

Numerical and Statistical Methods for Finance

Generating Discrete Random Variables

Pierpaolo De Blasi

University of Torino

email: pierpaolo.deblasi@unito.it

webpage: sites.google.com/a/carloalberto.org/pdeblasi/

Lecture no. 6
13 November 2013

The Inverse Transform Method

Suppose we want to generate a value of a discrete random variable X taking values in the set $\{0, 1, \dots\}$ with probabilities $\{p_0, p_1, \dots\}$ ($p_i \geq 0$ and $\sum_{i=0}^{\infty} p_i = 1$), i.e. $\Pr(X = j) = p_j$.

To do so, we generate $U \sim \text{Unif}(0, 1)$ and set

We have the pmf of X and we want to find the value of the RV X .

$$X = \begin{cases} 0 & \text{if } U \leq p_0 \\ 1 & \text{if } p_0 < U \leq p_0 + p_1 \\ \vdots & \\ j & \text{if } \sum_{i=0}^{j-1} p_i < U \leq \sum_{i=0}^j p_i \\ \vdots & \end{cases}$$

Since, for $0 < a < b < 1$, $\Pr(a < U \leq b) = (b - a)$,

$$\Pr(X = j) = \Pr\left\{\sum_{i=0}^{j-1} p_i < U \leq \sum_{i=0}^j p_i\right\} = p_j$$

and so X has the desired distribution.

If we let $F(k) = \Pr(X \leq k) = \sum_{i=0}^k p_i$ be the cumulative distribution function (cdf) of X , we have

$$X \text{ will equal } j \text{ if } F(j-1) < U \leq F(j) \quad (1)$$

We determine the value of X by finding the interval $(F(j-1), F(j)]$ in which U lies, or, equivalently, by finding the *inverse of F* , (hence the name *inverse transform method*).

Easy to extend (1) when the values of X are ordered, $x_0 < x_1 < \dots$ ($x_j = j$ is the prototypical example). Readily coded through a `while` loop:

```
# given U ~ Unif(0,1)
X<-0
while (F(X)<U) {
  X <- X+1
}
```

generate a rand number U
set X=0
if F(X)<U set X=0 and stop if not
X=X+1 aumenta di 1

Scriviamo il codice tramite
utilizzando un while loop

we generate only one value for X in this way.

When the algorithm terminates we have $F(X) \geq U$ and $F(X-1) < U$, that is $U \in (F(X-1), F(X)]$, thus $\Pr(X = j) = F(j) - F(j-1) = p_j$.

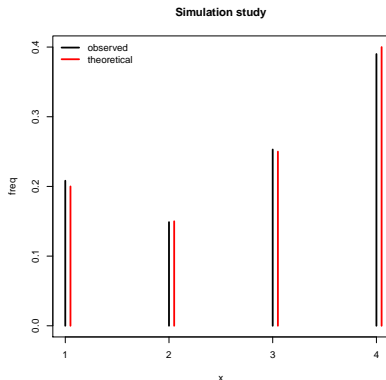
Example 4a page 50, Ross (2006)

Simulate a random variable X such that

$$p_1 = 0.20, \quad p_2 = 0.15, \quad p_3 = 0.25, \quad p_4 = 0.40,$$

```
x<-1:4
p<-c(0.2,0.15,0.25,0.40)
Fx<-cumsum(p)

N<-5000
set.seed(1)
U<-runif(N)
X<-rep(0,N) repeat 0 for N times
for (i in 1:N){
  j<-1
  while (Fx[j]<U[i]){
    j<-j+1
  }
  X[i]<-j
}
freq<-rep(0,4)
for (i in x) freq[i]<-sum(X==i)/N
```



In this way we are creating N PRN with that pmf

'sample()' in R

For simulating a finite (*i.e. discrete with finite number of values*) random variable R provides

```
sample(x, size, replace = FALSE, prob = NULL)
```



