HW 5 Random Number Generators

STAT 5400

Due: Oct 4, 2024 9:30 AM

Problems

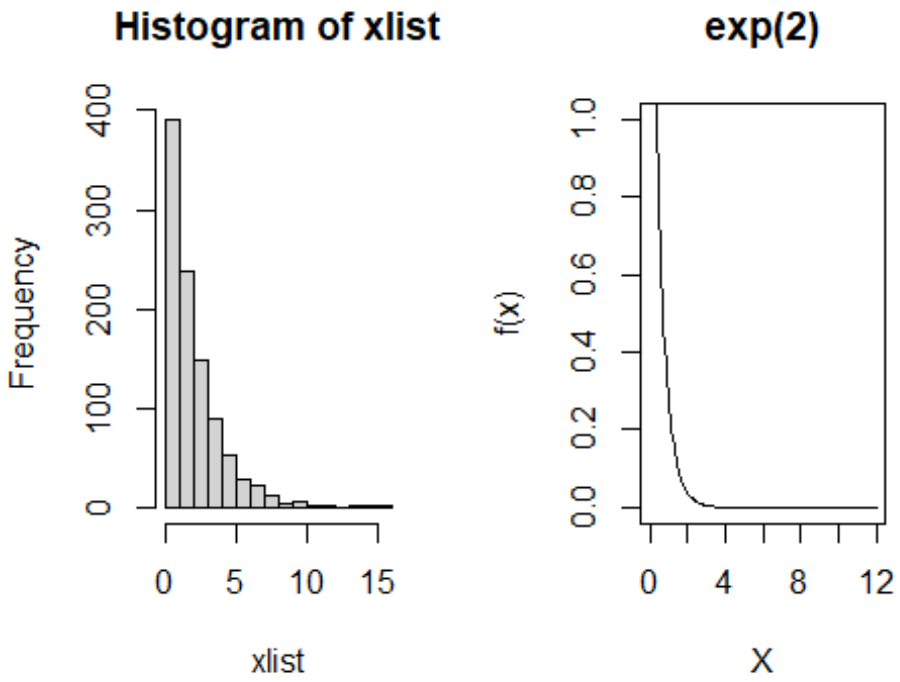
Submit your solutions as an .Rmd file and accompanying .pdf file. Include all the **relevant** R code and output. Always interpret your result whenever it is necessary.

Problems

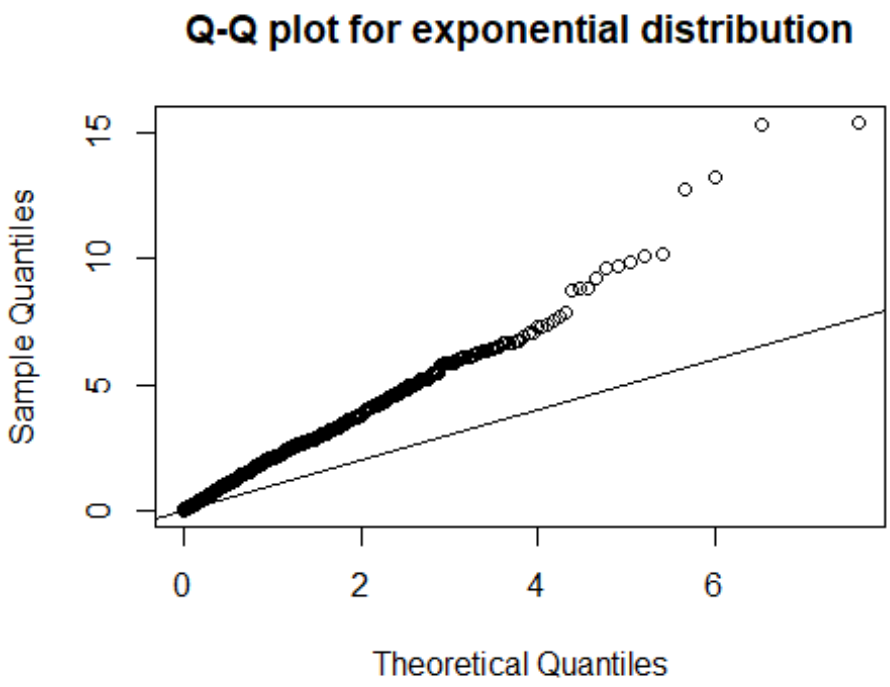
1. Generators of exponential distributions

