Assignment 4

Emre Beray Boztepe

17 12 2023

Question 1:

Consider a low dimensional setup: n = 20 and

a. mu_1: one needle of length $1.2\sqrt{2log(n)}$, other mu's are 0

b. mu_2: Five needles of length $1.02\sqrt{2log\left(\frac{n}{10}\right)}$, other mu's are 0

c. mu_2: i needles of length $\sqrt{2log\left(\frac{20}{i}\right)}$, other mu's are 0

Compare FWER, FDR and Power (proportion of identified alternative hypothesis among all alternative hypotheses) of the following procedures: - Bonferroni, - Sidak's procedure with $\alpha_n = 1 - (1-\alpha)^{1/n}$, - Holm, - Hochberg, - Benjamini-Hochberg.

Importing the library

```
library(knitr)

## Warning: package 'knitr' was built under R version 4.3.2
```

Define definitions

```
n = 20
alpha = 0.05
set.seed(2021)
```

Function does calculates all setups of size n with given its mu value.

```
all_mu = function(){
  mus = list(
    c(1.2*sqrt(2*log(n)), rep(0, n-1)),
    c(rep(1.02*sqrt(2*log(n/10)), 5), rep(0, n-5))
  )
  mu_3 = c()
  for(i in 1:10){
    mu_3 = append(mu_3, sqrt(2*log(20/i)))
  }
  mu_3 = append(mu_3, rep(0, n-10))
  mus[[3]] = mu_3
  return(mus)
}
```

Functions the apply the procedures

```
# Function to apply Bonferonni procedure
bonferroni_procedure = function(p_values){
  n = length(p_values)
  return(p_values <= (alpha / n))
}

# Function to apply Sidak procedure
sidak_procedure = function(p_values){
  n = length(p_values)
  return(p_values <= 1 - ((1-alpha)^(1/n)))
}

# Function to apply Holm procedure
holm_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  holm_procedure_formula = (p_values[ordered_p_values] <= (alpha / (n + 1 -
seq(n))))
  sapply(1:n, function(i)
all(holm_procedure_formula[1:i]))[perm_ordered_p_values]
}

# Function to apply Hochberg procedure
hochberg_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  hochberg_procedure_formula = (p_values[ordered_p_values] <= (alpha / (n + 1
- seq(n))))
  sapply(1:n, function(i)
any(hochberg_procedure_formula[i:n]))[perm_ordered_p_values]
}

# Function to apply Benjamini-Hochberg procedure
benjamini_hochberg_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  benjamini_hochberg_procedure_formula = (p_values[ordered_p_values] <=
(alpha * seq(n) / n))
```

```
  sapply(1:n, function(i)
any(benjamini_hochberg_procedure_formula[i:n]))[perm_ordered_p_values]
}
```

Functions to calculate FWER, FDR and Power

```
# Function to calculate Family-Wise Error Rate
FWER = function(true_values, test_results) {
  return(as.integer(any(test_results[which(!true_values)])))
}

# Function to calculate False Discovery Rate
FDR = function(true_values, test_results) {
  return(sum(test_results[which(!true_values)]) / max(sum(test_results), 1))
}

# Function to calculate Power
power = function(true_values, test_results) {
  return(mean(test_results[which(true_values)]))
}
```

This function is where the simulation starts. Basically generates n sized of random data and gives it to the related functions to apply procedures and calculates FWER, FDR and Power based on these results

```
simulate_tests = function(mu){
  n = length(mu)
  X = rnorm(n, mu)
  p_values = 2*(1 - pnorm(abs(X)))

  test_results = list(bonferroni_procedure = bonferroni_procedure(p_values),
                      sidak_procedure = sidak_procedure(p_values),
                      holm_procedure = holm_procedure(p_values),
                      hochberg_procedure = hochberg_procedure(p_values),
                      benjamini_hochberg_procedure =
benjamini_hochberg_procedure(p_values))

  results = sapply(test_results, function(test_result) list(FWER = FWER(mu >
0, test_result),
                                                            FDR = FDR(mu >
0, test_result),
                                                            power = power(mu
> 0, test_result)))
  return(results)
}
```

Function to start simulation with given replicate count

```
start_simulating = function(replicate_count, mu)
{
  simulation = replicate(replicate_count, simulate_tests(mu))
```

```
  apply(simulation, c(1, 2), function(x) mean(as.numeric(x)))
}
```

