Random Number and Sampling

Ivan Yi-Fan Chen

2023 Fall

Review of Probability

- Random: Events happen in an unpredictable manner / without obvious patterns.
- Examples:
    1. Dice tossing.
    2. Lottery number.
    3. Getting an SSR card.
    4. Brownian Motion.
- Probability: Numbers assigned to represent how likely an event takes place.
    - Bounded between 0 and 1.
    - Sum of probabilities of all events equals 1.
- Random variable: A random variable f is a function that maps each event x in the event space X to a real number $p \in \mathbb{R}$, $f : X \to \mathbb{R}$.
- Example: Tossing a fair dice once.
    - Event space: $X = \{x | x = 1, 2, 3, 4, 5, 6\}$.
    - Event: x is one of the numbers in $\{1, 2, 3, 4, 5, 6\}$.
    - Numbers assigned: $p_x = 1/6$ for each x.
- Discrete Random Variable
    - Events are discrete: $X = \{x_1, x_2, x_3, \dots\}$.
    - Numbers assigned: $f(x) \in [0, 1]$ and $\sum_{x \in X} f(x) = 1$.
    - The numbers $f(x)$ are **probability**.
- Continuous Random Variable
    - Events are continuous: $X = \{L \leq x \leq U\}$.
    - Numbers assigned: $f(x) \geq 0$ and $\int_{x \in X} f(x) dx = 1$.
    - The numbers are **probability density**, and the function that assigns a number to each event $f(x)$ is the **probability density function (pdf)**.
    - Cumulated Probability: $\Pr(L \leq x \leq \hat{X}) \equiv F(\hat{X}) = \int_L^{\hat{X}} f(x) dx$, which is increasing and is bounded in $[0, 1]$.

Generating “Random” Numbers

- We want to generate random numbers for the following purposes:
    - Drawing a sample, e.g., assigning seats and groups, determine control and treatment groups.
    - Resampling data, e.g., bootstrapping.
    - Numerical purposes, e.g., performing integration and optimization in Monte Carlo manner.
    - Simulation, e.g., simulating asset prices, simulating a distribution.
    - Encoding and encryption.
- Problems:
    - Difficult to characterize *true* randomness on computers.

    - Even if we can, it is nearly impossible to *replicate* as the probability to run into the same realization is 0.
- Pseudorandomness:
    - Behaves *as if* it is random.
    - In fact NOT random, but then it makes replication possible.
    - Most notable examples are probably computer games:
        - XCOM 2: https://www.youtube.com/watch?v=zAThQV-vj08 (https://www.youtube.com/watch?v=zAThQV-vj08)
        - SRW F: https://www.youtube.com/watch?v=TaQ8h89xt0c (https://www.youtube.com/watch?v=TaQ8h89xt0c)
        - Whether your shots connect are **pre-determined**, the probabilities of hits on your screen are just for show.
        - But how are the “probabilities” pre-determined? Depends on games: some are determined by the time you spent on the title screen, some are determined upon loading into the stages, some are determined upon you turned on your Playstation 5.
        - Thus we have the “Save & Load” and “Rebooting” black magics. Not just to avoid undesired outcome, but also to exploit the game mechanics, e.g., control the items dropped from mobs.
        - Anyway, as a player you won’t realize that those are in fact not random.
- Random Number Generators (RNGs):
    - Pseudorandom number generators actually…
    - An algorithm that generates a (very long) sequence of numbers given an initial input.
    - The initial input is what we refer to as a *random seed*.
    - Given the algorithm, the same random seed is guaranteed to generate the same sequence.
    - Illustrative example: given a seed, the algorithm generates 10 numbers.
        - Seed = 1 => generate {1, 3, 5, 10, 38, 50, 97, 24, 1, 6}
        - Seed = 2 => generate {2, 3, 543, 75, -6, 56, 8, 69, 132, 564}.
    - Suppose that we need 12 random numbers and we set the seed to 1. What are the 11th and 12th numbers? Can you predict the 53th random number if we need more?
    - Various algorithms are developed to improve the quality of “randomness” while maintaining reproducibility (and saving computational resources, too).
    - See Wikipedia: https://en.wikipedia.org/wiki/Pseudorandom_generator (https://en.wikipedia.org/wiki/Pseudorandom_generator)

Random Numbers on R

- Two ways to perform randomization: `sample()` and “Distributions”.

`sample()`

- `sample()` draws a randomized sample of a given size out from an object. Specifically,
    - `sample(x, size, replace)`
    - `x` is a vector that we want to draw our random sample.
    - `size` is a **positive integer** that determines the number of elements for the sample drawn.
    - `replace` is for replacement, and can be either `TRUE` or `FALSE`, or simply `T` or `F`. `TRUE / T` means that an element can be drawn multiple times.

```
#Simulating the Taiwan Lottery (draw 6 out of 49 numbers)
num<-c(1:49)
sample(x = num, size = 6, replace = F)
```

```
## [1] 24 41 12 46 22 27
```

```r
#Simulating tossing a coin for 10 times
sample(x = c("head","tail"), size = 10, replace = T)
```

```
##  [1] "head" "tail" "tail" "head" "head" "head" "head" "tail" "tail" "tail"
```

```r
#Replacement matters. You will get an error message here:
sample(x = c("head","tail"), size = 10, replace = F)
```

```
## Error in sample.int(length(x), size, replace, prob): 當 'replace = FALSE' 時不能取用比總體還要大的樣本
```

```r
#size <= length(x) must hold if replace = F
sample(x = c("head","tail"), size = 2, replace = F)
```

```
## [1] "tail" "head"
```

Distributions

- Distributions is NOT an R function, but a collection of randomization tools provided by R.
- Most of the frequently used distributions are provided, including Normal, Exponential, Uniform, Poisson, Binomial, F, Chi-squared, Cauchy, Beta, and Gamma and many more!
- Use `?Distributions` for more information on what R has to offer.
- Generally, the syntaxes are `dxxxx`, `pxxxx`, `qxxxx`, and `rxxxx` where `xxxx` is the name of the distribution.
    - `dxxxx(x, parameters)` returns the probability density at x, $f(x)$.
    - `pxxxx(q, parameters, lower.tail)` returns the cumulated probability with respect to q depending on what we provide to `lower.tail`
        - Returns cumulated probability $F(q)$ if `lower.tail` is set to `TRUE / T`.
        - Returns *right-tail probability* (more often referred to as *tail probability*) $1 - F(q)$ if `lower.tail` is set to `FALSE / F`.
        - By default, `lower.tail = T` if not specified.
    - `qxxx(p, parameters, lower.tail)` where `p` is a probability. Returns the *quantile* with respect to `p` depending on what we provide to `lower.tail`.
        - `lower.tail = T` gives us $q = F^{-1}(p)$, i.e., q is such that $p = F(q)$.
        - `lower.tail = F` gives us $q = F^{-1}(1-p)$, i.e., q is such that $p = 1 - F(q)$.
        - Once again, `lower.tail = T` by default.
    - `rxxxx(n, parameters)` generates a vector of random numbers of length `n` from the distribution.
        - `parameters` are the parameters of the distribution.
- Focus on `rxxxx` and demonstrate with Uniform and Normal. Will return to other functions later.

