LINKÖPINGS UNIVERSITET
Department of information and computer science
Statistics and Machine learning
Leif Jonsson
Examiner: Mattias Villani

Exam

732A94 Advanced Programming in R - Exam solutions

Time:	13:15-17:00, 2016-11-23
Material:	The extra material is included in the zip-file **exam_material.zip**.
Grades:	A = 19-20 points.
	B = 17-18 points.
	C = 12-16 points.
	D = 10-11 points.
	E = 8-9 points.
	F = 0-7 points.

Instructions

Write your code in an R script file named **Main.R**. The R code should be complete and readable code, possible to run by copying directly into a script. Comment directly in the code whenever something needs to be explained or discussed. Follow the instructions carefully.

Problem 1 (6 p)

a) Implement a function called `gen_shape()` that does not take any arguments. The function should return a **data frame** consisting of three variables, `t, x` and `y.`

`t` should contain a sequence of numbers from 1 to 2*pi in steps of 0.1.

`x` should contain the numbers generated by

$$f(t) = 16 * sin(t)^3$$

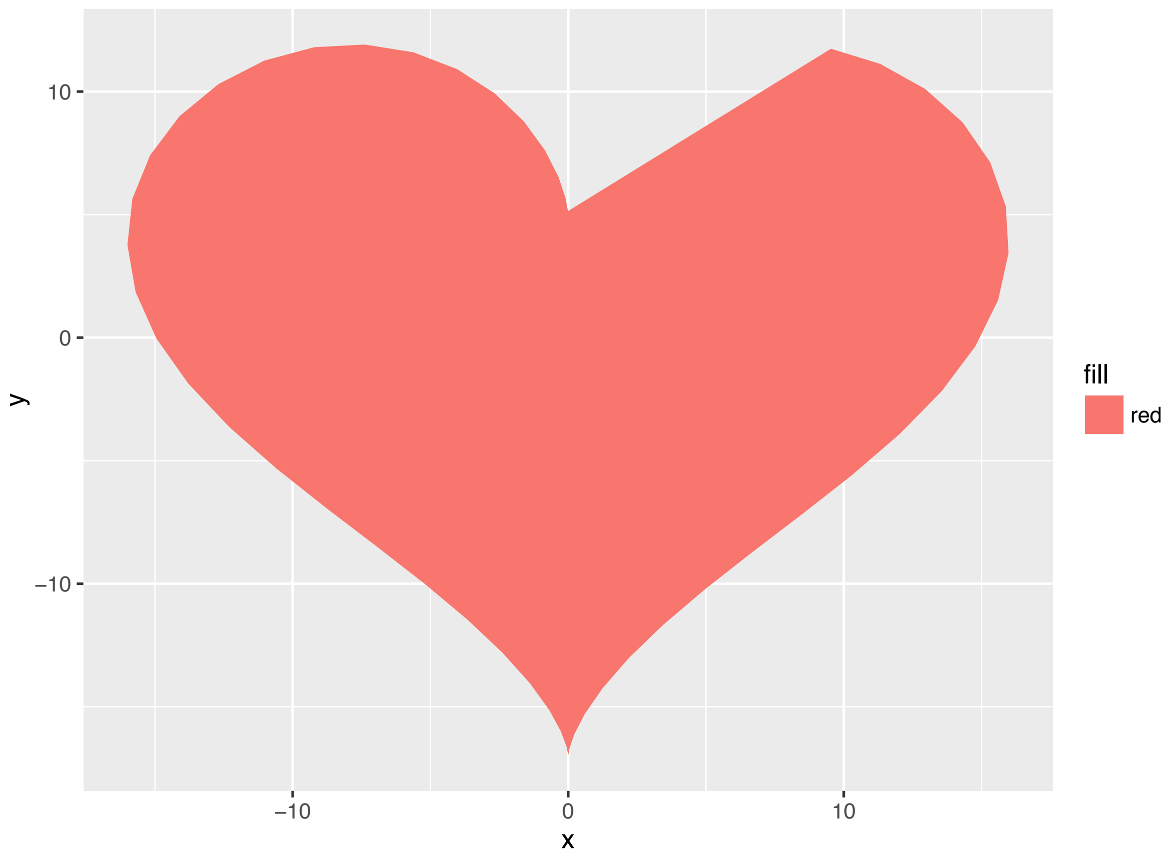
`y` should contain the numbers generated by

$$f(t) = 13 * cos(t) - 5 * cos(2 * t) - 2 * cos(3 * t) - cos(4 * t)$$

Call the function and use ggplot to display the resulting **data frame** using ggplot's **polygon geometric** with the `x` and `y` as the corresponding aesthetics.

Suggested solution:

```
function() {
  df <- data.frame(t=seq(1, 2*pi, by=0.1) )
  df$x <- 16*sin(df$t)^3
  df$y <- 13*cos(df$t)-5*cos(2*df$t)-2*cos(3*df$t)-cos(4*df$t)
  df
}
```



---

b) Write a testcase for you function using testthat. Check that the output type is correct and that `t`, `x` and `y` has correct values according to spec.

