HW1_MTH209

Group3

26/1/25

Question 1 : Estimation of e

General equations in which e appears

The number e is the limit of an expression that arises in the computation of compound interest.

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

Which can also expressed as,

$$e = \sum_{n=0}^{\infty}\frac{1}{n!}$$

Approaches to estimate e

(n+1) sided die method

Take n+1 sided die and roll it n times , the probability of **NOT** getting 1 each time is equal to

$$\left(\frac{n}{n+1}\right)^n$$

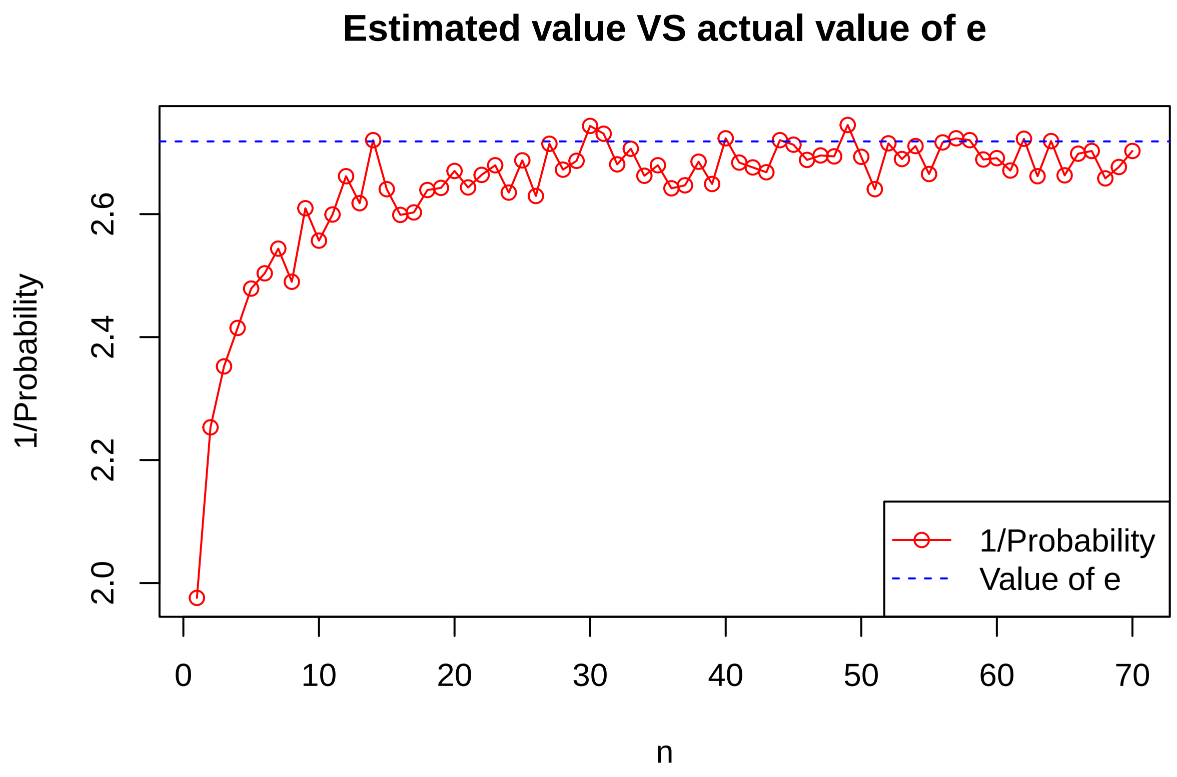
As $n \to \infty$, the expression

$$\left(\frac{n}{n+1}\right)^n \to \frac{1}{e}.$$

As we increase the value of n, the result approaches a fraction of e.

Here is code to estimate e

```
# Initialize an empty vector to store 1/probabilities
inverse_probabilities <- c()

# Set the maximum value of n to test
max_n <- 70  # You can adjust this to your desired limit
trials=1e4
# Loop through values of n
for (n in 1:max_n) {
  # Calculate the probability of not rolling 1 in n rolls of an (n+1)-sided die
  success=0
  for (i in 1:trials){
    rolls=sample(1:(n+1),n,replace = TRUE)
    if (all(rolls!=1)){
      success=success+1
    }
  }
  prob <- success/trials

  # Calculate 1/probability and append to the vector
  inverse_probabilities <- c(inverse_probabilities, 1 / prob)
}


# Plot the inverse probabilities to visualize the trend
plot(1:max_n, inverse_probabilities, type = "o", col = "red", xlab = "n", ylab = "1/Probability",
     main = "Estimated value VS actual value of e")

# Add a horizontal line for the actual value of e
abline(h = exp(1), col = "blue", lty = 2)

# Add a legend
legend("bottomright", legend = c("1/Probability", "Value of e"), col = c("red", "blue"),
       lty = c(1, 2), pch = c(1, NA))
```



Factorial-Based Estimation of e

1. Experiment Setup The experiment involves: - Choosing a random n from 1 to a fixed number X. - Generating a random arrangement of numbers 1 to n. - Checking if the arrangement is in ascending order.

2. Key Observation The total number of possible arrangements of 1 to n is given by:

$$n!$$

Out of these, only **one arrangement** is in perfect ascending order: $[1, 2, \ldots, n]$.

Thus, the probability of randomly generating the ascending order arrangement is:

$$P(\text{ascending}) = \frac{1}{n!}$$

3. Averaging Over n Since n is randomly chosen from 1 to X, the average success probability over all n is:

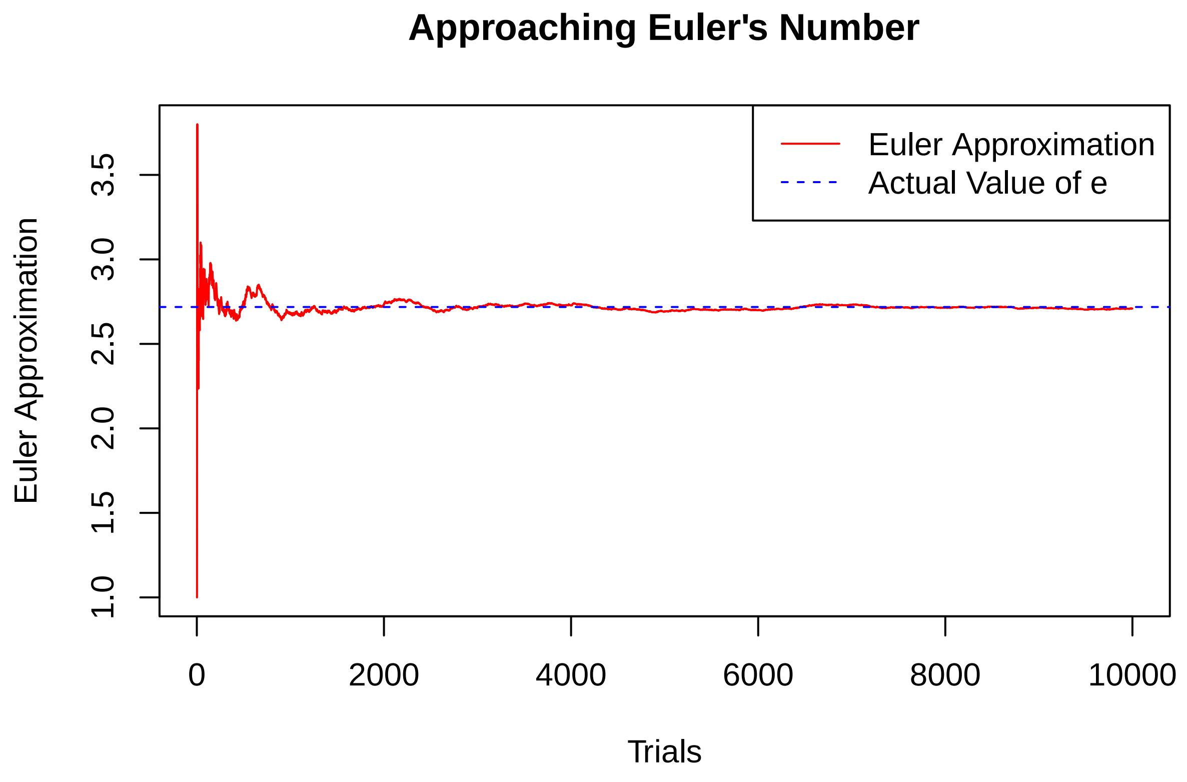
$$P_{\text{average}} = \frac{1}{\text{X}} \sum_{n=1}^{\text{X}} \frac{1}{n!}$$

