DCS 307: Simulation–Random Number Generator Streams

Eric Zhao

2023-02-01

Prep

```r
#function built
myArr1<-function(){rexp(1, 1)}
myArr2<-function(){rexp(1, 11/10)}
mySvc1<-function(){rexp(1, 10/9)}

#simulation
output11<-ssq(maxArrivals = 20, interarrivalFcn = myArr1, serviceFcn = mySvc1, seed = 8675309, saveAllS

output12<-ssq(maxArrivals = 20, interarrivalFcn = myArr2, serviceFcn = mySvc1, seed = 8675309, saveAllS
```

Results

```r
print(output11$interarrivalTimes)
```

```
##  [1] 1.66216344 1.22326524 0.34609182 0.67646507 0.02190231 0.49897147
##  [7] 0.14296101 0.43171886 3.00313699 2.34623787 0.63278097 1.85741936
## [13] 2.01721600 0.35873818 3.01868395 1.47366193 0.52454877 0.97513225
## [19] 0.93961356 1.58288361
```

```r
print(output12$interarrivalTimes)
```

```
##  [1] 1.51105767 1.11205931 0.31462893 0.61496825 0.01991119 0.45361043
##  [7] 0.12996455 0.57621243 0.10845545 0.30812606 0.64700100 2.13294351
## [13] 0.57525543 0.20751338 1.68856306 0.40275308 0.32612562 0.96832497
## [19] 0.87779602 1.33969266
```

```r
print(output11$serviceTimes)
```

```
##  [1] 0.6906071 0.2777054 0.1499705 0.6604378 0.5704503 0.1073709 0.3050448
##  [8] 0.6405310 1.4994352 0.2054382 0.8472361 0.3987256 0.9586417 0.8690181
## [15] 0.9447208 0.4549870 0.3656762 0.7658315 0.2638980 0.4963474
```

```r
print(output12$serviceTimes)
```

```
##  [1] 0.6906071 0.2777054 0.1499705 0.6604378 0.3885470 2.7028233 1.4994352
##  [8] 0.8472361 1.8154944 2.7168156 0.9447208 0.4549870 0.8776190 0.3656762
## [15] 0.8456522 0.7658315 1.4245953 0.2638980 0.7124820 0.4963474
```

Q1

The first ten service times from first run are:

```
output11$serviceTimes[1:10]
```

```
##  [1] 0.6906071 0.2777054 0.1499705 0.6604378 0.5704503 0.1073709 0.3050448
##  [8] 0.6405310 1.4994352 0.2054382
```

Q2
The first ten service times from second run are:

```
output12$serviceTimes[1:10]
```

```
##  [1] 0.6906071 0.2777054 0.1499705 0.6604378 0.3885470 2.7028233 1.4994352
##  [8] 0.8472361 1.8154944 2.7168156
```

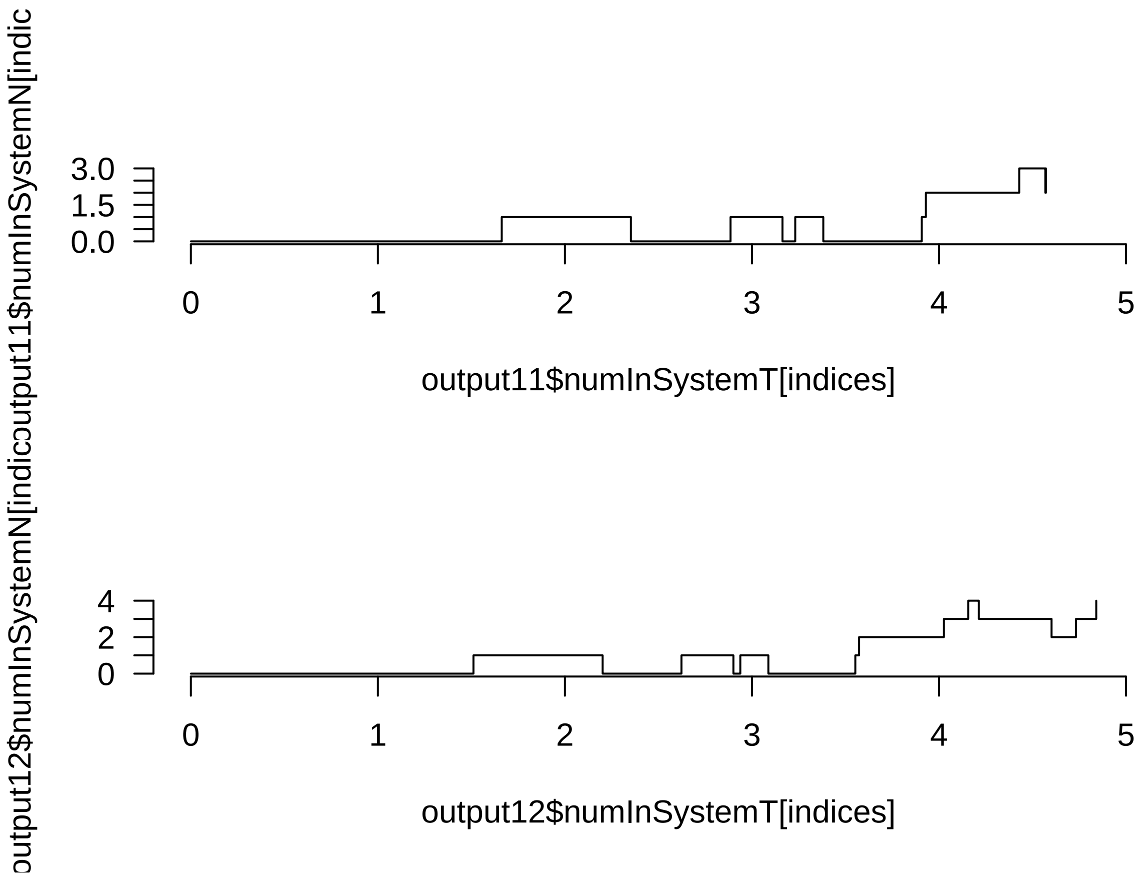
Q3
The divergence starts from the fifth customer and the service times for each customer are: 0.5704503 and 0.3885470.

