Monte Carlo Methods

Outline

1. Probability and Inference Refresher.
2. Statistical validation techniques.
3. Random numbers.
4. Approximation of probabilities and volumes.
5. Monte Carlo integration.

Introduction

The term "Monte Carlo methods" or "MC methods" generally refers to all those techniques that make use of artificial (i.e., computer generated) random variables to solve mathematical problems using random samples drawn from the corresponding populations.

Undoubtedly, this is not a very efficient way to obtain the solution to a problem, as the (often) time-consuming simulated sampling procedure gives a result that is always affected by the statistical error. In practice, however, we are often faced with situations in which it is too difficult, if not impossible, to use the standard numerical or analytical procedures; in all these cases, Monte Carlo methods become the only available alternative.

The application of these methods is not limited only to purely statistical problems, as it might seem from the use of probability distributions, but includes all those cases in which a connection can be found between the problem under consideration and the behaviour of a certain random system: for example, the value of a definite integral, which is certainly not a random quantity, can also be calculated using random numbers.

The theoretical foundations of the Monte Carlo methods (or simulation methods) have been known for a long time, and the first example of the use of random numbers for the resolution of definite integrals is even found in a book (Essai d'aritmetique moral) written in 1777 by Georges-Louis Leclerc, Comte de Buffon, French mathematician and naturalist, in which a procedure for the approximate calculation of the π value of is outlined.

However, for more than a century and a half, it was used only sporadically and above all for didactic purposes. Its first systematic application took place only in the first half of the 1940s in Los Alamos, by a team of scientists led by Enrico Fermi, who developed the project of the first atomic bombs. In the same period, the term "Monte Carlo" was also born, which refers to the town famous for its casino and, more precisely, to roulette, one of the simplest mechanical devices that can be used to generate random variables.

The authorship of the name is in particular attributed to the mathematicians J. von Neumann and S. Ulam, who adopted it as the code name of their secret research, conducted using random numbers, on the processes of diffusion and absorption of neutrons in fissile materials.

After 1950, these methods passed in a few years from the role of mathematical curiosity to that of an indispensable tool for scientific research thanks to the advent of computers. This has happened not only because computers provide rapid execution of long calculations that are often necessary to obtain a meaningful result, but also because they can easily generate random numbers. Currently, there are applications in many different research fields, from nuclear physics to chemistry to statistical mechanics to economics. [1]

Summary

- Stochastic simulation or Monte Carlo techniques are computation algorithms that rely on random sampling to approximate the solution of some real problems.
- A real system is first expressed as a stochastic model (set of variables related by mathematical equations, where uncertainty is present and replaced by randomness), then the model is run on a computer through simulations.
- Resampling bootstrapping: random sampling with replacement from an original sample to approximate the distribution of sampling statistics for inferential purposes.

[1]From *Probability, Statistics and Simulation with Application Programs Written in R* - Rotondi et al.

Probability and inference *refresher*

Laws of Large Numbers

The Laws of Large Numbers (LLN) are fundamental theorems in Probability Theory that describe the result of performing the same experiment a large number of times. According to these laws, the average of the results obtained from a large number of trials is close to the expected value and tends to become closer as more trials are performed. LLN form the basis for the Probability Theory and provides a solid foundation for statistical inference.

Suppose $\{X_n\}_n$ is a sequence of independently and identically distributed random variables (i.i.d.) with expected value $E[X_i] = \mu$. The i.i.d. condition means that each random variable X_i in the sequence has the same probability distribution as the others and is independent of them.

Weak Law of Large Numbers (WLLN)

The Weak Law of Large Numbers states that for any $\varepsilon > 0$,

$$\lim_{n\to\infty} P\left(\left|\overline{X}_n - \mu\right| \geq \varepsilon\right) = 0$$

where $\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean of independent and identically distributed (i.i.d.) random variables $\{X_1, X_2, \ldots, X_n\}$ with $\mathbb{E}[X_i] = \mu$ and $\mathrm{Var}(X_i) = \sigma^2 < \infty$.

Proof

Step 1: Rewrite the probability using Chebyshevs inequality

From Chebyshev's inequality, for any random variable Y with finite variance:

$$P(|Y - \mathbb{E}[Y]| \geq k) \leq \frac{\mathrm{Var}(Y)}{k^2}.$$

Here, substitute $Y = \overline{X}_n$, so that $\mathbb{E}[\overline{X}_n] = \mu$. Then:

$$P\left(\left|\overline{X}_n - \mu\right| \geq \varepsilon\right) \leq \frac{\mathrm{Var}(\overline{X}_n)}{\varepsilon^2}.$$

Step 2: Compute the variance of $\overline{X}_n$

Since $X_1, X_2, \ldots, X_n$ are i.i.d., the variance of the sample mean is:

$$\mathrm{Var}(\overline{X}_n) = \mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right).$$

Using the properties of variance:

$$\mathrm{Var}(\overline{X}_n) = \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}(X_i) = \frac{1}{n^2}\cdot n\cdot\sigma^2 = \frac{\sigma^2}{n}.$$

Thus:

$$P\left(\left|\overline{X}_n - \mu\right| \geq \varepsilon\right) \leq \frac{\sigma^2}{n\varepsilon^2}.$$

This implies that as the sample size n increases, the probability that the sample mean $\overline{X}_n$ deviates from the expected value μ by more than any positive amount ε becomes increasingly small. In simpler terms, as the number of trials increases, the sample mean converges in probability to the expected value.

Strong Law of Large Numbers (SLLN)

Let $\{X_i\}_{i=1}^{\infty}$ be a sequence of independent and identically distributed (i.i.d.) random variables with mean $\mu = \mathbb{E}[X_1]$ and finite variance $\sigma^2 = \mathrm{Var}(X_1) < \infty$. Define the sample mean as:

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

Then, the Strong Law of Large Numbers states that

$$P\left(\lim_{n\to\infty}\overline{X}_n = \mu\right) = 1.$$

Proof:

Let $S_n = \sum_{i=1}^{n} X_i$ be the partial sum of the i.i.d. sequence $\{X_i\}$. The sample mean can be expressed as:

$$\overline{X}_n = \frac{S_n}{n}.$$

We aim to show that $\overline{X}_n \to \mu$ almost surely as $n \to \infty$.

The sequence $\{X_i\}$ satisfies:

$$\mathbb{E}[S_n] = n\mu, \quad \mathrm{Var}(S_n) = n\sigma^2.$$

Thus, the variance of the sample mean is:

$$\mathrm{Var}(\overline{X}_n) = \mathrm{Var}\left(\frac{S_n}{n}\right) = \frac{\sigma^2}{n}.$$

This implies that as $n \to \infty$, the variance of $\overline{X}_n$ approaches 0.

Using Chebyshevs inequality, for any $\epsilon > 0$:

$$P\left(|\overline{X}_n - \mu| \geq \epsilon\right) \leq \frac{\text{Var}(\overline{X}_n)}{\epsilon^2} = \frac{\sigma^2}{n\epsilon^2}.$$

Summing over n, we have:

$$\sum_{n=1}^{\infty} P\left(|\overline{X}_n - \mu| \geq \epsilon\right) \leq \sum_{n=1}^{\infty} \frac{\sigma^2}{n\epsilon^2}.$$

The series $\sum_{n=1}^{\infty} \frac{\sigma^2}{n\epsilon^2}$ simplifies to:

$$\frac{\sigma^2}{\epsilon^2} \sum_{n=1}^{\infty} \frac{1}{n}.$$

The series $\sum_{n=1}^{\infty} \frac{1}{n}$ diverges (harmonic series).

The Borel-Cantelli Lemma states:

- If $\sum_{n=1}^{\infty} P(A_n) < \infty$, then $P(A_n \, \text{i.o.}) = 0$, where A_n is an event and "i.o." means "infinitely often."
- If $\sum_{n=1}^{\infty} P(A_n) = \infty$, then $P(A_n \, \text{i.o.})$ may or may not be zero depending on the independence or dependence of the events.

In this case, $P\left(|\overline{X}_n - \mu| \geq \epsilon\right)$ depends on n and decreases with $\frac{1}{n}$. However, because $\overline{X}_n$ is based on i.i.d. random variables and their averages, the sequence $\overline{X}_n$ satisfies the independence conditions needed to apply the second Borel-Cantelli Lemma.

While the series $\sum_{n=1}^{\infty} \frac{1}{n}$ diverges, the probabilities $P\left(|\overline{X}_n - \mu| \geq \epsilon\right)$ decay rapidly enough that the deviations from μ (large values of $|\overline{X}_n - \mu|$) occur only finitely often in practice.

As a result:

$$P\left(|\overline{X}_n - \mu| \geq \epsilon \, \text{infinitely often}\right) = 0.$$

This means that for any fixed $\epsilon > 0$, the sample mean $\overline{X}_n$ stays within ϵ of μ for all sufficiently large n, almost surely.

Then, we conclude that:

$$P\left(\lim_{n\to\infty} \overline{X}_n = \mu\right) = 1.$$

This law is a stronger statement compared to the Weak Law. It asserts that the sample mean $\overline{X}_n$ not only converges in probability but almost surely converges to the expected value μ as the sample size n goes to infinity. In other words, the sample mean will be equal to the expected value almost certainly in the limit of an infinite number of trials.

- The Weak Law of Large Numbers (WLLN) states that for any $\epsilon > 0$,

$$P\left(|\overline{X}_n - \mu| \geq \epsilon\right) \to 0 \quad \text{as } n \to \infty.$$

- This describes *convergence in probability*, which means that the probability of the sample mean $\overline{X}_n$ being far from μ becomes arbitrarily small as n increases.

- Intuitively, the WLLN implies that the sample mean is likely to be close to the population mean as the sample size grows, but it does not guarantee that this happens for *all* possible realizations of the sequence.

- The Strong Law of Large Numbers (SLLN) states that:

$$P\left(\lim_{n\to\infty} \overline{X}_n = \mu\right) = 1.$$

- This describes *almost sure convergence*, which means that the sequence of sample means $\{\overline{X}_n\}$ converges to the true mean μ for *almost every* realization of the random variables.

- Intuitively, the SLLN guarantees that the sample mean $\overline{X}_n$ will almost surely settle at the population mean μ as the sample size grows, regardless of the randomness in individual samples.

Key Intuitive Difference:

- The *Weak Law* guarantees that the sample mean will be close to the population mean *with high probability*, but allows for exceptions in certain realizations.

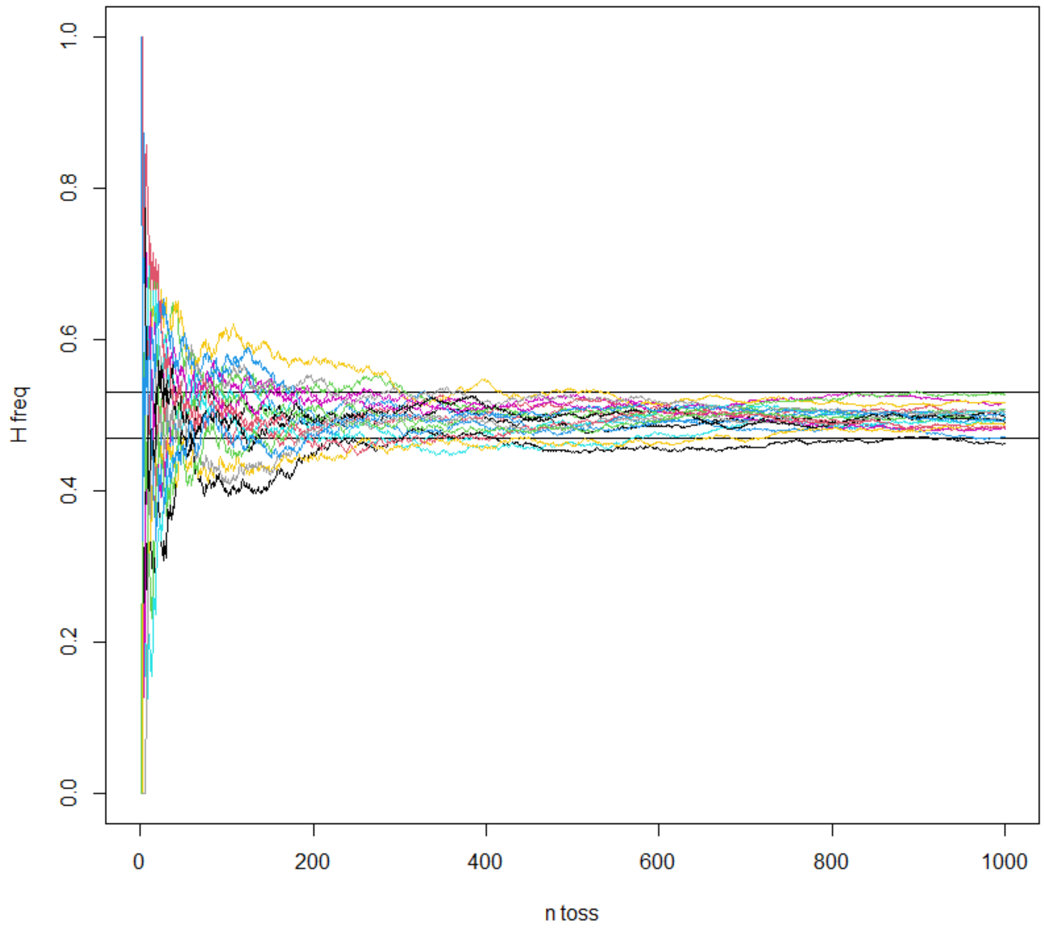
- The *Strong Law* guarantees that the sample mean will converge to the population mean for *almost all* realizations, leaving no room for sustained deviations in the long run.