Drawing from an Uniform Distribution

- If x follows an Uniform distribution on the interval $[min, max]$, then the pdf of x is given by

$$f(x) = \frac{1}{max - min}$$

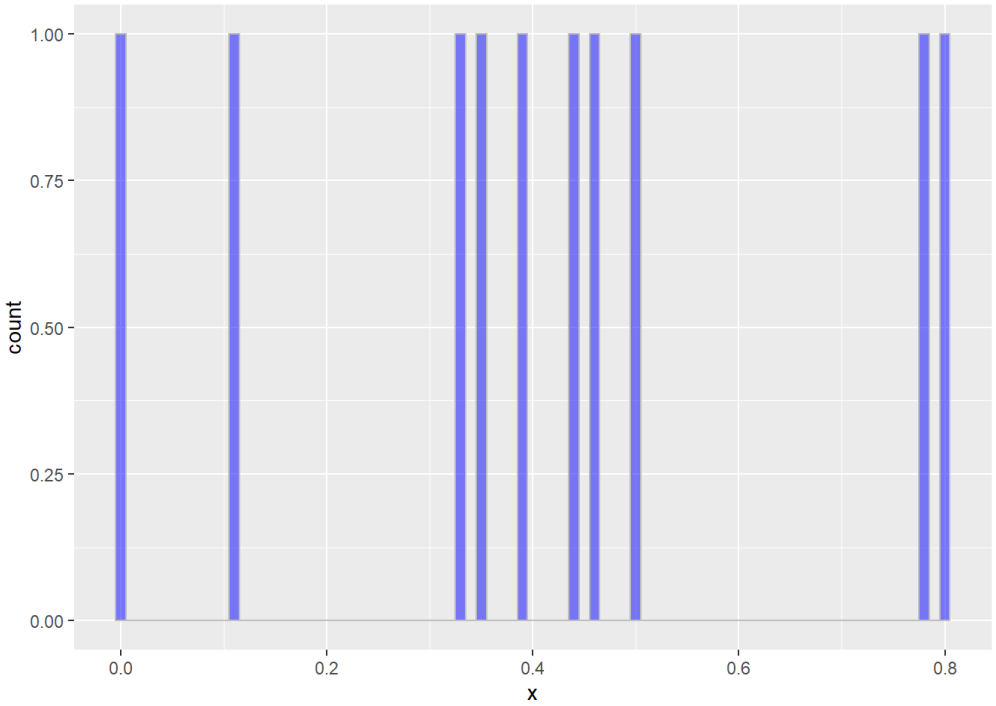
    - min and max are the *parameters* for Uniform.
    - Oftentimes we work with $min = 0$ and $max = 1$.
- We call by `runif(n, min, max)`

- `min` and `max` are set to 0 and 1 by default.

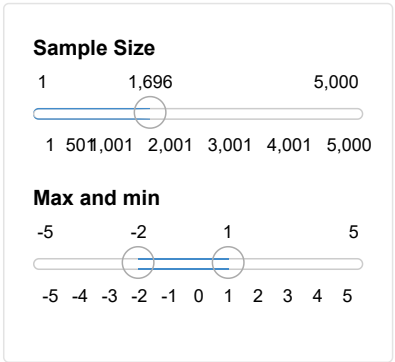
```
#Draw a sample of size 10 from U[0,1] and plot it out
num<-runif(n = 10, min = 0, max = 1)
num
```

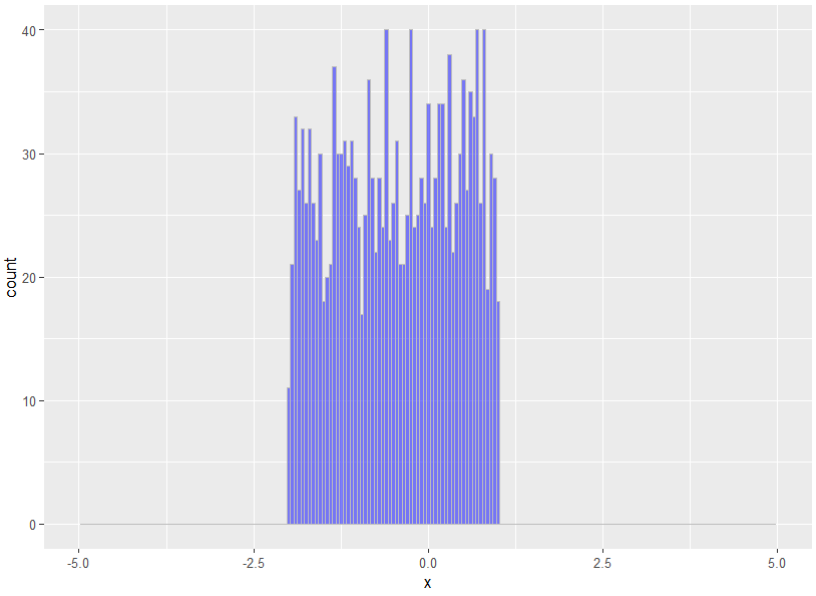
```
##  [1] 0.003841673 0.440347125 0.500512445 0.326407087 0.111349196 0.352220929
##  [7] 0.455470396 0.781586414 0.391258426 0.798977856
```

```
library(ggplot2) #Use ggplot2 to render plots.
ggplot(data=data.frame(x=num),aes(x)) +
   geom_histogram(binwidth=0.01,alpha=0.5,color="grey",fill="blue")
```



- Draw from a Uniform Distribution, plot it out with different sample sizes and parameters.





Drawing from a Normal Distribution

- The pdf of a normal distribution is defined on $x \in (-\infty, \infty)$ and is given by

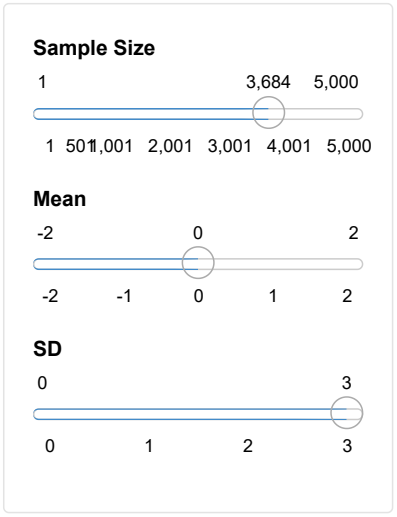
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