Skylineplots

```
par(mfrow=c(2,1))

indices<-seq_along(output11$numInSystemT[output11$numInSystemT<=5])
plot(output11$numInSystemT[indices], output11$numInSystemN[indices], type="s", xlim=c(0, 5), bty="n", la

indices<-seq_along(output12$numInSystemT[output12$numInSystemT<=5])
plot(output12$numInSystemT[indices], output12$numInSystemN[indices], type="s", xlim=c(0, 5), bty="n", la
```



Q4
According to the skyplots, we know that there are two arrivals after the fourth arrival occur before the fourth customer's service is finished.

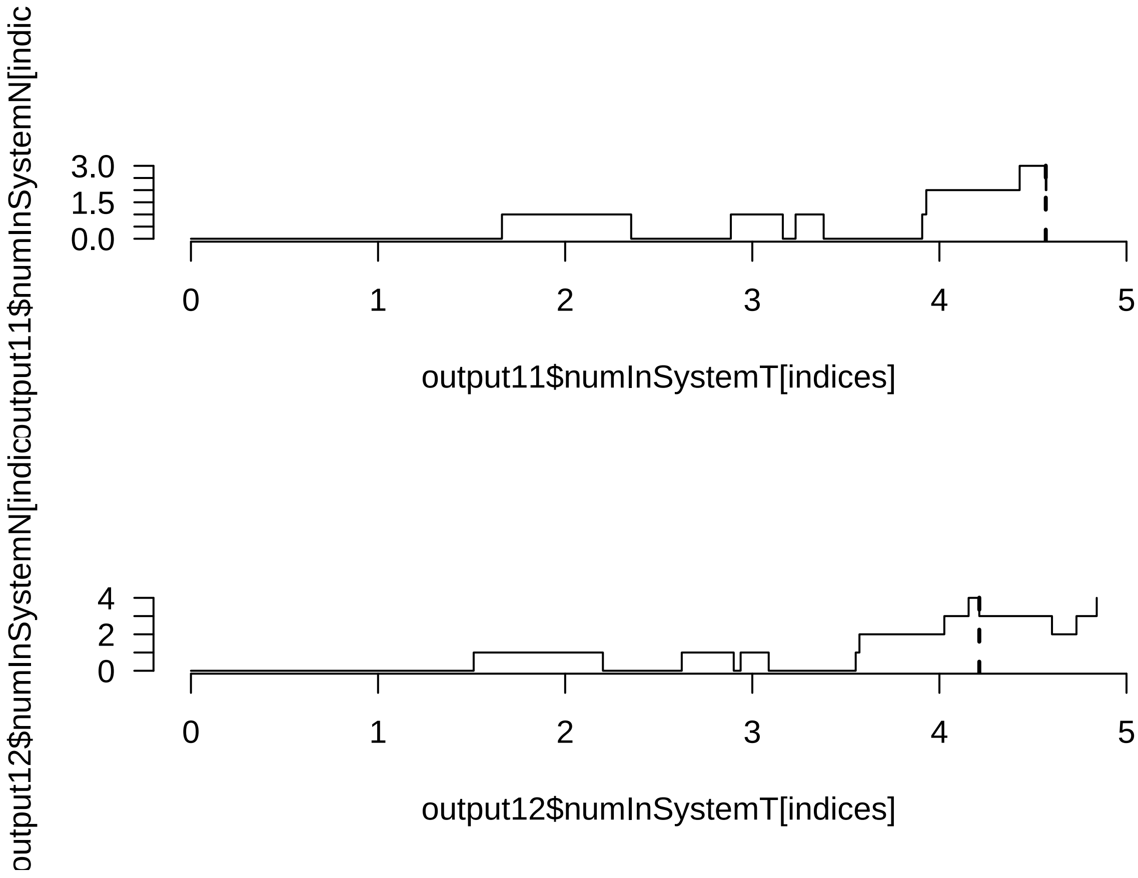
Q5
In the second run, we can observe that there are three more arrivals after the fourth arrival before that fourth customer complete the services.