Example

Observe that for the Bernoulli model, if $X_i \sim Bin(1,p)$, then $\overline{X}_n = \widehat{p}_n$, $\mu = p$, and $\widehat{p}_n$ converges to p, where

$$\widehat{p}_n = \frac{\#\{X_i \text{ with a characteristic}\}}{n}.$$

```r
# Set up an empty plot with specific axis labels and limits
# "n toss" on x-axis, "H freq" on y-axis, x-axis from 0 to 1000,
# y-axis from 0 to 1
plot(1, type="n", xlab="n toss", ylab="H freq",
     xlim=c(0, 1000), ylim=c(0, 1))

# Add horizontal lines at y = 0.53 and y = 0.47
abline(h=c(0.53, 0.47))

# Loop to generate and plot 20 different random coin toss simulations
for(i in 1:20) {
  set.seed(i)
  # Generate 1000 random coin tosses (0 for tails, 1 for heads),
  # compute cumulative mean, and plot the line with a unique color
  points(cumsum(sample(c(0,1), size=1000, replace=TRUE)) / (1:1000),
         type="l", col=i)
}
```



In Python:

```
# Illustration of LLN
import matplotlib.pyplot as plt
import numpy as np

plt.figure()
plt.xlabel("n toss")
plt.ylabel("H freq")
plt.xlim(0, 1000)
plt.ylim(0, 1)
plt.axhline(y=0.53, color='black')
plt.axhline(y=0.47, color='black')

for i in range(1, 21):
    np.random.seed(i)
    points = np.cumsum(np.random.choice([0, 1], size=1000, replace=True)) /
    np.arange(1, 1001)
    plt.plot(points, color=plt.cm.tab20(i))

plt.show()
```

Central Limit Theorem

Lyapunov's Central Limit Theorem

The Central Limit Theorem (CLT) is fundamental in Probability Theory, which states that, under certain conditions, the sum of a large number of random variables tends to follow a normal distribution, regardless of the underlying distribution of these variables. Lyapunov's CLT is a specific form of this theorem that applies to sequences of independent and identically distributed random variables.

Suppose $\{X_n\}_n$ is a sequence of independent and identically distributed random variables (i.i.d.), each with finite mean μ and variance σ^2. The theorem states that the normalized sum of these random variables converges in distribution to a standard normal random variable:

$$\frac{\sum_{i=1}^{n} X_i - n\mu}{\sigma\sqrt{n}} = \frac{\overline{X}_n - \mu}{\frac{\sigma}{\sqrt{n}}} \xrightarrow[n\to\infty]{d} Z,$$

where $\overline{X}_n$ is the sample mean of the first n random variables, and $Z \sim N(0,1)$ is a standard normal random variable.

The notation $\xrightarrow[n\to\infty]{d}$ signifies convergence in distribution, which means that the distribution of the normalized sum approaches the distribution of Z as n becomes large.

Convergence in distribution of a sequence of random variables $\{Y_n\}_n$ to a random variable Y is defined as:

$$\lim_n F_n(x) = F(x)$$

at every continuity point x of F, where F_n and F are the cumulative distribution functions (cdfs) of Y_n and Y, respectively.

In practical scenarios, the variance of the population σ^2 may not be known and is often replaced with a consistent estimate. A consistent estimate is a statistic that converges in probability to the parameter it estimates as the sample size increases. One common consistent estimate for σ^2 is the sample variance S_n^2. Using S_n in place of σ, we obtain the following expression:

$$\frac{\sum_{i=1}^n X_i - n\mu}{S_n\sqrt{n}} = \frac{\overline{X}_n - \mu}{\frac{S_n}{\sqrt{n}}} \xrightarrow[n\to\infty]{d} Z.$$

De Moivre-Laplace Theorem

For a binomial *r.v.* $X_n \sim Bin(n, p)$, it holds

$$\frac{X_n - np}{\sqrt{np(1-p)}} \xrightarrow[n\to\infty]{d} Z$$

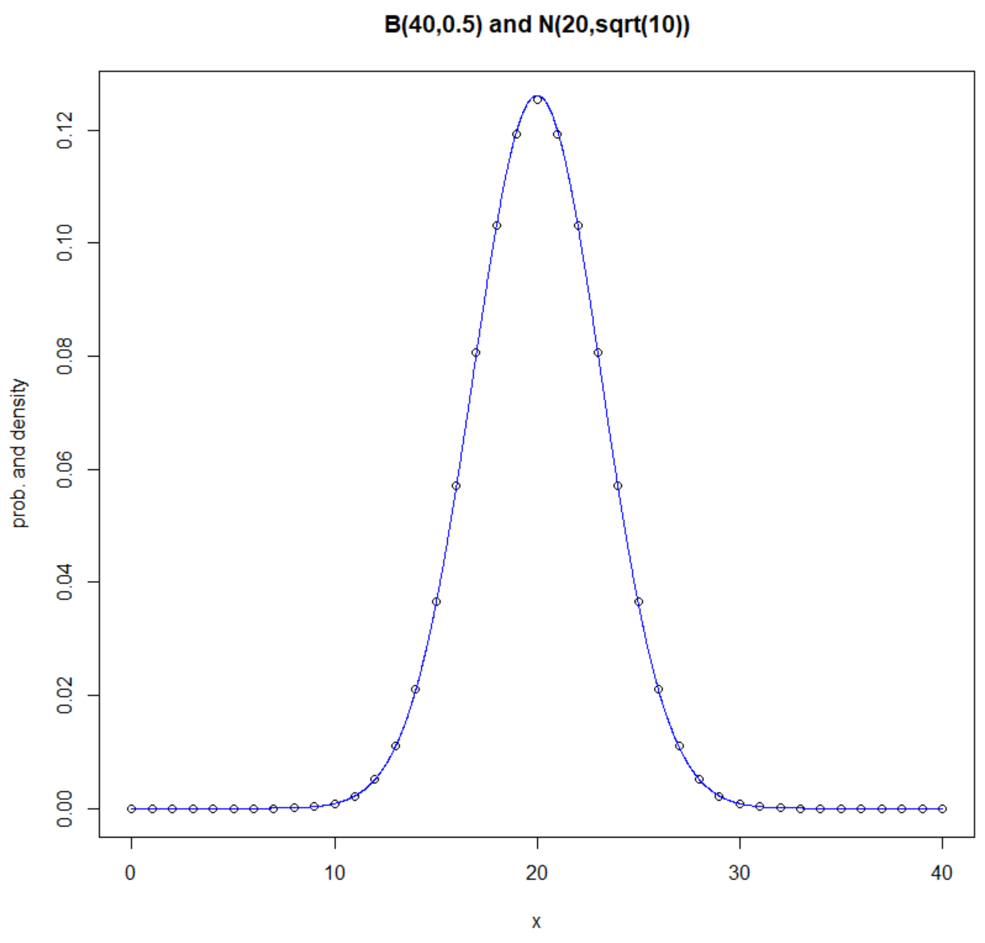
where we can divide the numerator and denominator by n to obtain

$$\frac{\widehat{p}_n - p}{\sqrt{\frac{p(1-p)}{n}}} \xrightarrow[n\to\infty]{d} Z$$

and then we can substitute the p's in the denominator by $\widehat{p}_n$ if needed.

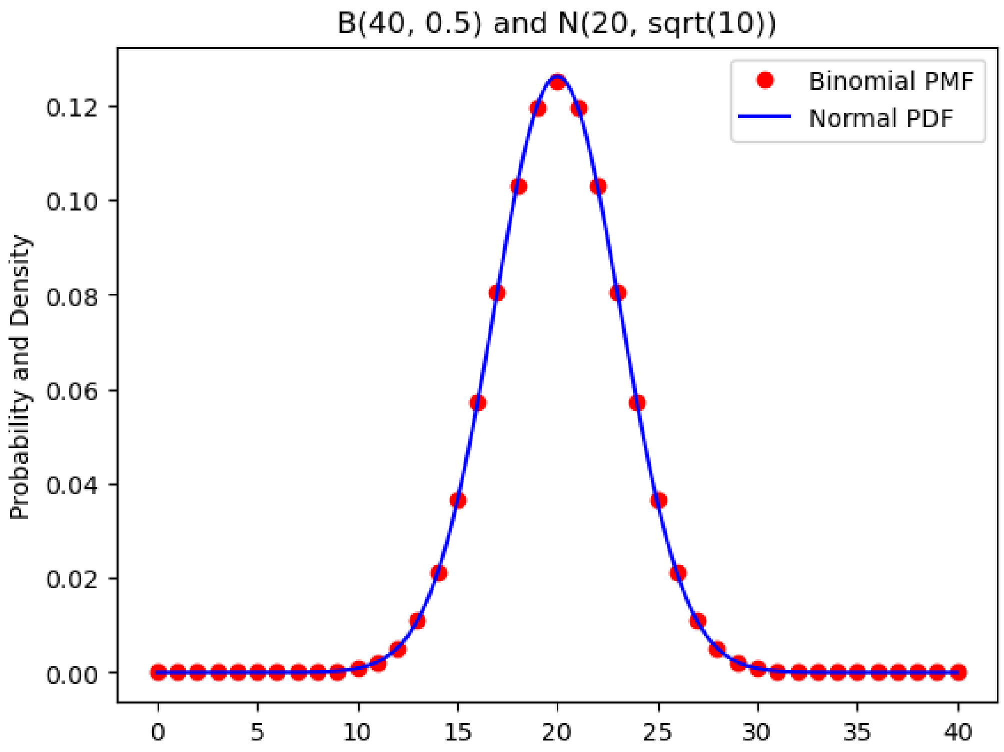
$$\frac{\widehat{p}_n - p}{\sqrt{\frac{\widehat{p}_n(1-\widehat{p}_n)}{n}}} \xrightarrow[n\to\infty]{d} Z$$

```r
x = 0:40

# Plot the binomial distribution for the given parameters
# Compute the probability mass function of the binomial distribution
plot(x, dbinom(x, size=40, prob=0.5),
     ylab="prob. and density",
     main="B(40,0.5) and N(20,sqrt(10))")

# Create a sequence from 0 to 40 with a step size of 0.01
t = seq(0, 40, by=0.01)

# Add a curve for the normal distribution as a line on the plot
# type="l" specifies that a line plot should be drawn
points(t, dnorm(t, mean=20, sd=sqrt(10)), type="l", col="blue")
```



In Python:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom, norm

# Binomial distribution
x = np.arange(0, 41)
binomial = binom.pmf(x, 40, 0.5)

# Normal distribution
t = np.arange(0, 40.01, 0.01)
normal = norm.pdf(t, 20, np.sqrt(10))

# Plotting
plt.plot(x, binomial, 'o', color='red', label='Binomial PMF')
plt.ylabel('Probability and Density')
plt.title('B(40, 0.5) and N(20, sqrt(10))')

plt.plot(t, normal, '-', color='blue', label='Normal PDF')

plt.legend()
plt.show()
```



Confidence Intervals

A confidence interval (CI) is a range of estimates for a parameter. Its confidence level represents the long-term proportion of CIs that contain the true value of the parameter.

Approximate $(1-\alpha)100\%$ CI on μ (the mean):

$$\left[\overline{x}_n \mp z_{\frac{\alpha}{2}} \frac{s_n}{\sqrt{n}}\right]$$

Approximate $(1-\alpha)100\%$ CI on p (a proportion):

$$\widehat{p}_n \mp z_{\frac{\alpha}{2}} \sqrt{\frac{\widehat{p}_n\left(1-\widehat{p}_n\right)}{n}}$$

where $z_{\frac{\alpha}{2}} = \texttt{qnorm}(1-\alpha/2)$.

Statistical validation techniques

Goodness of fit tests

Kolmogorov-Smirnoff Goodness-of-Fit test (continuous data, given F_0)

TEST. $H_0 : F = F_0$ vs. $H_1 : F \neq F_0$

TEST STATISTIC. $D = \max_x |\widehat{F}_n(x) - F_0(x)|$, with $\widehat{F}_n$ empirical cdf.

The distribution of D (under H_0) does not depend on F_0 (continuous).

