

# simTool

MarselScheer

December 2, 2011

## Contents

<b>1</b>	<b>Note</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>1</b>
2.1	Motivation . . . . .	2
2.1.1	Example 1 . . . . .	2
2.1.2	Example 2 . . . . .	2
2.2	Data generating function . . . . .	3
2.2.1	Illustration of generated Data . . . . .	3
2.3	A very simple example . . . . .	4
2.4	A simple example . . . . .	6
2.5	Plotting a bit . . . . .	10
2.6	Memory considerations . . . . .	14
<b>3</b>	<b>Reproducing some results from BKY (2006)</b>	<b>16</b>

## 1 Note

This SweaveFile is part of  $\mu$ Toss package. But because some parts of this document are computational intensiv the extension of the Sweavefile is ".actuallyRnw".

## 2 Introduction

This document will give an introduction to the use of simTool. We will start with a very simple application then raise the degree of complexity in a few steps and in the end reproduce some of the results from Benjamini, Krieger, Yekutieli (2006). Basically for a simulation we need the following things

1. a function that generates the data (for example  $p$ -values )
2. some procedures that evaluates the generated data (for example the bonferroni correction)
3. statistics we want to calculate. For example the the false discovery proportion (FDP).

If we are speak for example of 1000 replications, we mean that these 3 steps were repeated 1000 times. That means, after 1000 replications there have been 1000 data sets generated. Every procedure was applied to these 1000 data sets. If we have specified for example 3 procedures then every of these procedures will give us an output, this means 3000 "output objects". And every specified statistic is applied to these 3000 "output objects".

## 2.1 Motivation

### 2.1.1 Example 1

Suppose we want to compare some characteristics of the two procedures `BH` and `holm`. Denote by  $V_n(BH)$  and  $V_n(holm)$  the number of true hypotheses rejected. For example we are interested in the distribution of  $V_n(BH)$  and  $V_n(holm)$  and also in  $E[V_n(BH)]$  and  $E[V_n(holm)]$ . Lets have a look on the arguments of theses procedures:

```
> args(BH)

function (pValues, alpha, silent = FALSE)
NULL

> args(holm)

function (pValues, alpha, silent = FALSE)
NULL
```

Both have the argument `alpha`, which stands for the niveau of the corresponding error measure. `BH` controls the false discovery rate (FDR) and `holm` controls the family-wise error rate (FWER). One question may be how must `alpha` be set in `holm` such that for fixed `alpha = 0.1` in `BH` we have  $E[V_n(BH)] \approx E[V_n(holm)]$ ? And perhaps we are also interested in how a dependence structure affects the distribution of  $V_n(BH)$  and  $V_n(holm)$ .

### 2.1.2 Example 2

Suppose you have developed a new procedure controlling some error measure and you want to compare this new procedure with some other already established procedures. Then a simulation study will consist of creating data and gathering statistics for different sample sizes, dependence structures and parameter constellations.

Basically it is always the same story. Data is generated by a specific function, perhaps this depends on some parameters, and we want to apply different procedures, again depending on some paramters, to this generated data. Nearly everyone will say, "don't bother me with the details of programming, just do the simulation with the above information and give me a nice `data.frame` which I can analyze." And this is exactly the purpose of the `simTool`.

## 2.2 Data generating function

For now we only want to use the procedures `BH` and `holm`. Again, lets have a look on the arguments of these procedures:

```
> args(BH)

function (pValues, alpha, silent = FALSE)
NULL
```

```
> args(holm)

function (pValues, alpha, silent = FALSE)
NULL
```

Not much has to be specified. The parameter `alpha` is the level at which the corresponding error rate should be controlled and `pValues` are the  $p$ -values of the hypotheses that should be tested. In general, if we say that we reject a  $p$ -value this means that we reject the corresponding hypotheses. The following function generates data and will be used throughout the whole document. The data generating function **must have** an entry `$procInput`. All in `$procInput` will be used as input for the specified procedures. In our situation we will generate a list with 2 entries. `$procInput` will only consist of the generated  $p$ -values . And `$groundTruth` indicates which  $p$ -value corresponds to a true or false hypothesis.

```
> pValFromEquiCorrData <- function(sampleSize, rho, mu, pi0) {
+   nmbOfFalseHyp <- round(sampleSize * (1 - pi0))
+   nmbOfTrueHyp <- sampleSize - nmbOfFalseHyp
+   muVec <- c(rep(mu, nmbOfFalseHyp), rep(0, nmbOfTrueHyp))
+   Y <- sqrt(rho) * rnorm(1) + sqrt(1 - rho) * rnorm(sampleSize) +
+     muVec
+   return(list(procInput = list(pValues = 1 - pnorm(Y)), groundTruth = muVec ==
+     0))
+ }
```

### 2.2.1 Illustration of generated Data

We will now generate 1000  $p$ -Values independently ( $\rho = 0$ ). `sampleSize * (1-pi0) = 700` of them correspond to false hypotheses and 300 to true hypotheses.

```
> set.seed(123)
> data <- pValFromEquiCorrData(sampleSize = 1000, rho = 0, mu = 2,
+   pi0 = 0.3)
```