- Fill in the code on Slide 36 of S3P1.pdf. You may either use the pdf of exp(2) manually or call `dexp` in R.
- Fill in the code on Slide 37 of S3P1.pdf.
- (Negative exponential distribution) Suppose there is a distribution with pdf $\frac{1}{2}exp\{\frac{1}{2}(x)\}, x < 0$. Generate a random sample with 1000 observations from such distribution. Have a Q-Q plot to check whether the sample comforts with this distribution.

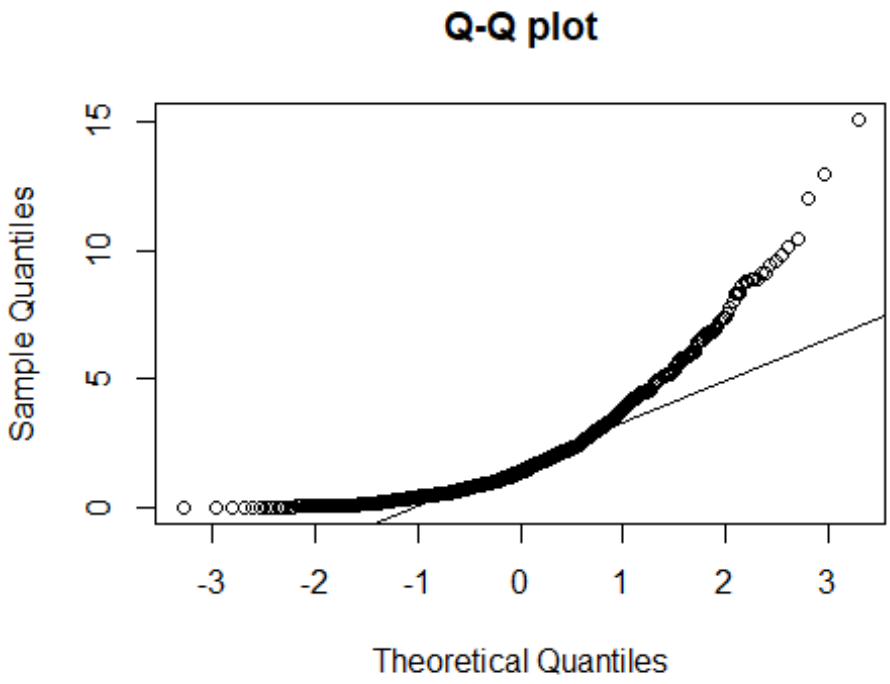
```
set.seed(123)
n <- 1000
u <- runif(n, 0, 1)  # Generate uniform random numbers
xlist <- -2 * log(u)  # Inverse CDF for negative exponential distribution
(rate = 1/2)



par(mfrow = c(1, 2))
hist(xlist)
plot(seq(0.001, 12, len = 200), 2 * exp(-2 * seq(0.001, 12, len = 200)),
type = "l", ylim = c(0, 1), xlab = "X", main = "exp(2)",
ylab = "f(x)")
```




```r
set.seed(5400)
Ulist <- runif(1000, 0, 1)
Xlist <- -2 * log(Ulist)
theoretical_quantiles <- qexp(ppoints(length(Xlist)))
plot(theoretical_quantiles, sort(Xlist), xlab = "Theoretical Quantiles",
ylab = "Sample Quantiles", main = "Q-Q plot for exponential distribution")
abline(0, 1)
```




```r
lambda <- 0.5
samples <- rexp(1000, rate = lambda)
qqnorm(samples, main = "Q-Q plot")
qqline(samples)
```