```
data = c(.56,.35,.89,.81,.36,.19,.65,.42,.12,.05)
ks.test(data,"punif")$p.value
```

```
[1] 0.8472806
```

In Python:

```
from scipy import stats

data = [.56, .35, .89, .81, .36, .19, .65, .42, .12, .05]

# Perform the Kolmogorov-Smirnov test for a uniform distribution
ks_statistic, p_value = stats.kstest(data, 'uniform')

# Print the statistics and the p-value
print("ks_statistic:",ks_statistic)
print("P-value:", p_value)
```

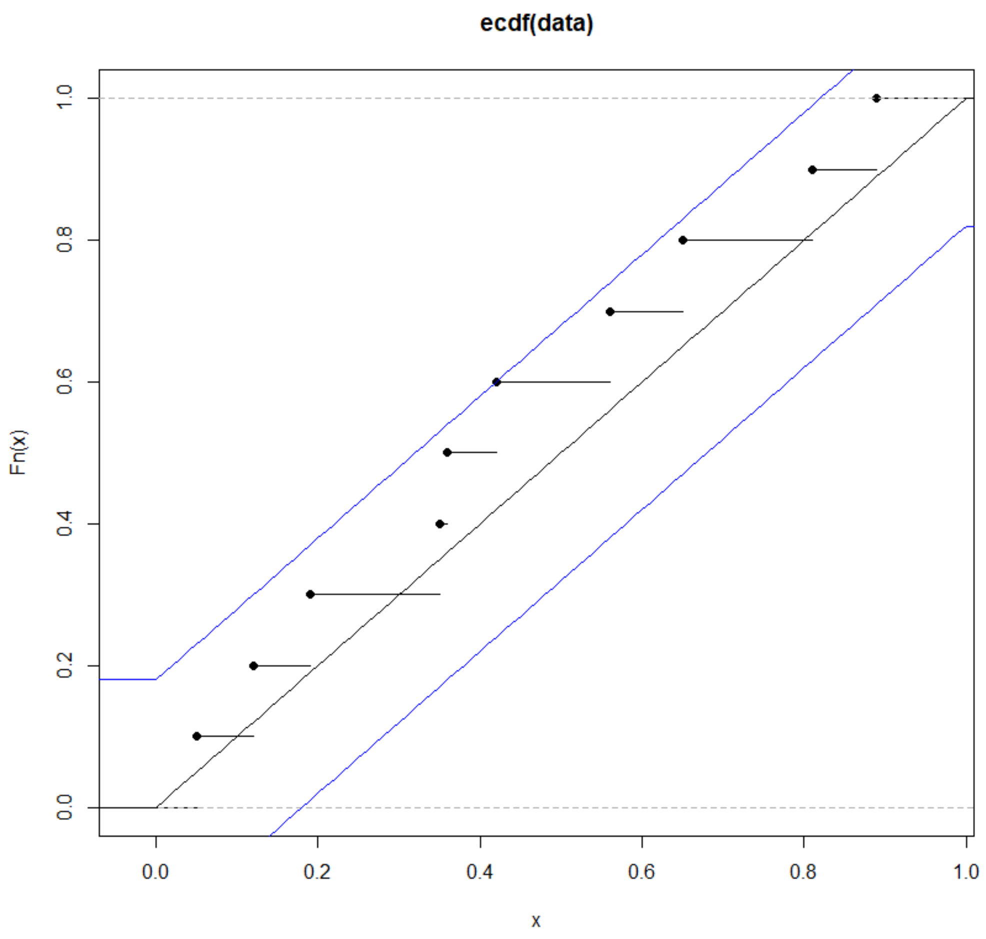
```
ks_statistic: 0.18
P-value: 0.8472806273219462
```

The **p-value** of a statistical test is the smallest significance level that would lead to rejection of H_0.

It is the observed level of significance.

```
# Plot the empirical cumulative distribution function (ECDF) of the data
plot(ecdf(data))

aux = seq(-.1, 1.1, 0.01)

# Overlay a uniform cumulative distribution function (CDF) line on the plot
points(aux, punif(aux), type = "l")

# Perform the Kolmogorov-Smirnov test to compare the data
# with a uniform distribution
# Extract the test statistic (D) from the test result
D = ks.test(data, "punif")$statistic
```

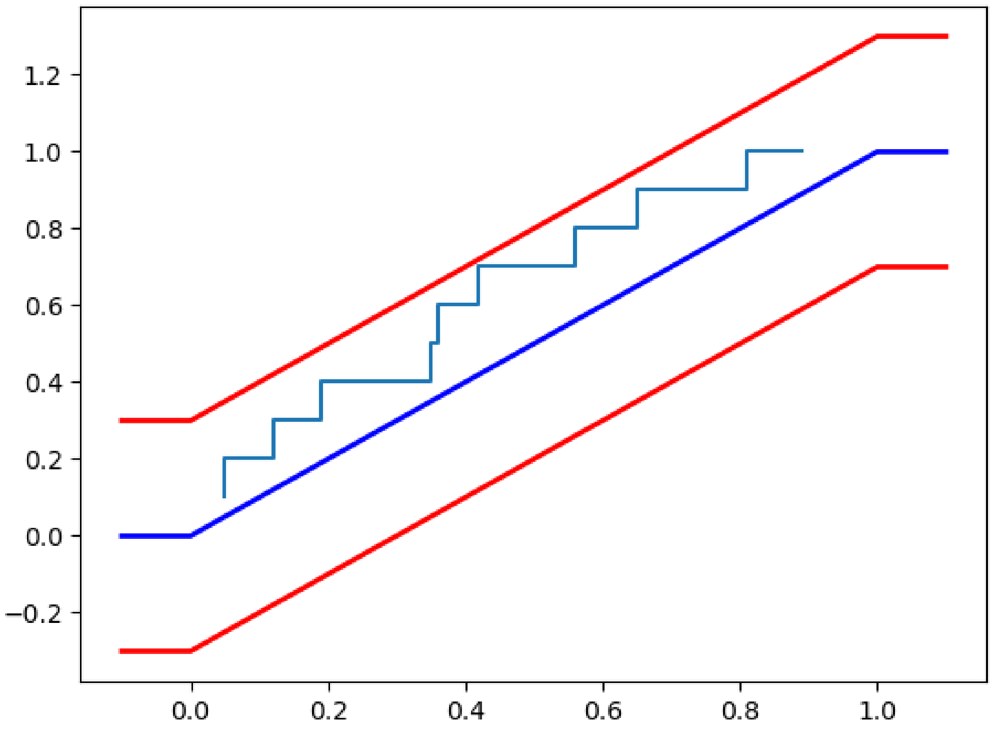
```r
# Add a line representing the upper and lower bounds of the confidence band
# for the uniform CDF
points(aux, punif(aux) + D, type = "l", col = "red")
points(aux, punif(aux) - D, type = "l", col = "red")
```



In Python:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform, ks_2samp

data = [.56, .35, .89, .81, .36, .19, .65, .42, .12, .05]

# Plot the ECDF of the data
plt.step(np.sort(data), np.arange(1, len(data)+1) / len(data))

# Generate a sequence of values and plot the uniform CDF
aux = np.arange(-0.1, 1.1, 0.01)
plt.plot(aux, uniform.cdf(aux), 'b-', lw=2)

# Perform the Kolmogorov-Smirnov test
D, p_value = ks_2samp(data, uniform.rvs(size=len(data)))
```

```
# Plot the uniform CDF + D and uniform CDF - D
plt.plot(aux, uniform.cdf(aux) + D, 'r-', lw=2)
plt.plot(aux, uniform.cdf(aux) - D, 'r-', lw=2)

plt.show()
```



If we want to calculate proper confidence intervals it is necessary to apply the Dvoretzky-Kiefer-Wolfowitz (DKW) inequality. These bands have widths that decrease as the sample size increases and may differ across the domain.

```
data = c(.56,.35,.89,.81,.36,.19,.65,.42,.12,.05)
n = length(data)  # Sample size
alpha = 0.05      # Significance level
epsilon = sqrt(-log(alpha / 2) / (2 * n))  # Half-width of the DKW band

aux = seq(-.1, 1.1, 0.01)
# Plot the ECDF with confidence bands
plot(ecdf(data), main = "ECDF with DKW Confidence Bands", ylim = c(-0.1, 1.1))
lines(aux, punif(aux) + epsilon, col = "blue", lty = 2)  # Upper band
lines(aux, punif(aux) - epsilon, col = "blue", lty = 2)  # Lower band
points(aux, punif(aux), type = "l")                      # Theoretical CDF
```

Chi-Square Goodness-of-Fit test (discrete data, given F_0)

TEST. $H_0 : F = F_0$ vs. $H_1 : F \neq F_0$

TEST STATISTIC.

$$X^2 = \sum_{i=1}^{k} \frac{(N_i - np_i)^2}{np_i},$$

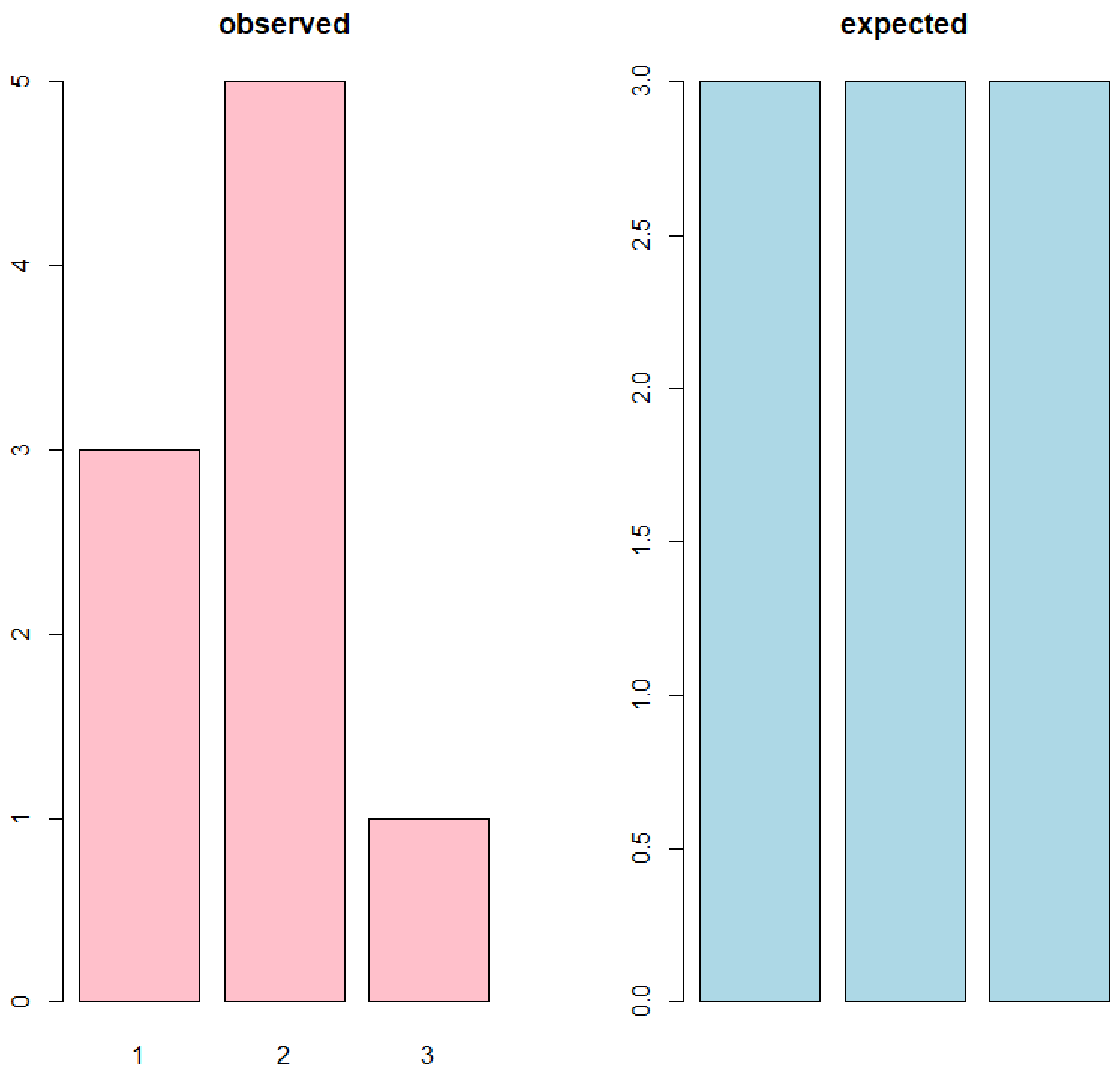
k levels of the variable, N_i observed counts, and np_i expected counts.