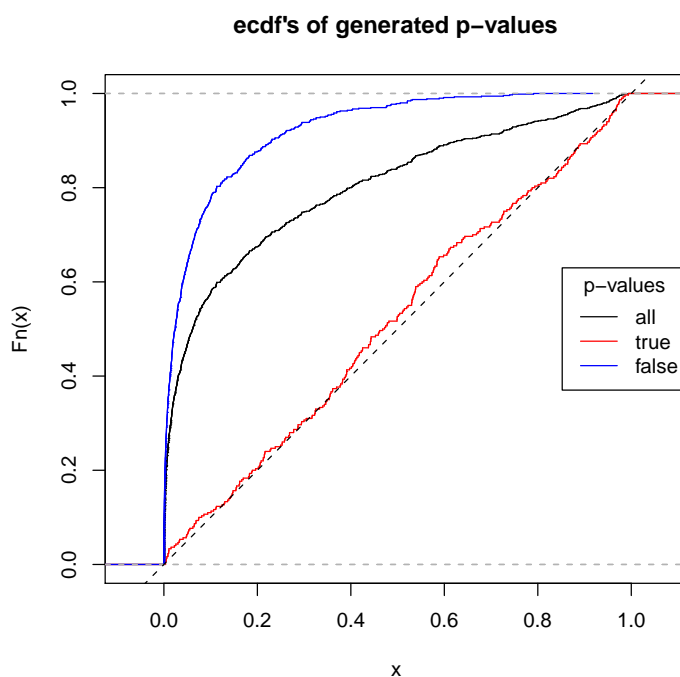
Lets visualize the different  $p$ -values .

```
> local({
+   pValues = data$procInput$pValues
+   groundTruth = data$groundTruth
+   plot(ecdf(pValues), do.points = FALSE, verticals = TRUE,
+     main = "ecdf's of generated p-values")
+ })
```

```

+   lines(ecdf(pValues[groundTruth]), do.points = FALSE, verticals = TRUE,
+         col = 2)
+   lines(ecdf(pValues[!groundTruth]), do.points = FALSE, verticals = TRUE,
+         col = 4)
+   abline(0, 1, lty = 2)
+   legend("right", legend = c("all", "true", "false"), col = c(1,
+     2, 4), lty = 1, title = "p-values")
+ })

```



### 2.3 A very simple example

Lets directly start with the function call and then analysing what happend. Just for now we implement a simple version of the data generating function in section 2.2 and a simple version of the BH.

```

> myGen <- function() {
+   pValFromEquiCorrData(sampleSize = 200, rho = 0, mu = 2, pi0 = 0.5)
+ }
> BH.05 = function(pValues) {
+   BH(pValues = pValues, alpha = 0.05, silent = TRUE)
+ }

> set.seed(123)
> sim <- simulation(replications = 10, list(funName = "myGen",
+   fun = myGen), list(list(funName = "BH.simple", fun = BH.05)))

```

The following happend:

1. Call `myGen`
2. Append generated data set to `sim$data`
3. Call `BH.05` with the generated  $p$ -values
4. Add to the results of `BH.05` the parameter constellation used by `myGen` and `BH.05` and also the position of the used data in `sim$data`
5. Append this extended results of `BH.05` to `sim$results`
6. repeat this 9 more times

Since `replications = 10` in `simulation` the object `sim` consists of:

```
> names(sim)
[1] "data"    "results"

> length(sim$data)
[1] 10

> length(sim$results)
[1] 10
```

The structure of one object in `sim$data` coincides with the structure of the return of `myGen`:

```
> names(myGen())
[1] "procInput" "groundTruth"

> names(sim$data[[1]])
[1] "procInput" "groundTruth"
```

Lets have a look at the additional information added by `simulation` to the object returned by `BH.05`. First the entries of the pure return value of `BH.05` and then of `sim$results`

```
> names(BH.05(runif(100)))
[1] "adjPValues"    "criticalValues" "rejected"      "errorControl"

> names(sim$results[[1]])
[1] "data.set.number" "parameters"     "adjPValues"    "criticalValues"
[5] "rejected"        "errorControl"
```

`data.set.number` is the number of the used data set in `sim$data`. Since every generated data set is used only once we have

```
> sapply(sim$results, function(x) x$data.set.number)
[1] 1 2 3 4 5 6 7 8 9 10
```

Thus it is possible to reproduce any result directly. Let us reproduce the results of the 6th replication

```
> idx <- sim$results[[6]]$data.set.number
> pValues <- sim$data[[idx]]$procInput$pValues
> all(BH.05(pValues)$adjPValues == sim$results[[6]]$adjPValues)

[1] TRUE
```

Since our data generating function and procedure did not really have any parameter, there is not much information in:

```
> sim$results[[1]]$parameters

$funName
[1] "myGen"

$method
[1] "BH.simple"
```

Note, BH.05 technically has the parameter `pValues` but since this parameter is the "input channel" for the output data of the data generating function it is not really regarded as parameter that affect the behaviour of the procedure like for example the level  $\alpha$  for the error measure.