Here, the simulation starts and the results are printed

```
mu_values = all_mu()
kable(start_simulating(100, mu_values[[1]]), digits=3, caption = "Option A",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option A

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.060	0.07	0.070	0.070	0.070
FDR	0.035	0.04	0.042	0.042	0.042
power	0.570	0.58	0.570	0.570	0.570

```
kable(start_simulating(100, mu_values[[2]]), digits=3, caption = "Option B",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option B

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.030	0.030	0.03	0.03	0.040
FDR	0.030	0.030	0.03	0.03	0.035
power	0.058	0.058	0.06	0.06	0.068

```
kable(start_simulating(100, mu_values[[3]]), digits=3, caption = "Option C",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option C

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.050	0.050	0.050	0.050	0.110
FDR	0.028	0.028	0.028	0.028	0.038
power	0.107	0.109	0.110	0.110	0.173

Comments:

All procedures controlling FWER (Bonferroni's, Sidak's, Holm's, Hochberg's) give similar results. As Expected: - FDR is not larger than FWER - So, it can be said that if test controls FWER, it also controls FDR - Holm and Hochberg is so close to each other by almost giving the same results for all (FWER, FDR, Power) - Benjamini-Hochberg's method does control FDR but it doesn't control FWER. In all cases, it has greater power (at the expense of more false discoveries - greater FDR and FWER).

Question 2:

Consider a large dimensional setup: n = 5000 and

a. mu_1: one needle of length $1.2\sqrt{2log(n)}$, other mu's are 0

b. mu_2: 100 needles of length $1.02\sqrt{2log\left(\frac{n}{200}\right)}$, other mu's are 0

c. mu_2: 100 needles of length $\sqrt{2log\left(\frac{n}{200}\right)}$, other mu's are 0

d. mu_3: 1000 needles of length $1.002\sqrt{2log\left(\frac{n}{2000}\right)}$

Compare FWER, FDR and Power (proportion of identified alternative hypothesis among all alternative hypotheses) of the following procedures: - Bonferroni, - Sidak's procedure with $\alpha_n = 1 - (1-\alpha)^{1/n}$, - Holm, - Hochberg, - Benjamini-Hochberg.

Define definitions

```
n = 5000
alpha = 0.05
set.seed(2021)
```

Function does calculates all setups of size n with given its mu value.

```
all_mu = function(){
  return(list(
    c(1.2*sqrt(2*log(n)), rep(0, n-1)),
    c(rep(1.02*sqrt(2*log(n/200)), 100), rep(0, n-100)),
    c(rep(sqrt(2*log(n/200)), 100), rep(0, n-100)),
    c(rep(1.002*sqrt(2*log(n/2000)), 1000), rep(0, n-1000))
  ))
}
```

Functions the apply the procedures

```
# Function to apply Bonferonni procedure
bonferroni_procedure = function(p_values){
  n = length(p_values)
  return(p_values <= (alpha / n))
}

# Function to apply Sidak procedure
sidak_procedure = function(p_values){
  n = length(p_values)
  return(p_values <= 1 - ((1-alpha)^(1/n)))
}

# Function to apply Holm procedure
```

```r
holm_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  holm_procedure_formula = (p_values[ordered_p_values] <= (alpha / (n + 1 -
seq(n))))
  sapply(1:n, function(i)
all(holm_procedure_formula[1:i]))[perm_ordered_p_values]
}

# Function to apply Hochberg procedure
hochberg_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  hochberg_procedure_formula = (p_values[ordered_p_values] <= (alpha / (n + 1
- seq(n))))
  sapply(1:n, function(i)
any(hochberg_procedure_formula[i:n]))[perm_ordered_p_values]
}

# Function to apply Benjamini-Hochberg procedure
benjamini_hochberg_procedure = function(p_values){
  n = length(p_values)

  ordered_p_values = order(p_values)
  perm_ordered_p_values = order(ordered_p_values)

  benjamini_hochberg_procedure_formula = (p_values[ordered_p_values] <=
(alpha * seq(n) / n))
  sapply(1:n, function(i)
any(benjamini_hochberg_procedure_formula[i:n]))[perm_ordered_p_values]
}
```

Functions to calculate FWER, FDR and Power