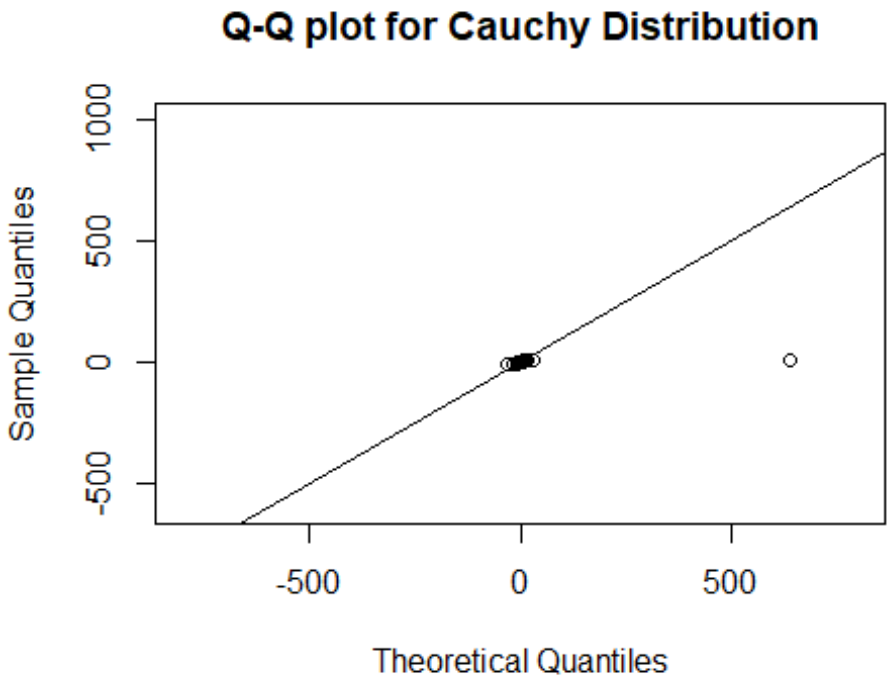


2. Generators of Cauchy distributions.

- Implement an algorithm to generate the standard Cauchy distribution using the inverse CDF.
- Check if the generated numbers conform to the standard Cauchy distribution. Basically, you may check this by filling in the code on Slide 42 of S3P1.pdf.

```
set.seed(124)
n <- 1000



setEPS()
postscript(file = "qqcauchy.eps", height = 4.5, width = 9)
# Generate uniform random numbers
u <- runif(n, 0, 1)

# Generate Cauchy distributed random variables using the inverse CDF
Xlist <- tan(pi * (u - 0.5))

# Generate theoretical quantiles for the standard Cauchy distribution
probs <- ppoints(1000)  # Generate probabilities for theoretical quantiles
sample_quantiles <- quantile(Xlist, probs)
theoretical_quantiles <- qcauchy(probs)  # Theoretical Cauchy quantiles

plot(theoretical_quantiles, sample_quantiles, xlab = "Theoretical Quantiles",
ylab = "Sample Quantiles", main = "Q-Q plot for Cauchy distribution")
```

```r
abline(0, 1)
dev.off()

## png
##   2

# Set seed for reproducibility
set.seed(124)

# Generate sample size
n <- 100

# Generate uniform random numbers
u <- runif(n, 0, 1)

# Generate Cauchy distributed random variables using qcauchy
Xlist <- qcauchy(u)  # Use qcauchy directly

# Generate theoretical quantiles for the standard Cauchy distribution
probs <- ppoints(1000)  # Generate probabilities for theoretical quantiles
theoretical_quantiles <- qcauchy(probs)  # Theoretical Cauchy quantiles

# Q-Q plot of the sample against theoretical Cauchy quantiles with Y-axis
limits
qqplot(theoretical_quantiles, Xlist,
       xlab = "Theoretical Quantiles", ylab = "Sample Quantiles",
       main = "Q-Q plot for Cauchy Distribution",
       xlim = c(-800, 800),
       ylim = c(-600, 1000))  # Set Y-axis limits

# Add a reference line
abline(0, 1)
```




```r
# Define the function to calculate X
calculate_X <- function(U1, U2, alpha) {
  # Calculate the denominator part of the formula
  denominator <- 2 * sqrt(1 / (1 + (1 / ((U1^(-1 / alpha) - 1) * cos(2 * pi *
U2)^2))))

  # Calculate X
  X <- (1 / 2) + (1 / denominator)

  return(X)
}

# Generate independent random variables U1, U2, U3 from U(0, 1)
set.seed(123)  # For reproducibility
U1 <- runif(1)
U2 <- runif(1)
U3 <- runif(1)

# Set the value for alpha
alpha <- 1  # You can adjust this value as needed

# Calculate X using the generated U1, U2, and alpha
X <- calculate_X(U1, U2, alpha)

# Check the condition for the indicator function
indicator_value <- ifelse(U3 <= 1 / 2, 1, -1)
```

```r
# Print the results
result <- list(U1 = U1, U2 = U2, U3 = U3, X = X, Indicator = indicator_value)
print(result)
```