This aligns with the series expansion for e:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Here is code to estimate e

```r
# Parameters
no <- 7          # Maximum number to consider
trials <- 1e4   # Number of trials
success <- 0    # Counter for successful trials

# Define a factorial function
factorial <- function(n) {
  if (n == 0) return(1)
  prod(1:n)
}

# Initialize a vector to store Euler approximations at each trial
euler_approximations <- c()

# Perform trials and track the approximations of Euler's number
for (i in 1:trials) {
  # Randomly pick n between 1 and `no`
  n <- sample(1:no, 1)

  # Generate a random arrangement of numbers 1 to n
  random_arrangement <- sample(1:n, n)

  # Check if the arrangement is in perfect ascending order
  if (all(random_arrangement == 1:n)) {
    success <- success + 1
  }
  # Calculate the Euler approximation at this point
  euler_approx <- (no * success / i) + 1
  euler_approximations <- c(euler_approximations, euler_approx)
}

# Plot the Euler approximations as the trials progress
plot(1:trials, euler_approximations, type = "l", col = "red", xlab = "Trials",
     ylab = "Euler Approximation", main = "Approaching Euler's Number")
abline(h = exp(1), col = "blue", lty = 2)  # Add a horizontal line for the actual value of e

# Add a legend
legend("topright", legend = c("Euler Approximation", "Actual Value of e"),
       col = c("red", "blue"), lty = c(1, 2))
```



Possible Errors When estimating e through simulations, two main errors can occur:

1. **Limited Trials**:
   With a small number of trials, the approximation can be inaccurate due to random fluctuations and insufficient data to converge to the true value.

2. **Effect of Large** n:
   As n increases, the probability of getting a perfect arrangement (ascending order) becomes very small, because $P = \frac{1}{n!}$ decreases rapidly. With limited trials, this small probability leads to fewer successful outcomes, causing the estimate of e to be lower than expected.

In essence, large n with limited trials leads to an underestimated e due to the rarity of successful outcomes.

Geometric probability based estimation of e

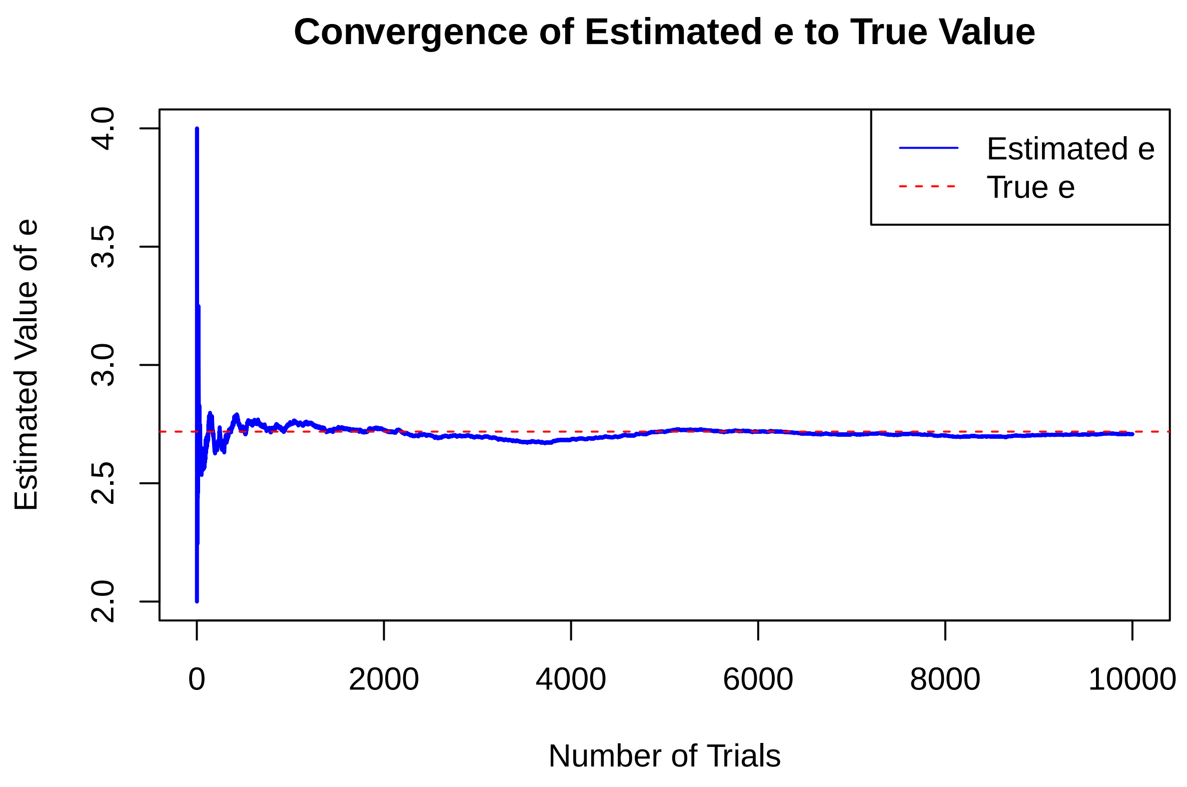
The idea behind the simulation is that the area under the curve $y = \frac{1}{x}$ from $x = 1$ to $x = 2$ can be approximated by generating random points in the square (1,2)x(0,1) and checking whether they lie below the curve $y = \frac{1}{x}$. The probability of a point falling below the curve corresponds to the area under the curve, which can be used to estimate the natural logarithm of 2, ln(2).

Key Formulae

1. **Total Area of the Square**: The total area of the square, where x is in $(1, 2)$ and y is in $(0, 1)$, is 1:
$$\text{Total Area} = 1 \times (2 - 1) = 1$$
2. **Probability of Falling Below the Curve**: For each point, the probability of falling below the curve $y = \frac{1}{x}$ is given by:
$$P(\text{below the curve}) = \frac{\text{Area under the curve}}{\text{Total Area}} = \{\ln(2)\}$$
3. **Estimation of** $\ln(2)$: Since the probability is related to the area under the curve, we can use the ratio of successful trials to total trials to estimate $\ln(2)$:
$$\{\ln(2)\} = \frac{\text{successes}}{\text{total trials}}$$
4. **Estimating Euler's Number** e: Using the relationship between $\ln(2)$ and Euler's number, we can estimate e as:
$$e = 2^{1/\hat{\ln(2)}}$$

Here is code to estimate e

```r
# Parameters
trials <- 1e4  # Number of trials
success <- 0   # Counter for successful trials

# Total area of the cube
total_area <- 1  # Since x is in (1, 2) and y is in (0, 1), the total area of the cube is 1.

# Initialize vectors to store x, y, success flag, and estimated e values
estimated_e_values <- numeric(trials)

# Function to generate random points and check if they lie under the curve
for (i in 1:trials) {
  # Generate random x in (1, 2) and y in (0, 1)
  x <- runif(1, 1, 2)
  y <- runif(1, 0, 1)

  # Check if the point lies below the curve y = 1/x
  if (y < 1 / x) {
    success <- success + 1
  }
  # Estimate the probability (area under the curve) after each trial
  probability <- success / i

  # Estimate ln(2) from the probability
  estimated_ln2 <- probability

  # Use the relationship e = 2^(1/ln(2)) to estimate e
  estimated_e <- 2^(1 / estimated_ln2)

  # Store the estimated e value
  estimated_e_values[i] <- estimated_e
}

# Plot how the estimated value of e approaches the true value as the number of trials increases
plot(1:trials, estimated_e_values, type = "l", col = "blue", lwd = 2,
     xlab = "Number of Trials", ylab = "Estimated Value of e",
     main = "Convergence of Estimated e to True Value")
abline(h = exp(1), col = "red", lty = 2)  # True value of e (Euler's number)
legend("topright", legend = c("Estimated e", "True e"), col = c("blue", "red"), lty = c(1, 2))
```



Monte Carlo estimation of e

In this experiment, we generate random numbers from a uniform distribution $U(0, 1)$ and keep summing them until the sum exceeds 1. We then track how many numbers it took for the sum to exceed 1. This experiment is repeated n times, and after each trial, the running average of the number of random numbers used is computed.

