2DI90: Probability Theory and Statistics

Simulation of Random Variables and Monte-Carlo Methods

1 Introduction

In this guide you'll find a set of computer experiments where you'll learn more about simulation of random variables and Monte-Carlo methods. We will make use of the statistical package R, which is free and available for many different platforms and operating systems. If you prefer you can also use other computing platforms, such as Matlab or Mathematica, but you'll have to translate the code listed below. If using R you'll need to install a working version of R, and have access to the command prompt.

The statistical software R is a command interpreter, and not a compiler. This means you can write scripts, and type commands in a command prompt. I'll assume you don't have any prior experience with R, and therefore will guide you through the various tasks needed to solve the questions. Note that you can read R code almost as easily as pseudo-code. In case you need there is a `help()` command can be useful sometimes. Just type it like above, or insert the command you are interested in between the brackets.

2 Simulation of Random Variables

In this part of the lab we will write functions that generate samples from some specific random variables.

2.1 Discrete Distributions

A Simple Discrete Distribution

In class we saw that we could generate samples from an arbitrary discrete distribution provided we have a way to generate standard uniform random variables. In R this is done using the command `runif()`. Let's use the approach in class to generate samples of a random variable X such that

$$P(X = x) = \begin{cases} 0.2 & \text{if } x = 0 \\ 0.5 & \text{if } x = 2 \\ 0.3 & \text{if } x = 3 \end{cases} .$$

Using the notation in class we have

$$p_1 = 0.2 \qquad p_2 = 0.5 \qquad p_3 = 0.3 \ ,$$

and

$$x_1 = 0 \qquad x_2 = 2 \qquad x_3 = 3 \ .$$

Therefore, we should partition the unit interval into three sets, respectively $A_1 = [0, p_1] = [0, 0.2]$, $A_2 = [p_1, p_1 + p_2] = [0.2, 0.7]$, and $A_3 = [p_1 + p_2, p_1 + p_2 + p_3] = [0.7, 1]$. The method consists on generating a standard uniform random variable U, and if $U \in A_i$ we return x_i for $i \in \{1, 2, 3\}$. Let's encode this using R. Type the following code on the command prompt. If you prefer type it on a script file (text file) and run it afterwards.

```
# function to compute n i.i.d. samples from the above distribution
Dist1<-function (n)
{
   #creates a vector of length n with i.i.d. uniform random variables
   U <- runif(n)

   X <- ((0<U)&(U<=0.2))*0+((0.2<U)&(U<=0.7))*2+((0.7<U)&(U<=1))*3
   return(X)
}
```

This function is now stored in memory, and ready to be invoked. Let's generate a sequence of 100 independent and identically distributed random variables with the specified distribution. This can be done by typing

```
X <- Dist1(100)
```

Let's check if the answer makes sense. Let's see what is the proportion samples taking the values $0, 2$ and 3. We can do this graphically using the command

```
hist(X,breaks=(0:5)-0.5)
```

The height of each bar is the number of samples that take each one of the values. Do the results make sense to you?

Experiment with different values of n. What happens when n is very large (e.g. 10000)?

Poisson Distribution

Let's use this approach to construct a function that generate samples from a Poisson random variable with parameter $\lambda > 0$. Recall that the probability mass function of a Poisson random variable is given by

$$f(k) = e^{-\lambda}\frac{\lambda^k}{k!}\ , \qquad k \in \mathbb{N}_0\ .$$

It is good to keep in mind that Poisson random variables can take ANY value in $\mathbb{N}_0$. Therefore one should be careful when implementing such a function, so that it is relatively efficient. Enter the following code in you R software.

```
Poisson <- function(lambda,n)
{
   U <- runif(n)

   X <- rep(0,n)
   k <- 0
   f <- exp(-lambda)
   F <- f
   while(F<max(U))
   {
      tmp <- (U>F)
      X <- X + tmp
      k <- k+1
      f <- f*lambda/k
      F <- F+f
   }
   return(X)
}
```

Try to understand this implementation of the method. It is done in this way to avoid several loops that are very inefficient. Instead we take advantage of the fact the R is very efficient when dealing with vectors and matrices.