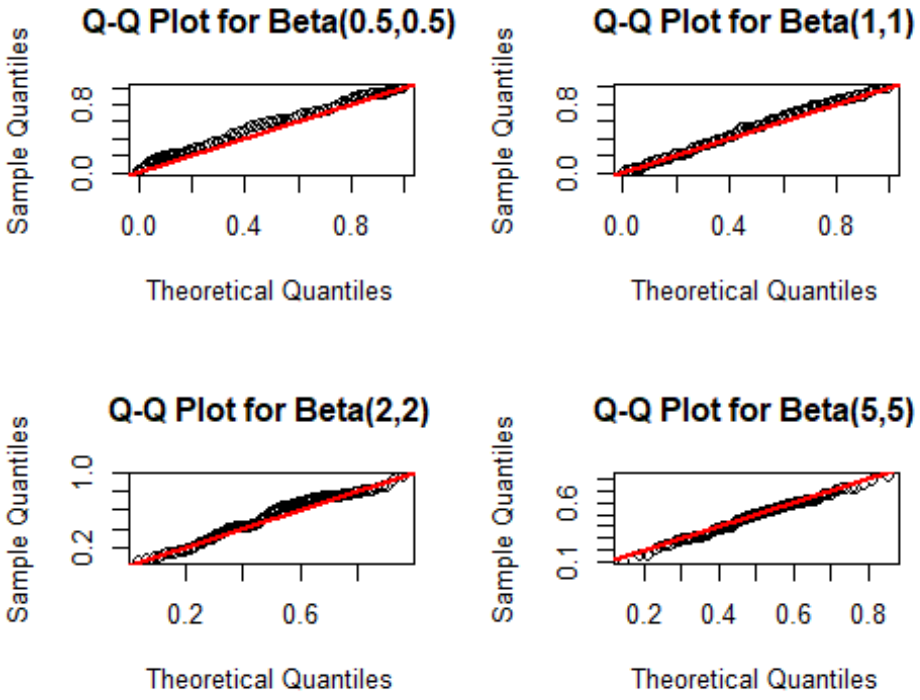
```
## $U1
## [1] 0.2875775
##
## $U2
## [1] 0.7883051
##
## $U3
## [1] 0.4089769
##
## $X
## [1] 1.923437
##
## $Indicator
## [1] 1
```

```r
# Load necessary libraries
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```r
# Function to generate samples and create Q-Q plot
generate_and_plot <- function(alpha) {
  # Generate 100 samples from Beta(α, α) adn 100 observations.
  samples <- rbeta(100, shape1 = alpha, shape2 = alpha)

  # Create a sequence of quantiles for the theoretical Beta distribution
  theoretical_quantiles <- qbeta(ppoints(100), shape1 = alpha, shape2 =
alpha)

  # Q-Q plot to compare the sample quantiles with the theoretical quantiles
  qqplot(theoretical_quantiles, samples,
         main = paste("Q-Q Plot for Beta(", alpha, ",", alpha, ")", sep =
""),
         xlab = "Theoretical Quantiles",
         ylab = "Sample Quantiles")

  # Add a reference line
  abline(0, 1, col = "red", lwd = 2)
}

# Set values for alpha
alpha_values <- c(0.5, 1, 2, 5)

# Generate Q-Q plots for different values of alpha
par(mfrow = c(2, 2))  # Set up the plotting area for 4 plots
```

```
for (alpha in alpha_values) {
  generate_and_plot(alpha)
}
```




```
# Reset plotting area to default
par(mfrow = c(1, 1))

calculate_t <- function(n, Y) {
  # Calculate the t-statistic
  Z <- sqrt(2 * n) * (Y - 0.5) / sqrt(Y * (1 - Y))

  return(Z)
}

n <- 10
Y <- 0.6
t_value <- calculate_t(n, Y)
print(t_value)
```

```
## [1] 0.9128709
```