The goal is to observe the convergence of the average number of trials to a certain constant value as the number of repetitions increases. It turns out that the expected number of terms required to exceed 1 is the constant e.

Theory Behind the Experiment

Let $X_1, X_2, \ldots$ be independent random variables, each following a uniform distribution $U(0, 1)$. The sum of the random variables after k terms is:

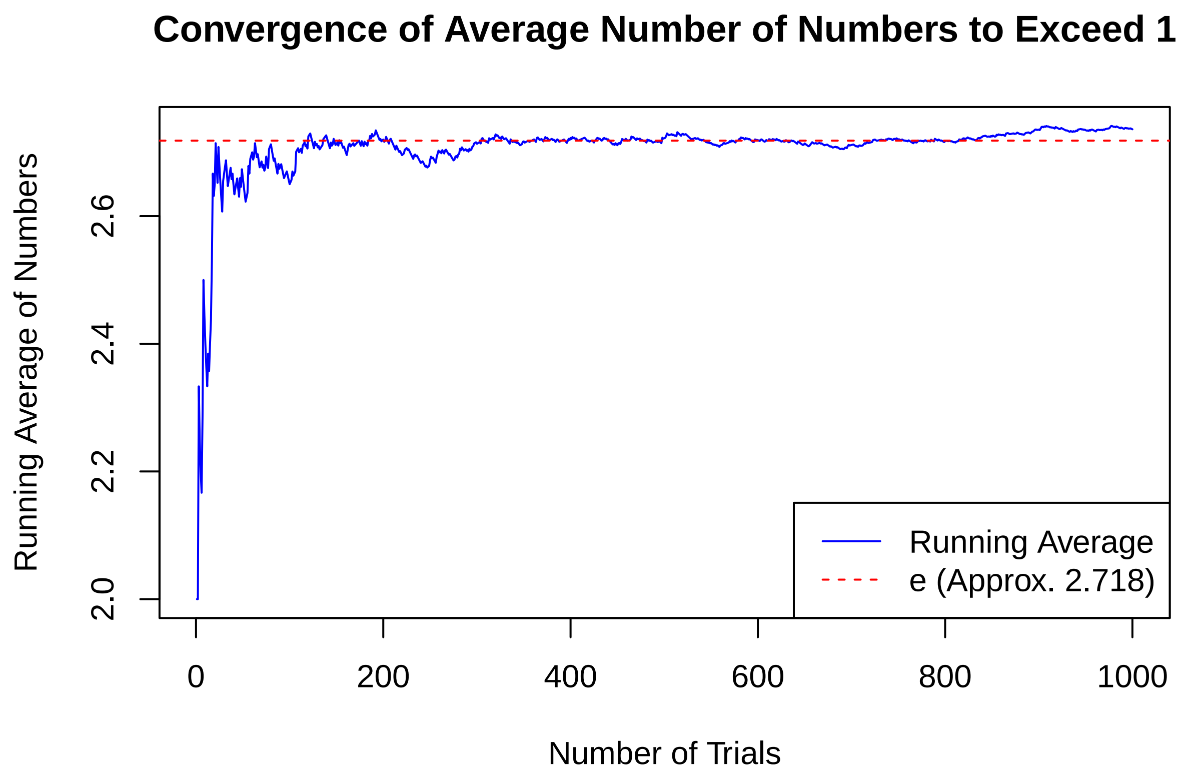
$$S_k = X_1 + X_2 + \cdots + X_k$$

We continue adding terms until the sum exceeds 1, i.e., $S_k > 1$. The expected number of terms, denoted as $E[k]$, is known to converge to the value e, as shown by the following reasoning:

This result comes from a well-known property in probability theory, where the expected number of uniform random variables required to exceed a certain threshold (like 1) is the mean of a geometric-like distribution, which converges to e.

Here is the code to estimate e

```r
set.seed(123)  # For reproducibility

max_trials <- 1000  # Max number of trials to run
results <- numeric(max_trials)  # Vector to store the results
avg_values <- numeric(max_trials)  # Vector to store running averages

for (n in 1:max_trials) {
  sum_val <- 0  # Initialize the sum for each trial
  count <- 0  # Initialize the count of random numbers

  # Keep generating random numbers until the sum exceeds 1
  while (sum_val <= 1) {
    sum_val <- sum_val + runif(1)  # Add random number to sum
    count <- count + 1  # Increment the count of random numbers
  }

  results[n] <- count  # Store the count of numbers for this trial
  avg_values[n] <- mean(results[1:n])  # Running average of the trials so far
}

# Plotting the running average vs. number of trials
plot(1:max_trials, avg_values, type = "l", col = "blue",
     xlab = "Number of Trials", ylab = "Running Average of Numbers",
     main = "Convergence of Average Number of Numbers to Exceed 1")
abline(h = exp(1), col = "red", lty = 2)  # Plot the value of e (approximately 2.718)
legend("bottomright", legend = c("Running Average", "e (Approx. 2.718)"), col = c("blue", "red"), lty =
```



Error in Monte Carlo Estimation of e

The Monte Carlo method for estimating e by summing random variables has inherent errors. These can be summarized as:

1. **Sampling Error**:
    - Randomness introduces variability in each simulation. While the estimate improves with more trials, fluctuations are inevitable.
2. **Convergence Rate**:
    - The running average takes time to converge to e, with error decreasing slowly as the number of trials increases.
3. **Finite Number of Trials**:
    - With too few trials, the estimate will be inaccurate. More trials help to reduce this error.
4. **Computational Constraints**:
    - Running a large number of trials may require significant computational time and resources, which can limit precision.

Question 2 : Estimation of π

Approaches to estimate π

Estimating π using a d-dimensional sphere - Monte Carlo method

Geometric Setup

1. **Unit Cube**:
   The D-dimensional cube spans $[-0.5, 0.5]^D$, with side length 1.

2. **Inscribed Sphere**:
   The sphere is centered at the origin $(0, 0, \ldots, 0)$ with radius $R = 0.5$. A point $(x_1, x_2, \ldots, x_D)$ is inside the sphere if:
   $$\sqrt{x_1^2 + x_2^2 + \cdots + x_D^2} \leq R$$
   or equivalently:
   $$x_1^2 + x_2^2 + \cdots + x_D^2 \leq R^2 = 0.25$$

A D-dimensional sphere of radius R is the set of points $(x_1, x_2, \ldots, x_D)$ satisfying:

$$x_1^2 + x_2^2 + \cdots + x_D^2 \leq R^2$$

The volume V_D of the sphere is calculated as:

$$V_D = \int_{x_1^2 + x_2^2 + \cdots + x_D^2 \leq R^2} 1 \, dV$$

Derivation of V_D

Step 1: Factorizing the Volume into Radial and Angular Components Using polar coordinates in D-dimensions, the volume integral can be rewritten as:

$$V_D = \int_0^R \int_{\text{angles}} r^{D-1} \, dr \, d\Omega$$

Here: - r: Radial distance from the origin. - Ω: Angular coordinates that describe the sphere's surface.