Under $H_0 : X \sim \chi^2_{k-1}$.

```r
observed = table(c(3,2,1,2,2,1,2,1,2))
observed
probs = rep(1/3,3)
chisq.test(x=observed,p=probs)
```

```
1 2 3
3 5 1

    Chi-squared test for given probabilities

data:  observed
X-squared = 2.6667, df = 2, p-value = 0.2636
```

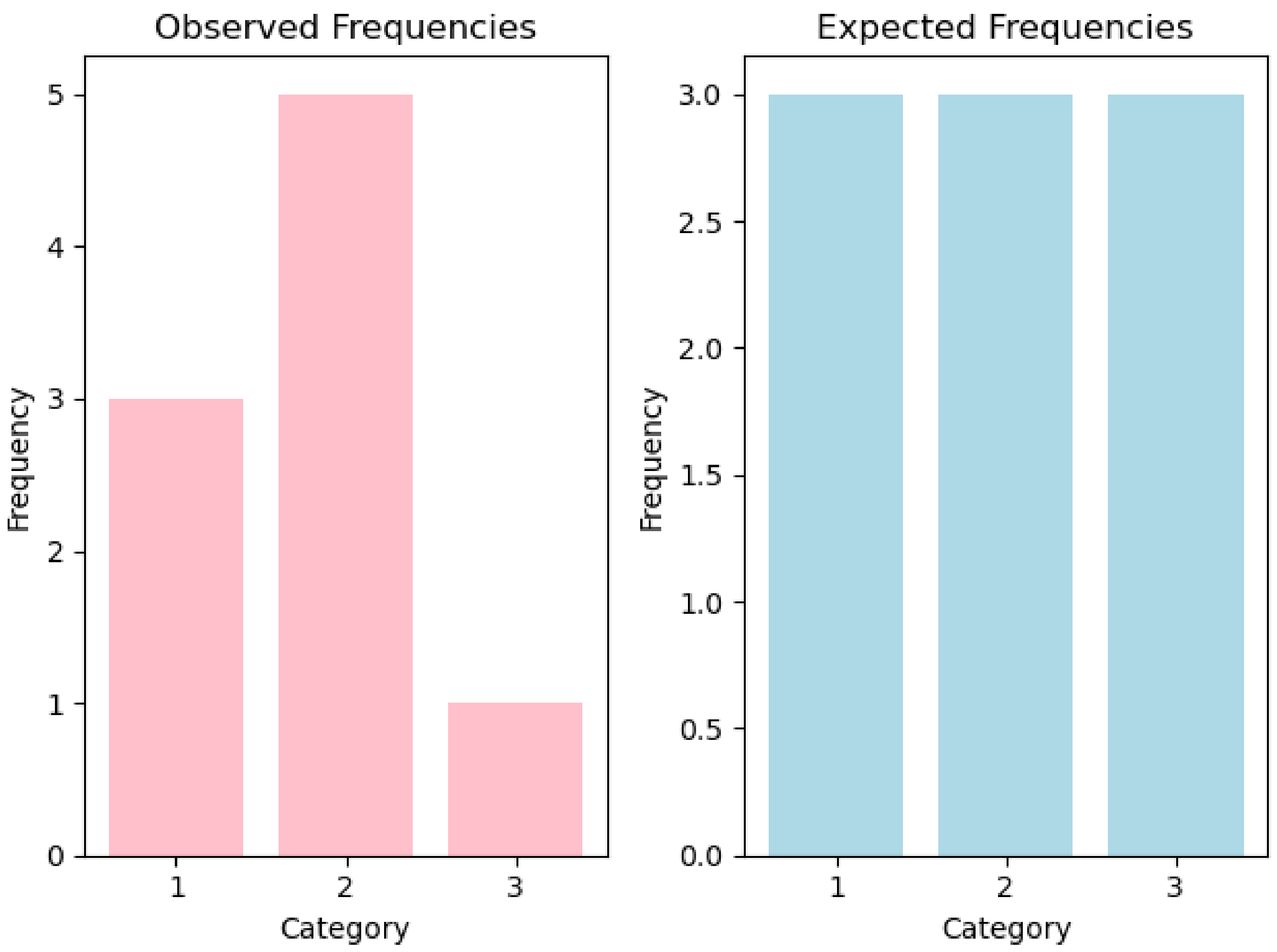
```r
n = sum(observed)
par(mfrow=c(1,2))
barplot(observed,main="observed", col="pink")
barplot(probs*n,main="expected", col="lightblue")
```


observed
expected
0
1
2
3
4
5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1
2
3

In Python:

```
import pandas as pd
import numpy as np
from scipy.stats import chisquare
import matplotlib.pyplot as plt

# Creating the observed frequencies
data = [3, 2, 1, 2, 2, 1, 2, 1, 2]
observed = pd.Series(data).value_counts().sort_index()

# Defining the probabilities
probs = np.repeat(1/3, 3)

# Performing the chi-squared test
chi_square_test, p_value = chisquare(f_obs=observed, f_exp=probs * sum(observed))

# Printing chi-square test result
print("p-value:", p_value)

# Plotting the observed frequencies
plt.subplot(1, 2, 1)
plt.bar(observed.index, observed, color='pink')
plt.title('Observed Frequencies')
plt.xlabel('Category')
plt.ylabel('Frequency')

# Plotting the expected frequencies
plt.subplot(1, 2, 2)
plt.bar(observed.index, probs * sum(observed), color='lightblue')
plt.title('Expected Frequencies')
plt.xlabel('Category')
plt.ylabel('Frequency')

# Displaying the plots
plt.tight_layout()
plt.show()
```

```
p-value: 0.26359713811572677
```



Statistical validation (Goodness-of-Fit tests, unspecified parameters)

For specific distributions (e.g. normal), we can use specific tests (e.g. Shapiro-Wilk normality test, `shapiro.test`) or graphical tools (e.g. QQplots, `qqnorm`).

When some parameters are not specified:

- Continuous distribution, use the KS test and simulate to estimate the critical value or p-value.
- Discrete distribution, use the Chi-square test, and subtract the number of estimated parameters from the degrees of freedom.

Randomness tests

Wald-Wolfowitz runs a randomness test for continuous data.

TEST. H_0 : random pattern of observations w.r.t. threshold (median) vs.

H_1 : pattern of observations is not random.

TEST STATISTIC.

A run of a sequence is a nonempty segment of the sequence consisting of adjacent equal elements. For example, the $N = 22$ elements sequence:

$$+ + + + - - - + + + - - + + + + + + - - - -$$

consists of 6 runs, 3 of which consist of $+$ and the others of $-$.

Let us denote U as the number of runs (group of consecutive values above or below the threshold: median).

Under the null hypothesis, the number of runs U in a sequence of N elements is a random variable whose conditional distribution given the observation of N_+ positive values and N_- negative values ($N = N_+ + N_-$) is approximately normal, with:

Mean:

$$\mu = \frac{2N_+N_-}{N} + 1$$

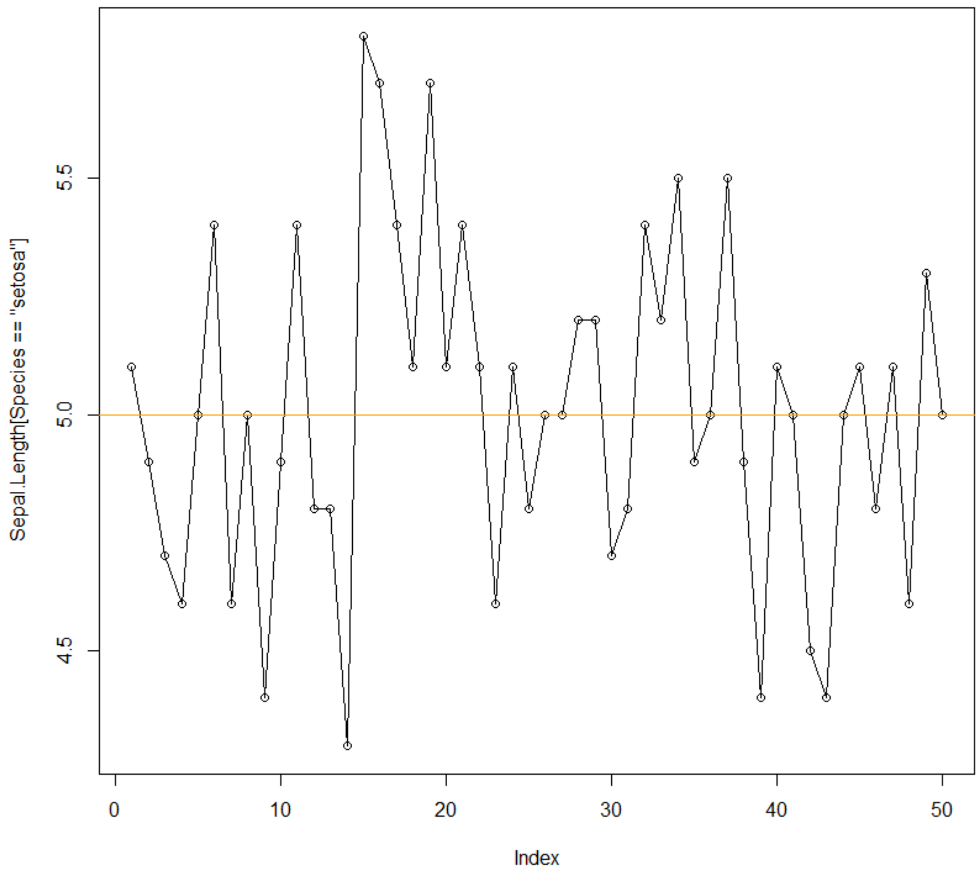
Variance:

$$\sigma^2 = \frac{2N_+N_-(2N_+N_- - N)}{N^2(N-1)} = \frac{(\mu-1)(\mu-2)}{N-1}$$

```
if (!require(randtests)) install.packages("randtests")

library(randtests)
data("iris")
runs.test(iris$Sepal.Length[iris$Species=="setosa"])$p.value
```

```
[1] 0.7428503
```

```
attach(iris)
plot(Sepal.Length[Species=="setosa"],type="l")
points(Sepal.Length[Species=="setosa"])
abline(h=median(Sepal.Length[Species=="setosa"]),col="orange")
detach(iris)
```



Independence tests

The Ljung-Box test is a statistical method used to determine whether a set of observations exhibits serial correlation (dependence between observations at different time lags). It is commonly applied to residuals of a time series model to assess its adequacy.

HYPOTHESES:

H_0 : The observations are independent (no serial correlation).

H_1 : The observations are not independent (autocorrelations at some lags are nonzero).

The Ljung-Box test statistic Q is given by:

$$Q = n(n+2) \sum_{k=1}^{h} \frac{\widehat{\rho}_k^2}{n-k},$$

where:

- n: Total number of observations in the sample.

- h: Number of lags being tested for autocorrelation.
- $\widehat{\rho}_k$: Sample autocorrelation at lag k.
- $n-k$: Adjustment for degrees of freedom as k increases.