```r
# Function to calculate Family-Wise Error Rate
FWER = function(true_values, test_results) {
  return(as.integer(any(test_results[which(!true_values)])))
}

# Function to calculate False Discovery Rate
FDR = function(true_values, test_results) {
  return(sum(test_results[which(!true_values)]) / max(sum(test_results), 1))
}
```

```
# Function to calculate Power
power = function(true_values, test_results) {
  return(mean(test_results[which(true_values)]))
}
```

This function is where the simulation starts. Basically generates n sized of random data and gives it to the related functions to apply procedures and calculates FWER, FDR and Power based on these results

```
simulate_tests = function(mu){
  n = length(mu)
  X = rnorm(n, mu)
  p_values = 2*(1 - pnorm(abs(X)))

  test_results = list(bonferroni_procedure = bonferroni_procedure(p_values),
                      sidak_procedure = sidak_procedure(p_values),
                      holm_procedure = holm_procedure(p_values),
                      hochberg_procedure = hochberg_procedure(p_values),
                      benjamini_hochberg_procedure =
benjamini_hochberg_procedure(p_values))

  results = sapply(test_results, function(test_result) list(FWER = FWER(mu >
0, test_result),
                                                            FDR = FDR(mu > 0,
test_result),
                                                            power = power(mu
> 0, test_result)))
  return(results)
}
```

Function to start simulation with given replicate count

```
start_simulating = function(replicate_count, mu)
{
  simulation = replicate(replicate_count, simulate_tests(mu))
  apply(simulation, c(1, 2), function(x) mean(as.numeric(x)))
}
```

Here, the simulation starts and the results are printed

```
mu_values = all_mu()
kable(start_simulating(100, mu_values[[1]]), digits=3, caption = "Option A",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option A

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.030	0.030	0.030	0.030	0.070

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FDR	0.017	0.017	0.017	0.017	0.038
power	0.830	0.830	0.830	0.830	0.830

```
kable(start_simulating(100, mu_values[[2]]), digits=3, caption = "Option B",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option B

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.040	0.040	0.040	0.040	0.400
FDR	0.014	0.014	0.014	0.014	0.048
power	0.035	0.035	0.035	0.035	0.100

```
kable(start_simulating(100, mu_values[[3]]), digits=3, caption = "Option C",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option C

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.040	0.050	0.040	0.040	0.440
FDR	0.010	0.015	0.010	0.010	0.054
power	0.032	0.032	0.032	0.032	0.085

```
kable(start_simulating(100, mu_values[[4]]), digits=3, caption = "Option D",
col.names = c("Bonferroni's", "Sidak's", "Holm's", "Hochberg's", "Benjamini-
Hochberg's"))
```

Option D

	Bonferroni's	Sidak's	Holm's	Hochberg's	Benjamini-Hochberg's
FWER	0.100	0.100	0.100	0.100	0.250
FDR	0.065	0.065	0.065	0.065	0.053
power	0.001	0.001	0.001	0.001	0.004

Comments:

All comments that has been made for the first question can also said for this question too. So, the main question is: Low dimensional case controls FWER, FDR and Power or Large dimensional? Since, n increases, the program starts to run slower as expected. Also, mu values changed too. For example, for mu_1, we had 2.937296 when n = 20, and 4.952728 when n = 5000. So, the signal becomes more powerful. It also reflects to the results. There is a noticeable decrease for FWER and FDR results in all tests. Also, the power is now closer to 0 with having 0.8300 in all tests. So, it can be said that large dimensional setup is more interesting. Also, when powerful signal count increases, the performation gets worse.

Question 3:

Apply two-step Fisher procedure using - Bonferroni, - chi-square test for the first step in the following cases n ∈ {20, 5000} and

a. mu_1: one needle of length $1.2\sqrt{2log(n)}$, other mu's are 0

b. mu_2: Five needles of length $1.02\sqrt{2log\left(\frac{n}{10}\right)}$, other mu's are 0

c. mu_2: i needles of length $\sqrt{2log\left(\frac{20}{i}\right)}$, other mu's are 0

d. mu_3: 1000 needles of length $1.002\sqrt{2log\left(\frac{n}{2000}\right)}$

Compare FWER (in the strong sense), FWER (in the weak sense), FDR and Power (proportion of identified alternative hypothesis among all alternative hypotheses).