The volume element in D-dimensional polar coordinates is $r^{D-1} dr d\Omega$.

Step 2: Surface Area of the Sphere The integral over $d\Omega$ gives the surface area of the unit sphere in D-dimensions, denoted by S_{D-1}. The volume becomes:

$$V_D = \int_0^R r^{D-1} S_{D-1} \, dr$$

Step 3: Evaluating the Radial Integral The radial integral is:

$$\int_0^R r^{D-1} \, dr = \frac{R^D}{D}$$

Substituting this, we have:

$$V_D = S_{D-1} \cdot \frac{R^D}{D}$$

Step 4: Recursive Formula for S_{D-1} The surface area S_{D-1} is related to the volume of the unit sphere in $D-1$-dimensions:

$$S_{D-1} = \frac{2\pi^{D/2}}{\Gamma(D/2)}$$

Substituting S_{D-1} into the volume formula:

$$V_D = \frac{2\pi^{D/2}}{\Gamma(D/2)} \cdot \frac{R^D}{D}$$

Final Formula The final formula for the volume of a D-dimensional sphere is:

$$V_D = \frac{\pi^{D/2} R^D}{\Gamma(D/2+1)}$$

Where: - $\Gamma(x)$ is the Gamma function, which generalizes the factorial: $\Gamma(n) = (n-1)!$ for integer n.

3. **Volume Ratio**:
   The ratio of the sphere's volume to the cube's volume is approximated by the fraction of points that fall inside the sphere:

$$\text{Volume Ratio} = \frac{\text{Points Inside Sphere}}{\text{Total Points}}$$

4. **Estimating π**:
   The volume of a D-dimensional sphere is given by:

$$V_{\text{sphere}} = \frac{\pi^{D/2} R^D}{\Gamma(D/2+1)}$$

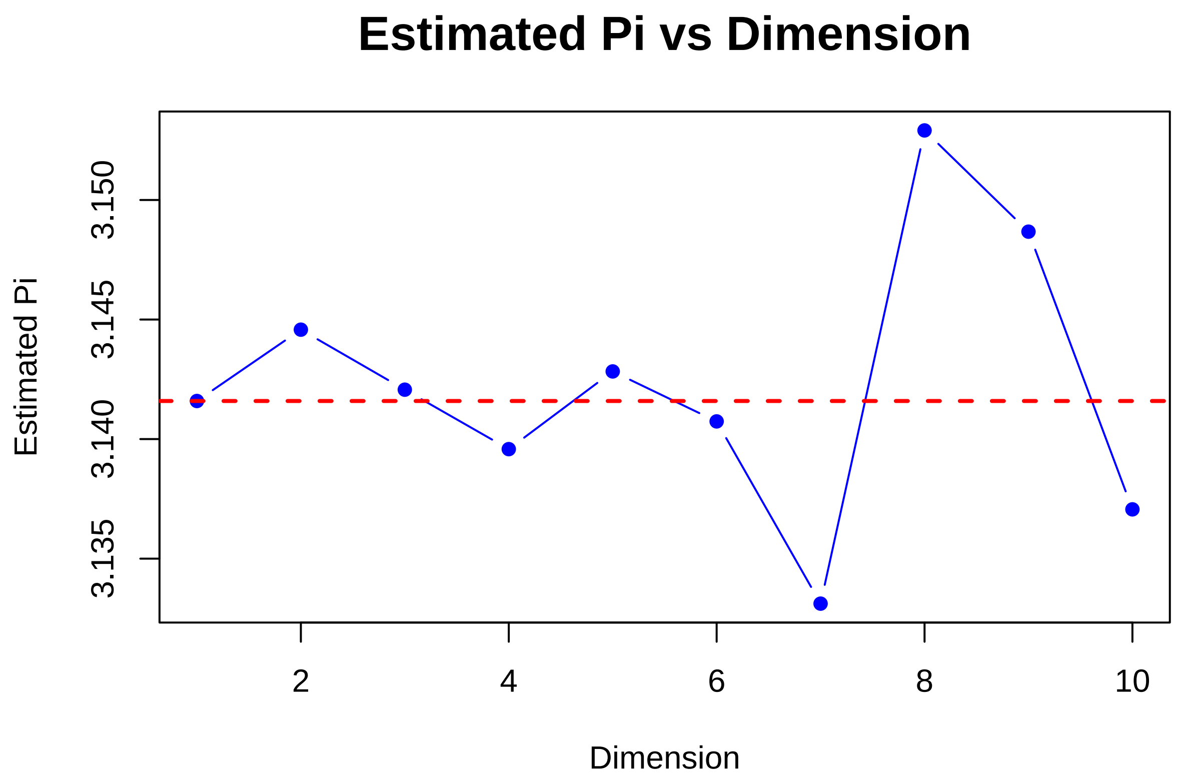
   For the unit cube, the volume is $(2R)^D = 1$. Rearranging to estimate π:

$$\pi = \left(\text{Volume Ratio} \cdot (2^D) \cdot \Gamma(D/2+1)\right)^{2/D}$$

Here is the code to estimate value of π

```r
# Function to estimate pi
estimate_pi <- function(dimensions, num_points) {
  points_inside <- 0
  radius <- 0.5

  for (i in 1:num_points) {
    point <- runif(dimensions, min = -radius, max = radius)
    distance <- sqrt(sum(point^2))
    if (distance <= radius) {
      points_inside <- points_inside + 1
    }
  }

  volume_ratio <- points_inside / num_points
  estimated_pi <- (volume_ratio * (2^dimensions) * gamma(dimensions / 2 + 1))^(2 / dimensions)

  return(estimated_pi)
}
```

```r
# Set dimensions and number of points
num_points <- 10000
dimensions <- 1:10
repeats <- 100

# Initialize vector to store mean pi estimates
mean_pi_estimates <- numeric(length(dimensions))

# Loop over each dimension
for (dim in dimensions) {
  pi_estimates <- numeric(repeats)

  for (r in 1:repeats) {
    pi_estimates[r] <- estimate_pi(dim, num_points)
  }

  mean_pi_estimates[dim] <- mean(pi_estimates)
}

# Actual value of Pi
actual_pi <- pi

# Base R Plotting
plot(dimensions, mean_pi_estimates,
     type = "b",
     pch = 16,
     col = "blue",
     xlab = "Dimension",
     ylab = "Estimated Pi",
     main = "Estimated Pi vs Dimension",
     cex.main = 1.5,
     font.main = 2)  # Making title bold

# Add horizontal line for actual Pi
abline(h = actual_pi, col = "red", lty = 2, lwd = 2)

# Add actual Pi text annotation
text(x = 5, y = actual_pi + 0.1, labels = paste("Actual Pi =", round(actual_pi, 4)), col = "red")
```



Sources of Error

1. **Sampling Error** The method relies on random sampling, and as the dimension D increases, the fraction of points inside the sphere becomes smaller. This increases the variance of the estimates, making the result less reliable.

2. **Curse of Dimensionality** As D increases, the volume of the sphere becomes much smaller relative to the volume of the cube. This makes it harder to detect points that lie inside the sphere, leading to less accurate estimates.

3.**Insufficient Sample Size** In higher dimensions, the number of samples needed to obtain accurate estimates increases exponentially. With insufficient samples, the estimates become less precise.

4. **Finite Precision of Computers** In high-dimensional spaces, the values involved in the calculations (such as distances) become very small, which may cause floating-point rounding errors. These errors degrade the accuracy of the result.