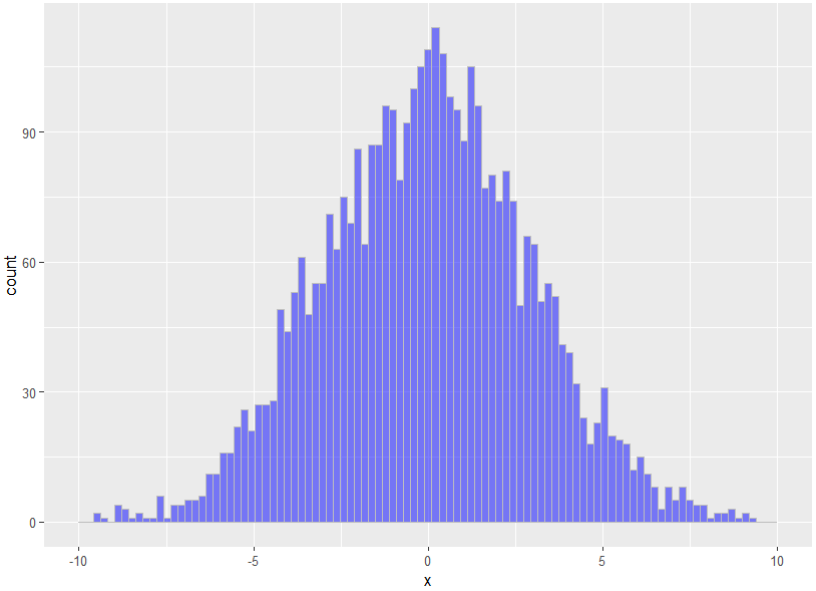
  - μ and $\sigma > 0$ are parameters such that $E(x) = \mu$ and $Var(x) = \sigma^2$.
  - We say that the distribution is a **Standard Normal Distribution** if $\mu = 0$ and $\sigma = 1$.
- We call by `rnorm(n, mean, sd)`, where `mean` corresponds to μ and `sd` corresponds to σ.
  - By default, `mean = 0` and `sd = 1`, i.e., standard normal.

```
#Draw a sample of size 10 from a Standard Normal Distribution
num<-rnorm(n = 10, mean = 0, sd = 1)
num
```

```
##  [1] -0.2570296  0.6669285 -2.8641244  0.3723155 -0.7644477 -1.2158095
##  [7]  1.0737798  0.2782275  0.4027353  0.9121564
```

- Try out sample sizes, mean and sd.





Random Seed and RNG

- R offers a wide range of RNG algorithms at your proposal, including *Mersenne-Twister*, *Wichmann-Hill*, *Super-Duper* and many others.
- `RNGkind(kind, normal.kind, sample.kind)`: Check and set up the RNG algorithms on R.
  - `kind`: takes a *string* as its input, sets the RNG algorithm.
  - `normal.kind`: takes a *string* as its input. Sets the RNG algorithm *specifically* for generating Normal random numbers. Can use *Kinderman-Ramage*, *Ahrens-Dieter*, *Box-Muller* and more.
  - `sample.kind`: No need to change. Only two options and one of them is an outdated buggy version kept for replicating old results.
  - For what algorithms R have to offer, check `?RNGkind()`.
- `RNGkind()` without specifying the arguments shows you the setting on the system.

```
#Check the RNG generators.
RNGkind()
```

```
## [1] "Super-Duper" "Inversion"   "Rejection"
```

- A vector of length 3 is returned, each corresponds to `kind`, `normal.kind` and `sample.kind`. - By default R uses *Mersenne-Twister* and *Inversion* for generic and normal random numbers.
- Use *Super-Duper* and *Box-Muller* for `kind` and `normal.kind`:

```
#Change RNG algorithms. Remember to use "".
RNGkind(kind = "Super-Duper", normal.kind = "Box-Muller")
RNGkind()
```

```
## [1] "Super-Duper" "Box-Muller"  "Rejection"
```

- Can be reverted back to the default setting using `"default"`

```
#Revert back to the default setting
RNGkind(kind = "default", normal.kind = "default")
RNGkind()
```

```
## [1] "Mersenne-Twister" "Inversion"        "Rejection"
```

- `set.seed(seed, kind, normal.kind, sample.kind)`: allows you to set the random seed for RNGs.
  - `seed` takes an integer value. Realizations are *guaranteed* to be the same for the same seed number given the RNG used… at least on the same machine (cross-machine reproducibility problem).
  - We have discussed `kind`, `normal.kind`, and `sample.kind`. The changes here will be applied **globally**. But usually we want to choose the RNG using `RNGkind()` already before tweaking with seeds.
- The seeds are re-initialized each time `set.seed(seed)` is executed.
- Demonstrate with seed number 48763.

```
#Set seed to 48763, then draw from uniform and do a sampling.
set.seed(seed = 48763)
u1<-runif(n = 10, min = 0, max = 1)
s1<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)
u1
```

```
##  [1] 0.50530856 0.44668138 0.70166146 0.81306954 0.01714303 0.47186362
##  [7] 0.85617196 0.29569995 0.08882903 0.51287889
```

```
s1
```

```
##  [1] "Kirito" "Leafa"  "Kirito" "Yui"    "Asuna"  "Yui"    "Asuna"  "Leafa"
##  [9] "Kirito" "Yui"
```

- Do the same but without changing the seed.

```
#Draw from uniform and do a sampling without changing the seed.
u2<-runif(n = 10, min = 0, max = 1)
s2<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)
u2
```

```
##  [1] 0.7421049 0.1972381 0.5185563 0.5344426 0.2692721 0.2283831 0.4335493
##  [8] 0.6881719 0.6683697 0.1897726
```

```
s2
```

```
##  [1] "Kirito" "Kirito" "Yui"    "Kirito" "Kirito" "Kirito" "Asuna"  "Yui"
##  [9] "Kirito" "Kirito"
```

- Now reinitialize the seed as 48763 and do the same.

```
set.seed(seed = 48763)
u1.r<-runif(n = 10, min = 0, max = 1)
s1.r<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)
u2.r<-runif(n = 10, min = 0, max = 1)
s2.r<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)
```