## 2.4 A simple example

Indeed, for the simple example above the `simTool` is not very helpful. Let us increase the complexity of our simulation. As a first step, we want to increase the parameter `rho` of the data generating function gradually and investigate  $EV_n(BH)$  and  $EV_n^2(BH)$ .

```
> set.seed(123)
> sim <- simulation(replications = 10, list(funName = "pVGen",
+   fun = pValFromEquiCorrData, sampleSize = 100, rho = seq(0,
+     1, by = 0.2), mu = 2, pi0 = 0.5), list(list(funName = "BH",
+   fun = BH, alpha = c(0.5, 0.25), silent = TRUE)))
```

The following happend:

1. Call `myGen` with `rho = 0`
2. Append generated data set to `sim$data`
3. Call `BH` with `alpha = 0.5` and the generated  $p$ -values
4. Add to the results the parameter constellation used by `myGen` and `BH` and also the position of the used data in `sim$data`
5. Append this extended results to `sim$results`
6. Call `BH` with `alpha = 0.25` and the **same**  $p$ -values as in 3.
7. Add to the results the parameter constellation used by `myGen` and `BH` and also the position of the used data in `sim$data`

8. Append this extended results of to `sim$results`
9. repeat this 9 more times
10. Call `myGen` with `rho = 0.2`
11. and so on

So we have now a bunch of data sets and results.

```
> length(sim$data)
```

```
[1] 60
```

```
> length(sim$results)
```

```
[1] 120
```

60 data sets have been generated because we have `rho = 0, 0.2, 0.4, 0.6, 0.8, 1` and for each `rho` we generated 10 data sets. We have applied BH with `alpha = 0.5` to all 60 data sets and BH with `alpha = 0.25`, yielding 120 objects in `results`. We see from the `data.set.number` that after generating a data set it is used two times in series

```
> sapply(sim$results, function(x) x$data.set.number)
```

```
[1]  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 11 12 12 13
[26] 13 14 14 15 15 16 16 17 17 18 18 19 19 20 20 21 21 22 22 23 23 24 24 25 25
[51] 26 26 27 27 28 28 29 29 30 30 31 31 32 32 33 33 34 34 35 35 36 36 37 37 38
[76] 38 39 39 40 40 41 41 42 42 43 43 44 44 45 45 46 46 47 47 48 48 49 49 50 50
[101] 51 51 52 52 53 53 54 54 55 55 56 56 57 57 58 58 59 59 60 60
```

In order to gather how many true hypotheses have been rejected corresponding to the different parameter constellations we need a function that is able to calculate it.

```
> NumberOfType1Error <- function(data, result) sum(data$groundTruth *
+   result$rejected)
> V2 <- function(data, result) NumberOfType1Error(data, result)^2
```

Lets us calculate for the first result object the number of rejected true hypotheses explicitly.

```
> idx = sim$results[[1]]$data.set.number
> data = sim$data[[idx]]
> NumberOfType1Error(data, sim$results[[1]])
```

```
[1] 15
```

This means for the following parameter constellation

```
> sim$results[[1]]$parameters
```

```
$funName
[1] "pVGen"
```

```
$sampleSize
[1] 100
```

```
$rho
[1] 0
```

```
$mu
[1] 2
```

```
$pi0
[1] 0.5
```

```
$method
[1] "BH"
```

```
$alpha
[1] 0.5
```

```
$silent
[1] TRUE
```

we one time observed

```
[1] 15
```

rejected true null hypotheses. In order to estimate  $EV_n(BH)$  and  $EV_n^2(BH)$  we have to search in `sim$results` for the other 9 results using the same parameter constellation. In order to facilitate this task we provide a function.

```
> result <- gatherStatistics(sim, list(V = NumberOfType1Error,
+   V2 = V2), mean)
> result
```

```
$statisticDF
```

	funName	sampleSize	rho	mu	pi0	method	alpha	silent	V.mean	V2.mean
1	pVGen	100	0	2	0.5	BH	0.5	TRUE	15.1	245.3
2	pVGen	100	0	2	0.5	BH	0.25	TRUE	5.9	40.9
3	pVGen	100	0.2	2	0.5	BH	0.5	TRUE	21.5	591.9
4	pVGen	100	0.2	2	0.5	BH	0.25	TRUE	9.2	163.2
5	pVGen	100	0.4	2	0.5	BH	0.5	TRUE	14.6	401.6
6	pVGen	100	0.4	2	0.5	BH	0.25	TRUE	6.2	126.6
7	pVGen	100	0.6	2	0.5	BH	0.5	TRUE	22.3	860.7
8	pVGen	100	0.6	2	0.5	BH	0.25	TRUE	9.3	265.7
9	pVGen	100	0.8	2	0.5	BH	0.5	TRUE	18.1	711.1
10	pVGen	100	0.8	2	0.5	BH	0.25	TRUE	8.0	328.6
11	pVGen	100	1	2	0.5	BH	0.5	TRUE	20.0	1000.0
12	pVGen	100	1	2	0.5	BH	0.25	TRUE	15.0	750.0



```

$name.parameters
[1] "funName"      "sampleSize" "rho"         "mu"          "pi0"
[6] "method"      "alpha"      "silent"

```

```

$name.statistics
[1] "V.mean" "V2.mean"

```

```

$name.avgFun
[1] "mean"