The sample autocorrelation at lag k, denoted by $\widehat{\rho}_k$, is calculated using the formula:

$$\widehat{\rho}_k = \frac{\sum_{t=k+1}^{n}(x_t - \bar{x})(x_{t-k} - \bar{x})}{\sum_{t=1}^{n}(x_t - \bar{x})^2},$$

where:

- x_t is the observation at time t,
- $\bar{x}$ is the sample mean, calculated as $\bar{x} = \frac{1}{n}\sum_{t=1}^{n} x_t$,
- n is the total number of observations,
- $t-k$ refers to the time point lagged by k.

The numerator computes the covariance between the observations x_t and x_{t-k} separated by k lags, while the denominator normalizes this value by the total variance of the observations.

Then, the squared sample autocorrelation, used in the Ljung-Box test statistic ensures that only the magnitude of the autocorrelation contributes to the test, irrespective of its sign.

The test statistic aggregates the squared sample autocorrelations ($\widehat{\rho}_k^2$), scaled to account for sample size and lag adjustments. The scaling factor $n(n+2)/(n-k)$ ensures that the test is robust to varying sample sizes and the inclusion of multiple lags.

If the null hypothesis (H_0) is true indicating no serial correlationthe test statistic Q approximately follows a chi-squared (χ^2) distribution with h degrees of freedom:

$$Q \sim \chi_h^2.$$

Steps to Perform the Ljung-Box Test:

1. Compute the sample autocorrelations $\widehat{\rho}_1, \widehat{\rho}_2, \ldots, \widehat{\rho}_h$ for the desired number of lags h.
2. Plug these values into the formula for Q to calculate the test statistic.
3. Determine the critical value from the chi-squared distribution with h degrees of freedom.

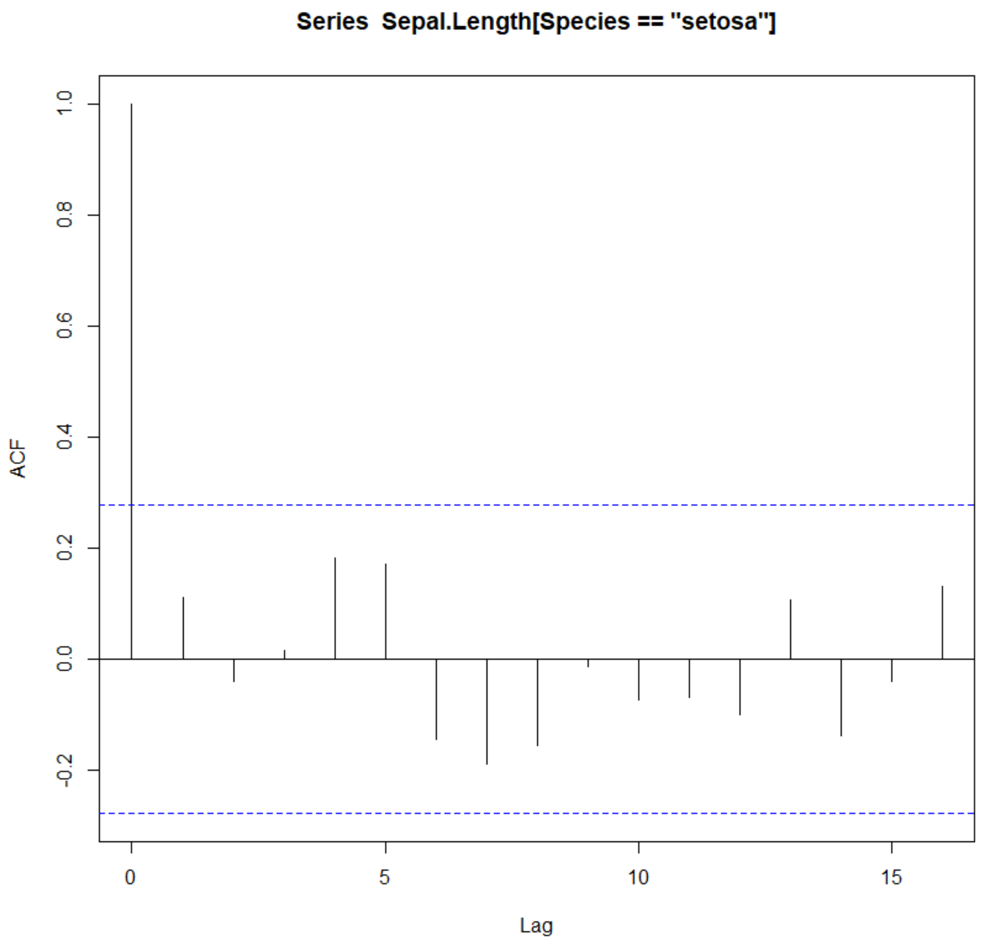
4. Make a decision based on the comparison of Q with the critical value.

The Ljung-Box test is particularly useful in model diagnostics for time series, such as checking whether the residuals of a fitted model behave like white noise (i.e., are uncorrelated).

```
Box.test(iris$Sepal.Length[iris$Species=="setosa"],
         lag=10,type="Ljung")
```

```
    Box-Ljung test

data:  iris$Sepal.Length[iris$Species == "setosa"]
X-squared = 9.5803, df = 10, p-value = 0.4781
```

```
acf(Sepal.Length[Species=="setosa"])
```



(Pseudo)random number generation

DILBERT By Scott Adams
TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

The first matter in simulation is to obtain random numbers in the $(0,1)$ interval (independent random variables with a $U(0,1)$ distribution).

Is it possible to build such random numbers?

If fair coins exist, it is so.

Consider a sequence of tosses of a fair coin $x_1, x_2, x_3, \dots$ where

$$x_i = \begin{cases} 1 & \text{if} \quad \text{head on } i\text{-th toss} \\ 0 & \text{if} \quad \text{tail on } i\text{-th toss} \end{cases}$$

We can interpret the sequence as the binary expansion of a number in the $[0,1]$ interval,

$$\sum_{i=1}^{\infty} \frac{x_i}{2^i}.$$

The first step in conducting a simulation is to generate random numbers within the interval $(0,1)$. These random numbers should be independent and uniformly distributed, which means they follow a $U(0,1)$ distribution.

A fundamental question arises: Is it possible to generate such random numbers?

The answer is affirmative, provided we have access to a *fair coin.* A fair coin is one where the probability of landing heads is exactly 0.5 and the probability of landing tails is also 0.5.

This property ensures that each toss of the coin is independent of others and equally likely to produce a head or a tail.

To construct random numbers in the $[0,1)$ interval using a fair coin, we proceed as follows:

1. Define a Sequence of Coin Tosses: Consider a sequence of independent tosses of the fair coin. Denote the outcomes of the tosses as $x_1, x_2, x_3, \ldots$, where each x_i takes a value of 1 (if the outcome is heads) or 0 (if the outcome is tails). Formally, we define:

$$x_i = \begin{cases} 1 & \text{if the } i\text{-th toss results in heads,} \\ 0 & \text{if the } i\text{-th toss results in tails.} \end{cases}$$

2. Binary Representation: Using the outcomes of the coin tosses, we interpret the sequence $x_1, x_2, x_3, \ldots$ as the binary expansion of a real number in the interval $[0, 1)$.

   The number is given by the infinite series:

$$r = \sum_{i=1}^{\infty} \frac{x_i}{2^i}.$$

   In this representation:

   - x_1 determines the first binary digit after the decimal point, contributing $\frac{x_1}{2}$,
   - x_2 determines the second binary digit, contributing $\frac{x_2}{4}$,
   - and so on.

3. Uniformity of the Distribution: By the properties of the fair coin and the independence of each toss, the resulting random variable r is uniformly distributed over the interval $[0, 1)$.

   Each binary digit x_i corresponds to an independent Bernoulli random variable with parameter 0.5, and the sum converges to a uniformly distributed random number.

This method demonstrates that, under the assumption of a truly fair coin, it is indeed possible to generate independent random numbers uniformly distributed in the $(0, 1)$ interval.

```r
# Set the number of Monte Carlo simulations to 200
MC = 200

# Create a vector to store the results of the simulations,
# initializing it with length MC
simul.bin = vector(length=MC)

set.seed(1)

# Loop through the Monte Carlo simulations
for(i in 1:MC){
  # Generate a sample of 100 random values from the set {0, 1} with replacement
  # This simulates 100 tosses of a fair coin,
  # where 0 represents tails and 1 represents heads
  samp = sample(c(0,1), replace=TRUE, size=100)

  # Compute the binary expansion value of the generated sequence
  # Each sample value (0 or 1) is divided by 2 raised to
  # the power of its position index (1 to 100)

  # The sum of these values represents the random number in the interval [0, 1)
  simul.bin[i] = sum(samp / 2^(1:100))
}

head(simul.bin)

ks.test(simul.bin,punif)$p.value
```

```
[1] 0.284208307 0.879024303 0.861395814 0.014649777 0.086113814 0.001975348

[1] 0.9731958
```

In Python:

```python
import numpy as np
from scipy.stats import ks_2samp

MC = 100
simul_bin = np.zeros(MC)
np.random.seed(1)

for i in range(MC):
    samp = np.random.choice([0, 1], size=20, replace=True)
    simul_bin[i] = np.sum(samp / 2 ** np.arange(1, 21))

print("p-value: ",ks_2samp(simul_bin, np.random.uniform(0, 1, MC)).pvalue)
```

```
p-value:  0.9084105017744525
```