5.**Computational Complexity** As the dimension increases, the computational cost of performing the necessary trials also increases. This may limit the number of trials that can be run, affecting the accuracy of the estimation.

Conclusion As the dimension D increases, the accuracy of the π estimation decreases due to the sampling error, curse of dimensionality, volume decrease, insufficient sample size, finite precision, and computational complexity. This method becomes less reliable and more computationally expensive as D grows.

Estimation of π using Basel problem

The goal of this experiment is to estimate the value of π using a probabilistic approach based on the summation of the reciprocals of squares, leveraging the known formula:

$$\sum_{x=1}^{\infty} \frac{1}{x^2} = \frac{\pi^2}{6}$$

The approach for estimating π relies on the following steps:

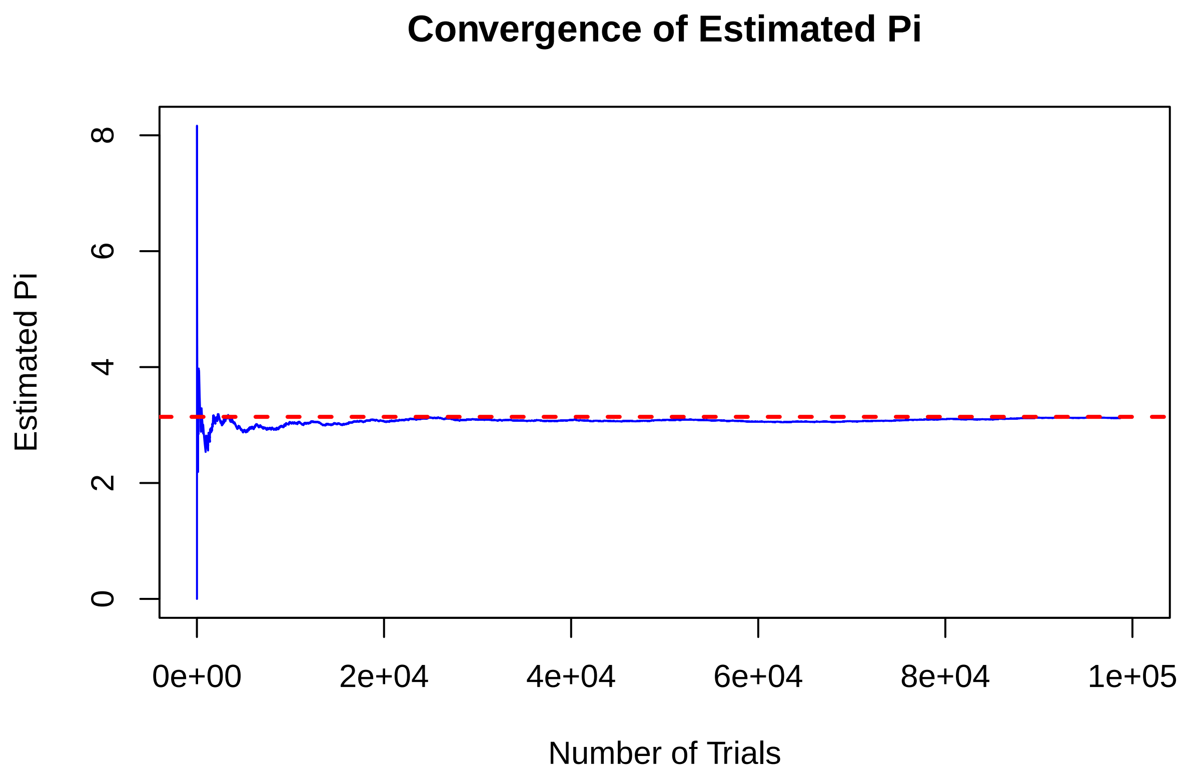
1. **Grid Selection**: We define a grid of size $x \times x$ where x is a randomly chosen value between 1 and n. Here, n is the maximum grid size, chosen to be 100.
2. **Probability Calculation**: The probability of selecting the point (x, x) in the grid is $\frac{1}{x^2}$, as there are x^2 possible points in the grid, and only one of them corresponds to this point (Henceforth called the origin).
3. **Sampling**: For each trial, we:
   - Randomly select a grid size x between 1 and n.
   - Randomly choose two coordinates a and b from 1 to x.
   - If both a and b equal x, we count it as a success (indicating that we have selected the origin).
4. **Estimating π**: By summing the probabilities of selecting the origin across all trials, we can estimate π using the formula:

$$\frac{1}{n}\sum_{x=1}^{n} \frac{1}{x^2} = \frac{\text{success}}{\text{trials}} = P(\text{successes})$$

$$\pi \approx \sqrt{\frac{6n \times \text{successes}}{\text{trials}}}$$

where successes refers to the number of times the origin was selected, and trials is the total number of trials conducted. **Here is code to estimate π**

```r
# Define parameters
n = 100             # Maximum grid size
trials = 1e5        # Number of trials to run
success = 0         # Initialize success count
estimated_pi_values = numeric(trials)  # Initialize vector to store estimated pi values

# Loop through trials
for (i in 1:trials){
  x = sample(1:n, size=1)  # Sample a random value of x between 1 and n
  a = sample(1:x, size=1)  # Sample a random value for coordinate a from 1 to x
  b = sample(1:x, size=1)  # Sample a random value for coordinate b from 1 to x

  # Check if both a and b equal x (finding the origin)
  if (a == x && b == x) {
    success = success + 1
  }
  # Estimate pi based on the success ratio up to this point
  estimated_pi_values[i] = (success * n * 6 / i) ^ 0.5
}

# Plot the estimated pi values
plot(1:trials, estimated_pi_values, type="l", col="blue",
     main="Convergence of Estimated Pi", xlab="Number of Trials",
     ylab="Estimated Pi")
abline(h = pi, col = "red", lwd = 2, lty = 2)  # Actual value of pi for reference
```



###** Estimation of π using Stochastic process** estimates the value of π using a stochastic process that can be conceptually related to the Leibniz formula for π, which is expressed as:

$$\pi = 4\sum_{n=1}^{\infty}\frac{(-1)^{n-1}}{2n-1}$$

This formula converges slowly but gives an approximation of π as more terms are added.

##**Estimation of π using Leibniz formula**

Random Sampling: - The code samples a random number x from 1 to n and generates coordinates of the form (a, x) and (x, b). - These coordinates can be seen as sampling points on a **lattice structure**, where these points form the edge of the x by x square from the (x-1) by (x-1) square. Thus it has

$$x^2 - (x-1)^2 = 2x - 1$$

Success Count: - The success counter is updated when a randomly selected coordinate matches (x, x) with it's respective probability being

$$\frac{1}{2x-1}$$

. - The sign of the update alternates based on whether x is odd or even, mimicking the alternating nature of the Leibniz series.

Fluctuations: - Early trials result in larger fluctuations (noise), but as the number of trials increases, the estimate of π stabilizes.