```

As you can see every parameter constellation has its own row in `$statisticDF`. By the way, again, we see that  $p$ -values is a parameter of BH but since it is contained in `$procInput` it is not considered as "real parameter". If we are interested in more than one statistics, then we simply provide a list with "average functions"

```

> result <- gatherStatistics(sim, list(V = NumberOfType1Error,
+   V2 = V2), list(mean = mean, sd = function(vec) round(sd(vec),
+   1)))
> result$statisticDF

```

	funName	sampleSize	rho	mu	pi0	method	alpha	silent	V.mean	V2.mean	V.sd	V2.sd
1	pVGen	100	0	2	0.5	BH	0.5	TRUE	15.1	245.3	4.4	131.7
2	pVGen	100	0	2	0.5	BH	0.25	TRUE	5.9	40.9	2.6	34.9
3	pVGen	100	0.2	2	0.5	BH	0.5	TRUE	21.5	591.9	12.0	597.3
4	pVGen	100	0.2	2	0.5	BH	0.25	TRUE	9.2	163.2	9.3	329.0
5	pVGen	100	0.4	2	0.5	BH	0.5	TRUE	14.6	401.6	14.5	684.8
6	pVGen	100	0.4	2	0.5	BH	0.25	TRUE	6.2	126.6	9.9	318.5
7	pVGen	100	0.6	2	0.5	BH	0.5	TRUE	22.3	860.7	20.1	981.2
8	pVGen	100	0.6	2	0.5	BH	0.25	TRUE	9.3	265.7	14.1	508.8
9	pVGen	100	0.8	2	0.5	BH	0.5	TRUE	18.1	711.1	20.6	1051.5
10	pVGen	100	0.8	2	0.5	BH	0.25	TRUE	8.0	328.6	17.1	801.7
11	pVGen	100	1	2	0.5	BH	0.5	TRUE	20.0	1000.0	25.8	1291.0
12	pVGen	100	1	2	0.5	BH	0.25	TRUE	15.0	750.0	24.2	1207.6

Another possibility is to call `gatherStatistics` without and "average function". Then every result get his own row in `$statisticDF`

```

> result <- gatherStatistics(sim, list(V = NumberOfType1Error,
+   V2 = V2))
> head(result$statisticDF)

```

	funName	sampleSize	rho	mu	pi0	method	alpha	silent	V	V2
1	pVGen	100	0	2	0.5	BH	0.5	TRUE	15	225
2	pVGen	100	0	2	0.5	BH	0.5	TRUE	10	100
3	pVGen	100	0	2	0.5	BH	0.5	TRUE	20	400
4	pVGen	100	0	2	0.5	BH	0.5	TRUE	11	121
5	pVGen	100	0	2	0.5	BH	0.5	TRUE	19	361
6	pVGen	100	0	2	0.5	BH	0.5	TRUE	16	256

```

> tail(result$statisticDF)

```

```

      funName sampleSize rho mu pi0 method alpha silent V V2
115  pVGen      100    1  2 0.5    BH  0.25  TRUE  0  0
116  pVGen      100    1  2 0.5    BH  0.25  TRUE  50 2500
117  pVGen      100    1  2 0.5    BH  0.25  TRUE  50 2500
118  pVGen      100    1  2 0.5    BH  0.25  TRUE  50 2500
119  pVGen      100    1  2 0.5    BH  0.25  TRUE  0  0
120  pVGen      100    1  2 0.5    BH  0.25  TRUE  0  0

```

## 2.5 Plotting a bit

```

> if (!is.element("sim.plot", ls())) {
+   sim.plot <- simulation(replications = 1000, list(funName = "pVGen",
+     fun = pValFromEquiCorrData, sampleSize = 100, rho = seq(0,
+     1, by = 0.2), mu = 0, pi0 = 1), list(list(funName = "BH",
+     fun = function(pValues, alpha) BH(pValues, alpha, silent = TRUE),
+     alpha = c(0.5, 0.75))))
+ }
> length(sim.plot$data)

[1] 6000

> length(sim.plot$results)

[1] 12000

```

There are already different kind of R functions that take data.frames and generate histograms, boxplots and so on from them. For some plot example we will need the lattice package.

```
> require(lattice)
```

First we calculate V for every MutossSim object and then plot a histogram and a boxplot of V, see Figure 1 (p. 11), Figure 2 (p. 12).

```
> result.all <- gatherStatistics(sim.plot, list(V = NumberOfType1Error))
```

Also after the "average process" we again get an data.frame which can be used to generate plots.

```

> result <- gatherStatistics(sim.plot, list(V = NumberOfType1Error),
+   list(Ml = function(x) mean(x) - 2 * sd(x)/sqrt(length(x)),
+     Mu = function(x) mean(x) + 2 * sd(x)/sqrt(length(x)),
+     M = mean, SD = sd))
> subset(result$statisticDF, alpha == "0.5")[1:3, -1]

```

	sampleSize	rho	mu	pi0	method	alpha	V.Ml	V.Mu	V.M	V.SD
1	100	0	0	1	BH	0.5	1.610993	1.973007	1.792	2.861980
3	100	0.2	0	1	BH	0.5	8.833764	11.544236	10.189	21.428169
5	100	0.4	0	1	BH	0.5	12.131991	15.860009	13.996	29.472574

```

> print(histogram(~V | rho, data = subset(result.all$statisticDF,
+     alpha == "0.5")))