3. Generators of Beta distributions and t distributions.

- Generate a Beta(α, α) distribution using

$$X = \frac{1}{2} + \frac{\mathbb{I}_{U_3 \le 1/2}}{2\sqrt{1 + \frac{1}{\left(U_1^{-1/\alpha} - 1\right)\cos^2(2\pi U_2)}}}.$$

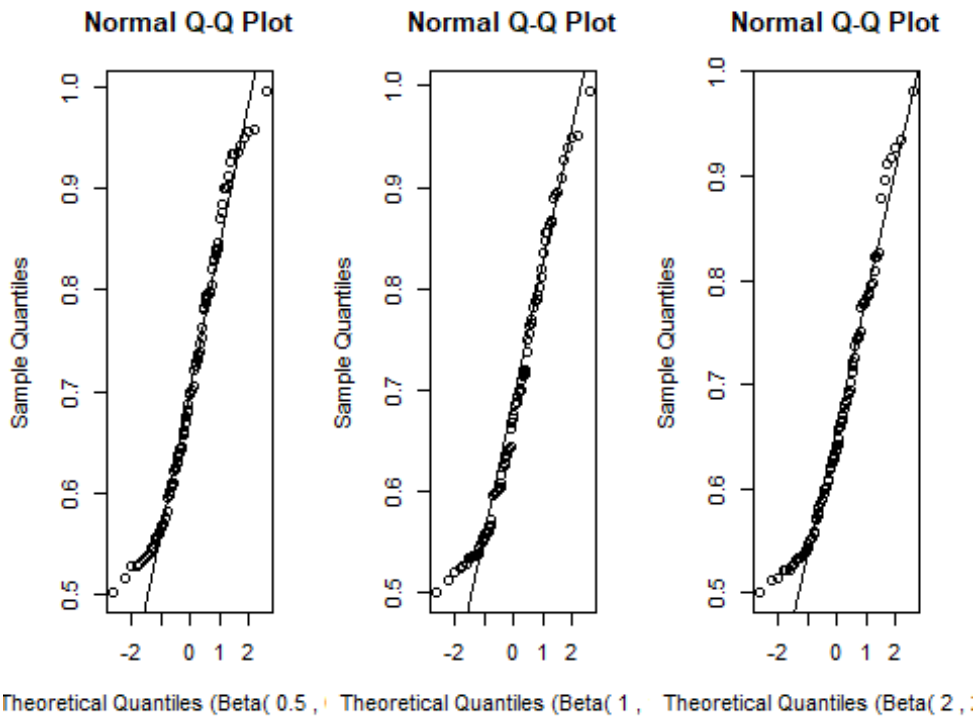
where U_1, U_2, and U_3 are independently generated random vaiables on (0,1), and the indicator function $\mathbb{I}_{U_3 \le 1/2}$ takes the value of 1 if the condition is true or the value of -1 otherwise.