Relation to Leibniz Formula

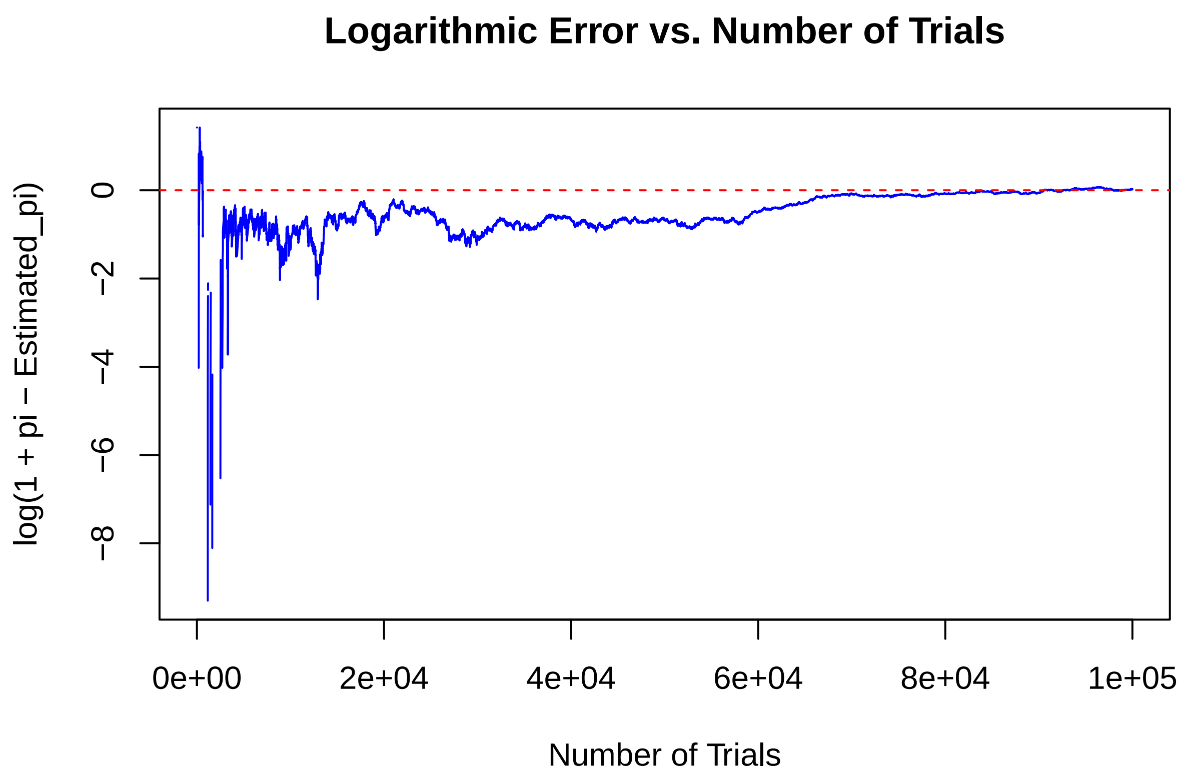
The alternating success mechanism (increasing or decreasing based on the parity of x) is akin to the alternating terms in the Leibniz formula. This results in a **dynamical system** that fluctuates until it stabilizes at the true value of π.

Here is code to estimate pi

```r
# Parameters
set.seed(42)
n <- 100
trials <- 1e5
success <- 0
estimated_pi <- numeric(trials)  # To store the estimated pi values

# Loop to perform trials and calculate estimated pi at each step
for (i in 1:trials) {
  x <- sample(1:n, 1)  # Randomly sample x from 1 to n
  coords_list <- list()  # Initialize an empty list

  # Add (x, x) to the list
  coords_list[[1]] <- c(x, x)

  # Generate coordinates of the form (a, x)
  if (x > 1) {
    for (a in 1:(x - 1)) {
      coords_list[[length(coords_list) + 1]] <- c(a, x)  # Append (a, x)
    }

    # Generate coordinates of the form (x, b)
    for (b in 1:(x - 1)) {
      coords_list[[length(coords_list) + 1]] <- c(x, b)  # Append (x, b)
    }
  }

  # Randomly sample one coordinate from the list
  sampled_coord <- coords_list[[sample(1:length(coords_list), 1)]]

  # Check if the sampled coordinate equals (x, x)
  if (all(sampled_coord == c(x, x))) {
    if (x %% 2 == 1) {  # If x is odd
      success <- success + 1
    } else {  # If x is even
      success <- success - 1
    }
  }
  # Store the estimated value of pi at the current trial
  estimated_pi[i] <- 4 * n * success / i
}

# Calculate log(1 + pi - estimated_pi) for each trial
log_diff <- log(1 + pi - estimated_pi)
```

```
## Warning in log(1 + pi - estimated_pi): NaNs produced
```

```
# Plot log(1 + pi - estimated_pi) vs. number of trials
plot(1:trials, log_diff, type = "l", col = "blue",
     xlab = "Number of Trials", ylab = "log(1 + pi - Estimated_pi)",
     main = "Logarithmic Error vs. Number of Trials")

# Add horizontal line at y = 0
abline(h = 0, col = "red", lty = 2)  # Red dashed line at y = 0
```



Errors and Inefficiencies in the Estimation Method

While the code provides a way to estimate π, it introduces several errors and inefficiencies that impact its accuracy and computational feasibility:

1. **Computational Error:**

- As the number of trials increases, the time and resources required to compute the results grow significantly.

- For each trial, the code performs random sampling, generating and checking multiple coordinates, which leads to increasing computational complexity.
- With a large number of trials (e.g., 10^6 or more), the system becomes inefficient and slower, making this method impractical for precise estimations.

2. **Slow Convergence:**

- This method converges slowly towards the true value of π, especially when compared to more efficient algorithms.
- This slow convergence makes the method inefficient for practical purposes, especially when faster algorithms like Monte Carlo simulations or the Leibniz series are available.

3. **Random Fluctuations:**

- The random nature of the process introduces fluctuations in the estimate, leading to higher variability in the results as the number of trials is lower.
- Even with a large number of trials, the method continues to have inherent fluctuations, making it less reliable than deterministic methods for estimating π.

4. **Memory and Resource Constraints:**

- The list used to store coordinates grows larger as more trials are performed. As the number of trials increases, the memory usage becomes substantial, further reducing the efficiency of this approach.

Conclusion Although this method provides an interesting way to estimate π, its **computational inefficiency** and **slow convergence** make it impractical for large-scale calculations. More efficient methods like the **Monte Carlo method** or **Leibniz series** are better suited for accurate and quick estimation of π.

###**Estimation of π using Zeta function** 1. Fixing values n (grid size) and x (dimension/power). 2. Randomly selecting a number k from 1 to n. 3. Generating an array of size $2x$ with elements randomly sampled from 1 to k. 4. Checking the condition where all elements are equal to 1. 5. Using the success probability and a mathematical formula to estimate π.

Physical Interpretation

The code is essentially simulating a **probabilistic experiment** where:

1. **Fixing x and n:** These parameters define the grid and the size of the random arrays we generate. x controls the dimension of the system, and n sets the grid size.
2. **Random Sampling:** We randomly choose k from 1 to n. Then, we generate an array of size $2x$ from values between 1 and k. The goal is to check if all values in the array are equal to 1.
3. **Probability Calculation:** The probability of the array containing only 1s is $\frac{1}{k^{2x}}$, as each element independently has a $1/k$ chance of being 1.
4. **Estimating π:** After running multiple trials, the success ratio is used in a formula involving **Bernoulli numbers** and **factorials** to approximate π.

The final estimate of π is derived from the **zeta function**. The zeta function at even integers $2n$ is related to π and Bernoulli numbers. The formula for the zeta function at even integers is:

$$\zeta(2n) = \sum_{k=1}^{\infty} \frac{1}{k^{2n}} = \frac{(-1)^{n+1} \cdot B_{2n} \cdot (2\pi)^{2n}}{2 \cdot (2n)!}$$

where: - B_{2n} is the $2n$-th Bernoulli number, - $(2\pi)^{2n}$ represents the power of 2π, - $(2n)!$ is the factorial of $2n$.