```

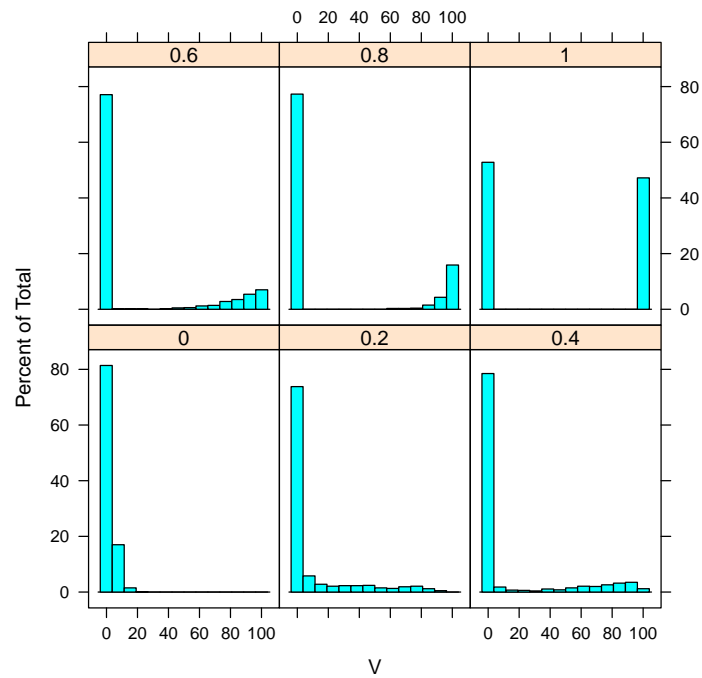


Figure 1: Histogram of the number of rejected true hypotheses for alpha equals 0.5

```
> print(bwplot(V ~ rho | alpha, data = subset(result.all$statisticDF)))
```

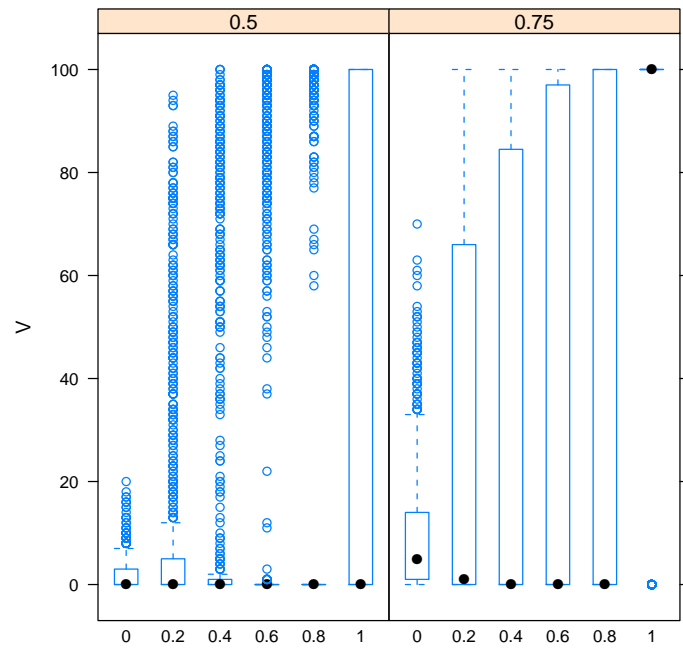


Figure 2: Boxplot of the number of rejected true hypotheses

```

> print(xyplot(V.Ml + V.M + V.Mu ~ rho | alpha, data = result$statisticDF,
+   type = "a", auto.key = list(space = "right", points = FALSE,
+   lines = TRUE)))

```

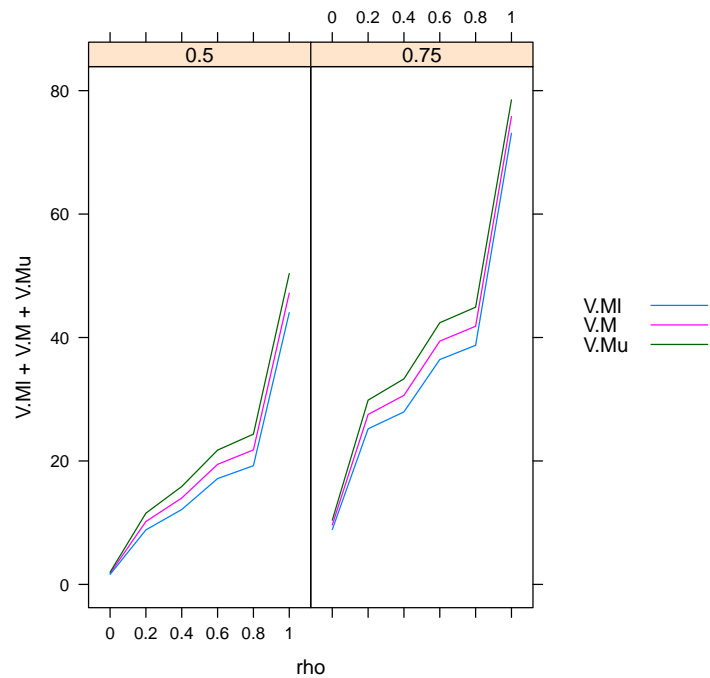


Figure 3: XYPlot of the mean number of rejected true hypotheses with asymptotic 95% confidence intervals (pointwise)