Suggested solution:

```
library(testthat)
test_that("gen_shape() is working", {
  df <- gen_shape()
 expect_is(df, "data.frame")
  t <- seq(1, 2*pi, by=0.1)
  expect_equal(df$t, t)
  expect_equal(df$x, 16*sin(t)^3)
  expect_equal(df$y, 13*cos(t)-5*cos(2*t)-2*cos(3*t)-cos(4*t))
})
```

---

c) Describe the fundamental environments related to functions and relate how they come into play in the gen_shape example in Task 1.a

---

Suggested solution:
The gen_shape function is bound to a name in the Global environment which is gen_shape's **binding environment.** The **df** data frame is declared in the **execution environment** of the gen_shape function. The **calling environment** of the gen_shape function is the Global environment. The **enclosing environment** of gen_shape is also the Global environment.

Problem 2 (7 p)

a) In this task you should use object oriented programming in S3 or RC to implement a simulator for the bombings of London during World War II. The first task is to implement a function called build_city that returns a "city" object. The build_city function should take two arguments, a **name** and the **number of "regions"** that the city is divided into. The object shall represent each region ID as an integer value, and also the number of hits during a bombing raid that each region has taken.

```
# S3 and RC call to build_city function
london <- build_city(name="London", nr_regions=576)
```

Suggested solution:

```
function(name, nr_regions) {
  city <- list(name=name, bombings=data.frame(region=1:nr_regions, hits=numeric(nr_regions)))
  class(city) <- "city"
  city
}
```

```
# Solution using RC and reference semantics

city_builder <- setRefClass("city_rc",
                     fields = list(name = "character",
                                  bombings = "data.frame"),
                    methods = list(
                      simulate_bombings = function(nr_raids) {
                        for(raid in 1:nr_raids) {
                          for(region in 1:nrow(bombings)) {
                            bombings$hits[region] <<- bombings$hits[region] + rpois(1,lambda=0.93)
                          }
                        }
                      }))

build_city_rc <- function(name, nr_regions) {
  df <- data.frame(region=1:nr_regions, hits=numeric(nr_regions))
  city <- city_builder$new(name="london",bombings=df)
  city
}

london_rc <- build_city_rc(name="London", nr_regions=576)
```

```
london_rc$simulate_bombings(71)

plot.city_rc <- function (x,...) {
  cat("Plotting", x$name, "...")
  print(ggplot(x$bombings) + geom_bar(aes(x=region , y=hits), colour="blue",stat="identity"))
}

plot(london_rc)
```

---

b) Now implement a function called `simulate_bombings` to simulate the bombing raids. If you use RC, you can add the function to your city object that you implemented in Task 2.a if you like otherwise use an ordinary function. The function should take two arguments (only one argument (the number of raids) is needed if using RC), a city and the number of raids.

The simulation should loop for “number of raids” times. In each raid it should loop over each region in the city and “bomb” it. The number of bombs that hit a region is Poisson distributed with `lambda=0.93`. The function should return a city with the updated bombing statistics for each region in the city.

```
# S3 and RC call to simulate_bombings function
bombed_london <- simulate_bombings(london,71)

# It is also ok to use the following OO style of simulation if using RC
# london$simulate_bombings(71)
```

Suggested solution:
For RC, see answer to Task2.a

```
function(city, nr_raids) {
  for(raid in 1:nr_raids) {
    for(region in 1:nrow(city$bombings)) {
      city$bombings$hits[region] <- city$bombings$hits[region] + rpois(1,lambda=0.93)
    }
  }
  city
}
```

---

c) What is the complexity of the `simulate_bombings` algorithm with regard to the number of raids and the number of regions?

Suggested solution:
The complexity is $O(raids * regions)$.

---

d) Implement a `plot` function for your city objects. The function should use ggplot to visualize the bombing statistics for each region. You are free to choose how to visualize it yourself!

