1 Introduction

The purpose of the simTool package is to disengage the research from any kind of administrative source code which is usually an annoying necessity of a simulation study.

This vignette will give an introduction into the simTool package mainly by examples of growing complexity. The workhorse is the function evalGrids. Every parameter of this function will be discussed briefly and the functionality is illustrated by at least one example.

2 Workflow

The workflow is quite easy and natural. One defines two data.frames, the first one represents the functions that generate the data sets and the second one represents the functions that analyze the data. These two data.frames are passed to evalGrids which conducts the simulation. Afterwards, the results can nicely be displayed as a data.frame be coercing the object returned by evalGrids to a data.frame.

3 Defining the data.frames for data generation and analyzation

There are 3 rules:

- the first column (a character vector) defines the functions to be called
- the other columns are the parameters that are passed to function specified in the first column
- The entry NA will not be passed to the function specified in the first column.

The function expandGrid is a convenient function for defining such data.frames.

We now define the data generation functions for our first simulation.

```
library(simTool)
library(plyr)
library(reshape)

##
## Attaching package:  'reshape'
##
## The following objects are masked from 'package:plyr':
##
##    rename, round_any
```

```r
print(dg <- rbind.fill(
  expandGrid(fun="rexp", n=c(10, 20), rate=1:2),
  expandGrid(fun="rnorm", n=c(10, 20), mean=1:2)))

##     fun  n rate mean
## 1  rexp 10    1   NA
## 2  rexp 20    1   NA
## 3  rexp 10    2   NA
## 4  rexp 20    2   NA
## 5 rnorm 10   NA    1
## 6 rnorm 20   NA    1
## 7 rnorm 10   NA    2
## 8 rnorm 20   NA    2
```

This `data.frame` represents 8 `R`-functions. For instance, the second row represents a function that generates 20 exponential distributed random variables with `rate` 1. Since `mean=NA` in the second row, this parameter is not passed to `rexp`.

Similar, we define the `data.frame` for data analyzing functions.

```r
print(pg<-rbind.fill(
  expandGrid(proc="min"),
  expandGrid(proc="mean", trim=c(0.1, 0.2))))

##   proc trim
## 1  min   NA
## 2 mean  0.1
## 3 mean  0.2
```

Hence, this `data.frame` represents 3 `R`-functions i.e. calculating the minimum and the arithmetic mean with `trim=0.1` and `trim=0.2`.

4 The workhorse `evalGrids`

The workhorse `evalGrids` has the following simplified pseudo code:

```
convert dg to R-functions  $\{g_1, ..., g_k\}$
convert pg to R-functions  $\{f_1, ..., f_\ell\}$
initialize result object
append dg and pg to the result object
t1 = current.time()
for g in  $\{g_1, \ldots, g_k\}$
    for r in 1:replications (optionally in a parallel manner)
        data = g()
        for f in  $\{f_1, \ldots, f_\ell\}$
            append f(data) to the result object
```

```
        optionally append data to the result object
      optionally summarize the result object over all
      replications but separately for $f_1, \ldots, f_\ell$
    optionally save the results so far obtained to HDD
t2 = current.time()
Estimate the number of replications per hour from t1 and t2
```

In general, the object returned by `evalGrids` is a `list` of class `evalGrid` and can be coerced into a `data.frame`. Later on, we will investigate the case if this is not the case.

```
dg = expandGrid(fun="rnorm", n=10, mean=1:2)
pg = expandGrid(proc="min")
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 2)

## [1] "Estimated replications per hour:  44086115"

as.data.frame(eg)

##   i j   fun  n mean proc replication       V1
## 1 1 1 rnorm 10    1  min           1  0.26875
## 2 1 1 rnorm 10    1  min           2 -0.98311
## 3 2 1 rnorm 10    2  min           1  0.78409
## 4 2 1 rnorm 10    2  min           2  0.09762
```

As you can see, the function always estimates the number of replications that can be done in one hour.

The object returned by `evalGrids` will be discussed at the end of this section. But specific points about this object will be explained earlier.

