Homework 2 in R Markdown

*Cosmin Borsa**

20 September 2018

Abstract

In this document we will approximate the probability distribution function of a standard normal distribution using Monte-Carlo methods. We will also talk about some topics related to the double-precision binary floating-point format.

Keywords: Template; R Markdown; **bookdown**; **knitr**; **Pandoc**

1 Introduction

The standard normal distribution is probably the most popular continuous probability distribution. It is used not only in statistics, but also in the natural and social sciences. The probability density function of the standard normal distribution is given by the following formula:

$$\Phi(t) = \int_{-\infty}^{t} \frac{1}{\sqrt{2\pi}} e^{\frac{-y^2}{2}} dy. \tag{1}$$

The probability density function of the standard normal distribution $\Phi(t)$ cannot be expressed in terms of elementary functions; thus, it doesn't have a closed-form expression. In evaluate the probability density function $\Phi(t)$, we will use Monte Carlo simulations.

To estimate the probability density function $\Phi(t)$, we are going to first generate n normally distributed random observations X_i. Then, we are going to count how many of them are less than a given value t. To keep things fairly straight forward, we are going to have $n \in \{100, 1000, 10000\}$ and $t \in \{0.0, 0.67, 0.84, 1.28, 1.65, 2.32, 2.58, 3.09, 3.72\}$. The Monte-Carlo estimate of the probability density function is going to be given by the formula (2) and is represented by $\hat{\Phi(t)}$.

$$\hat{\Phi}(n, t) = \frac{1}{n} \sum_{i=1}^{n} I(X_i \leq t). \tag{2}$$

2 Monte Carlo Implementation

For implementation of the Monte-Carlo method into R, we are going to define a function `MC_normal_dist`. This function will estimate the probability density function of the standard normal distribution $\Phi(t)$ using the formula (2).

*cosmin.borsa@uconn.edu; M.S. in Applied Financial Mathematics, Department of Mathematics, University of Connecticut.

```r
MC_normal_dist<-function (n,t)
{
inc<-0
randv<-rnorm(n)
for(i in 1:n){
  if(randv[i] <= t){inc = inc + 1}
  }
estprob <- inc/n
return(estprob)
}
```

Next, we are going to use the function `MC_normal_dist` to create a table with the various values of n and t. The function `MC_table` does this task. It saves the values of t in a vector and uses a for loop to compute the estimates for each n. In order to display the table in an orderly fashion, we have rounded the results of the estimation to 3 decimal places. We have also used the `set.seed(1)` function in order to fix the random generated numbers.

```r
MC_table<-function ()
{
  set.seed(1)

  t <- c(0.0,0.67,0.84,1.28,1.65,2.32,2.58,3.09,3.72)
  t_column <-numeric(length(t))
  est100 <-numeric(length(t))
  est1000 <-numeric(length(t))
  est10000 <-numeric(length(t))
  true_val <-numeric(length(t))

  for (i in 1:length(t)){
    t_column[i] <- t[i]
    est100[i] <- signif(MC_normal_dist(100,t[i]), digits = 3)
    est1000[i] <- signif(MC_normal_dist(1000,t[i]), digits = 3)
    est10000[i] <- signif(MC_normal_dist(10000,t[i]), digits = 3)
    true_val[i] <- signif(pnorm(t[i]), digits = 3)
  }

  matr <-cbind(t_column,true_val, est100,est1000,est10000)
  colnames(matr) <- c("t", "True Value", "n = 100", "n = 1,000", "n = 10,000")

    return(matr)
}
```

Next we would like to generate the table and display it using the function `knitr::kable`.

3 Box Plots

In this section we are going to evaluate how well the Monte-Carlo method has estimated the the probability density function $\Phi(t)$ at various levels of t. For this task we have computed the