Define definitions

```
set.seed(2021)
alpha = 0.05
trajectory = 1000
```

Functions for calculating FWER (strong and weak), FDR and Power

```
# Function for calculating FWER strong
FWER_strong = function(true_values, test_results) {
  return(as.integer(any(test_results[which(!true_values)])))
}

# Function for calculating FWER weak
FWER_weak = function(true_values, test_results) {
  return(as.integer(sum(test_results[which(!true_values)])))
}

# Function for calculating FDR
FDR = function(true_values, test_results) {
  return(sum(test_results[which(!true_values)]) / max(sum(test_results), 1))
}

# Function for calculating Power
power = function(true_values, test_results) {
  return(mean(test_results[which(true_values)]))
}
```

Function for calculating bonferroni closure

```
bonferroni_closure = function(p_values)
{
```

```
  n = length(p_values)
  return(p_values <= (alpha / n))
}
```

Function for calculating chisq closure by calculating squares and critical values and comparing them. The function returns values when n is low but when n increases, calculating chi-squared closer becomes really hard

```
chisq_closure = function(X) {
  chi_sq_values = X^2
  chi_sq_cutoff = qchisq(1 - alpha/2, df = length(X), lower.tail = FALSE)
  return(chi_sq_values >= chi_sq_cutoff)
}
```

This function is where the simulation starts. Basically generates n sized of random data and gives it to the related functions to apply procedures and calculates FWER strong, FWER weak, FDR and Power based on these results

```
simulate_tests = function(mu_vec) {
  n = length(mu_vec)
  X = rnorm(n, mu_vec)
  p_values = 2*(1 - pnorm(abs(X)))

  test_results = list(bonferroni_closure = bonferroni_closure(p_values),
                      chisq_closure = chisq_closure(X))

  results = sapply(test_results, function(test_result) list(FWER_strong =
FWER_strong(mu_vec > 0, test_result),
                                                            FWER_weak =
FWER_weak(mu_vec > 0, test_result),
                                                            FDR = FDR(mu_vec
> 0, test_result),
                                                            power =
power(mu_vec > 0, test_result)))
  return(results)
}
```

Function to start simulation

```
start_simulating = function(mu_vec, m) {
  simulation = replicate(m, simulate_tests(mu_vec))
  apply(simulation, c(1, 2), function(x) mean(as.numeric(x)))
}
```

Function does calculates all setups of size n with given its mu value.

```
all_mu = function(n)
{
  if(n == 5000){
    mus = list(
      c(1.2 * sqrt(2 * log(10)), rep(0, n-1)),
```

```
      c(rep(1.02 * sqrt(2 * log(n/10)), 5),  rep(0,  n-5)))
    mu_3 = c()
    for(i in 1:10){
      mu_3 = append(mu_3, sqrt(2*log(20/i)))
    }
    mu_3 = append(mu_3, rep(0, n-10))
    mus[[3]] = mu_3
    mus[[4]] = c(rep(1.002 * sqrt(2 * log(n/2000)), 1000),  rep(0,  n-1000))
    return(mus)
  }
  else{
    mus = list(
      c(1.2 * sqrt(2 * log(10)), rep(0, n-1)),
      c(rep(1.02 * sqrt(2 * log(n/10)), 5),  rep(0,  n-5)))
    mu_3 = c()
    for(i in 1:10){
      mu_3 = append(mu_3, sqrt(2*log(20/i)))
    }
    mu_3 = append(mu_3, rep(0, n-10))
    mus[[3]] = mu_3

    return(mus)
  }
}
```

Calculate all mu values when n = 20 and apply bonferroni and chi-squared closure

```
mu_values_n_20 = all_mu(20)
mu_1_n_20 = mu_values_n_20[[1]]
mu_2_n_20 = mu_values_n_20[[2]]
mu_3_n_20 = mu_values_n_20[[3]]

kable(start_simulating(mu_1_n_20, trajectory), digits=3, caption = "Option A,
n = 20",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option A, n = 20

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.049	0.038
FWER_weak	0.051	0.040
FDR	0.039	0.031
power	0.351	0.325

```
kable(start_simulating(mu_2_n_20, trajectory), digits=3, caption = "Option B,
n = 20",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option B, n = 20

