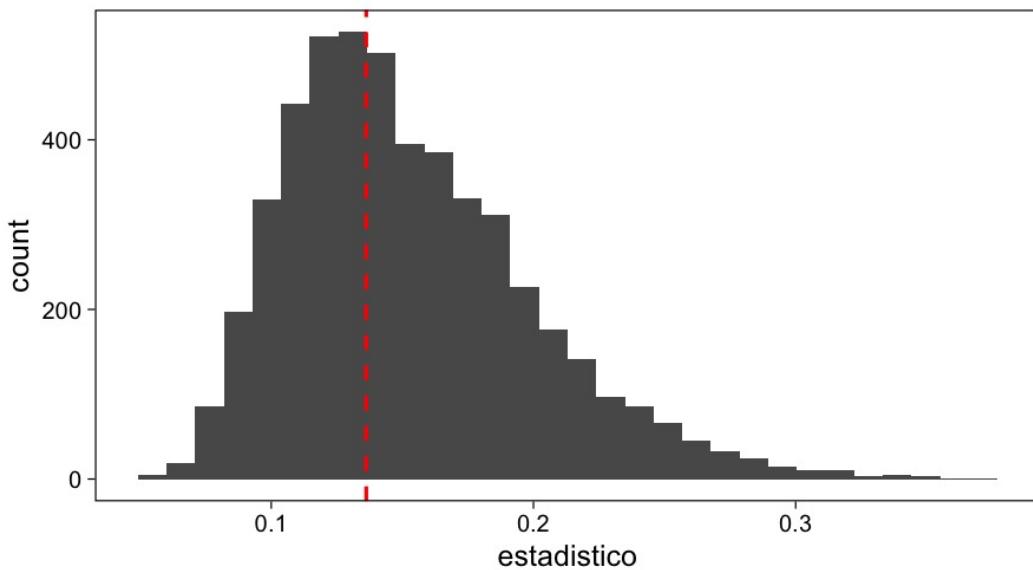
Para el panel de la izquierda, la distancia máxima es

```
1 Pn <- ecdf(samples$x)
2 x_ <- seq(0, 1, length = 1000)
3 Dn <- max(abs(Pn(x_) - punif(x_)))
4 print(paste("Distancia: ", Dn))
```

```
1 [1] "Distancia: 0.136236236236236"
```

Esta cantidad estimada de la muestra, D_n , depende precisamente de la muestra que generamos. ¿Qué tan extraño fue haber observado dicha muestra?

Si asumimos que los datos son generados por una uniforme observaríamos una distribución de posibles valores D_n como se muestra



Utilizando los datos que construyeron el histograma podemos calcular la probabilidad de haber observado un estadístico tan extremo. Es decir, la probabilidad de haber observado lo que observamos si el generador fuera el generador uniforme.

```
1 print(paste("Probabilidad: ", mean(replicas$estadistico ≥ Dn)))
```

```
1 [1] "Probabilidad: 0.577"
```

Este análisis se conoce como **prueba Kolmogorov-Smirnov**, y como vimos, sirve para detectar cuando una muestra aleatoria proviene de una distribución en particular. Como vamos empezando el curso, nos interesa saber si nuestro generador de datos es un buen generador de muestras uniformes.

```
1 ks.test(samples$x, "punif")
```

```
1
2      Exact one-sample Kolmogorov-Smirnov test
```

6 Documentación

```
3 data: samples$x
4 D = 0.14, p-value = 0.6
5 alternative hypothesis: two-sided
```

La prueba KS es una ejercicio estadístico típico, conocido como **contraste de hipótesis** donde ponemos a prueba

$$H_0 : \mathbb{P}(x) = \mathbb{P}_0(x) \quad \forall x \quad \text{contra} \quad H_1 : \mathbb{P}(x) \neq \mathbb{P}_0(x) \text{ para alguna } x. \quad (11)$$

6. Documentación

Puedes consultar la documentación de R, utilizando el comando `?Random` en la consola.