- A quick comparison shows that the results are the same after reinitialization

```
#Side-by-side visual comparison
library(tidyverse) #Call tidyverse for pipe-line operators "%>%".
cbind(c(u1,u2),c(u1.r,u2.r))%>%'colnames<-'(c("1st","2nd"))
```

```
##              1st        2nd
##  [1,] 0.50530856 0.50530856
##  [2,] 0.44668138 0.44668138
##  [3,] 0.70166146 0.70166146
##  [4,] 0.81306954 0.81306954
##  [5,] 0.01714303 0.01714303
##  [6,] 0.47186362 0.47186362
##  [7,] 0.85617196 0.85617196
##  [8,] 0.29569995 0.29569995
##  [9,] 0.08882903 0.08882903
## [10,] 0.51287889 0.51287889
## [11,] 0.74210489 0.74210489
## [12,] 0.19723815 0.19723815
## [13,] 0.51855634 0.51855634
## [14,] 0.53444259 0.53444259
## [15,] 0.26927206 0.26927206
## [16,] 0.22838305 0.22838305
## [17,] 0.43354925 0.43354925
## [18,] 0.68817191 0.68817191
## [19,] 0.66836974 0.66836974
## [20,] 0.18977256 0.18977256
```

```
cbind(c(s1,s2),c(s1.r,s2.r))%>%'colnames<-'(c("1st","2nd"))
```

```
##       1st      2nd
##  [1,] "Kirito" "Kirito"
##  [2,] "Leafa"  "Leafa"
##  [3,] "Kirito" "Kirito"
##  [4,] "Yui"    "Yui"
##  [5,] "Asuna"  "Asuna"
##  [6,] "Yui"    "Yui"
##  [7,] "Asuna"  "Asuna"
##  [8,] "Leafa"  "Leafa"
##  [9,] "Kirito" "Kirito"
## [10,] "Yui"    "Yui"
## [11,] "Kirito" "Kirito"
## [12,] "Kirito" "Kirito"
## [13,] "Yui"    "Yui"
## [14,] "Kirito" "Kirito"
## [15,] "Kirito" "Kirito"
## [16,] "Kirito" "Kirito"
## [17,] "Asuna"  "Asuna"
## [18,] "Yui"    "Yui"
## [19,] "Kirito" "Kirito"
## [20,] "Kirito" "Kirito"
```

- Check similarity with logical operation.

```
#Checking with logical operation
sum(c(u1,u2)!=c(u1.r,u2.r))
```

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

```
sum(c(s1,s2)!=c(s1.r,s2.r))
```

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

- `!=` performs element-wise comparisons between objects. Returns `TRUE` if the elements are *different*, and returns `FALSE` if elements are the same.
- R treats `TRUE` and `FALSE` as 1 and 0. Hence the sum of `!=` operation returns 0 if all elements are the same.
- Now use 65536 as the seed and compare the results.

```
#Use 65536 as the seed and compare
set.seed(seed = 65536)
u1.r<-runif(n = 10, min = 0, max = 1)
s1.r<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)
u2.r<-runif(n = 10, min = 0, max = 1)
s2.r<-sample(x = c("Kirito", "Asuna", "Yui", "Leafa"), size = 10, replace = T)

cbind(c(u1,u2),c(u1.r,u2.r))%>%'colnames<-'(c("s48763","s65536"))
```

```
##           s48763     s65536
##  [1,] 0.50530856 0.56468435
##  [2,] 0.44668138 0.34512149
##  [3,] 0.70166146 0.64719228
##  [4,] 0.81306954 0.48449493
##  [5,] 0.01714303 0.76464675
##  [6,] 0.47186362 0.07771642
##  [7,] 0.85617196 0.91509428
##  [8,] 0.29569995 0.27076328
##  [9,] 0.08882903 0.89629137
## [10,] 0.51287889 0.17812100
## [11,] 0.74210489 0.98286264
## [12,] 0.19723815 0.62230530
## [13,] 0.51855634 0.98264551
## [14,] 0.53444259 0.52651414
## [15,] 0.26927206 0.08835121
## [16,] 0.22838305 0.29423874
## [17,] 0.43354925 0.07623665
## [18,] 0.68817191 0.95283261
## [19,] 0.66836974 0.42467125
## [20,] 0.18977256 0.68328902
```

```
cbind(c(s1,s2),c(s1.r,s2.r))%>%'colnames<-'(c("s48763","s65536"))
```

```
##       s48763   s65536
##  [1,] "Kirito" "Yui"
##  [2,] "Leafa"  "Asuna"
##  [3,] "Kirito" "Leafa"
##  [4,] "Yui"    "Asuna"
##  [5,] "Asuna"  "Kirito"
##  [6,] "Yui"    "Asuna"
##  [7,] "Asuna"  "Asuna"
##  [8,] "Leafa"  "Kirito"
##  [9,] "Kirito" "Asuna"
## [10,] "Yui"    "Asuna"
## [11,] "Kirito" "Kirito"
## [12,] "Kirito" "Asuna"
## [13,] "Yui"    "Kirito"
## [14,] "Kirito" "Kirito"
## [15,] "Kirito" "Leafa"
## [16,] "Kirito" "Asuna"
## [17,] "Asuna"  "Asuna"
## [18,] "Yui"    "Leafa"
## [19,] "Kirito" "Kirito"
## [20,] "Kirito" "Yui"
```

```
sum(c(u1,u2)==c(u1.r,u2.r)) #Sum will be less than 20 if any of the elements are different.
```

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

```
sum(c(s1,s2)==c(s1.r,s2.r)) #Sum will be less than 20 if any of the elements are different.
```

```
## [1] 5
```

- Let's use *Super-Duper* as our RNG and see how the realizations differ from *Mersenne-Twister*.

```
#Use the default RNG to generate a random vector using seed 48763.
RNGkind()
```

```
## [1] "Mersenne-Twister" "Inversion"        "Rejection"
```

```
set.seed(seed = 48763)
drw.MT<-runif(n = 10, min = 0, max = 1)

#Use Super-Duper and generate a random vector using the same seed.
RNGkind(kind = "Super-Duper")
set.seed(seed = 48763)
drw.SD<-runif(n = 10, min = 0, max = 1)

#Compare:
drw.MT
```

```
##  [1] 0.50530856 0.44668138 0.70166146 0.81306954 0.01714303 0.47186362
##  [7] 0.85617196 0.29569995 0.08882903 0.51287889
```

```
drw.SD
```

```
##  [1] 0.33614158 0.54181592 0.76705691 0.29961992 0.12271298 0.10703084
##  [7] 0.69835006 0.80798265 0.95212424 0.06113896
```

```
sum(drw.MT==drw.SD) #Sum will be less than 10 if any of the elements are different.
```

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

Quick Takeaways

- R provides various ways for randomization.
- It is NOT real randomness! RNGs and seeds matter for the realizations.
- ALWAYS set up a random seed to keep track of your results and for reproducibility, especially when turning in your assignments.
- For performing randomized simulations, sticking with the same seed can be problematic as the realizations are actually predictable.
- For example, *Mersenne-Twister* generates $2^{19937}-1$ random numbers given the random seed, but people most likley only look at the very first 10^5 numbers in the list.
- Possible ways to improve the quality of randomness:
    1. Take the middle: Suppose that you need 10^4 random numbers. You generate a random vector of length 10^8 and drop the very first 10^3 and very last 10^1.
    2. Randomized randomization: Similar to the previous point, but now you choose another random seed and then sample 10^4 elements from the 10^8 random numbers you generated in the first round.
    3. Seed-of-seeds: Given a seed, generate a vector of length 10^2 for example. Then use these numbers as seeds and draw 10^2 of numbers from each of them. Together you will get a vector of random numbers of $10^2 * 10^2 = 10^4$ as desired.
    4. Use different RNGs.