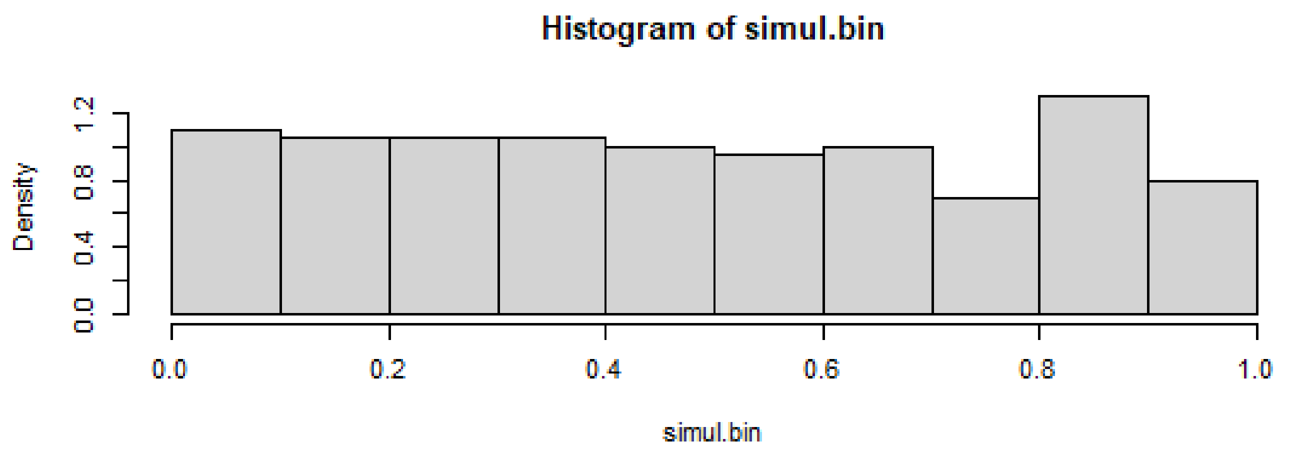
```
runs.test(simul.bin)$p.value
```

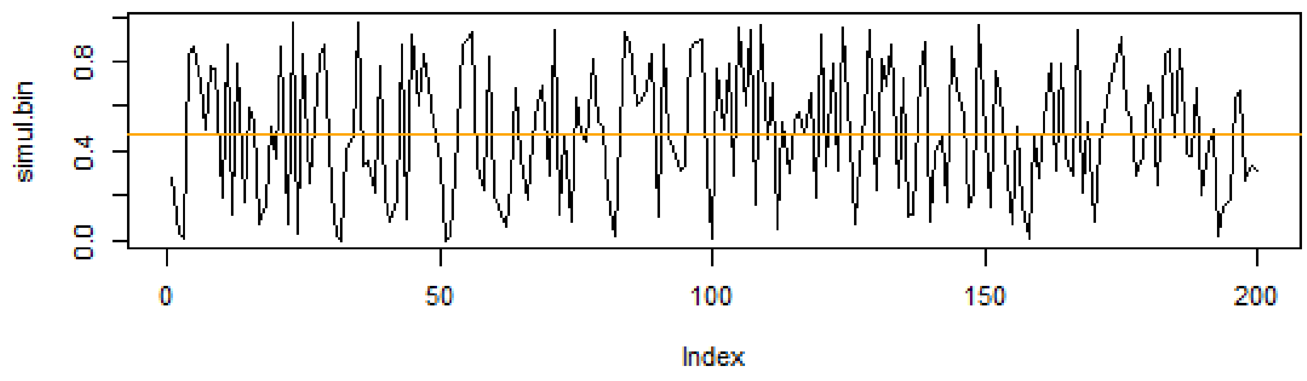
```
[1] 0.3949537
```

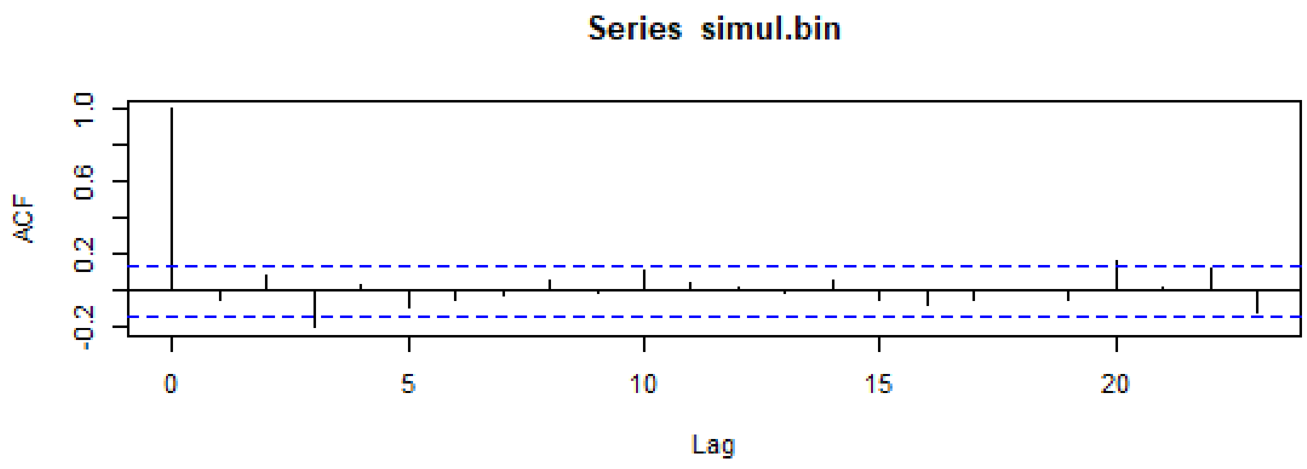
```
Box.test(simul.bin,lag=10,type="Ljung")$p.value
```

```
[1] 0.1061696
```

```r
par(mfrow=c(3, 1))

hist(simul.bin,probability=T)
plot(simul.bin,type="l")
abline(h=median(simul.bin),col="orange")
acf(simul.bin)
```







Pseudorandom numbers

The numbers we work with have been deterministically generated, so they are not truly random.

A **multiplicative congruential generator** works as follows:

1. Set m (prime number) large.
2. Set $a < m$.
3. Set seed x_0 (initial value).
4. Set $x_n \equiv ax_{n-1}(m)$, which means that x_n is the remainder of ax_{n-1}/m.
5. For every n return x_n/m.

```
MC = 1000
seed = 1
m = 2^35-31
a = 5^5
simul = vector(length=MC)
simul[1] = (a*seed)%%m

for(i in 2:MC) simul[i] = (a*simul[i-1])%%m

simul = simul/m

ks.test(simul,"punif")$p.value
```

```
[1] 0.9174393
```

In Python:

```
import numpy as np
from scipy.stats import kstest

MC = 1000
seed = 1
m = 2**35 - 31
a = 5**5
simul = np.zeros(MC)
simul[0] = (a*seed) % m
for i in range(1, MC):
    simul[i] = (a*simul[i-1]) % m
simul = simul / m

p_value = kstest(simul, 'uniform').pvalue
print("p-value:", p_value)
```

```
p-value: 0.9119786743096102
```

In Rcpp:

```
library(Rcpp)

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector linearCongruentialGenerator(int MC, double seed) {
    double m = pow(2, 35) - 31;
    double a = pow(5, 5);
    NumericVector simul(MC);

    simul[0] = fmod(a * seed, m);

    for (int i = 1; i < MC; ++i) {
        simul[i] = fmod(a * simul[i - 1], m);
    }

    for (int i = 0; i < MC; ++i) {
        simul[i] = simul[i] / m;
    }

    return simul;
}
')

result = linearCongruentialGenerator(100, 666)
result
```

```
runs.test(simul)$p.value
```

```
[1] 0.3113298
```

```
Box.test(simul,lag=10,type="Ljung")$p.value
```

```
[1] 0.5634981
```

Replicability and efficiency vs. true randomness

The methods that we use do not produce truly random numbers, but they have two properties that are essential in statistics:

- We can regenerate our results. By fixing a seed (initial value) we always obtain the same sequence of (pseudo)random numbers.

- They are efficient: fast and simple to implement.

If you are interested in R's algorithms for random number generation, type `help(RNG)`.

```
MC = 1000000
ptm = proc.time()
set.seed(1)

simul.unif = runif(MC)
proc.time() - ptm
```

```
   user  system elapsed
   0.03    0.00    0.07
```

```
ptm = proc.time()
seed = 1
m = 2^35-31
a = 5^5

simul = vector(length=MC)
simul[1] = (a*seed)%%m

for(i in 2:MC) simul[i] = (a*simul[i-1])%%m
simul = simul/m
proc.time() - ptm
```

```
   user  system elapsed
   0.33    0.00    0.45
```

Approximation of probabilities and volumes

We can approximate the volume of some given compact set $C \subset \mathbb{R}^d$ by considering a d-dimensional box B containing it, $C \subset B$.

We can simulate points in the box at random, then the volume of C is approximately the fraction of points in C times the volume of B.

Consider $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$

1. Set *inside* $= 0$.
2. Repeat steps 3 to 4 n times.
3. Generate U_1 a random number in $[a_1, b_1]$, U_2 a random number in $[a_2, b_2], \ldots,$ U_d a random number in $[a_d, b_d]$.
4. If $(U_1, \ldots, U_d) \in C$ then $\mathit{inside} = \mathit{inside} + 1$
5. Set $\mathit{vol} = (\mathit{inside}/n) \cdot (b_1 - a_1) \cdots (b_d - a_d)$ and stop.

Approximate the area of a circle of radius 1

The following R code aims to approximate the area of a quarter-circle with radius 1. A Monte Carlo simulation method is used for the approximation.

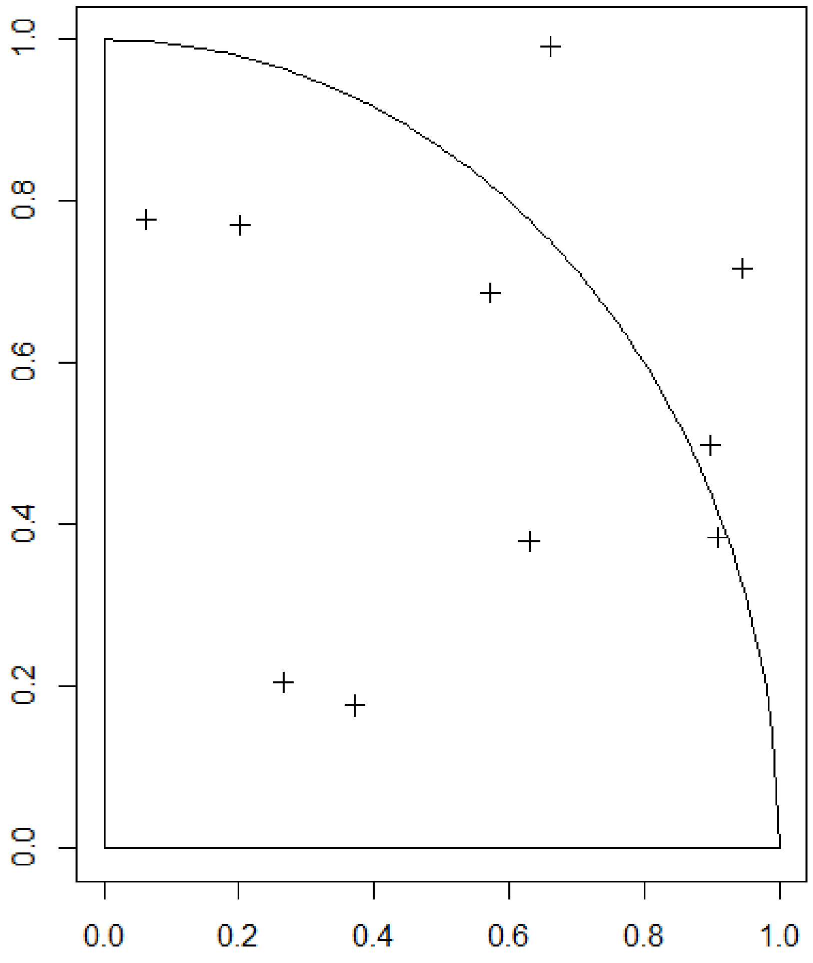
```r
# Define a sequence of x-values from 0 to 1 with increments of 0.01.
aux = seq(0,1,.01)

# Plot the upper edge of a quarter-circle of radius 1 in the first quadrant.
# The formula sqrt(1 - aux^2) represents the y-coordinates
# of the circle's edge.
plot(aux, sqrt(1-aux^2), type="l", xlab="", ylab="")

# Add a triangle to the plot by drawing lines connecting
# the points (1,0), (0,0), and (0,1).
lines(c(1,0,0), c(0,0,1))

# Set the number of Monte Carlo points to generate.
MC = 10

set.seed(1)
```

```
# Generate MC random points in the unit square [0,1] x [0,1].
# These points are plotted with a "+" symbol (pch=3) on the existing plot.
points(runif(MC), runif(MC), pch=3)
```

The aux sequence is used to discretize the x-axis for plotting the curve of the quarter-circle. The y-values are calculated as $\sqrt{1-x^2}$, which corresponds to the equation of a circle centered at the origin with radius 1.

The random points generated in the unit square can be used to estimate the area of the quarter-circle. By counting how many points fall inside the quarter-circle and dividing by the total number of points, we can approximate the fraction of the square's area occupied by the quarter-circle.

This fraction, multiplied by 4, provides an estimate for the area of the full circle (which is π for a radius of 1).



```r
MC = 1000
set.seed(1)

x = runif(MC)
y = runif(MC)
sum(x^2+y^2<=1)/MC*4
```

```
[1] 3.148
```

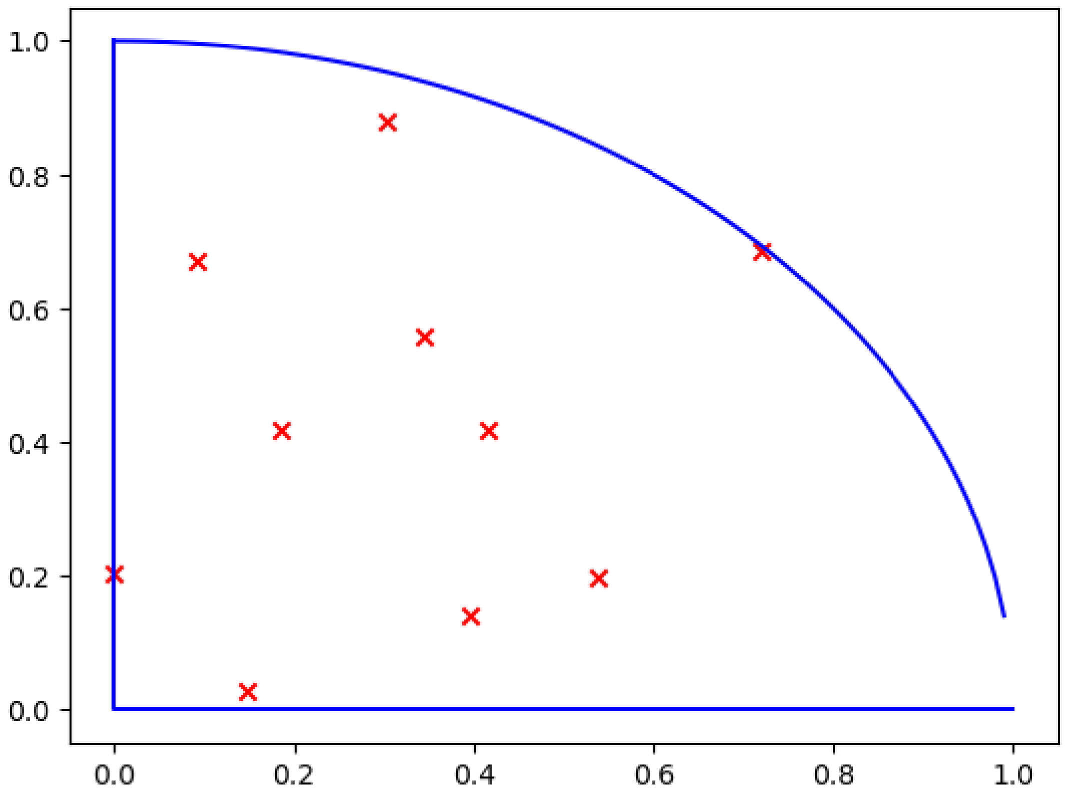
```r
prop.test(sum(x^2+y^2<=1), n=MC, conf.level=0.95)$conf.int*4
```

```
[1] 3.040123 3.246918
attr(,"conf.level")
[1] 0.95
```