Let's experiment with this function. Begin by generating 100 samples from a Poisson distribution with parameter $\lambda = 4.5$.

```
X <- Poisson(4.5,100)
```

Type `hist(X,breaks=(0:round(1.5*max(X)+2))-0.5)` to check the number of samples taking each of values in $\mathbb{N}_0$. Does this appear reasonable to you?

As you know, by the law of large number, if we normalize the heights of the bars from the plot you just did you should recover approximately the values of the probability mass function. In other words, the normalized height of the bin centered in $k \in \mathbb{N}_0$ should be approximately

$$f(k) = e^{-\lambda}\frac{\lambda^k}{k!} \ .$$

We can plot the normalized height using the command

```
hist(X,breaks=(0:round(1.5*max(X)+2))-0.5, freq=FALSE)
```

Let's super-impose the true probability mass function $f(\cdot)$ on top of the plot we just did

```
lambda <- 4.5
k <- (0:round(1.5*max(X)+2))
f=exp(-lambda)*lambda^k/gamma(k+1)
points(k,f,col='red',pch=19)
```

Do the results seem satisfactory to you? What if you increase n (try really large numbers too, like 1000000). Experiment with different values of λ as well.

The Normal Approximation to the Poisson Distribution

You might have noticed that, as λ gets larger the probability mass function of the Poisson random variables starts looking more and more like a bell-shape curve. As we have seen in class this is the case in particular when $\lambda > 5$. To help us visualize this let's superimpose the density of a normal random variable (with the same mean and variance of the Poisson) to the plots we just made:

```
n <- 100
lambda <- 4.5
X <- Poisson(lambda,n)
hist(X,breaks=(0:round(1.5*max(X)+2))-0.5,freq=FALSE)

k <- (0:round(1.5*max(X)+2))
f=exp(-lambda)*lambda^k/gamma(k+1)
points(k,f,col='red',pch=19)

t <- seq(from=-5, to=1.5*max(X), by=0.05)
f_n <- 1/sqrt(2*pi*lambda)*exp(-(t-lambda)^2/2/lambda)
lines(t,f_n,col='blue')
```

Experiment with various values of λ in the range 0.1 to 100. This should give you some qualitative insight when the normal approximation to the Poisson distribution is reasonable.

2.2 Continuous Distributions

In class you also learned how to generate samples from continuous distributions. We saw there are various methods that can be used. We'll explore these in the following exercises.

Generating Samples from the Weibull Distribution

As you already know, Weibull random variables (with parameters $\delta > 0$ and $\beta > 0$) have probability density function

$$f(x) = \begin{cases} \frac{\beta}{\delta}\left(\frac{x}{\delta}\right)^{\beta-1} e^{-\left(\frac{x}{\delta}\right)^{\beta}} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} .$$

In this case we can analytically compute the cumulative distribution function

$$F(x) = \int_{-\infty}^{x} f(t)dt \ ,$$

With a little work, and a change of variables, we conclude that

$$F(x) = \begin{cases} 1 - e^{-\left(\frac{x}{\delta}\right)^{\beta}} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} .$$

(Check that this is indeed correct, by computing $F'(x)$ and seeing that you get $f(x)$).

This function in continuous, and furthermore we can compute its inverse, simply by solving for x the equation

$$u = F(x) \ ,$$

with $0 < u < 1$. Do this, and check that you get

$$x = F^{-1}(u) = \delta\left(-\ln(1-u)\right)^{1/\beta} \ .$$

We are now ready to apply the result in class that tells us that if U is a standard normal random variable then $F^{-1}(U)$ is a random variable with the density specified above. Therefore the following function will generate samples from a Weibull distribution:

```
Weibull <- function(delta,beta,n)
{
   U <- runif(n)
   X <- delta*(-log(1-U))^(1/beta)
   return(X)
}
```