4.1 Parameter `replications`

Of course, this parameter controls the number of replications conducted.

```
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 3)

## [1] "Estimated replications per hour:  62308780"

as.data.frame(eg)

##   i j   fun  n mean proc replication      V1
## 1 1 1 rnorm 10    1  min           1 -0.2308
## 2 1 1 rnorm 10    1  min           2 -0.7510
## 3 1 1 rnorm 10    1  min           3 -0.1170
## 4 2 1 rnorm 10    2  min           1  1.2573
## 5 2 1 rnorm 10    2  min           2  1.3779
## 6 2 1 rnorm 10    2  min           3  0.8037
```

4.2 Parameter discardGeneratedData

`evalGrids` saves ALL generated data sets. In general, it is sometimes very handy to have the data sets in order to investigate unusual or unexpected results. But saving the generated data sets can be very memory consuming. Stop saving the generated data sets can be obtained by setting `discardGeneratedData = TRUE`. Confer command line 11 in the pseudo code.

```
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000)

## [1] "Estimated replications per hour:  89601139"

object.size(eg)

## 1244648 bytes

eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000,
    discardGeneratedData = TRUE)

## [1] "Estimated replications per hour:  87596268"

object.size(eg)

## 908808 bytes
```

The object returned by `evalGrids` will be discussed at the end of this vignette.

4.3 Parameter progress

This parameter activates a text progress bar in the console. Usually, this does not make sense if one uses `Sweave` or `knitr`, but for demonstration purpose we do this here.

```
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 10,
    progress = TRUE)

##
  |
  |                                                                  |   0%
  |
  |================================                                  |  50%
  |
  |==================================================================| 100%
## [1] "Estimated replications per hour:  29143976"
```

The progress bar increases every time a new element is chosen in command line 6 of the pseudo code.

4.4 Parameter `post.proc`

As stated in command line 12 we can summarize the result objects over all replications but separately for all data analyzing functions.

```
dg = expandGrid(fun="runif", n=c(10,20,30))
pg = expandGrid(proc=c("min", "max"))
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000,
    post.proc=mean)

## [1] "Estimated replications per hour:  19753187"

as.data.frame(eg)

##   i j   fun  n proc value      V1
## 1 1 1 runif 10  min (all) 0.09424
## 2 1 2 runif 10  max (all) 0.91075
## 3 2 1 runif 20  min (all) 0.04433
## 4 2 2 runif 20  max (all) 0.94950
## 5 3 1 runif 30  min (all) 0.03126
## 6 3 2 runif 30  max (all) 0.96894

eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000,
    post.proc=c(mean, sd))

## [1] "Estimated replications per hour:  21515748"

as.data.frame(eg)

##   i j   fun  n proc value V1_mean   V1_sd
## 1 1 1 runif 10  min (all) 0.09664 0.08937
## 2 1 2 runif 10  max (all) 0.91214 0.08140
## 3 2 1 runif 20  min (all) 0.04722 0.04432
## 4 2 2 runif 20  max (all) 0.94972 0.04699
## 5 3 1 runif 30  min (all) 0.03055 0.03066
## 6 3 2 runif 30  max (all) 0.96566 0.03275
```

Note, by specifying the parameter **`post.proc`** the generated data sets and all individual result objects are discarded. In this example we discard 3×1000 data sets and $3 \times 1000 \times 2$ individual result objects. Although the function **`as.data.frame`** or to be more precise **`as.data.frame.evalGrid`** has also a parameter **`post.proc`** that serves the same purpose, it may be necessary to summarize the results as soon as possible to spare memory.

We now briefly show that `post.proc` in `evalGrids` and `as.data.frame` yield the same results.

```r
set.seed(1234)
# summarize the result objects as soon as possible
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000,
    post.proc=mean)

## [1] "Estimated replications per hour:  24216768"

as.data.frame(eg)

##   i j   fun  n proc value      V1
## 1 1 1 runif 10  min (all) 0.09211
## 2 1 2 runif 10  max (all) 0.90852
## 3 2 1 runif 20  min (all) 0.04986
## 4 2 2 runif 20  max (all) 0.95271
## 5 3 1 runif 30  min (all) 0.03026
## 6 3 2 runif 30  max (all) 0.96821

set.seed(1234)
# keeping the result objects
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 1000)

## [1] "Estimated replications per hour:  40987685"

# summarize the result objects by as.data.frame
as.data.frame(eg, post.proc=mean)

##   i j   fun  n proc value      V1
## 1 1 1 runif 10  min (all) 0.09211
## 2 1 2 runif 10  max (all) 0.90852
## 3 2 1 runif 20  min (all) 0.04986
## 4 2 2 runif 20  max (all) 0.95271
## 5 3 1 runif 30  min (all) 0.03026
## 6 3 2 runif 30  max (all) 0.96821
```