In Python

```python
import numpy as np
import matplotlib.pyplot as plt

# Creating the auxiliary sequence
aux = np.arange(0, 1, 0.01)

# Plotting the quarter circle
plt.plot(aux, np.sqrt(1 - aux**2), 'b-') # 'b-' specifies a blue line
plt.xlabel('')
plt.ylabel('')

# Adding lines to form a square
plt.plot([1, 0, 0], [0, 0, 1], 'b-')

# Setting the seed for reproducibility
np.random.seed(1)

# First Monte Carlo simulation with 10 points
MC = 10
points_x = np.random.uniform(0, 1, MC)
points_y = np.random.uniform(0, 1, MC)
plt.scatter(points_x, points_y, c='red', marker='x') # 'x' marks for the points

# Second Monte Carlo simulation with 1000 points
MC = 1000
x = np.random.uniform(0, 1, MC)
y = np.random.uniform(0, 1, MC)
pi_estimate = np.sum(x**2 + y**2 <= 1) / MC * 4

plt.show() # To display the plot

print("Estimated value of p:", pi_estimate)
```

```
Estimated value of p: 3.08
```



Approximation of probabilities by simulation

For each random experiment, we can simulate a random number in the interval $(0, 1)$ and assume that an event with an assigned probability p occurs if the simulated number is less than p.

The expression `runif(1) < p` is a logical value (`TRUE` or `FALSE`), which is interpreted as the numerical value 1 (`TRUE`) or 0 (`FALSE`) when used in calculations.

The intersection of two events occurs whenever both events occur simultaneously. This is represented by the logical operator `AND` and corresponds to the product of the events' numerical values.

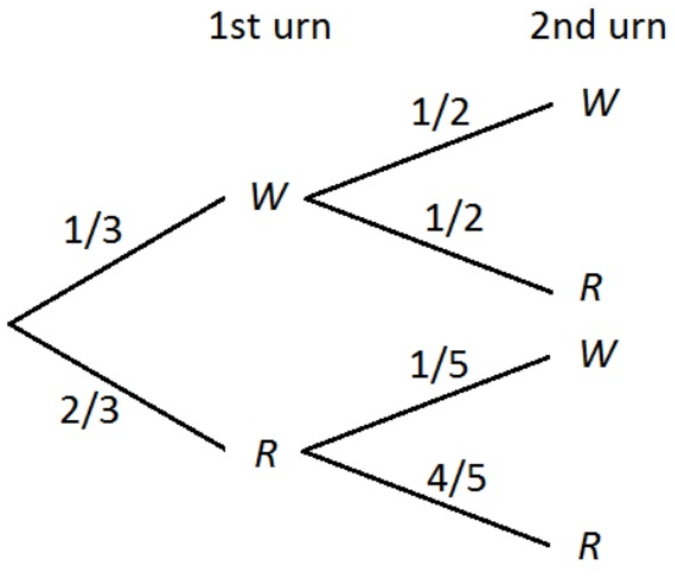
The union of two events occurs whenever at least one of the events occurs. This is represented by the logical operator `OR` and corresponds to the maximum of the events' numerical values.

Approximation of probabilities by simulation (example)

An urn contains 1 white ball and 2 red balls. One ball is taken from the urn at random.

If it is a white ball, it is returned to the urn together with another white ball. If it is a red ball, it is returned together with the other 2 red balls.

In the second stage, another ball is randomly taken from the urn.



Events

- $W_1 \equiv$ first ball is white
- $R_1 \equiv$ first ball is red
- $W_2 \equiv$ second ball is white

- $R_2 \equiv$ second ball is red

Probabilities

- $P(W_1) = 1/3$; $P(R_1) = 2/3$
- $P(W_2|W_1) = 1/2$; $P(R_2|W_1) = 1/2$
- $P(W_2|R_1) = 1/5$; $P(R_2|R_1) = 4/5$

Total probability rule

What is the probability that the second ball extracted from the urn is red?

$$P(R_2) = P(W_1)P(R_2|W_1) + P(R_1)P(R_2|R_1) = \frac{1}{3} \cdot \frac{1}{2} + \frac{2}{3} \cdot \frac{4}{5} = 0.7.$$

Bayes' rule

If the second ball extracted from the urn is red, what is the probability that the first ball is also red?

$$P(R_1|R_2) = \frac{P(R_1)P(R_2|R_1)}{P(R_2)} = \frac{\frac{2}{3} \cdot \frac{4}{5}}{0.7} = 0.762.$$

```r
MC = 1000
set.seed(1)
ball.1 = runif(MC)
ball.2 = runif(MC)
R1 = (ball.1 < 2/3)
R2 = (R1&(ball.2<4/5))|((!R1)&(ball.2<1/2))
p1 = sum(R2)/MC
p1
```

```
[1] 0.703
```

- `MC = 1000`: The number of Monte Carlo iterations is set to 1000. This determines how many times the random experiment will be simulated.
- `ball.1 = runif(MC)` and `ball.2 = runif(MC)`: Two sets of random numbers are generated, each uniformly distributed in the interval $[0, 1]$, representing the outcomes of the first and second draws.

- `R1 = (ball.1 < 2/3)`: The first ball is red if the first random number is less than $\frac{2}{3}$, as the probability of selecting a red ball on the first draw is $P(R_1) = \frac{2}{3}$.
- `R2 = (R1 &(ball.2<4/5))|((!R1)&(ball.2<1/2))`: The second ball is red based on the following conditions:
  - If the first ball was red (`R1 = TRUE`), the probability of the second ball being red is $P(R_2|R_1) = \frac{4}{5}$.
  - If the first ball was white (`R1 = FALSE`), the probability of the second ball being red is $P(R_2|W_1) = \frac{1}{2}$.

  The logical expression combines these conditions to determine whether the second ball is red.
- `p1 = sum(R2)/MC`: The proportion of simulations where the second ball is red is calculated as the estimate for $P(R_2)$.
- `p1`: The resulting value is printed, showing the simulated probability $P(R_2) \approx 0.703$, which is close to the theoretical value 0.7.

```
p2 = sum(R1&R2)/sum(R2)
p2
```

```
[1] 0.7496444
```

- `p2 = sum(R1&R2)/sum(R2)`: This calculates the conditional probability $P(R_1|R_2)$. The numerator counts the cases where both the first and second balls are red (`R1 = TRUE` and `R2 = TRUE`), while the denominator counts the cases where the second ball is red (`R2 = TRUE`).
- `p2`: The resulting value is printed, showing the simulated conditional probability $P(R_1|R_2) \approx 0.750$, which is close to the theoretical value 0.762.

95% CIs on the probabilities

```
p1-sqrt(p1*(1-p1)/MC)*qnorm(.975)
```

```
[1] 0.6746793
```

```
p1+sqrt(p1*(1-p1)/MC)*qnorm(.975)
```

```
[1] 0.7313207
```

```
n2 = sum(R2)
p2-sqrt(p2*(1-p2)/n2)*qnorm(.975)
```

```
[1] 0.7176203
```

```
p2+sqrt(p2*(1-p2)/n2)*qnorm(.975)
```

```
[1] 0.7816685
```

Alternative and more accurate intervals:

```
library(binom)

# Parameters
MC = 1000              # Number of Monte Carlo iterations
p1 = 0.703             # Estimated probability for p1
p2 = 0.7496444         # Estimated probability for p2
n2 = round(p1 * MC)    # Approximation for the count of R2 occurrences

# Wilson score interval for p1
binom.confint(x = round(p1 * MC), n = MC, methods = "wilson")

# Wilson score interval for p2
binom.confint(x = round(p2 * n2), n = n2, methods = "wilson")
```

```
  method   x    n  mean    lower     upper
1 wilson 703 1000 0.703 0.673946 0.7305003

  method   x   n      mean     lower     upper
1 wilson 527 703 0.7496444 0.7163219 0.7802534
```

Monte Carlo integration

Consider a function $g(x)$ defined for $x \in (0,1)$. We want to evaluate the integral

$$\int_0^1 g(x)\,dx.$$

This quantity can be interpreted in a probabilistic framework as the expected value (mean) of $g(x)$ when x is a random variable distributed uniformly on $(0,1)$. More precisely, if U is a random variable such that $U \sim U(0,1)$, then its probability density function (pdf) $f_U(x)$ is given by

$$f_U(x) = \begin{cases} 1, & 0 \leq x \leq 1, \\ 0, & \text{otherwise.} \end{cases}$$

By definition, the expected value of $g(U)$ is

$$E[g(U)] = \int_{-\infty}^{\infty} g(x)\, f_U(x)\,dx.$$

Since $f_U(x) = 0$ outside the interval $(0,1)$ and $f_U(x) = 1$ for x in $(0,1)$, this integral simplifies to

$$\int_{-\infty}^{\infty} g(x)\, f_U(x)\,dx = \int_0^1 g(x)\,(1)\,dx = \int_0^1 g(x)\,dx.$$

Hence,

$$\int_0^1 g(x)\,dx = E[g(U)].$$

Estimating the Integral using Monte Carlo Method

Based on the Law of Large Numbers, the integral can be approximated by the sample mean of $g(U)$, calculated from a series of random samples. The steps are as follows:

1. Generate n random numbers $U_1, \ldots, U_n$, each uniformly distributed on $(0,1)$.
2. Transform each random number U_i into $Y_i = g(U_i)$.
3. Approximate the integral by the sample mean: $\frac{1}{n}\sum_{i=1}^{n} Y_i$.

Asymptotic Confidence Interval for the Integral

To assess the reliability of the Monte Carlo estimate, we can construct an approximate confidence interval for the mean μ of $g(U)$.

For a $(1-\alpha)100\%$ confidence interval (CI), the formula is:

$$\overline{x}_n \mp z_{\frac{\alpha}{2}} \frac{s_n}{\sqrt{n}}$$

Here, $\overline{x}_n$ is the mean of the sample, s_n is the standard deviation of the sample, n is the sample size, and $z_{\frac{\alpha}{2}}$ is the critical value of the normal standard distribution, which can be obtained using the quantile function (e.g., `qnorm`$(1-\alpha/2)$ in R).

To construct the confidence interval for our Monte Carlo estimate, follow the following steps:

1. Generate n random numbers $U_1, \ldots, U_n$, uniformly distributed over $(0, 1)$.
2. Transform each U_i into $Y_i = g(U_i)$.
3. Calculate the mean of the sample $\overline{y}_n$ and the variance of the sample s_n^2.
4. Construct the confidence interval: $\overline{y}_n \mp z_{\frac{\alpha}{2}} \frac{s_n}{\sqrt{n}}$.

This method provides a probabilistic estimate of the integral along with a measure of its precision.

Example: Area of a quarter circle

$$\int_0^1 \sqrt{1-x^2}dx$$

```
MC =1000
set.seed(1)
y = sqrt(1-runif(MC)^2)
mean(y)
```

```
[1] 0.7845459
```

```
t.test(y)$conf.int
```

```
[1] 0.7704274 0.7986644
attr(,"conf.level")
[1] 0.95
```

So, the area of a full circle is 4 times the previous one.

```
4*t.test(y)$conf.int
```

```
[1] 3.081710 3.194658
attr(,"conf.level")
[1] 0.95
```

Integrals over (a, b)

Consider a function $g(x)$ defined on the interval (a, b). We want to evaluate the integral

$$\int_a^b g(x)\,dx.$$

A useful technique is to transform this integral over (a, b) into an integral over $(0, 1)$.