The inputs are

- **x**: a vector giving the possible values the rv can take;
- **size**: how many rv's to simulate;
- **replace**: set this to **TRUE** to generate an iid sample, otherwise the rv's will be conditioned to be different from each other;
- **prob**: a vector giving the probabilities of the values in **x**. If omitted then the values in **x** are assumed to be equally likely.

```
X<-sample(x,size=N,replace=T,prob=p)
```

in this case the replace=TRUE and so the sample is iid this means that all the values in x have the same prob to be chosen and is equal to p even why i think its not necessary since is a result 1/the nr of possible values of x

Discrete uniform random variable

Suppose we want to generate the value of X which is equally likely to take any of the values $1, \dots, n$. Using (1),

$$X = j \quad \text{if} \quad \frac{j-1}{n} < U \leq \frac{j}{n}$$

in other words

$$X = \lfloor nU \rfloor + 1$$

sbagliato :(

where $\lfloor a \rfloor$ is the integer part of $a \in \mathbb{R}$.

<code>n<-10</code>	(using sample)
<code>X<-floor(n*runif(N))+1</code>	<code>X<-sample(1:n,size=N,replace=T)</code>

We next use discrete uniform random variables for generating *random permutations*, see Example 4b page 51, Ross (2006).

It uses the function `floor(x)`, whose value is the largest integer smaller than x .

Random Permutation

We are interested in generating a permutation of the integers $1, 2, \dots, n$ which is such that all $n!$ possible ordering are equally likely.

First choose one of the numbers $1, \dots, n$ at random and put that number in position n ; then choose at random one of the remaining $n - 1$ numbers and put that numbers in position $n - 1$; and so on.

The above algorithm can be written as follows:

- Step 1: Let π_1, \dots, π_n be any permutation of $1, \dots, n$ (e.g., choose $\pi_j = j$, $j = 1, \dots, n$).
- Step 2: Set $k = n$.
- Step 3: Generate $U \sim \text{Unif}(0, 1)$ and set $I = \lfloor kU \rfloor + 1$.
- Step 4: Interchange π_I and π_k .
- Step 5: Set $k = k - 1$ and if $k > 1$ go to Step 3.
- Step 6: π_1, \dots, π_n is the final random permutation.

```

n<-10
pi<-1:n -this is the permutation that we want to trans into rand.permutation
k<-n
while (k>1){
  U<-runif(1)
  I<-floor(k*U)+1
  x<-pi[k] ci serve dopo
  pi[k]<-sample(1:n)
  pi[I]<-pi[I] diamo a pi[k] il valore del nr scelto a random
  pi[I]<-x diamo al nr scelto a random il valore precedent di pi[k] che lo abbiamo registrato prima come x
  k<-k-1
}

```

when k becomes equal to 1 the algorithm stops generating.

Random permutation



To generate a **random subset**, say of **size r** , of integers $1, 2, \dots, n$, follow the algorithm until the positions $n, n-1, \dots, n-r+1$ are filled. **The elements in these positions form the random subset.**

Recall that there exist $\binom{n}{r}$ subsets of r elements from $1, 2, \dots, n$, hence the random subset has probability $1/\binom{n}{r}$ to be chosen (*why?*).

When $r > n/2$, it is more efficient to choose a random subset of size $n-r$ and let the elements not in this subset be the random subset of size r (*why?*).

Binomial Random Variable


We present next some code to simulate a binomial rv. Note that R has superior binomial probability and simulation functions, compared with those we present below. See `?rbinom` for more information.