In-class Exercise

1. Use the seed-of-seeds approach to generate 100 random numbers from Standard Normal with 5 different seeds. You may want to have each seed to generate 20 numbers, but let's just be flexible about it.
2. Generate a 100×5 matrix from an Exponential Distribution. You are free to choose the rate parameter.
3. Draw 10000 random numbers out from an Uniform distribution on $[-2, 5]$. Compute the average of the numbers generated and compare with the theoretical mean of this distribution.

Transformation Methods

- We have been working on random variables that R has to offer.
- What if we need to simulate with self-defined random variables?
- Or, what if R does not have the random variable we want?
    - Modern quantitative trade models are almost always characterize firm productivity with either a Pareto or a Frechet Distribution. But R offers NEITHER of them!
- For well-known distributions like Frechet and Pareto, one might search on CRAN to see if there are packages providing these distributions.
- For home-made distributions, we are on our own! This is when the transformation method comes in.

PDF and CDF Transformation

PDF Transformation:

Consider a continuous random variable x with its pdf given by $f(x)$. Suppose that the variable y relates to x by a function as $y = h(x)$. Assume that the function $h(x)$ is continuous, one-to-one and unique, then the inverse function $x = h^{-1}(y)$ is defined. The pdf of y denoted by $g(y)$ is therefore given by

$$g\left(y\right) = f\left(h^{-1}\left(y\right)\right)\left|\frac{dh^{-1}\left(y\right)}{dy}\right|$$

- Example 1: If x follows $U[0, 1]$ (the pdf is hence $f(x) = 1$) and we have $y = x^2$. Then the inverse function $h^{-1}(x)$ is $x = y^{\frac{1}{2}}$ The pdf of y is thus given by

$$g(y) = f(y^{\frac{1}{2}})\frac{1}{2}y^{-\frac{1}{2}} = \frac{1}{2}y^{-\frac{1}{2}}$$

- Example 2: If $y > 0$ follows a Standard Normal Distribution, and we know that $y = \ln x$. Then $x = \exp y$ follows a Log-Normal Distribution with $\ln \mu = 0$ and $\ln \sigma = 1$. See Wikipedia: https://en.wikipedia.org/wiki/Log-normal_distribution (https://en.wikipedia.org/wiki/Log-normal_distribution)

- We require knowledge on derivatives to perform PDF transformation. But sometimes it is not very easy to compute if the pdf $f(x)$ is complicated.

CDF Transformation:

Consider a random variable x with its cdf given by $\Pr(X \leq x) \equiv F(x)$. Suppose that $y = h(x)$ and the inverse function $x = h^{-1}(y)$ is defined. The cdf of y denoted by $\Pr(Y \leq y) \equiv G(y)$ is obtained from $F(x)$ as

$$G(y) = F(h^{-1}(y)).$$

- A very simple proof: Recall that $y = h\left(x\right) \iff x = h^{-1}\left(y\right)$. Then we have

$$\Pr\left(X \leq x\right) = \Pr\left(X \leq h^{-1}\left(y\right)\right) = \Pr\left(h\left(X\right) \leq y\right) \equiv \Pr(Y \leq y),$$

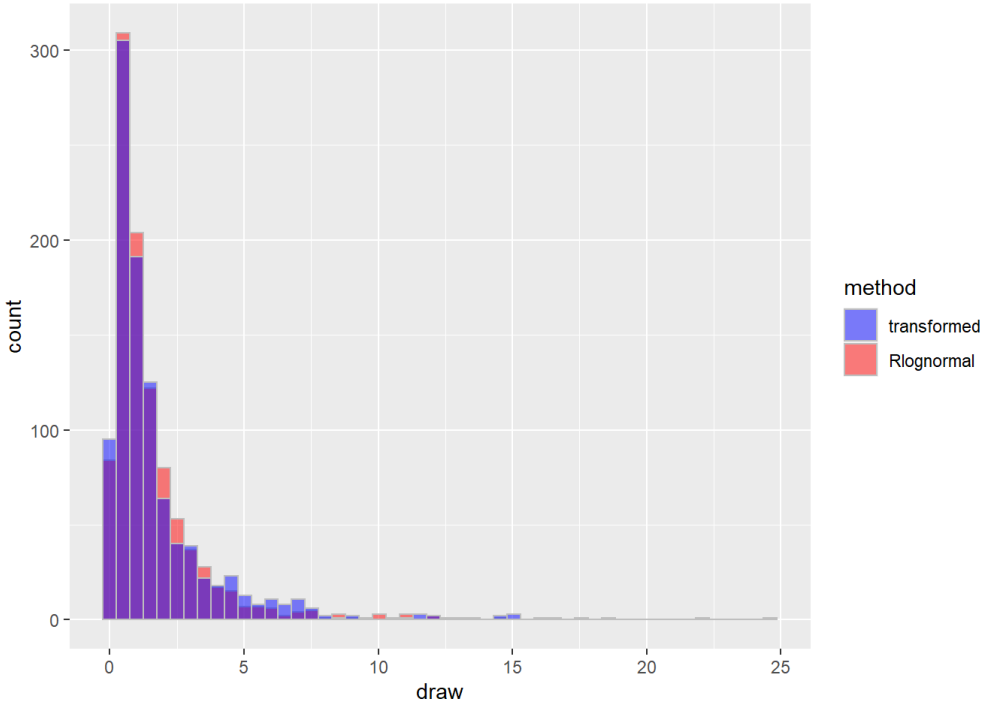
where we define $Y \equiv h(X)$. Since by definition $F(x) \equiv \Pr(X \leq x)$, $\Pr(X \leq h^{-1}(y)) = F(h^{-1}(y))$, and $G(y) \equiv \Pr(Y \leq y)$, we conclude that $G(y) = F(h^{-1}(y))$.

- Example 3: We do Example 1 again, but this time by the CDF Transformation method. Here we have $F(x) = x$. Then $G(y) = F(y^{\frac{1}{2}}) = y^{\frac{1}{2}}$. It is easily checked that the results are equivalent by the fact that $f(x) = \frac{dF(x)}{dx}$.
- This is easier to work with than the pdf transformation method if we know the cdf directly.

Implementation on R

- If R has the distribution of x, and we know that the desired random variable y is related to x by $y = h(x)$. Then we can simply draw x using `rxxx`, and then apply $y = h(x)$ to get the random vector we want.
- Demonstrate using Standard Normal: we want to draw a vector of random numbers from a Log-Normal Distribution with log-scaled mean and standard deviation being 0 and 1. Denote this random variable by y. We perform the following steps:
  1. Draw x from Standard Normal.
  2. For the desired variable y, the definition of Log-Normal says that $x = \ln y$ is a Standard Normal. Meaning that we recover y from x by applying $y = \exp x$.
  3. R has a built-in Log-Normal Generator, we can draw from it and compare with y that we obtained from the previous step. They should be looking similar if we do it right. Check `?rlnorm()` for details of drawing from Log-Normal.

```
x<-rnorm(n = 1000, mean = 0, sd = 1) #Draw from Standard Normal.
y<-exp(x) #Transform x to y, which can be proven to be Log-Normal.
z<-rlnorm(n = 1000, meanlog = 0, sdlog = 1) #R's built-in log-normal generator.
df<-data.frame(c(y,z),c(rep("transformed",1000),rep("Rlognormal",1000)))%>%
  'colnames<-'(c("draw","method"))
ggplot(data=df) +
  geom_histogram(aes(x=draw,fill=method),color="grey",bins=50,alpha=0.5,position="identity")+
  scale_fill_manual(values=c("blue","red"),
                    breaks=c("transformed","Rlognormal"))
```



- Comes in handy when performing quantitative simulations with models with heterogeneity.