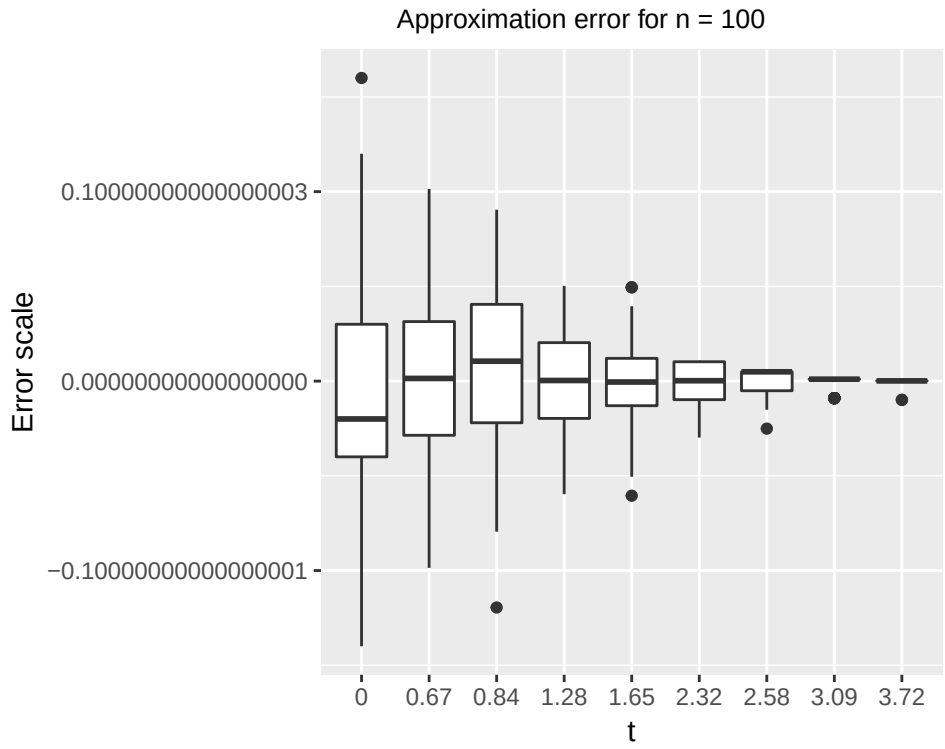
Table 1: Comparison of Monte-Carlo estimates and the true values of the probability density function.

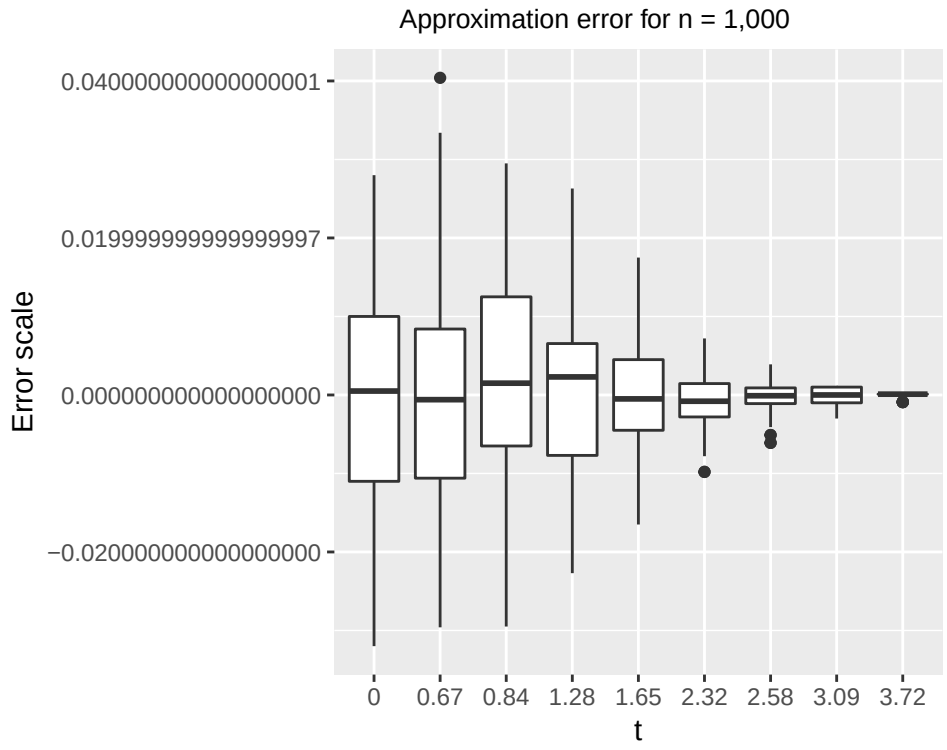
t	True Value	n = 100	n = 1,000	n = 10
0.00000000000000000	0.50000000000000000	0.46000000000000002	0.52100000000000002	0.506000000000000
0.67000000000000004	0.74900000000000000	0.76000000000000001	0.74299999999999999	0.753000000000000
0.83999999999999997	0.80000000000000004	0.76000000000000001	0.79400000000000004	0.796000000000000
1.28000000000000003	0.90000000000000002	0.87000000000000000	0.90800000000000003	0.902000000000000
1.64999999999999991	0.95099999999999996	0.95999999999999996	0.95399999999999996	0.950999999999999
2.31999999999999984	0.98999999999999999	1.00000000000000000	0.99500000000000000	0.990999999999999
2.58000000000000007	0.99500000000000000	1.00000000000000000	0.99900000000000000	0.995000000000000
3.08999999999999986	0.99900000000000000	1.00000000000000000	1.00000000000000000	0.999000000000000
3.72000000000000020	1.00000000000000000	1.00000000000000000	1.00000000000000000	1.000000000000000