4.5 Parameter `ncpus` and `clusterSeed`

By specifying **`ncpus`** larger than 1 a cluster objected is created for the user and passed to the parameter **`cluster`** discussed in the next section.

```r
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 10,
    ncpus=2, post.proc=mean)

## Loading required package:  parallel

## [1] "Estimated replications per hour:  126867"

as.data.frame(eg)
```

```
##   i j   fun  n proc value      V1
## 1 1 1 runif 10  min (all) 0.08837
## 2 1 2 runif 10  max (all) 0.91211
## 3 2 1 runif 20  min (all) 0.06557
## 4 2 2 runif 20  max (all) 0.94724
## 5 3 1 runif 30  min (all) 0.03624
## 6 3 2 runif 30  max (all) 0.96856
```

As it is stated in command line 6, the replications are parallelized. In our case, this means that roughly every CPU conducts 5 replications.

The parameter `clusterSeed` must be an integer vector of length 6 and serves the same purpose as the function `set.seed`. By default, `clusterSeed` equals `rep(12345, 6)`. Note, in order to reproduce the simulation study it is also necessary that `ncpus` does not change.

4.6 Parameter `cluster`

The user can create a cluster on its own. This also enables the user to distribute the replications over different computers in a network.

```
require(parallel)
cl = makeCluster(rep("localhost", 2), type="PSOCK")
eg = evalGrids(dataGrid = dg, procGrid = pg, replications = 10,
    cluster=cl, post.proc=mean)

## [1] "Estimated replications per hour:  125604"

as.data.frame(eg)

##   i j   fun  n proc value      V1
## 1 1 1 runif 10  min (all) 0.08837
## 2 1 2 runif 10  max (all) 0.91211
## 3 2 1 runif 20  min (all) 0.06557
## 4 2 2 runif 20  max (all) 0.94724
## 5 3 1 runif 30  min (all) 0.03624
## 6 3 2 runif 30  max (all) 0.96856

stopCluster(cl)
```

As you can see our cluster consists of 3 workers. Hence, this reproduces the results from the last code chunk above. Further note, if the user starts the cluster, the user also has to stop the cluster. A cluster that is created within `evalGrids` by specifying `ncpus` is also stop within `evalGrids`.

4.7 Parameter `clusterLibraries` and `clusterGlobalObjects`

A newly created cluster is "empty". Hence, if the simulation study requires libraries or objects from the global environment, they must be transferred to the cluster.

Lets look at standard example from the `boot` package.

```
library(boot)
ratio <- function(d, w) sum(d$x * w)/sum(d$u * w)
city.boot <- boot(city, ratio, R = 999, stype = "w",
    sim = "ordinary")
boot.ci(city.boot, conf = c(0.90, 0.95),
    type = c("norm", "basic", "perc", "bca"))

## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 999 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = city.boot, conf = c(0.9, 0.95), type = c("norm",
##     "basic", "perc", "bca"))
##
## Intervals :
## Level      Normal              Basic
## 90%   ( 1.096,  1.853 )   ( 1.047,  1.755 )
## 95%   ( 1.023,  1.926 )   ( 0.814,  1.777 )
##
## Level     Percentile            BCa
## 90%   ( 1.285,  1.994 )   ( 1.308,  2.047 )
## 95%   ( 1.264,  2.226 )   ( 1.275,  2.293 )
## Calculations and Intervals on Original Scale
```