Let $X \sim \text{binom}(n, p)$, that is $p_X(x) := \Pr(X = x)$ given by

$$p_X(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, \dots, n$$

In order to use the inverse transform method, we need to create a function, `binom.cdf`, which calculates the distribution function $F_X(x)$.

```
binom.cdf<-function(x,n,p){  
  Fx<-0  
  for (i in 0:x){  
    Fx<-Fx+choose(n, i)*p^i*(1-p)^(n-i)  
  }  
  return(Fx)  
}
```



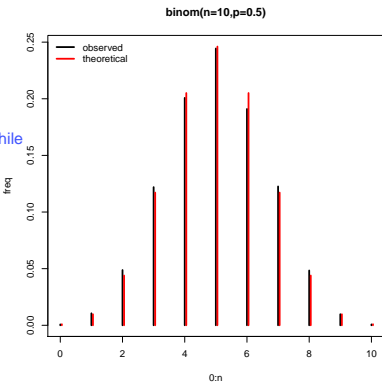
With this code we are creating the cumulative distrib function of a binomial.

The function `cdf.sim` takes as first argument a function F, which is assumed to calculate the cdf of a non-negative integer valued random variable. `cdf.sim` also uses the arguments `...` to pass parameters to the function F.

```
cdf.sim<-function(F,...){  
  X<-0  
  U<-runif(1)  
  while (F(X,...)<U){  
    X<-X+1  
  }  
  return(X) ritorna questo quando la condiz di while  
non viene soddisf cioè F(x)>U  
}
```

Simulation study:

```
N<-5000  
n<-10  
p<-0.5  
X<-rep(0,N)  
for (i in 1:N){  
  X[i]<-cdf.sim(binom.cdf,n,p)  
}
```



complic way to generate R.numb from a binomialRV

In the program above, suppose that U is close to 1. In this case we will need to calculate $F_X(x)$ for many values of x . But if we look at how `binom.cdf` is defined, each time we calculate $F_X(x)$ we recalculate $p_X(0), p_X(1), \dots$, which is rather inefficient.

We can avoid this by combining the loop in `cdf.sim`, which checks $F(X, \dots) < U$, with the loop in `binom.cdf`, which calculates F_X .

To improve the efficiency further we use a recursive formula to calculate $p_X(x)$:

$$p_X(x) = \frac{(n - x + 1)p}{x(1 - p)} p_X(x - 1)$$

```
binom.sim <- function(n,p){
  X<-0
  px<-(1-p)^n
  Fx<-px
  U<-runif(1)
  while (Fx<U) {
    X<-X+1
    px<-px*((n-X+1)*p)/
      (X*(1-p))
    Fx<-Fx+px
  }
  return(X)
}
```

```
system.time(
  for (i in 1:N){
    X[i]<-cdf.sim(binom.cdf,n,p)
  }
)
  user    system elapsed
0.523    0.003    0.524

system.time(
  for (i in 1:N){
    X[i]<-binom.sim(n,p)
  }
)
  user    system elapsed
0.135    0.001    0.137
```

Questa è la parte import per generare una binomial random variable

Sequences of Independent Trials

For random variables that are defined using sequences of independent trials (binomial, geometric and negative binomial), we have alternative methods. Given $U \sim \text{Unif}(0, 1)$ we can generate a Bernoulli rv B with parameter p using

```
# given U ~ Unif(0,1)
if (U<p) {
  B<-1
} else B<-0
```

-----Bernoulli(p)

Thus, given n and p , to generate a **binom(n, p)** rv X we can use

```
X<-0
for (i in 1:n){
  U<-runif(1)
  if (U<p) X<-X+1
}
```

(*why?*)

because a binomial is the sum of n bernulli which can take value 0 or 1, the sum of this values is just the sum for when the bernulli are equal to 1 because when they are 0 has no sence to add them up.

This is simpler and faster than the previous algorithm, however it requires a larger number of uniforms (n as opposed to 1) so that if we generate a lot of binomial rv's with this algorithm, then the "random" numbers will start repeating themselves sooner (*not a serious problem in simple simulation studies*).