Generate 100 samples from a Weibull distribution with $\delta = 3000$ and $\beta = 2$ using the command `X <- Weibull(3000,2,100)`. Let's see what this sample looks like by typing

```
hist(X)
```

We can actually estimate the probability density from these samples using the command

```
hist(X,freq=FALSE)
```

Let's check if this is reasonable, by superimposing the plot of the density of a Weibull random variable with the same parameters

```
n <- 100
beta <- 2
delta <- 3000
X <- Weibull(3000,2,n)
hist(X,freq=FALSE)

t <- seq(from=0, to=max(X)*1.5, by=1)
f_w <- beta/delta*(t/delta)^(beta-1)*exp(-(t/delta)^beta)
lines(t,f_w,col='blue')
```

Experiment with different parameters and samples sizes. Comment on your results.

Changing `1-U` to `U`

Replace `1-U` by `U` in the Weibull function code, and run the above experiments again. Did your results changed? Why? Use this code to generate samples from an exponential distribution with parameter $\lambda = 0.1$.

COMPUTATIONAL PROBLEM:

Explain how you can use the inversion method to generate samples from a distribution with density

$$f(x) = \begin{cases} \frac{3}{2}x^2 & \text{if } -1 < x < 1 \\ 0 & \text{otherwise} \end{cases} .$$

Derive all the formulas necessary, and write a simple function that generates samples from that distribution. Check that this function is indeed doing what you want it to do.

Note: you must be a bit careful when implementing the inverse c.d.f..

The Rejection Method

Let's solve the previous problem using instead the rejection method. Note that in this case the density is bounded (the largest value it can take is 1.5) and the density support is also bounded and equal to $[-1, 1]$. The rejection method begins by generating two independent uniform random variables U and V over $[-1, 1]$ and $[0, 1.5]$ respectively. Then you reject the sample it it satisfies $V > f(U)$, otherwise you return $X = U$. Let's code this up:

```
Dist3 <- function(n)
{
   X <- rep(0,n);
   for(k in 1:n)
   {
      repeat
      {
         U <- -1+(1-(-1))*runif(1)
         V <- 1.5*runif(1)
         if(V<=1.5*U^2) {break}
      }
      X[k] <- U
   }
   return(X)
}

n <- 1000
X <- Dist3(n)
hist(X,freq=FALSE)

t <- seq(from=-1, to=1, by=0.01)
f <- 1.5*t^2
lines(t,f,col='blue')
```

Experiment with different values of n. Which code is more effective, and which was easier to deploy? Note that the code of function `Dist3` is not very efficient, because of the two nested loops. There are ways of speeding this up, but we will not worry about this now.

3 Solving Problems using Monte-Carlo Methods

In this section we will use the tools at our disposal to solve some problems that would be otherwise hard to solve analytically. Let's do this for a simple, yet illustrative problem.

Service Time by a Server

A computer server allocates each incoming task to one of two different processors (denoted by A and B). These have different characteristics: the time (in milliseconds) to process a task by processor A is Weibull distributed with parameters $\delta = 260$ and $\beta = 7$. On the other hand, the time processor B takes to complete a task follows a Weibull distribution, with parameters and $\delta = 110$ and $\beta = 2$. The choice of processor to use is made randomly by the server (and independent of the task and everything else). With probability $p = 0.3$ the server allocates an incoming task to processor A, and otherwise the task is allocated to processor B. Let T be the time it takes to complete a task by this server. What can be say about T? In particular:

1. How can we generate samples for this process?
2. What is the probability that it takes more than 250ms to complete a task in this server?
3. What is the expected value of T?
4. What is the variance of T?