- With several values of α, generate a sample of 100 observations from Beta(α, α). Have a Q-Q plot to check whether the sample comforts with the beta distribution. You may use the built-in function in R to give the inverse CDF function.
- Using the above method to generate $Y \sim$ Beta(n, n). Generate a t-distribution with degree of freedom $2n$ using

$$Z = \frac{\sqrt{n}(Y - 1/2)}{2\sqrt{Y(1-Y)}}.$$

- Generate a sample of 100 observations from $t(1)$, namely Cauchy distribution. Have a Q-Q plot to check whether the sample comforts with the $t(1)$ distribution. Compare the Q-Q plot with the one you plotted in the previous question.

```r
# Define the parameters
alpha_vals <- c(0.5, 1, 2)   # Different values of alpha
n_samples <- 100
U1 <- runif(n_samples)       # Generate random numbers between 0 and 1 for V1
U2 <- runif(n_samples)       # Generate random numbers between 0 and 1 for V2
U3 <- runif(n_samples) / 2   # Generate random numbers between 0 and 0.5 for
V3

# Beta transformation function
beta_transform <- function(alpha, U1, U2, U3) {
  Y <- 0.5 + U3 / sqrt(1 + 1 / ((U1^(-1/alpha) - 1)) * cos(2 * pi * U2)^2)
  return(Y)  # Ensure the function returns the value of Y
}

# Apply the transformation for different alpha values
transformed_samples <- lapply(alpha_vals, function(alpha)
beta_transform(alpha, U1, U2, U3))

# Create Q-Q plots for each set of samples
par(mfrow = c(1, length(alpha_vals)))  # Set up a grid for multiple plots
for (i in seq_along(alpha_vals)) {
  qqnorm(transformed_samples[[i]],
         xlab = paste("Theoretical Quantiles (Beta(", alpha_vals[i], ",",
alpha_vals[i], "))"),
         ylab = "Sample Quantiles")
  qqline(transformed_samples[[i]])  # Add a Q-Q line for reference
}
```

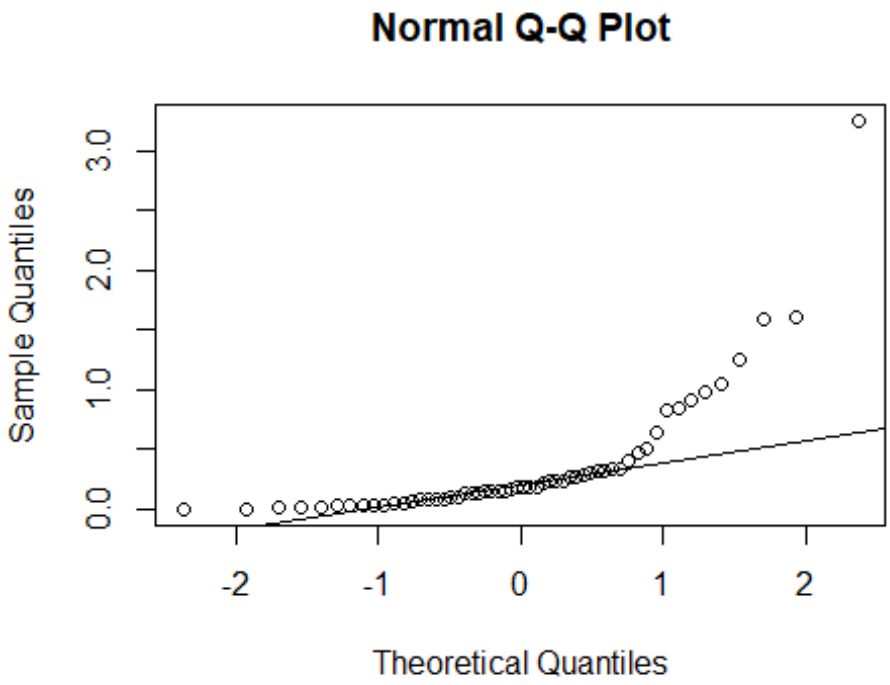



```r
par(mfrow = c(1, 1))

# Function
generate_t_distribution <- function(n, num) {
  # Initialize a vecto
  t_distribution <- numeric(num)

  # Loop over the
  for (i in 1:num) {
    # Generate Y ~ Beta(n, n)
    Y <- beta_transform(n, runif(1), runif(1), runif(1))

    # Calculate Z
    Z <- (sqrt(n) * (Y - 1/2)) / (2 * sqrt(Y * (1 - Y)))

    # Store Z in the vector
    t_distribution[i] <- Z
  }

  return(t_distribution)
}


n <- 0.5
num <- 100
```

```r
# Generate t-distributed random variables
t_samples <- generate_t_distribution(n, num)
```

```
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
## Warning in sqrt(Y * (1 - Y)): NaNs produced
```

```r
# normal distribution
qqnorm(t_samples)
qqline(t_samples)  # reference line
```