- For example, firms are different in their productivities φ. Firm's revenue thus depends on its productivity, and potentially on an exogenous variable say tariff rate τ. Then the model will be giving you the functional form of revenue as $r(\varphi, \tau)$.
- How do we simulate the impact of the doubling of the tariff rate τ on the *distribution* of firm sales? Well:

1. Usually we have the distribution of φ, so we can draw it on R.
2. Then we apply $r(\varphi, \tau)$ given the level of τ to get the revenue of firms hence the distribution.
3. To see the effect of the doubling of τ, simply apply $r(\varphi, \tau)$ and $r(\varphi, 2\tau)$ to our draw of φ. Then we have two distributions, one is before and the other is after the doubling of tariffs.

- This is more about quantitative modeling: We care about how exogenous shocks affect the distribution of y generated from the underlying heterogeneity x. But for quantitative purpose, how $g(y)$ looks like does not always matter. All we need is the pdf $f(x)$ and the mapping $y = h(x)$.
- What if $f(x)$ is not even available in R? The approach here requires substantial knowledge on R's built-in generator and transformation mapping. Not always doable…

Probability Integral Transformation (PIT)

- The more practical approach to generate random numbers from a general distribution. But let's introduce the concept of quantile function before diving in.

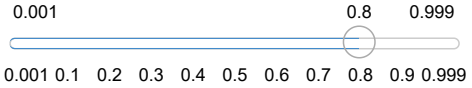
Quantile Function: Suppose that the cumulated probability of x is given by the cdf $p = F(x) = \Pr(X \leq x)$, the *quantile function* of this distribution is such that

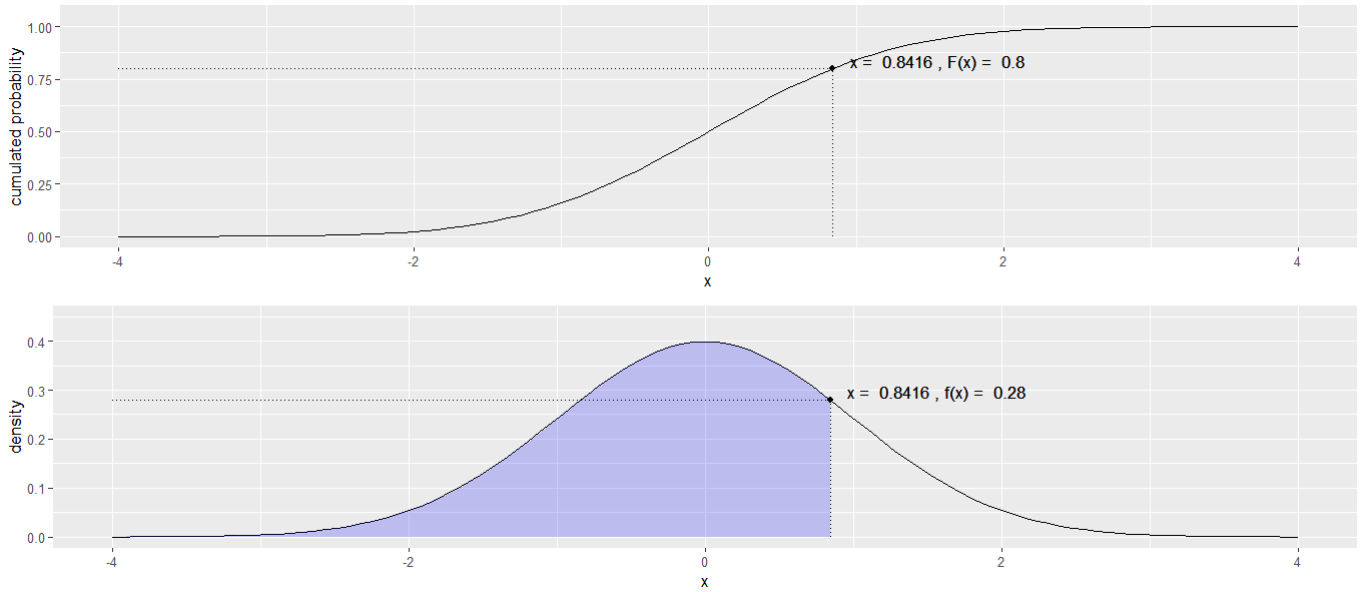
$$F^{-1}(p) = \inf\{x : F(x) \geq p\},\ p \in (0, 1)$$

In a more sloppy manner, if a distribution is such that $p = F(x)$ then the quantile function is that $x = F^{-1}(p)$.

- Example 1: Suppose that x follows an uniform distribution on $[0, 5]$. Then:
  - The cdf is then $F(x) = \frac{x}{5}$; the quantile function is $x = 5p$.
  - If we think of x as restaurant ranking, then a restaurant getting $x = 4$ beats 80% of restaurants, i.e., $F(4) = \frac{4}{5} = 0.8$
  - Then how high should the ranking be if the restaurant is to beat 80% of its competitors? Well, $x = F^{-1}(p) = 5 \times 0.8 = 4$.
- Example 2: visualization with a Standard Normal Distribution.