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.035	0.030

	Bonferroni_Closure	Chi-squared_Closure
FWER_weak	0.036	0.031
FDR	0.033	0.029
power	0.033	0.028

```
kable(start_simulating(mu_3_n_20, trajectory), digits=3, caption = "Option C,
n = 20",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option C, n = 20

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.020	0.016
FWER_weak	0.020	0.016
FDR	0.010	0.008
power	0.112	0.099

Calculate all mu values when n = 5000 and apply bonferroni and chi-squared closure

```
mu_values_n_5000 = all_mu(5000)
mu_1_n_5000 = mu_values_n_5000[[1]]
mu_2_n_5000 = mu_values_n_5000[[2]]
mu_3_n_5000 = mu_values_n_5000[[3]]
mu_4_n_5000 = mu_values_n_5000[[4]]

kable(start_simulating(mu_1_n_5000, trajectory), digits=3, caption = "Option
A, n = 5000",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option A, n = 5000

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.048	0
FWER_weak	0.051	0
FDR	0.048	0
power	0.031	0

```
kable(start_simulating(mu_2_n_5000, trajectory), digits=3, caption = "Option
B, n = 5000",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option B, n = 5000

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.047	0
FWER_weak	0.047	0
FDR	0.028	0
power	0.205	0

```
kable(start_simulating(mu_3_n_5000, trajectory), digits=3, caption = "Option
C, n = 5000",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option C, n = 5000

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.048	0
FWER_weak	0.050	0
FDR	0.048	0
power	0.005	0

```
kable(start_simulating(mu_4_n_5000, trajectory), digits=3, caption = "Option
D, n = 5000",  col.names = c("Bonferroni_Closure", "Chi-squared_Closure"))
```

Option D, n = 5000

	Bonferroni_Closure	Chi-squared_Closure
FWER_strong	0.044	0
FWER_weak	0.044	0
FDR	0.027	0
power	0.001	0

Comments:

When n = 20, according to the results, it can be said that Bonferroni has better power in all mu setups but higher FWER, FDR values. As expected, chi-squared closure deals better with detecting distributed effects. When n increases, Bonferroni starts to perform worse. Calculating chi-squared becomes really hard. The chi-squared distribution with n−1 degrees of freedom becomes very large, leading to a high critical value. If the chi-squared values calculated from X are not reaching or exceeding this critical value, the function consistently returns 0.

Question 4:

For n = 5000 simulate 1000 trajectories of the empirical process $U_n(t) = \sqrt{n}(F_n(t) - t)$ $t \in$ [0,1] and 1000 trajectories of the Brownian bridge B(t), t ∈ [0, 1] (see BBridge {SDE}). Plot 5 trajectories for each of these processes on the same graph. Based on these simulations estimate the α quantile of the K-S statistics under the null hypothesis as well as α quantile of T = supt∈[0;1] |B(t)| for α = 0, 8; 0, 9; 0, 95. Discuss the results.

Importing necessary libraries

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.2
```

```
library(reshape2)
```

```
## Warning: package 'reshape2' was built under R version 4.3.2
```

Defining variables

```
n = 5000
trajectory = 1000
plotting_trajectory = 5
alphas =  c(0.8,  0.9,  0.95)
plotting_data = data.frame()
set.seed(2021)
```

Function for empirical process by its formula

```
empirical_process = function(p_val) {
  n = length(p_val)
  t_stats =  seq(n) / n
  CDF = ecdf(p_val)
  U = sqrt(n) * (CDF(t_stats) - t_stats)
  return(U)
}
```

Function for brownian bridge by its formula. Since it seems a bit hard to implement BB from sde library, I decided to go for its formula by checking in the internet.