The following data generating function is extremely boring because it always returns the data set `city` from the library `boot`.

```
returnCity = function(){
  city
}
bootConfInt = function(data){
city.boot <- boot(data, ratio, R = 999, stype = "w",
    sim = "ordinary")
boot.ci(city.boot, conf = c(0.90, 0.95),
    type = c("norm", "basic", "perc", "bca"))
}
```

The function `ratio` exists at the moment only in our global environment. Further we had to load the `boot` package. Hence, we load the `boot` package by setting `clusterLibraries = c("boot")` and transfer the function `ratio` by setting `clusterGlobalObjects = c("ratio")`.

```
dg = expandGrid(fun="returnCity")
pg = expandGrid(proc="bootConfInt")
eg = evalGrids(dg, pg, replications=10, ncpus=2,
    clusterLibraries=c("boot"),
    clusterGlobalObjects=c("ratio"))

## [1] "Estimated replications per hour:  122237"
```

Of course, it is possible to set `clusterGlobalObjects=ls()`, but then all objects from the global environment are transferred to all workers.

4.8 Parameter `fallback`

If the user is afraid of a power black out, server crashes, or something else interrupting the simulation study, the user can pass a character to `fallback`. Then every time a new element in command line 6 is chosen, the results obtained so far are written to the file specified in `fallback`.

```
genData = function(n){
  n
}
anaData = function(data){
  if (data == 4)
    stop("Simulated error that terminates the simulation")
  data^2
}
dg = expandGrid(fun="genData", n=1:5)
pg = expandGrid(proc="anaData")
try(eg <- evalGrids(dg, pg, replications=2,
    fallback="simTool_fbTest"))

## [1] "With fallback!"
```

Loading the `Rdata`-file creates an `R`-object `fallBackObj` of the class `evalGrid`. Of course, some results are missing which is indicated by the column `.evalGridComment` in the resulting `data.frame`.

```
# clean the current R-session
rm(list=ls())
load("simTool_fbTest.Rdata")
as.data.frame(fallBackObj)

##   i j     fun n    proc replication V1 .evalGridComment
```

```
## 1 1 1 genData 1 anaData          1  1             <NA>
## 2 1 1 genData 1 anaData          2  1             <NA>
## 3 2 1 genData 2 anaData          1  4             <NA>
## 4 2 1 genData 2 anaData          2  4             <NA>
## 5 3 1 genData 3 anaData          1  9             <NA>
## 6 3 1 genData 3 anaData          2  9             <NA>
## 7 4 1 genData 4 anaData       <NA> NA  Results missing
## 8 5 1 genData 5 anaData       <NA> NA  Results missing
```

4.9 Parameter `envir`

The function `evalGrids` generates in a first step function calls from `dataGrid` and `procGrid`. This is achieved by applying the `R`-function `get`. By default, `envir=globalenv()` and thus `get` searches the global environment of the current R session. An example shows how to use the parameter `envir`.

```
# masking summary from the base package
summary = function(x) sd(x)
g = function(x) quantile(x, 0.1)
someFunc = function(){
  summary = function(x) c(sd=sd(x), mean=mean(x))

  dg = expandGrid(fun="runif", n=100)
  pg = expandGrid(proc=c("summary", "g"))

  # the standard is to use the global
  # environment, hence summary defined outside
  # of someFunc() will be used
  print(as.data.frame(evalGrids(dg, pg)))
  cat("--------------------------------------------------\n")
  # will use the local defined summary, but g
  # from the global environment, because
  # g is not locally defined.
  print(as.data.frame(evalGrids(dg, pg, envir=environment())))
}
someFunc()

## [1] "Estimated replications per hour:  5672236"
##   i j   fun   n    proc replication     V1    10%
## 1 1 1 runif 100 summary           1 0.2866     NA
## 2 1 2 runif 100       g           1     NA 0.1124
## --------------------------------------------------
## [1] "Estimated replications per hour:  12069939"
##   i j   fun   n    proc replication     sd   mean    10%
## 1 1 1 runif 100 summary           1 0.2871 0.5386     NA
```

```
## 2 1 2 runif 100       g           1      NA      NA 0.1958
```