Probability





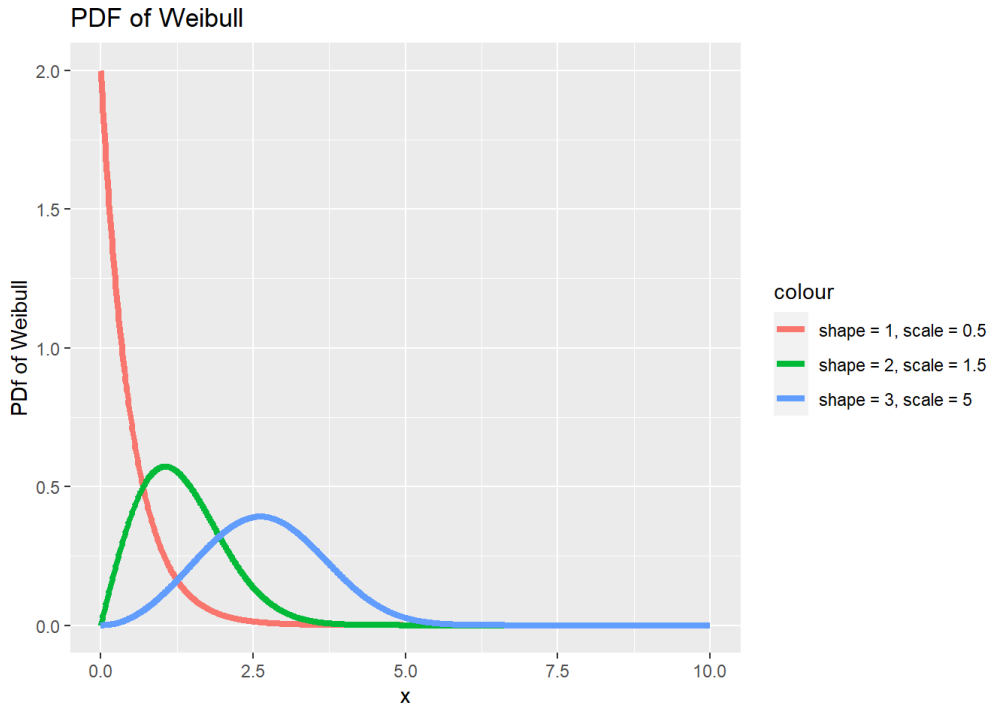
- R provides quantile functions for several distributions as we have mentioned in the beginning. For examples, `pnorm()`, `pexp()` and `punif()`. Check `Help` for details.

Probability Integral Transformation (PIT): Consider a continuous random variable X with a continuous cdf $p = F(x)$. The probability $F(x)$ by itself is also a random variable and follows an uniform distribution on the closed interval $[0, 1]$.

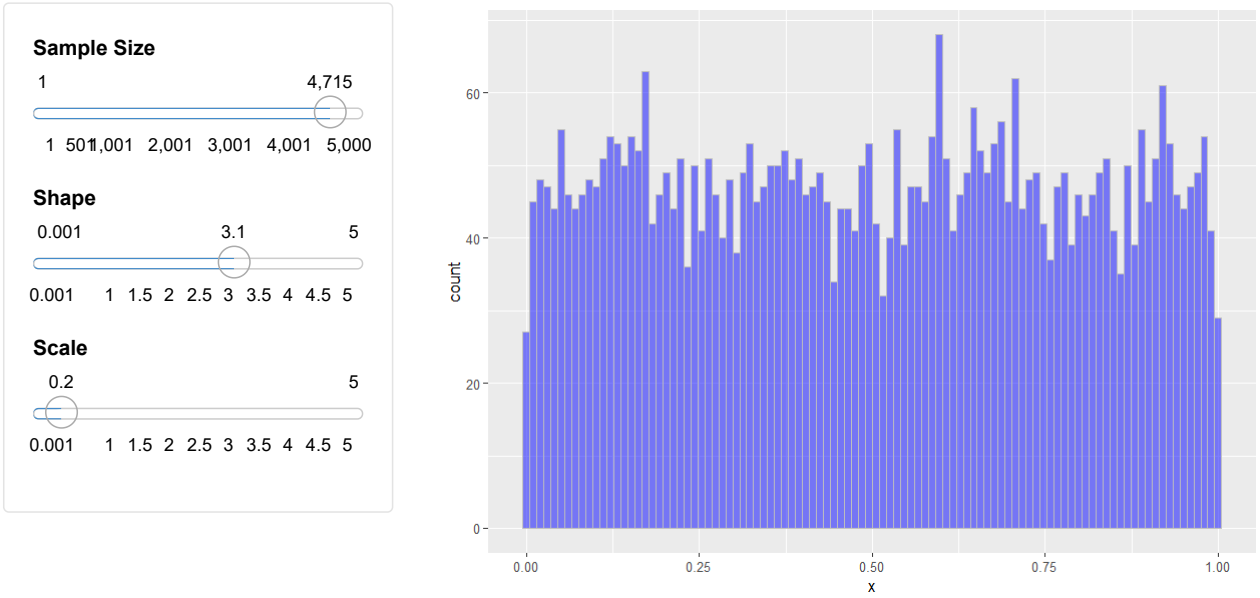
- Demonstrate using a Weibull Distribution.
  - A Weibull distribution is defined on $(0, \infty)$ with its pdf given by

$$f(x) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-\left(\frac{x}{\lambda}\right)^k}$$

    where $\lambda > 0$ and $k > 0$ represent the scale and shape parameters, respectively.
  - The pdf of Weibull with different shape and scale parameters. It can have an asymptote around $x = 0$, i.e., density goes to ∞ as x gest arbitrarily close to 0.



- Demonstrate that the probability from Weibull is still uniform.



- The PIT theorem allows us to generate random numbers from a continuous distribution using draws from $U[0,1]$, provided that we know the *quantile function* of the concerned distribution.

Implementation on R

- We need `runif()` to generate a random vector from $U[0,1]$.
- We need to know the quantile function of the distribution to our interest, $x = F^{-1}(p)$.
- Implementation:
  - **Step 1** Draw a vector from $U[0,1]$. Let's denote this vector by p.
  - **Step 2** Plug p into $F^{-1}(p)$ to obtain x, which is what we want.
- Use Frechet for demonstration. A Frechet random variable is defined on $x \in (m, \infty)$, and its cdf and quantile functions are respectively given as follows:

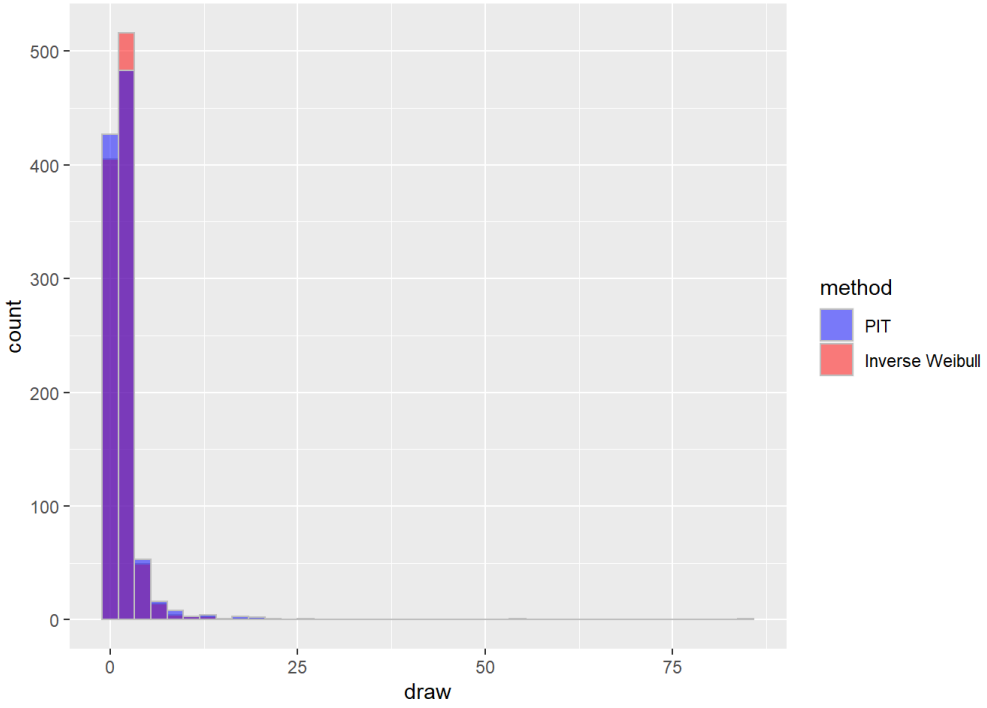
$$p = F(x) = e^{-\left(\frac{x-m}{s}\right)^{-\alpha}}$$