Geometric Random Variable

Let $Y \sim \text{geom}(p)$, that is

$$\Pr(Y = y) = p(1 - p)^{y-1}, \quad y = 1, 2, \dots$$

(number of independent trials necessary to observe the first success)

To generate a Y , we can use a sequence of independent trials as in

`Y<-0` -----this count the nr of trials up to when we will have a succes

```
success <-FALSE
while (!success) {
  U<-runif(1)
  if (U<p) {
    success <-TRUE
  } else {
    Y<-Y+1
  }
}
```

In alternative, we can use the inverse transform method by exploiting the formula

$$F_Y(y) = \sum_{i=1}^y \Pr(Y = i) = 1 - (1 - p)^y$$

(why?)

Equation (1) corresponds to

$$Y = y \quad \text{if} \quad (1 - p)^y \leq (1 - U) < (1 - p)^{y-1}$$

that is, we can define Y by

$$Y = \min\{y : (1 - p)^y \leq 1 - U\}$$

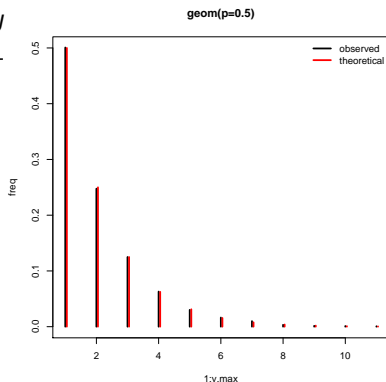
We can use the fact that $1 - U \stackrel{\mathcal{L}}{=} U$ (*why?*), the monotonicity of the logarithm and $\log(1 - p) < 0$ to get

$$Y = \min \left\{ y : y > \frac{\log U}{\log(1 - p)} \right\}$$

so that

$$Y = \lfloor \log U / \log(1 - p) \rfloor + 1$$

```
U<-runif(1)
Y<-floor(log(U)/log(1-p))+1
```



Generatin the geometric random number by using the inverse trasform method

Poisson Random Variable

Let $X \sim \text{Pois}(\lambda)$, $\lambda > 0$, that is $\Pr(X = x)$ given by

$$p_X(x) = e^{-\lambda} \frac{\lambda^x}{x!}, \quad x = 0, 1, \dots \quad (2)$$

We may implement an inverse transform algorithm similar to those used for binomial by exploiting the following **recursive formula**:

$$p_X(x) = \frac{\lambda}{x} p_X(x-1)$$

here we define the function which
for any lamda I will give us the random
number distrib a poisson

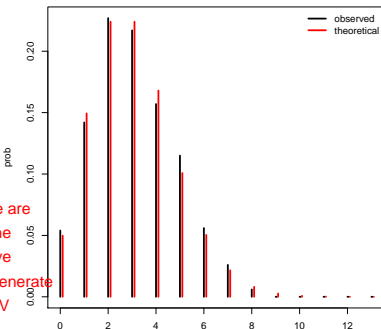
This is the recursive formula of the Poisson
random variable

```
pois.sim <- function(1){  
  X<-0  
  px<-exp(-1) ---this is the pdf when x=0  
  Fx<-px  
  U<-runif(1)  
  while (Fx<U) {  
    X<-X+1  
    px<-px*1/X  
    Fx<-Fx+px  
  }  
  return(X)  
}
```

non è 1 ma l(lambda)

here we are
using the
recursive
formula to generate
a poisson RV

Pois($\lambda=3$), N = 1000



The above algorithm successively checks whether the Poisson value is 0, then whether it is 1, then 2, and so on. Thus, the number of comparisons needed will be equal the generated value of the Poisson. Hence, on average, λ searches (*why?*).

Whereas this is fine when λ is small, it can be improved upon when λ is large. Since a Poisson rv with mean λ is most likely to take one of the two integers closest to λ (*why?*), a more efficient algorithm would first check one of these values, rather than starting at 0 and working upward.