Formula for Estimating Pi

The formula used in this method to estimate π is:

$$\pi \approx \frac{1}{2} \left(\frac{\text{success ratio} \cdot n \cdot 2 \cdot (2x)!}{B_{2x}} \right)^{\frac{1}{2x}}$$

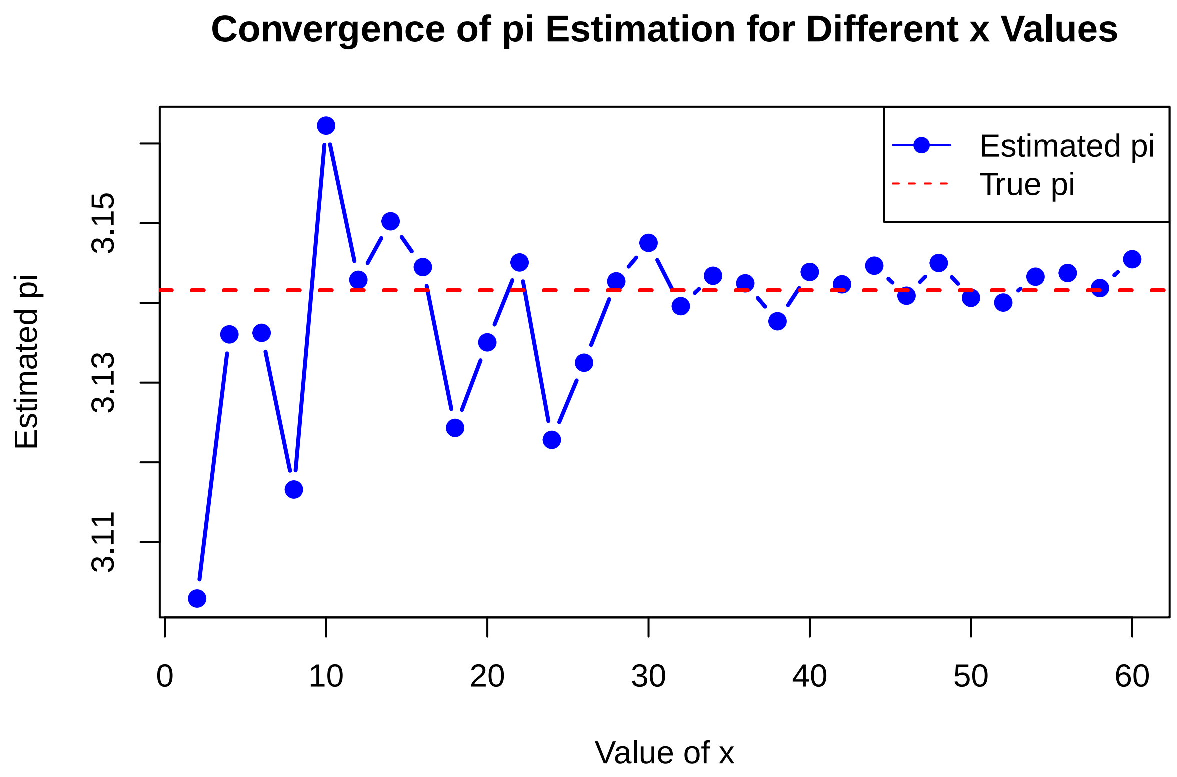
where: - success ratio $= \frac{\text{success}}{\text{trials}}$, - B_{2x} is the $2x$-th Bernoulli number, - $(2x)!$ is the factorial of $2x$.

Here is the code to estimate π

```
library(pracma)
```

```
## Warning: package 'pracma' was built under R version 4.4.2
```

```
nth_bernoulli <- function(n) {
  abs(bernoulli(n)[n+1])
}

n = 100          # Maximum grid size
trials = 1e4       # Number of trials to run
x_values = seq(2, 60, by = 2)  # Range of x values to test (even numbers)
pi_estimates = numeric(length(x_values))  # Store pi estimates for each x

for (j in seq_along(x_values)) {
  x = x_values[j]
  success = 0  # Initialize success count

  # Run trials for the current x
  for (i in 1:trials) {
    k = sample(1:n, size = 1)  # Sample a random value of x between 1 and n
    a = sample(1:k, size = 2*x, replace = TRUE)  # Sample a random value for coordinates a and b from 1

    # Check if all values in a equal k
    if (all(a == k)) {
      success = success + 1
    }
  }

  # Estimate pi based on the success ratio
  pi_estimates[j] = ((((success / trials) * n * 2 * factorial(2*x)) / nth_bernoulli(2*x))^(1/(2*x)))/2
}

# Plot the results
plot(x_values, pi_estimates, type = "b", col = "blue", pch = 19, lwd = 2,
     xlab = "Value of x", ylab = "Estimated pi",
     main = "Convergence of pi Estimation for Different x Values")
abline(h = pi, col = "red", lty = 2, lwd = 2)
legend("topright", legend = c("Estimated pi", "True pi"), col = c("blue", "red"), lty = c(1, 2), pch =
```



Question 3 : Testing if a sequence is really random

1. Basic Runs Test

Description: The basic runs test is a statistical method used to assess the randomness of a sequence of data. In this test, a "run" is defined as a sequence of consecutive identical values (either 0 or 1) in a binary sequence. The test examines the number of runs in the sequence and compares it to the expected number of runs if the sequence were truly random.

To apply the basic runs test: - A sequence of random numbers is generated and classified into binary values based on a chosen threshold (e.g., values greater than the median are classified as 1, and values less than or equal to the median are classified as 0). - The number of runs (i.e., the number of times the sequence changes from 0 to 1 or vice versa) is counted. - The expected number of runs and the variance are computed under the assumption that the sequence is random. - The observed number of runs is compared to the expected number to calculate a Z-statistic. If the number of observed runs significantly deviates from the expected value, the sequence is considered non-random.

Advantages:

- *Simple to Apply:* The basic runs test is easy to implement and understand, making it accessible for a wide range of users.
- *Applicable to Binary Data:* It works well with binary data and sequences that have been converted to binary form (e.g., using the median or some threshold).
- *Quick Analysis:* The test provides a quick assessment of whether a sequence is random, making it useful for preliminary analysis of data.

Limitations:

- *Requires Binary Data:* The basic runs test is limited to binary data. This means the data must be transformed into a binary format, which may not always capture the full complexity of the original data.
- *Sensitivity to Sample Size:* For very small sample sizes, the runs test may not provide reliable results, and the expected number of runs may be skewed.
- *Not Suitable for Continuous Data:* The test is not directly applicable to continuous data without transformation, which might lose some important characteristics of the data.

Here is the code

```r
# Step 1: Generate a sequence of random numbers between 0 and 1
set.seed(123)  # For reproducibility
n <- 100  # Length of the sequence
random_numbers <- runif(n)  # Generate random numbers between 0 and 1

# Step 2: Classify the numbers based on the median
median_value <- median(random_numbers)  # Find the median of the sequence
binary_classification <- ifelse(random_numbers > median_value, 1, 0)  # Classify based on the median

# Step 3: Identify the runs in the binary sequence
runs <- 1  # Initialize the number of runs
for (i in 2:n) {
  if (binary_classification[i] != binary_classification[i - 1]) {
    runs <- runs + 1  # Increment runs when the sequence changes
  }
}
runs=runs/(n-1)
# Step 4: Calculate the expected number of runs and variance
p0 <- sum(binary_classification == 0) / n  # Proportion of 0's
p1 <- sum(binary_classification == 1) / n  # Proportion of 1's

expected_runs <- (2 * p0 * p1 * n) / (n - 1)
variance_runs <- (expected_runs * (n - 1)) / (n - 2)

# Step 5: Compute the test statistic
z <- (runs - expected_runs) / sqrt(variance_runs)

# Step 6: Check if the statistic is significant (typically use alpha = 0.05)
alpha <- 0.05
z_critical <- qnorm(1 - alpha / 2)  # Two-tailed test

if (abs(z) > z_critical) {
  print("The sequence is not random (reject null hypothesis).")
} else {
  print("The sequence is random (fail to reject null hypothesis).")
}
```

```
## [1] "The sequence is random (fail to reject null hypothesis)."
```

```r
# Output the results
cat("Number of runs:", runs, "\n")
```

```
## Number of runs: 0.5252525
```

```
cat("Expected number of runs:", expected_runs, "\n")
```

```
## Expected number of runs: 0.5050505
```

```
cat("Variance of the runs:", variance_runs, "\n")
```

```
## Variance of the runs: 0.5102041
```

```
cat("Z statistic:", z, "\n")
```

```
## Z statistic: 0.02828283
```

---

2. Modified Runs Test on Absolute Deviations

Description: The modified runs test extends the basic runs test by applying it to the *absolute deviations* from the mean, rather than the raw data or binary classification based on a threshold. In this version: - The random numbers are generated as usual. - The *absolute deviation* of each number from the mean of the sequence is calculated. - Each absolute deviation is classified into binary values: 1 if the deviation is greater than the median of the absolute deviations, and 0 if the deviation is less than or equal to the median. - The number of runs in this new binary sequence is then calculated. - The observed number of runs is compared to the expected number of runs under the assumption of randomness using a Z-statistic.