## 2.6 Memory considerations

Up to now we have kept any information in memory. Sometimes, it is useful not to discard any generated information. For example, if some objects in results appear odd, they can directly be reproduced and of course with all information available we can investigate anything like distribution or any statistic we did not consider before the simulation. The price for this liberty is in general an extensive memory usage which of course restricts the size of our simulation. On the other hand if we exactly know that we are only interested in  $EV_n$  the mean number of true rejected hypotheses, why should we keep the generated  $p$ -values or the index of the rejected  $p$ -values? Thus it would be enough that an object in `$results` contains  $V_n$  the number of rejected true hypotheses. We will now show an example where only the information needed is kept in memory. The following is basically the "simple example" from section 2.4. The main issue is to save memory thus we will calculate the  $V_n$  right after applying the procedure to the generated data set. If done so, we can also discard the generated data!

```
> V.BH <- function(pValues, groundTruth, alpha) {
+   out <- BH(pValues, alpha, silent = TRUE)
+   V <- sum(out$rejected * groundTruth)
+   return(list(V = V))
+ }
```

As already mentioned anything in `$procInput` will be used as an input for the procedures. Since our procedure now has a parameter `groundTruth` our data generating function has to be adapted. We just move `groundTruth` in `$procInput`.

```
> pValFromEquiCorrData2 <- function(sampleSize, rho, mu, pi0) {
+   nmbOfFalseHyp <- round(sampleSize * (1 - pi0))
+   nmbOfTrueHyp <- sampleSize - nmbOfFalseHyp
+   muVec <- c(rep(mu, nmbOfFalseHyp), rep(0, nmbOfTrueHyp))
+   Y <- sqrt(rho) * rnorm(1) + sqrt(1 - rho) * rnorm(sampleSize) +
+     muVec
+   return(list(procInput = list(pValues = 1 - pnorm(Y), groundTruth = muVec ==
+     0)))
+ }
```

Obviously, it is unnecessary to store this generated data, because the number of Type 1 Errors we are interested in will be directly calculated. Setting `discardProcInput = TRUE` in `simulation` removes `$procInput` from the object generated by the data generating function after `$procInput` has been used by all specified procedures. In our case these will be the 2 procedures `V.BH` with `alpha = 0.5` and `alpha = 0.25`.

```
> set.seed(123)
> sim <- simulation(replications = 10, list(funName = "pVGen2",
+   fun = pValFromEquiCorrData2, sampleSize = 100, rho = seq(0,
+     1, by = 0.2), mu = 2, pi0 = 0.5), list(list(funName = "V.BH",
+   fun = V.BH, alpha = c(0.5, 0.25))), discardProcInput = TRUE)
> result <- gatherStatistics(sim, list(V = function(data, results) results$V),
+   mean)
> result
```

```

$statisticDF
  funName sampleSize rho mu pi0 method alpha V.mean
1  pVGen2          100  0  2 0.5  V.BH  0.5  15.1
2  pVGen2          100  0  2 0.5  V.BH  0.25  5.9
3  pVGen2          100 0.2  2 0.5  V.BH  0.5  21.5
4  pVGen2          100 0.2  2 0.5  V.BH  0.25  9.2
5  pVGen2          100 0.4  2 0.5  V.BH  0.5  14.6
6  pVGen2          100 0.4  2 0.5  V.BH  0.25  6.2
7  pVGen2          100 0.6  2 0.5  V.BH  0.5  22.3
8  pVGen2          100 0.6  2 0.5  V.BH  0.25  9.3
9  pVGen2          100 0.8  2 0.5  V.BH  0.5  18.1
10 pVGen2          100 0.8  2 0.5  V.BH  0.25  8.0
11 pVGen2          100  1  2 0.5  V.BH  0.5  20.0
12 pVGen2          100  1  2 0.5  V.BH  0.25  15.0

```

```

$name.parameters
[1] "funName" "sampleSize" "rho" "mu" "pi0"
[6] "method" "alpha"

```

```

$name.statistics
[1] "V.mean"

```

```

$name.avgFun
[1] "mean"

```

Here the corresponding table from section 2.4.

```

> set.seed(123)
> sim.simple <- simulation(replications = 10, list(funName = "pVGen",
+   fun = pValFromEquiCorrData, sampleSize = 100, rho = seq(0,
+   1, by = 0.2), mu = 2, pi0 = 0.5), list(list(funName = "BH",
+   fun = BH, alpha = c(0.5, 0.25), silent = TRUE)))
> result.simple <- gatherStatistics(sim.simple, list(V = NumberOfType1Error),
+   mean)
> print(result.simple)