4. More on accept-reject sampling.

- Generate n values from the truncated normal distribution:

$$Y_i \sim (X|a < X < b), \text{where } X \sim N(\mu, \sigma^2).$$

Hint: generate observations from normal distribution using `rnorm``` and discard it if it falls outside the interval.
This function should have five arguments:

  - `n`: number of observations,
  - `mu`: the value of mean μ,
  - `sigma`: the value of standard deviation σ,
  - `a`: the left endpoint of the interval of `-Inf`,
  - `b`: the right endpoint of the interval of `Inf`, This function should return a vector of n truncated normal variables.

```r
generate_truncated <- function(n, mu, sigma, a, b) {
  observe <- numeric(n)
  counter <- 1
  while (counter <= n) {
    sample <- rnorm(1, mu, sigma)
    if (sample > a && sample < b) {
      observe[counter] <- sample
```

```
      counter <- counter + 1
    }
  }
  return(observe)
}

n <- 100  # Number of observations
mu <- 5    # Mean
sigma <- 2 # Standard deviation
a <- 2     # Left endpoint
b <- 8     # Right endpoint

# Generate truncated normal distribution
generate_truncated(n, mu, sigma, a, b)
```

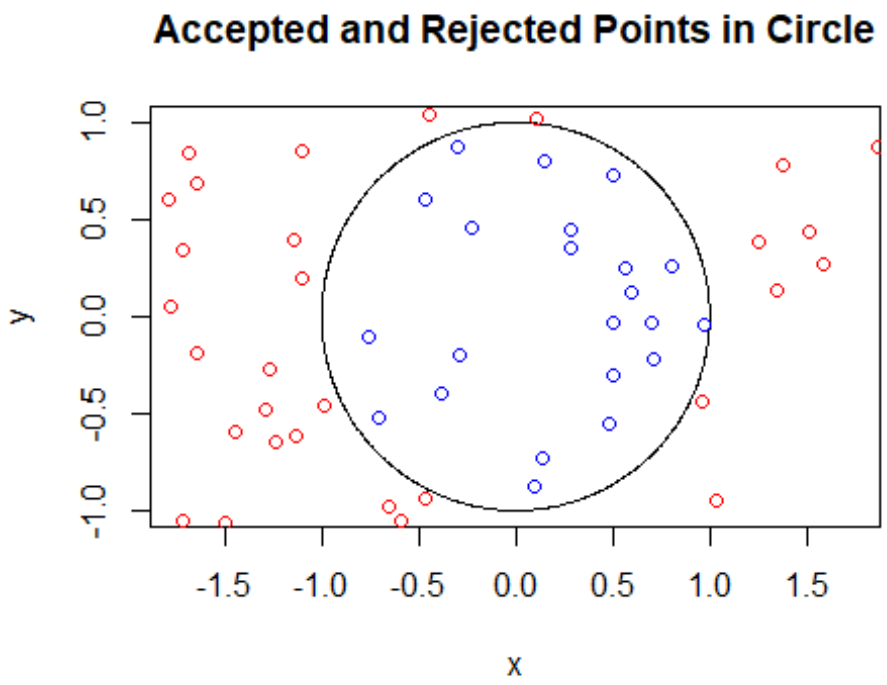
```
##   [1] 2.569197 6.784351 4.592700 6.914140 6.449450 6.770668 3.322178
4.685636
##   [9] 3.526354 5.284873 4.665727 5.490427 3.474247 4.596126 4.391391
3.254694
##  [17] 4.661114 4.588097 3.454220 6.018819 4.566699 3.651115 3.534744
4.881341
##  [25] 6.501052 5.265864 3.818443 4.625634 2.299708 4.470959 4.056202
3.861628
##  [33] 4.021911 3.461751 3.067602 7.536100 7.206665 4.275916 6.324494
4.666914
##  [41] 4.990050 2.238226 2.486946 5.650491 4.445999 4.221978 7.420788
5.469833
##  [49] 2.130643 3.156556 3.915808 2.697575 6.835115 2.728294 4.847283
6.228003
##  [57] 4.458464 4.204915 3.516583 6.442198 7.080016 3.221157 7.488995
5.020519
##  [65] 5.794353 7.474647 3.697871 6.663032 6.901007 7.296893 2.433465
6.178911
##  [73] 4.403268 5.107563 7.402617 6.573104 5.727994 4.493630 4.444056
3.872262
##  [81] 5.295920 6.319432 5.805837 5.681938 3.863282 5.195927 4.114186
5.490182
##  [89] 2.233412 7.299209 7.012935 5.785141 3.227613 2.256491 6.143101
5.479962
##  [97] 5.586887 7.286294 5.063484 2.544832
```