$$x = F^{-1}(x) = s\left(-\frac{1}{\ln p}\right)^{\frac{1}{\alpha}} + m$$

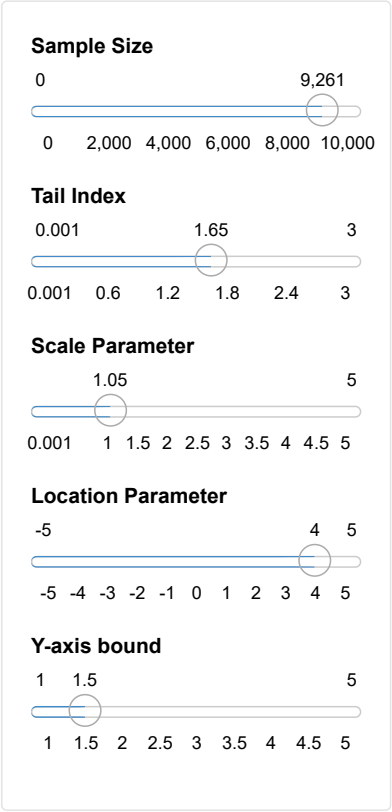
where $\alpha > 0$ is the tail index, $s > 0$ is the scale parameter, and $m \in (-\infty, \infty)$ is the location parameter.

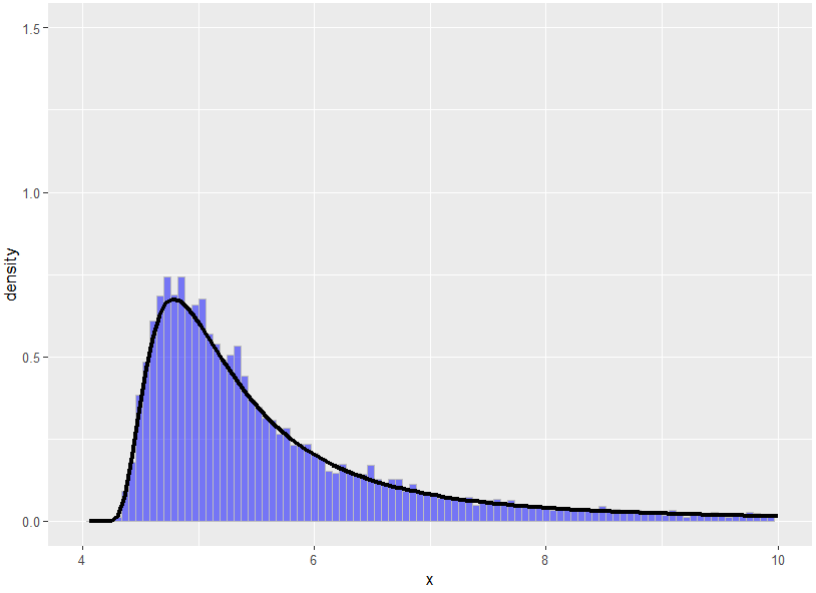
- For x following a Frechet distribution with $m = 0$, its inverse $y = 1/x$ follows a Weibull distribution with shape parameter $\lambda \equiv \alpha$ and scale parameter $k \equiv 1/s$. See Wikipedia: https://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution (https://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution)
- Since R has a built-in Weibull distribution, we can check if the PIT approach is consistent.

```
#Focus on a Frechet distribution with m = 0, alpha = 2 and s = 1.
p<-runif(n = 1000, min = 0, max = 1) #Draw 1000 should be enough.
frechet.PIT<-(-1/log(p))^(1/2) #Plug p into Frechet's quantile function.

y<-rweibull(n = 1000, shape = 2, scale = 1) #Use R's built-in Weibull generator.
frechet.invweib<-1/y #Inverse of Weibull becomes Frechet as discussed.

df<-data.frame(c(frechet.PIT,frechet.invweib),
               c(rep("PIT",1000),rep("Inverse Weibull",1000)))%>%
  'colnames<-'(c("draw","method"))
ggplot(data=df) +
  geom_histogram(aes(x=draw,fill=method),color="grey",bins=40,alpha=0.5,position="identity")+
  scale_fill_manual(values=c("blue","red"),
                    breaks=c("PIT","Inverse Weibull"))
```



- End this chapter by generating Frechet sample and pdf under different settings.





Assignments

1. Senior undergraduate students at NUK AE are required to take a Seminar before graduation. In this course, students will be divided into 3 groups and perform a group project. Assume that there are in total 34 students taking this course. Denote the groups by A, B, C, and denote the students by numbers 1 to 34. Write a code to randomly assign the students into the 3 groups as equally as possible.

2. A Pareto random variable x is defined on $x \in (x_{min}, \infty)$, with its pdf given by

$$f(x) = x_{min}^{k} k x^{-k-1}$$

where $x_{min} > 0$ is the location parameter and $k > 0$ is the tail index. Let x_{min} be 1 plus the modulus of dividing the last digit of your student ID by 5. For example, $x_{min} = 4$ if your last digit is 3. Then let $k = x_{min}/2$. Draw a random sample of size 1000 from the Pareto distribution given x_{min} and k using the PIT approach. Then compute the mean of your sample.

3. Continued from the question above. Law of Large Number (LLN) says that the sample mean should eventually converge to the population mean for large enough sample size with probability one. Let n be the sample size you drawn from the Pareto distribution above, and $\overline{x}(n) \equiv \frac{\sum_1^n x_i}{n}$ be the sample average when the sample size is n. The LLN implies that there exists some N such that for all $n >= N$ we must have $|\overline{x}(n) - \overline{x}(n+1)|$ to be small enough. Find such a N.

   To do this, you will need to use `while` loop. The idea is that you setup an arbitrarily small number, say $c = 10^{-8}$ and define $diff \equiv |\overline{x}(n+1) - \overline{x}(n)|$. Then you keep increasing your sample size by 1 and computing $diff$. Then you stop when $diff < c$. The sample size such that you stop computing $diff$ is exactly N. But there is a possibility that you can never achieve $diff < c$ because of a special property of Pareto distribution. Therefore you will need to set up a maximum number of trials N_c. That is, you stop when either you get $diff < c$ or when n exceeds N_c.