This test checks whether the sequence's deviations from the mean exhibit a random pattern, which could indicate potential underlying structure or systematic behavior in the data.

Advantages:

- *Sensitive to Magnitude of Deviations:* This test considers not just whether a number is larger or smaller than the median but also the magnitude of deviations from the mean. This allows the test to detect patterns related to the variability of the data.
- *Better for Continuous Data:* By focusing on absolute deviations, the test is more appropriate for continuous data where the values are not simply binary. It captures fluctuations in the data that might be missed by the basic runs test.
- *Increased Robustness:* This version of the test provides a more nuanced analysis of randomness, as it accounts for how much each value deviates from the mean, making it potentially more robust to subtle patterns in the data.

Limitations:

- *Still Requires Transformation to Binary Format:* Like the basic runs test, the data still needs to be transformed into a binary sequence (based on the absolute deviations), which could obscure some information about the original distribution of the data.
- *Does Not Handle All Types of Data Well:* Although more flexible than the basic runs test, the modified test still requires classification into binary values, which may limit its applicability in cases where detailed distributional properties are important.
- *Potential Loss of Information:* The binary classification of deviations based on the median could lose information about the degree of deviation.

Comparison:

- *Scope of Application:* The basic runs test is more suited for binary or dichotomous data, whereas the modified runs test on absolute deviations is better for continuous data or data with variability. The modified test provides more insight into the structure of the data by focusing on deviations from the mean, which can reveal underlying patterns.
- *Sensitivity:* The modified runs test is more sensitive to the structure and variability of data. By considering absolute deviations, it detects patterns in how data points fluctuate around the mean, which could be missed by the simpler binary classification in the basic runs test.
- *Complexity:* The basic runs test is simpler to apply, as it only requires the data to be converted into a binary sequence. In contrast, the modified runs test involves additional steps to compute the absolute deviations and then apply the binary classification on them.

Conclusion: Both the basic runs test and the modified runs test on absolute deviations provide valuable tools for assessing the randomness of a sequence. The basic runs test is best suited for binary sequences and simpler analyses, while the modified test is more appropriate for continuous data and can capture more nuanced information about the randomness of the deviations from the mean. Each test has its advantages and limitations, and the choice between them depends on the nature of the data and the specific hypotheses being tested.

Here is the code

```
# Step 1: Generate a sequence of random numbers between 0 and 1
set.seed(123)  # For reproducibility
n <- 100  # Length of the sequence
random_numbers <- runif(n)  # Generate random numbers between 0 and 1

# Step 2: Classify the numbers based on the median
median_value <- median(random_numbers)  # Find the median of the sequence
binary_classification <- ifelse(random_numbers > median_value, 1, 0)  # Classify based on the median

# Step 3: Calculate the deviation of each value from the mean (absolute deviation)
mean_value <- mean(random_numbers)
absolute_deviations <- abs(random_numbers - mean_value)

# Step 4: Apply runs test on the binary classification
runs_binary <- 1  # Initialize the number of runs for binary classification
for (i in 2:n) {
  if (binary_classification[i] != binary_classification[i - 1]) {
    runs_binary <- runs_binary + 1  # Increment runs when the sequence changes
  }
}

runs_binary=runs_binary/(n-1)

# Step 5: Apply runs test on the absolute deviations
# Classify the absolute deviations: 1 if deviation is greater than the median of absolute deviations, e
median_deviation <- median(absolute_deviations)
binary_deviation_classification <- ifelse(absolute_deviations > median_deviation, 1, 0)
```

```r
runs_deviation <- 1  # Initialize the number of runs for deviation classification
for (i in 2:n) {
  if (binary_deviation_classification[i] != binary_deviation_classification[i - 1]) {
    runs_deviation <- runs_deviation + 1  # Increment runs when the sequence changes
  }
}

runs_deviation=runs_deviation/(n-1)

# Step 6: Calculate the expected number of runs and variance for binary classification
p0_binary <- sum(binary_classification == 0) / n  # Proportion of 0's in binary sequence
p1_binary <- sum(binary_classification == 1) / n  # Proportion of 1's in binary sequence

expected_runs_binary <- (2 * p0_binary * p1_binary * n) / (n - 1)
variance_runs_binary <- (expected_runs_binary * (n - 1)) / (n - 2)

# Step 7: Calculate the expected number of runs and variance for deviation-based binary sequence
p0_deviation <- sum(binary_deviation_classification == 0) / n  # Proportion of 0's in deviation classif
p1_deviation <- sum(binary_deviation_classification == 1) / n  # Proportion of 1's in deviation classif

expected_runs_deviation <- (2 * p0_deviation * p1_deviation * n) / (n - 1)
variance_runs_deviation <- (expected_runs_deviation * (n - 1)) / (n - 2)

# Step 8: Compute the test statistic for both binary and deviation sequences
z_binary <- (runs_binary - expected_runs_binary) / sqrt(variance_runs_binary)
z_deviation <- (runs_deviation - expected_runs_deviation) / sqrt(variance_runs_deviation)

# Step 9: Check if the statistics are significant (typically use alpha = 0.05)
alpha <- 0.05
z_critical <- qnorm(1 - alpha / 2)  # Two-tailed test

# Step 10: Output the results
random_binary <- abs(z_binary) <= z_critical
random_deviation <- abs(z_deviation) <= z_critical

# Final decision: If either test rejects the null hypothesis, it's not random
if (random_binary & random_deviation) {
  print("The sequence is random (fail to reject null hypothesis).")
} else {
  print("The sequence is not random (reject null hypothesis).")
}
```

```
## [1] "The sequence is random (fail to reject null hypothesis)."
```

```r
# Output the results for both tests
cat("Number of runs (binary classification):", runs_binary, "\n")
```

```
## Number of runs (binary classification): 0.5252525
```

```r
cat("Expected number of runs (binary classification):", expected_runs_binary, "\n")
```

```
## Expected number of runs (binary classification): 0.5050505
```

```r
cat("Variance of the runs (binary classification):", variance_runs_binary, "\n")
```

```
## Variance of the runs (binary classification): 0.5102041
```

```r
cat("Z statistic (binary classification):", z_binary, "\n")
```

```
## Z statistic (binary classification): 0.02828283
```

```r
cat("\n")
```

```r
cat("Number of runs (deviation classification):", runs_deviation, "\n")
```

```
## Number of runs (deviation classification): 0.5555556
```

```r
cat("Expected number of runs (deviation classification):", expected_runs_deviation, "\n")
```

```
## Expected number of runs (deviation classification): 0.5050505
```

```r
cat("Variance of the runs (deviation classification):", variance_runs_deviation, "\n")
```

```
## Variance of the runs (deviation classification): 0.5102041
```

```r
cat("Z statistic (deviation classification):", z_deviation, "\n")
```

```
## Z statistic (deviation classification): 0.07070707
```