```
brownian_bridge = function(z_stat) {
  n = length(z_stat)
  Ws = cumsum(z_stat) / sqrt(n)
  Bs = Ws - seq(n) / n * Ws[n]
  return(Bs)
}
```

Simulation for the both stats. store the values inside an array and return it

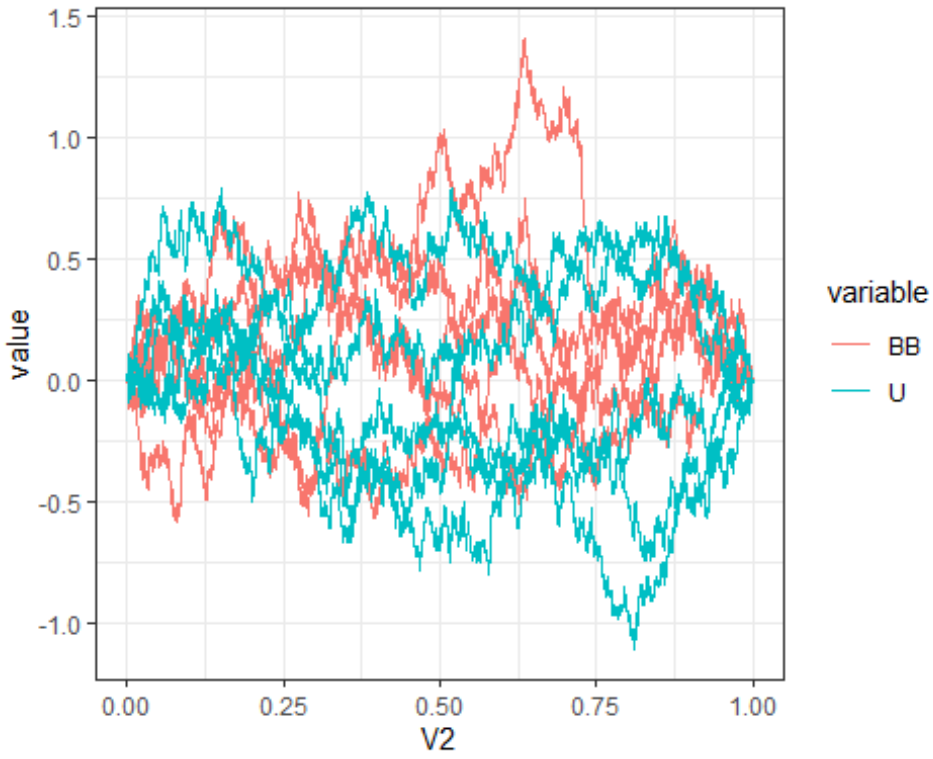
```
apply_simulation = function(n) {
  data = rnorm(n)
  p_values = 2 * (1 - pnorm(abs(data)))
  a = array(dim=c(2, n), dimnames = list(c("BB", "U")))
  a["BB", ] = brownian_bridge(data)
  a["U", ] = empirical_process(p_values)
  return(a)
}

simulation = replicate(trajectory, apply_simulation(n))
```

Store five trajectories from results for both brownian bridge and empririal process for plotting

```
for (i in 1:plotting_trajectory) {
  plotting_data = rbind(plotting_data, cbind(i, seq(n)/n, sapply(c("BB",
"U"), function(sim) simulation[sim, , i])))
}
```

Adjust the graph by updating x and y axes, a legend to show which line represents which stats, adding colors to distinguish between them and so on. Finally, plot the graph

```
plotting_data  = melt(plotting_data, id.vars=c("i", "V2"),  measure.vars =
c("BB",  "U"))
plotting_data$i = factor(plotting_data$i,  ordered=T)
ggplot(plotting_data, aes(x=V2, y=value,  color=variable,
fantom_variable=i)) + geom_line() + theme_bw()
```



Apply T stats on Brownian Bridge statistic by the formula: T = supt∈[0;1] |B(t)|

```
T_stat = sapply(seq(dim(simulation)[3]),  function(i)
max(abs(simulation["BB",  , i])))
```

Generating tables as results

```
# print the result as a table
kable(t(quantile(T_stat, alphas)), caption = "T quantiles", digits = 3)
```

T quantiles

80%	90%	95%
1.081	1.255	1.366

```
# apply K-S statistics for empirical process under the null hypothesis T =
supt∈[0;1] |Un(t)|
KS_stat = sapply(seq(dim(simulation)[3]),  function(i)
max(abs(simulation["U",  , i])))

# print the result as a table
kable(t(quantile(KS_stat, alphas)), caption = "KS quantiles", digits = 3)
```

KS quantiles

80%	90%	95%
1.061	1.202	1.288

Comments:

It can be seen that, as expected, the quantiles for both K-S stats and T-stats are almost the same with having very similar values. Also, the value increases when the quantile increases.