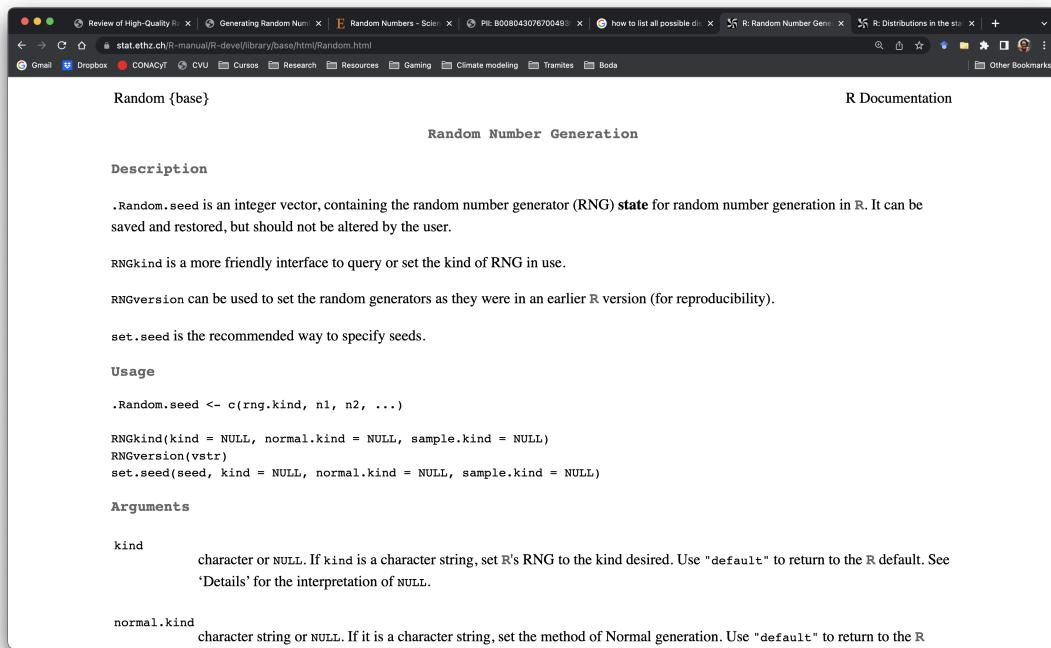


FIGURA 5. Documentación de *Random*.

El método *default* en muchos lenguajes de programación es el de **Mersenne-Twister** y en particular presenta un periodo de $2^{19937} - 1$.

```
1 sessionInfo()
```



```
1 R version 4.3.1 (2023-06-16)
2 Platform: x86_64-apple-darwin20 (64-bit)
3 Running under: macOS Ventura 13.4.1
4
5 Matrix products: default
6 BLAS: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/
    libRblas.0.dylib
7 LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/
    libRlapack.dylib; LAPACK version 3.11.0
```

```
8 locale:  
9 [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8  
10  
11 time zone: America/Mexico_City  
12 tzcode source: internal  
13  
14 attached base packages:  
15 [1] stats      graphics   grDevices datasets  utils      methods    base  
16  
17 other attached packages:  
18 [1] scales_1.2.1    patchwork_1.1.2 lubridate_1.9.2 forcats_1.0.0  
19 [5] stringr_1.5.0    dplyr_1.1.2     purrr_1.0.1    readr_2.1.4  
20 [9] tidyverse_2.0.0  tidyr_1.3.0     tibble_3.2.1    ggplot2_3.4.2  
21  
22 loaded via a namespace (and not attached):  
23 [1] vctrs_0.6.3      cli_3.6.1       rlang_1.1.1     stringi_1.7.12  
24 [5] renv_1.0.0       generics_0.1.3  labeling_0.4.2   glue_1.6.2  
25 [9] colorspace_2.1-0 hms_1.1.3      fansi_1.0.4     grid_4.3.1  
26 [13] munsell_0.5.0    tzdb_0.4.0      lifecycle_1.0.3 compiler_4.3.1  
27 [17] timechange_0.2.0 pkgconfig_2.0.3  farver_2.1.1    R6_2.5.1  
28 [21] tidyselect_1.2.0 utf8_1.2.3     pillar_1.9.0    magrittr_2.0.3  
29 [25] tools_4.3.1      withr_2.5.0     gtable_0.3.3  
30
```

Referencias

- [1] C. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer Science & Business Media, mar 2013.
[1](#), [7](#)
- [2] S. M. Ross. *Simulation*. 2013. [1](#)