Specifically, with the change of variable

$$t = \frac{x-a}{b-a},$$

Then

$$x = a + (b-a)t,$$

and

$$dx = (b-a)\,dt.$$

When $x = a$, we have $t = 0$. When $x = b$, we have $t = 1$. Thus,

$$\int_a^b g(x)\,dx = \int_0^1 g\big(a + (b-a)t\big)\,(b-a)\,dt = (b-a)\int_0^1 g\big(a + (b-a)t\big)\,dt.$$

Since $U \sim U(0, 1)$ has density

$$f_U(u) = \begin{cases} 1, & 0 \le u \le 1, \\ 0, & \text{otherwise,} \end{cases}$$

the random variable $a + (b-a)U$ is uniformly distributed on (a, b). Therefore,

$$\int_a^b g(x)\,dx = (b-a)\int_0^1 g\big(a + (b-a)t\big)\,dt = E\big[g\big(a + (b-a)U\big)\big] \times (b-a).$$

To generate random numbers uniformly in (a, b), it is sufficient to generate uniform random numbers U in $(0, 1)$ and then map them to (a, b) via $x = a + (b-a)\,U$.

In R: `runif(MC, min=a, max=b)`.

Integrals over (a, ∞)

In many applications, we encounter integrals of the form

$$\int_0^{+\infty} g(x)\,dx.$$

One useful technique for evaluating or transforming such integrals is to map the interval $(0, \infty)$ onto $(0, 1)$.

Let us consider the substitution

$$t \;=\; (1+x)^{-1},$$

where $x \in (0, \infty)$. Then:

1. **Define the variable change:**

$$t \;=\; \frac{1}{1+x}.$$

2. **Express x in terms of t:**

$$x \;=\; \frac{1}{t} - 1.$$

3. **Determine the differential dx:**

$$dx \;=\; -\frac{1}{t^2}\,dt.$$

The minus sign appears because x decreases as t increases. We will handle the direction of the limits accordingly.

4. **Limits of integration:**

$$x = 0 \quad \longrightarrow \quad t = \frac{1}{1+0} = 1,$$
$$x = +\infty \quad \longrightarrow \quad t = \frac{1}{1+\infty} = 0.$$

Thus, as x goes from 0 to $+\infty$, t goes from 1 down to 0. We can switch these limits to integrate from 0 to 1 and remove the minus sign (or we can keep the original order and keep the minus sign).

5. **Substitute into the integral:**

$$\int_0^{+\infty} g(x)\,dx \;=\; \int_{x=0}^{x=\infty} g(x)\,dx \;=\; \int_{t=1}^{t=0} g\Big(\tfrac{1}{t} - 1\Big)\,\big(-t^{-2}\big)\,dt.$$

Changing the order of integration limits (from $t = 0$ to $t = 1$) removes the negative sign:

$$= \int_0^1 t^{-2}\,g\Big(\tfrac{1}{t} - 1\Big)\,dt.$$

Hence,

$$\int_0^{+\infty} g(x)\,dx \;=\; \int_0^1 t^{-2}\,g\Big(t^{-1} - 1\Big)\,dt.$$

Example: The Gaussian Integral

We can apply this idea, for instance, to the Gaussian (normal) integral

$$\int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{x^2}{2}\Bigr)\, dx \;=\; 1.$$

(Recall that this integral equals 1 when we include the normalizing factor $\frac{1}{\sqrt{2\pi}}$.)

However, if we restrict the integral to the positive half-line and then use symmetry, we get

$$\int_{0}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{x^2}{2}\Bigr)\, dx \;=\; \tfrac{1}{2}.$$

Using the substitution $t = (1+x)^{-1}$ on the half-line integral, we have:

$$\int_{0}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{x^2}{2}\Bigr)\, dx = \int_{0}^{1} t^{-2} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{\left(t^{-1}-1\right)^2}{2}\Bigr)\, dt.$$

Because the entire integral over $\mathbb{R}$ gives 1, it follows that the integral from $-\infty$ to $+\infty$ can be written as 2 times the integral from 0 to $+\infty$. Hence,

$$\int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{x^2}{2}\Bigr)\, dx \;=\; 2\int_{0}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{x^2}{2}\Bigr)\, dx \;=\; 2\int_{0}^{1} t^{-2} \frac{1}{\sqrt{2\pi}} \exp\Bigl(-\tfrac{\left(t^{-1}-1\right)^2}{2}\Bigr)\, dt.$$

Thus, the integral over the entire real line can be seen as a special case of transforming from $(0,\infty)$ to $(0,1)$ and then doubling it to account for the symmetry about the origin.

```
MC = 10000
set.seed(1)
u = runif(MC)

y = 2 * (u^(-2)/sqrt(2*pi))*exp(-(1/u-1)^2/2)
t.test(y)$conf.int
```

```
[1] 0.9803571 1.0074944
attr(,"conf.level")
[1] 0.95
```

Numerical integration in R

The main function for numerical integration is `integrate()` at base R.

As an example, we will integrate the function $f(x) = e^{-x}\cos(x)$ from 0 to π:

```r
f = function(x) exp(-x) * cos(x)
(q = integrate(f, 0, pi))
```

```
0.521607 with absolute error < 7.6e-15
```

Example

We wish to calculate the surface area obtained by revolving the curve

$$y = \sin(x)$$

about the x-axis, over the interval $x \in [0, 2\pi]$. The general formula for the surface area of a curve $y = f(x)$ revolved about the x-axis, from $x = a$ to $x = b$, is:

$$S \;=\; 2\pi \int_a^b f(x)\,\sqrt{1 + \big(f'(x)\big)^2}\,dx.$$

The classical formula above, assumes $f(x) \geq 0$ on $[a, b]$, so that $f(x)$ can serve directly as the radius of revolution around the x-axis.

For the function $y = \sin(x)$, we have $\sin(x) \geq 0$ on $[0, \pi]$ and $\sin(x) \leq 0$ on $[\pi, 2\pi]$. Strictly speaking, if we revolve the entire graph from 0 to 2π, the portion for $x \in [\pi, 2\pi]$ is below the x-axis. When computing a surface of revolution about the x-axis, one typically uses the absolute value of the radius:

$$S \;=\; 2\pi \int_a^b \big|\,f(x)\big|\,\sqrt{1 + \big(f'(x)\big)^2}\,dx.$$

Because $\sin(x)$ from π to 2π is just the negative mirror of $\sin(x)$ from 0 to π, the total surface area from 0 to 2π is in fact

$$S_{0\to 2\pi} \;=\; 2 \times \Big(\text{surface area from 0 to } \pi\Big).$$

Hence, once you compute the area on $[0, \pi]$, you can just multiply by 2 if you want the entire surface from 0 to 2π.

Although we know analytically that $\frac{d}{dx}\sin(x) = \cos(x)$, suppose we pretend we do not know this derivative and want to use a numerical method to estimate the derivative. In the statistical software R, one can use the package `numDeriv` and its `grad` function to approximate derivatives.

Below is an example R script showing how to compute the integral numerically on $[0, \pi]$. (You may double it afterward if you want the entire range $[0, 2\pi]$.)

```
library(numDeriv)
fn = sin
gr = function(x) grad(fn, x)
F = function(x) fn(x) * sqrt(1 + gr(x)^2)
( I = integrate(F, 0, pi) )
```

```
2.295587 with absolute error < 2.1e-05
```

```
S = 2*pi * I$value
S
```

```
[1] 14.4236
```

Monte Carlo Integration in R

```
if (!require(SI)) install.packages("SI")
library(SI)

N = 100000
MVMresult = SI.MVM(F,0,pi,N)
I2 = MVMresult[[1]]
VarI2 = MVMresult[[2]]
I2
VarI2
```

```
> I2
[1] 2.292575
```

```
> VarI2
[1] 9.021468e-06
```

Multiple integrals

We can consider a function of two or more variables: The function g maps from a 2-dimensional real space $\mathbb{R}^2$ to real numbers $\mathbb{R}$, that is $g : \mathbb{R}^2 \to \mathbb{R}$.

This represents a function that depends on two variables, for example, $g(x_1, x_2)$.

The given integral of a function over a region

$$\int_0^1 \int_0^1 g(x_1, x_2)\, dx_1\, dx_2$$

represents the double integral of the function g on the unit square $[0, 1] \times [0, 1]$. This integral calculates the *accumulated value* of g in this square region.

The integral can be interpreted as the expectation (average value) of the function g when applied to two independent random variables X_1 and X_2, each uniformly distributed over the interval $(0, 1)$ (denoted $U(0, 1)$).

Then

$$\int_0^1 \int_0^1 g(x_1, x_2)\, dx_1\, dx_2 = E[g(X_1, X_2)],$$

where E denotes the expected value.

Approximation Using Laws of Large Numbers

1. **Generate Random Numbers**: Create two sets of n random numbers each, within the interval $(0, 1)$. These numbers represent samples of the random variables X_1 and X_2.

2. **Transformation**: Transform each pair of random numbers (U_i, U_i') into a new random variable Y_i using the function g. This step evaluates the function g at each pair of sampled points.

3. **Average to Approximate Integral**: Calculate the average of all the values of Y_i. This average is an approximation of the integral and becomes more accurate as n increases, thanks to the Law of Large Numbers, which states that the average of a large number of trials converges to the expected value.

Algorithm

1. Generate $U_1, \ldots, U_n$ and $U_1', \ldots, U_n'$ random numbers in $(0, 1)$.

2. Transform each pair (U_i, U_i') into $Y_i = g(U_i, U_i')$.

3. Return $\frac{1}{n}\sum_{i}^{n} Y_i$ and stop.

Handling different integration regions: Adapting the algorithm for non-unit square regions

- **Transforming the Region**: If the integration region is not the unit square, it can be transformed into a unit square. Once transformed, the same method can be applied as if it were the unit square.
- **Wider Region and Discarding Points**: Alternatively, a larger region encompassing the desired integration region can be considered. Points that fall outside the target region are discarded. This process continues until a sufficient number of points are obtained within the desired region.
- **Modified Step in the Algorithm**: An additional step is introduced between generating random numbers and transforming them. This step involves verifying whether each random pair (U_i, U_i') lies within the target region. If a pair does not, a new pair is generated.

To incorporate this modification, the algorithm is updated by introducing a new step between Steps 1 and 2 of the previous procedure:

1.–2. For each generated pair (U_i, U_i'), check if it lies inside the desired region. If not, generate a new pair.

Example 1

Define the function $g(U_1, U_2)$ for the transformation

$$g(U_1, U_2) = U_1 \cdot U_2$$

in the region

$$(U_1 - 0.5)^2 + (U_2 - 0.5)^2 \leq 0.5^2$$

```r
# Define the function g(U1, U2) for the transformation
g = function(U1, U2) {
  return(U1 * U2)
}
```

```
# Monte Carlo Integration Algorithm with Region Handling
monte_carlo_integration = function(n, region_check = NULL) {
  # Initialize variables
  Y = numeric(n)
  count = 0

  # Generate n pairs (U1, U2)
  while (count < n) {
    U1 = runif(1)
    U2 = runif(1)

    # Check if (U1, U2) is in the desired region
    if (is.null(region_check) || region_check(U1, U2)) {
      count = count + 1
      Y[count] = g(U1, U2)
    }
  }

  # Compute the average of Y values
  result = mean(Y)
  return(result)
}
```

We define a region check function for a non-unit square region: A circular region of radius 0.5 centered at $(0.5, 0.5)$

```
circular_region = function(U1, U2) {
  return((U1 - 0.5)^2 + (U2 - 0.5)^2 <= 0.5^2)
}

n = 10000 # Number of samples

result = monte_carlo_integration(n, circular_region)
cat("Result for circular region:", result, "\n")
```

```
Result for circular region: 0.2519465
```

Example 2

Find the integral

$$\int_0^{0.5}\int_0^{0.5}\int_0^{0.5}\frac{2}{3}(x_1+x_2+x_3)dx_1dx_2dx_3$$

To integrate a scalar function over a multidimensional rectangle, use the function `adaptIntegrate()` from the library `cubature`.

```r
if (!require(cubature)) install.packages("cubature")
library(cubature)

f = function(x){
  (2/3)*(x[1]+x[2]+x[3])
}

adaptIntegrate(f, lowerLimit = c(0,0,0), upperLimit = c(0.5,0.5,0.5))
```

```
$integral
[1] 0.0625

$error
[1] 1.387779e-17
```