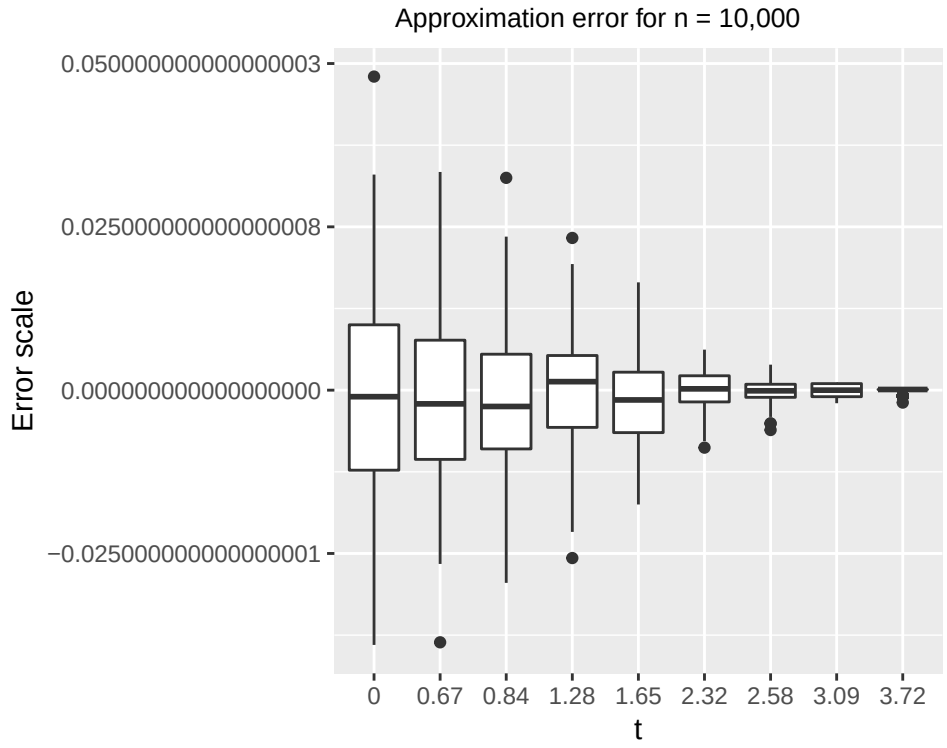
approximation error $\epsilon(t)$ by substracting the true value of the probability density function from each Monte-Carlo estimation.

$$\epsilon(t) = \hat{\Phi}(t) - \Phi(t)$$

In order to vizualize the errors at each level of t, we have generated 100 approximation errors for each pair of n and t. The results are displayed using `ggplot2` in the following 3 figures.




Approximation error for n = 1,000
Error scale
0.040000000000000001
0.019999999999999997
0.000000000000000000
−0.020000000000000000
0
0.67
0.84
1.28
1.65
2.32
2.58
3.09
3.72
t


Approximation error for n = 10,000
Error scale
0.050000000000000003
0.025000000000000008
0.000000000000000000
−0.025000000000000001
0
0.67
0.84
1.28
1.65
2.32
2.58
3.09
3.72
t

4 Floating-point arithmetic

In this section we are going to explain how `.Machine$double.xmax`, `.Machine$double.xmin`, `.Machine$double.eps`, and `.Machine@double.neg.eps` are defined using the 64-bit double precision floating point arithmetic. First of all, `Machine()` is a function that returns information on the numeric characteristics of the computer that R is running on. Such characteristics include the largest double or the machine's precision. `.Machine` is the variable that stores these information.

`.Machine$double.xmax` gives the largest finite floating-point number. In the double-precision binary floating-point format this number is represented by:

0 11111111110 1111111111111111111111111111111111111111111111111111

On this computer the value of `.Machine$double.xmax` is:

```
## [1] 1.7976931348623157e+308
```

`.Machine$double.xmin` gives the smallest non-zero normalized floating-point number. In the double-precision binary floating-point format this number is represented by:

0 00000000001 0000000000000000000000000000000000000000000000000000

On this computer the value of `.Machine$double.xmin` is:

```
## [1] 2.2250738585072014e-308
```

`.Machine$double.eps` returns the smallest positive floating-point number x such that $1 + x! = 1$. In the double-precision binary floating-point format this number is represented by:

0 01111001011 0000000000000000000000000000000000000000000000000000

On this computer the value of `.Machine$double.eps` is:

```
## [1] 2.2204460492503131e-16
```

`.Machine$double.neg.eps` returns the smallest positive floating-point number x such that $1-x! = 1$. In the double-precision binary floating-point format this number is represented by:

0 01111001010 0000000000000000000000000000000000000000000000000000

. On this computer the value of `.Machine$double.neg.eps` is:

```
## [1] 1.1102230246251565e-16
```


Acknowledgment

I would like togive a special thanks to Professor Jun Yan for granting me a deadline extension.