```

```

$statisticDF
  funName sampleSize rho mu pi0 method alpha silent V.mean
1  pVGen          100  0  2 0.5  BH  0.5  TRUE  15.1
2  pVGen          100  0  2 0.5  BH  0.25  TRUE  5.9
3  pVGen          100 0.2  2 0.5  BH  0.5  TRUE  21.5
4  pVGen          100 0.2  2 0.5  BH  0.25  TRUE  9.2
5  pVGen          100 0.4  2 0.5  BH  0.5  TRUE  14.6
6  pVGen          100 0.4  2 0.5  BH  0.25  TRUE  6.2
7  pVGen          100 0.6  2 0.5  BH  0.5  TRUE  22.3
8  pVGen          100 0.6  2 0.5  BH  0.25  TRUE  9.3
9  pVGen          100 0.8  2 0.5  BH  0.5  TRUE  18.1
10 pVGen          100 0.8  2 0.5  BH  0.25  TRUE  8.0
11 pVGen          100  1  2 0.5  BH  0.5  TRUE  20.0
12 pVGen          100  1  2 0.5  BH  0.25  TRUE  15.0

```

```

$name.parameters
[1] "funName" "sampleSize" "rho" "mu" "pi0"
[6] "method" "alpha" "silent"

```

```

$name.statistics
[1] "V.mean"

```

```

$name.avgFun
[1] "mean"

```

The same results but the memory usage differ much:

```

> print(s1 <- object.size(sim))

127192 bytes

> print(s2 <- object.size(sim.simple))

565912 bytes

> unclass(s2/s1)

[1] 4.449274

```

### 3 Reproducing some results from BKY (2006)

Now, we will reproduce figure 1 from the publication. For this we need to estimate the FDR. This will be done by calculating the FDP for every object in `$results` and then calculating the empirical mean. In this simulation where we exactly know what we want and thus retain only the necessary information. To be able to calculate the FDP right after applying the procedure we need to know which pValue belongs to a true or false hypothesis. Thus, like in the foregoing section, `$groundTruth` will be an element of `$procInput`.

```

> pValFromEquiCorrData2 <- function(sampleSize, rho, mu, pi0) {
+   nmbOfFalseHyp <- round(sampleSize * (1 - pi0))
+   nmbOfTrueHyp <- sampleSize - nmbOfFalseHyp
+   muVec <- c(rep(mu, nmbOfFalseHyp), rep(0, nmbOfTrueHyp))
+   Y <- sqrt(rho) * rnorm(1) + sqrt(1 - rho) * rnorm(sampleSize) +
+     muVec
+   return(list(procInput = list(pValues = 1 - pnorm(Y), groundTruth = muVec ==
+     0)))
+ }

```

We also need a function that calculates the FDP.

```

> FDP <- function(pValues, groundTruth, proc = c("BH", "M-S-HLF")) {
+   if (proc == "BH")
+     out <- BH(pValues, alpha = 0.05, silent = TRUE)
+   else out <- adaptiveSTS(pValues, alpha = 0.05, silent = TRUE)
+   R <- sum(out$rejected)
+   if (R == 0)

```



```

+       return(list(FDP = 0))
+   V <- sum(out$rejected * groundTruth)
+   return(list(FDP = V/R))
+ }

```

Now, the (partial) reproduction of the results can start!

```

> set.seed(123)
> date()

```

```
[1] "Thu Dec 01 18:05:29 2011"
```

```

> sim <- simulation(replications = 10000, list(funName = "pVGen2",
+   fun = pValFromEquiCorrData2, sampleSize = c(16, 32, 64, 128,
+   256, 512), rho = c(0, 0.1, 0.5), mu = 5, pi0 = c(0.25,
+   0.75)), list(list(funName = "FDP", fun = FDP, proc = c("BH",
+   "M-S-HLF"))), discardProcInput = TRUE)
> date()

```

```
[1] "Thu Dec 01 18:39:30 2011"
```

```

> result <- gatherStatistics(sim, list(FDP = function(data, results) results$FDP),
+   mean)> result

```

```
$statisticDF
```

	funName	sampleSize	rho	mu	pi0	method	proc	FDP.mean
1	pVGen2	16	0	5	0.25	FDP	BH	0.01243507
2	pVGen2	16	0	5	0.25	FDP	M-S-HLF	0.04680482
3	pVGen2	32	0	5	0.25	FDP	BH	0.01254447
4	pVGen2	32	0	5	0.25	FDP	M-S-HLF	0.04933171
5	pVGen2	64	0	5	0.25	FDP	BH	0.01245073
6	pVGen2	64	0	5	0.25	FDP	M-S-HLF	0.05027779
7	pVGen2	128	0	5	0.25	FDP	BH	0.01250933
8	pVGen2	128	0	5	0.25	FDP	M-S-HLF	0.05000333
9	pVGen2	256	0	5	0.25	FDP	BH	0.01259937
10	pVGen2	256	0	5	0.25	FDP	M-S-HLF	0.05021692
11	pVGen2	512	0	5	0.25	FDP	BH	0.01246457
12	pVGen2	512	0	5	0.25	FDP	M-S-HLF	0.04993815
13	pVGen2	16	0.1	5	0.25	FDP	BH	0.01243852
14	pVGen2	16	0.1	5	0.25	FDP	M-S-HLF	0.04929676
15	pVGen2	32	0.1	5	0.25	FDP	BH	0.01290907
16	pVGen2	32	0.1	5	0.25	FDP	M-S-HLF	0.05355816
17	pVGen2	64	0.1	5	0.25	FDP	BH	0.01271804
18	pVGen2	64	0.1	5	0.25	FDP	M-S-HLF	0.05626510
19	pVGen2	128	0.1	5	0.25	FDP	BH	0.01233749
20	pVGen2	128	0.1	5	0.25	FDP	M-S-HLF	0.05637963
21	pVGen2	256	0.1	5	0.25	FDP	BH	0.01231818
22	pVGen2	256	0.1	5	0.25	FDP	M-S-HLF	0.05619898
23	pVGen2	512	0.1	5	0.25	FDP	BH	0.01248149
24	pVGen2	512	0.1	5	0.25	FDP	M-S-HLF	0.05719747
25	pVGen2	16	0.5	5	0.25	FDP	BH	0.01269308
26	pVGen2	16	0.5	5	0.25	FDP	M-S-HLF	0.06255181