We will answer these questions using Monte-Carlo simulation, which requires us to generate independent samples from T. This can be easily done using the functions we already created:

```
SampleT <- function(n)
{
   A <- Weibull(260,7,n);
   B <- Weibull(110,2,n);
   choice <- (runif(n)<0.3)
   T <- (choice==1)*A+(choice==0)*B
}
```

So, we can now generate sample of T with the command `X <- SampleT(n)`. Let's use this to estimate the probability that it takes more than 250ms to complete a task. We begin by generating n samples of T, and count the number of them that are above 250. Dividing this number by n yields the desired result.

```
n <-100
X <- SampleT(n)
sum(X>250)/n
```

Run the above code several times. What can you say about the quality of your estimate of $P(T > 250)$? Suppose now you want to guarantee that the error in your estimate is smaller

than $\epsilon = 0.001$ with probability greater than 0.95 (don't forget that the estimate you obtain by Monte-Carlo methods is also random, and with some probability it might be "off"). We can use the Chernoff bound we saw in class and conclude that we should take

$$n \geq \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\alpha}\right) = \frac{1}{0.000002} \ln\left(\frac{2}{0.05}\right) = 1844440 \ .$$

Run the code above several times for this choice of n. Comment on the outcomes you see.

Since the Chernoff bound is often very conservative let's use instead the normal approximation, also discussed in class. In this case it tells us that we should pick the sample size so that

$$n \geq \left(\frac{z_{\alpha/2}}{2\epsilon}\right)^2 = \left(\frac{1.96}{0.002}\right)^2 = 960400 \ .$$

Experiment with this value and comment on your results. Note that this sample size is about half of what was recommended by using the more conservative approach, therefore speeding up the computations by a factor of two.

To estimate the expected value we can proceed exactly in the same fashion. To compute $\mathbb{E}[T]$ we can simply run:

```
n <-100000
X <- SampleT(n)
sum(X)/n
```

It is worth mentioning that the last line can be simply replaced by `mean(X)`.

To estimate the variance we can use the following expression (presented in class without further justification):

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \overline{X})^2 \ ,$$

where $\overline{X} = \frac{1}{n} \sum_{i=1}^{n} X_i$. This can be done easily with the code

```
n <-100000
X <- SampleT(n)
sum((X-mean(X))^2)/(n-1)
```

Again, the last line can be simply replaced by `var(X)`.

It turns out that you can actually compute the exact form of the density of T, as well as $\mathbb{E}[T]$ and $\mathrm{Var}(T)$. These are respectively:

$$f_T(x) = \begin{cases} 0.3\frac{7}{260}\left(\frac{x}{260}\right)^{7-1} e^{-\left(\frac{x}{260}\right)^7} + 0.7\frac{2}{110}\left(\frac{x}{110}\right)^{2-1} e^{-\left(\frac{x}{110}\right)^2} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \ .$$

$$\mathbb{E}[T] = 0.3 \times 260\Gamma\left(1+\frac{1}{7}\right) + 0.7 \times 110\Gamma\left(1+\frac{1}{2}\right) \ .$$

$$\begin{aligned}\mathrm{Var}(T) &= 0.3 \times 260^2\Gamma\left(1+\frac{2}{7}\right) + 0.7 \times 110^2\Gamma\left(1+\frac{2}{2}\right) \\ &\quad - \left(0.3 \times 260\Gamma\left(1+\frac{1}{7}\right) + 0.7 \times 110\Gamma\left(1+\frac{1}{2}\right)\right)^2 \ .\end{aligned}$$

We can easily compute the values using the gamma function in R, and we get

$$\mathbb{E}[T] = 141.2036 \qquad \text{and} \qquad \mathrm{Var}(T) = 6778.415 \ .$$

Compare these with your estimates, and experiment with different sample sizes. Finally, superimpose the true density to the simulated samples you created, by using the code:

```
n <-100000
X <- SampleT(n)
hist(X,freq=FALSE)
t <- seq(from=0, to=max(X)*1.5, by=0.5)
f <- 0.3*7/260*(t/260)^(7-1)*exp(-(t/260)^7)+
     0.7*2/110*(t/110)^(2-1)*exp(-(t/110)^2)
lines(t,f,col='blue')
```

4 Final Remarks

With this guide you should have gained more familiarity with the basics of computer simulation, and become more familiar with some implementation details.