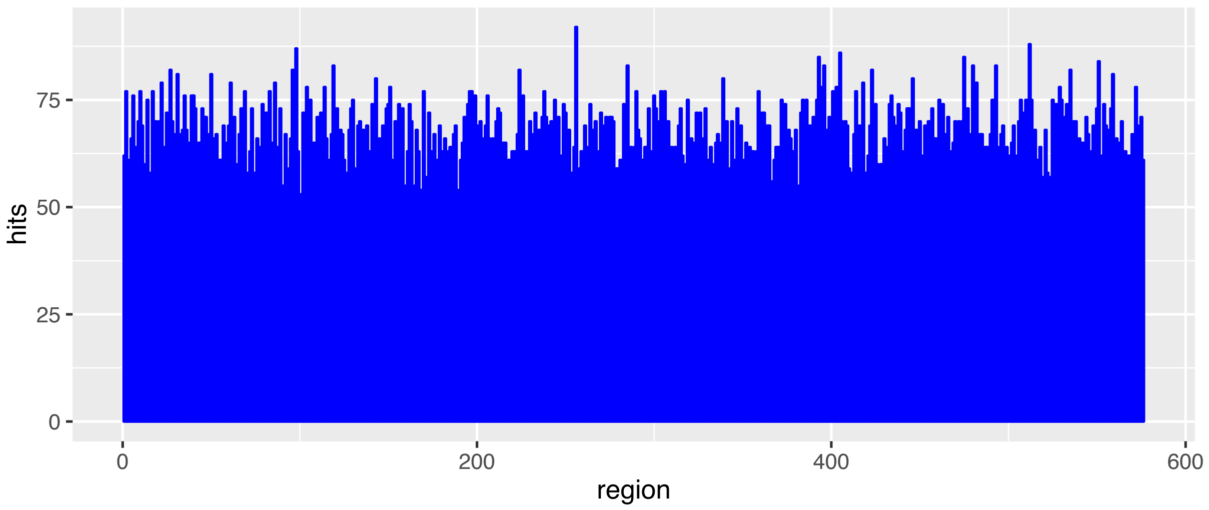
```
# S3 and RC call to print function
plot(bombed_london)

Plotting London ...

# If you use RC with reference semantics, it is also ok to have
# plot(london)
```

Suggested solution:
For RC, see answer to Task2.a

```
plot.city <-  function (x, ...)
{
    cat("Plotting", x$name, "...")
    print(ggplot(x$bombings) + geom_bar(aes(x = region, y = hits),
        colour = "blue", stat = "identity"))
}
Plotting London ...
```



Problem 3 (7 p)

a) You have moved on to the construction business realizing that this Big Data stuff is just a hype anyway. You are building a simulator for a nuclear power plant.

Implement a function called `power_plant_factory` that takes no arguments and returns a **function** (lets call it `power_plant`) that represents a power plant. Each power plant has an internal `heat` (0 at start) and `in_meltdown` (FALSE at start) states.

The function **returned** should take one argument, the number of watts that an operator wants to take out of the plant. Each time the `power_plant` function is called there is a chance that it will go into meltdown, this chance is distributed by a Poisson with `lambda=<current heat>` that is, the more heat the bigger the chance of a meltdown. If the draw from the Poisson with `lambda=heat` exceeds 11 Mega Watt, the plant will go into meltdown and always print “Warning: Reactor is in MELTDOWN!” when called after that.

If it does **not** go into meltdown and the heat is **below** 10 Mega Watt, the heat is **increased** with the **number of watts requested** and the function prints “Extracting more power!”. If the current heat is **10 or more** Mega Watt, no more power can be taken out of the plant and the function prints “Warning: Reactor overheated, cannot extract more power!” and **decreases** the heat with **2** Mega Watt as it cools down.

```
set.seed(4712)
power_plant <- nuclear_plant_factory()
power_plant(5)

[1] "Extracting more power!"

power_plant(5)

[1] "Extracting more power!"

power_plant(5)

[1] "Warning: Reactor overheated, cannot extract more power!"

power_plant(5)

[1] "Extracting more power!"

power_plant(5)

[1] "Warning: Reactor has gone unstable, NUCLEAR MELTDOWN!"

power_plant(5)

[1] "Warning: Reactor is in MELTDOWN!"
```

Suggested solution:

```
function(critical_limit=3) {
  heat <- 0
  in_meltdown <- FALSE
  function(power) {
    if(in_meltdown) {
      print("Warning: Reactor is in MELTDOWN!")
      return(invisible(heat))
    }
    drw <- rpois(1,lambda = heat)
    if(drw>=11) {
      print("Warning: Reactor has gone unstable, NUCLEAR MELTDOWN!")
      in_meltdown <<- TRUE
      return(invisible(heat))
    }
    if(heat<10) {
      print("Extracting more power!")
      heat <<- heat + power
    } else {
      print("Warning: Reactor overheated, cannot extract more power!")
      heat <<- heat - 2
    }
invisible(heat)
  }
}
```

---

b) Construction was too dull! You are now part of the **Scorpion** team, a crack team of geniuses that helps save the world. Your mission is to save the power plant that has gone into meltdown and protect the plant city from annihilation.

You should write a function that takes a power plant as an argument and resets the heat in it to 0 and resets the meltdown status to false and then prints a happy message "Yay, Scorpion has saved the reactor!"

```
save_plant(power_plant)

[1] "Yay, Scorpion has saved the reactor!"

power_plant(5)

[1] "Extracting more power!"
```

Suggested solution:

```
function(power_plant) {
  penv <- environment(power_plant)
  penv$heat <- 0
  penv$in_meltdown <- FALSE
  print("Yay, Scorpion has saved the reactor!")
}
```

Good luck!