27	pVGen2	32	0.5	5	0.25	FDP	BH	0.01227550
28	pVGen2	32	0.5	5	0.25	FDP	M-S-HLF	0.06710637
29	pVGen2	64	0.5	5	0.25	FDP	BH	0.01183768
30	pVGen2	64	0.5	5	0.25	FDP	M-S-HLF	0.07187446
31	pVGen2	128	0.5	5	0.25	FDP	BH	0.01200182
32	pVGen2	128	0.5	5	0.25	FDP	M-S-HLF	0.07523846
33	pVGen2	256	0.5	5	0.25	FDP	BH	0.01245500
34	pVGen2	256	0.5	5	0.25	FDP	M-S-HLF	0.07724869
35	pVGen2	512	0.5	5	0.25	FDP	BH	0.01193485
36	pVGen2	512	0.5	5	0.25	FDP	M-S-HLF	0.07817799
37	pVGen2	16	0	5	0.75	FDP	BH	0.03687048
38	pVGen2	16	0	5	0.75	FDP	M-S-HLF	0.05066989
39	pVGen2	32	0	5	0.75	FDP	BH	0.03771246
40	pVGen2	32	0	5	0.75	FDP	M-S-HLF	0.04999444
41	pVGen2	64	0	5	0.75	FDP	BH	0.03736944
42	pVGen2	64	0	5	0.75	FDP	M-S-HLF	0.04979089
43	pVGen2	128	0	5	0.75	FDP	BH	0.03716345
44	pVGen2	128	0	5	0.75	FDP	M-S-HLF	0.04963708
45	pVGen2	256	0	5	0.75	FDP	BH	0.03812883
46	pVGen2	256	0	5	0.75	FDP	M-S-HLF	0.05079616
47	pVGen2	512	0	5	0.75	FDP	BH	0.03731167
48	pVGen2	512	0	5	0.75	FDP	M-S-HLF	0.05002008
49	pVGen2	16	0.1	5	0.75	FDP	BH	0.03624063
50	pVGen2	16	0.1	5	0.75	FDP	M-S-HLF	0.06107526
51	pVGen2	32	0.1	5	0.75	FDP	BH	0.03702100
52	pVGen2	32	0.1	5	0.75	FDP	M-S-HLF	0.06206310
53	pVGen2	64	0.1	5	0.75	FDP	BH	0.03771594
54	pVGen2	64	0.1	5	0.75	FDP	M-S-HLF	0.06404561
55	pVGen2	128	0.1	5	0.75	FDP	BH	0.03677325
56	pVGen2	128	0.1	5	0.75	FDP	M-S-HLF	0.06185353
57	pVGen2	256	0.1	5	0.75	FDP	BH	0.03694803
58	pVGen2	256	0.1	5	0.75	FDP	M-S-HLF	0.06317781
59	pVGen2	512	0.1	5	0.75	FDP	BH	0.03694764
60	pVGen2	512	0.1	5	0.75	FDP	M-S-HLF	0.06267116
61	pVGen2	16	0.5	5	0.75	FDP	BH	0.03533002
62	pVGen2	16	0.5	5	0.75	FDP	M-S-HLF	0.11075864
63	pVGen2	32	0.5	5	0.75	FDP	BH	0.03501351
64	pVGen2	32	0.5	5	0.75	FDP	M-S-HLF	0.11841870
65	pVGen2	64	0.5	5	0.75	FDP	BH	0.03331830
66	pVGen2	64	0.5	5	0.75	FDP	M-S-HLF	0.12028660
67	pVGen2	128	0.5	5	0.75	FDP	BH	0.03413304
68	pVGen2	128	0.5	5	0.75	FDP	M-S-HLF	0.12464657
69	pVGen2	256	0.5	5	0.75	FDP	BH	0.03561711
70	pVGen2	256	0.5	5	0.75	FDP	M-S-HLF	0.12532959
71	pVGen2	512	0.5	5	0.75	FDP	BH	0.03463151
72	pVGen2	512	0.5	5	0.75	FDP	M-S-HLF	0.12510925

\$name.parameters

```
[1] "funName"      "sampleSize" "rho"          "mu"           "pi0"
[6] "method"      "proc"
```

```
$name.statistics
[1] "FDP.mean"
```

```
$name.avgFun
[1] "mean"
```

```
> print(xyplot(FDP.mean ~ sampleSize | pi0 * rho, data = result$statisticDF,
+   group = proc, type = "a", auto.key = list(space = "right",
+     points = FALSE, lines = TRUE)))
```