5. Uniformly generating points within a circle.

- Implement an algorithm to perform the following steps to uniformly generating points within a circle.
  - Generate a random angle θ uniformly distributed in the range $[0,2\pi)$.
  - Genreate a random radius r uniformly distributed in the range [0, 1) with a square root scale, i.e., r <- sqrt(runif(1))`.
  - Give polar coordinates (r, θ), get the Cartesian coordinates $x = r\cos\theta$ and $y = r\sin\theta$.

- Generate 500 points and use the code on Slide 48 of S3P1.pdf to plot the points.
- Compare the run time between this algorithm and the one based on accept/reject sampling (Slide 45 of S3P1.pdf).

```
generate_points_in_circle <- function(n, radius) {
  theta <- runif(n, 0, 2 * pi)   # Generate random angles in [0, 2pi)
  r <- sqrt(runif(n)) * radius   # Generate random radii with square root
scale
  x <- r * cos(theta)            # Convert to Cartesian coordinates
  y <- r * sin(theta)
  return(data.frame(x, y))
}

n <- 500
radius <- 5
points <- generate_points_in_circle(n, radius)

# Accept if V_1^2 + V_2^2 <= 1.
accept <- (points[, 1]^2 + points[, 2]^2) <= 1
mean(accept)  # This should be approximately pi/4

## [1] 0.044

# Generate points
V <- points[accept, ]
V_out <- points[!accept, ]

# Define a sequence of angles to draw a circle
theta <- seq(0, 2 * pi, length.out = 1000)

# Draw the unit circle
plot(sin(theta), cos(theta), type = "l", xlab = "x", ylab = "y", asp = 1,
     main = "Accepted and Rejected Points in Circle")

# Plot accepted points (inside the circle) in blue
points(V$x, V$y, pch = 1, col = "blue")

# Plot rejected points (outside the circle) in red
points(V_out$x, V_out$y, pch = 1, col = "red")
```



6. (Optional) Advanced readings on accept-reject sampling.

Read more on accept-reject sampling on Section 4.7 and 4.8 of http://statweb.stanford.edu/~owen/mc/Ch-nonunifrng.pdf. Implement Algorithm 4.8, which generates a gamma distribution using the accept-reject sampling.