Let $I = \lfloor \lambda \rfloor$ and use recursive formula (2) to determine $F_X(I)$. Now generate $X \sim \text{Pois}(\lambda)$ by first generating a random number U ,

$$\text{if } F_X(I) > U \text{ then } X > I, \quad \text{else } X \leq I$$

Then search downward starting from I in the case $X \leq I$ and upward starting from $I + 1$ otherwise.


```

pois.sim1 <- function(l){
  X<-floor(l)
  px<-rep(exp(-l),3*X)
  Fx<-px[1]
  for (i in 1:X){
    px[i+1]<-px[i]*l/i
    Fx<-Fx+px[i+1]
  }
  U<-runif(1)
  if (Fx<U) {
    while (Fx<U) {
      X<-X+1
      px[X+1]<-px[X]*l/X
      Fx<-Fx+px[X+1]
    }
  } else {
    while (Fx>=U) {
      Fx<-Fx-px[X+1]
      X<-X-1
    }
    X<-X+1
  }
  return(X)
}

```

The number of searches needed by this algorithm is roughly equal to

$$E[|X - \lambda|]$$

the absolute difference between the random variable X and its mean λ . For λ large, it is approximately equal to $\sqrt{\lambda}E(|Z|)$ where $Z \sim N(0, 1)$ (*why?*).

Note that $E(|Z|) \approx 0.798$ (*why?*).

Rejection Method

Suppose we have a method for simulating from a random variable having probability mass function $\{q_j, j \geq 0\}$.

We can use this as a basis for simulating from the distribution with mass function $\{p_j, j \geq 0\}$ by first simulating a rv Y where $\Pr(Y = j) = q_j$ and then accepting this simulated value with a probability proportional to p_Y/q_Y .

Let $c > 0$ such that

$$p_j/q_j \leq c \quad \text{for all } j \text{ such that } p_j > 0$$

The algorithm for simulating X where $\Pr(X = j) = p_j$ is as follows:

Step 1: Generate Y where $\Pr(Y = j) = q_j$.

Step 2: Generate $U \sim \text{Unif}(0, 1)$.

Step 3: If $U < p_Y/cq_Y$, set $X = Y$ and stop. Otherwise return to Step 1.

The number of iteration of the algorithm needed to obtain X is a geometric random variable with mean c (i.e. $\Pr(\text{Acceptance}) = 1/c$).

Example 4f page 59, Ross (2006)

$p = (0.11, 0.12, 0.09, 0.08, 0.12, 0.10, 0.09, 0.09, 0.10, 0.10)$

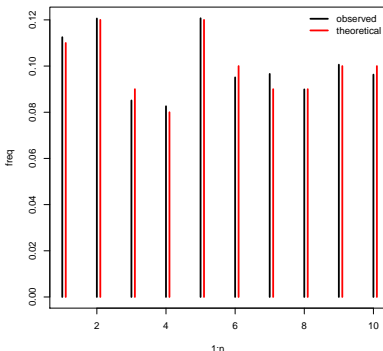
Y discrete uniform on $1, \dots, n$, i.e. $q_j = 0.1$.

$$c = \max_{j=1, \dots, n} p_j / q_j = 1.2$$

```
prob.accept<-0
iter<-0
U<-1
while (U>=prob.accept){
  Y<-floor(n*runif(1))+1
  prob.accept<-p[Y]/(c*q[Y])
  U<-runif(1)
  iter<-iter+1
}
X<-Y

mean(num.iter)
[1] 1.1954
```

Example 4f, Ross (2006)



Exercises

- Chapter 4, Ross (2006)
Ex 1, Ex 3, Ex 4, Ex 6, Ex 7, Ex 10, Ex 11, Ex 12
- Chapter 18, Owen, et al. (2007)
Ex 3, Ex 4, Ex 11, Ex 12

Resources

- BOOKS

- Owen J., Maillardet R. and Robinson A. (2009).
Introduction to Scientific Programming and Simulation Using R.
Chapman & Hall/CRC.
- Ross, S. (2006).
Simulation. 4th edn. Academic Press.

- WEB

- R software:
<http://www.r-project.org/>
- Owen, et al. (2009): <http://www.ms.unimelb.edu.au/spuRs/>