4.10 The result object

Usually, the user has not work with the object returned by `evalGrids` because `as.data.frame` can coerce it to a `data.frame`. Nevertheless, we want to discuss the return value of `evalGrids`.

```
dg = rbind.fill(
  expandGrid(fun="rexp", n=c(10, 20), rate=1:2),
  expandGrid(fun="rnorm", n=c(10, 20), mean=1:2))
pg = rbind.fill(
  expandGrid(proc="min"),
  expandGrid(proc="mean", trim=c(0.1, 0.2)))
```

Now we conduct a simulation study and discuss the result object

```
eg = evalGrids(dg, pg, replications=100)

## [1] "Estimated replications per hour:  4567120"
```

The returned object is a `list` of class `evalGrid`:

```
names(eg)

## [1] "call"               "dataGrid"           "procGrid"
## [4] "simulation"         "post.proc"          "est.reps.per.hour"
## [7] "sessionInfo"
```

The important element is `simulation` which itself is a `list`. It optionally contains ALL data that were generated and optionally contains ALL objects returned by the data analyzing functions. The structure is as follows. `eg$simulation[[i]][[r]]$data` is the data generated by the ith row in `dg` in the rth replication and `eg$simulation[[i]][[r]]$results[[j]]` is the object returned by the jth parameter constellation of `pg` applied to `eg$simulation[[i]][[r]]$data`. For instance, let $i = 7, r = 22$, and $j = 3$. We generated the data according to

```
dg[7,]

##     fun  n rate mean
## 7 rnorm 10   NA    2
```

that is 10 normal distributed random variables with mean 2 and analyzed it with

```
pg[3,]

##   proc trim
## 3 mean  0.2
```

In the 22nd replication this leads to

```
eg$simulation[[7]][[22]]$results[[3]]

## [1] 2.782
```

which can be replicated by

```
mean(eg$simulation[[7]][[22]]$data, trim=0.2)

## [1] 2.782
```

5 Converting results to `data.frame`

We have already applied `as.data.frame.evalGrid` many times. This function also has the parameters `post.proc` and `progress`. The functionality of these parameter resembles the corresponding parameters of `evalGrids`. Hence, it remains to explain `value.fun`. Sometimes, the objects returned by the analyzing functions can not be automatically converted to `data.frame`. In such cases, the parameter `value.fun` enables the user to pre-process the result objects. We exemplify this by calculating linear regression models.

```
genRegData <- function(){
  data.frame(
    x = 1:10,
    y = rnorm(10, mean=1:10))
}
```

```
eg <- evalGrids(
  expandGrid(fun="genRegData"),
  expandGrid(proc="lm", formula=c("y ~ x", "y ~ x + I(x^2)")),
  replications=100)

## [1] "Estimated replications per hour:  1006231"

class(eg$simulation[[1]][[1]]$results[[1]])

## [1] "lm"
```

An object of class `lm` can easily be converted by calling `coef`.

```
head(df<-as.data.frame(eg, value.fun=coef))

##   i j       fun proc formula replication (Intercept)      x I(x^2)
## 1 1 1 genRegData   lm   y ~ x           1      0.1184 1.0614     NA
## 2 1 1 genRegData   lm   y ~ x           2      0.7990 0.9287     NA
## 3 1 1 genRegData   lm   y ~ x           3     -0.6973 1.1399     NA
## 4 1 1 genRegData   lm   y ~ x           4     -0.5630 0.9967     NA
## 5 1 1 genRegData   lm   y ~ x           5     -0.2929 0.9978     NA
## 6 1 1 genRegData   lm   y ~ x           6     -1.4514 1.2381     NA
```

Of course, this can be combined with `post.proc`

```
as.data.frame(eg, value.fun=coef, post.proc=c(mean, sd))

##   i j       fun proc        formula value (Intercept)_mean (Intercept)_sd
## 1 1 1 genRegData   lm          y ~ x (all)         -0.04609         0.7228
## 2 1 2 genRegData   lm y ~ x + I(x^2) (all)         -0.06337         1.3401
##   x_mean   x_sd I(x^2)_mean I(x^2)_sd
## 1  1.004 0.1114          NA        NA
## 2  1.012 0.5677  -0.0007857   0.04935
```