Add 4th completion time

```
par(mfrow=c(2,1))

arrivalTimes<-cumsum(output11$interarrivalTimes)
completitionTimes<-arrivalTimes+output11$sojournTimes
indices<-seq_along(output11$numInSystemT[output11$numInSystemT<=5])
plot(output11$numInSystemT[indices], output11$numInSystemN[indices], type="s", xlim=c(0, 5), bty="n", la
abline(v=completitionTimes[4], lty=2, lwd = 2)

arrivalTimes<-cumsum(output12$interarrivalTimes)
completitionTimes<-arrivalTimes+output12$sojournTimes
indices<-seq_along(output12$numInSystemT[output12$numInSystemT<=5])
plot(output12$numInSystemT[indices], output12$numInSystemN[indices], type="s", xlim=c(0, 5), bty="n", la
abline(v=completitionTimes[4], lty=2, lwd = 2)
```



RNG__Streams

```r
set.seed(8675309)
#The first three numbers (rate = 1) are interarrivals, the second three (rate = 10/9) are service times
rexp(1, 1)
```

```
## [1] 1.662163
```

```r
rexp(1, 1)
```

```
## [1] 1.223265
```

```r
rexp(1, 1)
```

```
## [1] 0.7673412
```

```r
rexp(1, 10/9)
```

```
## [1] 0.3114826
```

```r
rexp(1, 10/9)
```

```
## [1] 0.2777054
```

```r
rexp(1, 10/9)
```

```
## [1] 0.6088186
```

Q6
The arrival times(cumulative interarrival times) are 1.662, 2.885 and 3.653.

Q7
The service times are 0.311, 0.278 and 0.609.

```r
set.seed(8675309)
#The first three numbers (rate = 1) are interarrivals, the second three (rate = 10/9) are service times
rexp(1, 1)
```

```
## [1] 1.662163
```

```r
rexp(1, 10/9)
```

```
## [1] 1.100939
```

```r
rexp(1, 1)
```

```
## [1] 0.7673412
```

```r
rexp(1, 10/9)
```

```
## [1] 0.3114826
```

```r
rexp(1, 1)
```

```
## [1] 0.3085615
```

```r
rexp(1, 10/9)
```

```
## [1] 0.6088186
```

Q8
The arrival times are 1.6622, 2.4295 and 2.7381.

Q9
The service times are 1.1009, 0.3115 and 0.6088.

Using vexp() with stream option

```
set.seed(8675309)

#repeat the exercise using vexp rather than rexp, and use stream 1 for interarrivals and stream 2 for s
vexp(1, 1, stream = 1)
```

```
## [1] 0.2334595
```

```
vexp(1, 1, stream = 1)
```

```
## [1] 1.07828
```

```
vexp(1, 1, stream = 1)
```

```
## [1] 0.7855611
```

```
vexp(1, 10/9, stream = 2)
```

```
## [1] 0.0305915
```

```
vexp(1, 10/9, stream = 2)
```

```
## [1] 0.1224158
```

```
vexp(1, 10/9, stream = 2)
```

```
## [1] 1.42499
```

Q10
The arrival times are 0.2335, 1.3117 and 2.0973.

Q11
The service times are 0.0306, 0.1224 and 1.4250.

Interleave the time generation

```
set.seed(8675309)

vexp(1, 1, stream = 1)
```

```
## [1] 0.2334595
```

```
vexp(1, 10/9, stream = 2)
```

```
## [1] 0.0305915
```

```
vexp(1, 1, stream = 1)
```

```
## [1] 1.07828
```

```r
vexp(1, 10/9, stream = 2)
```

```
## [1] 0.1224158
```

```r
vexp(1, 1, stream = 1)
```

```
## [1] 0.7855611
```

```r
vexp(1, 10/9, stream = 2)
```

```
## [1] 1.42499
```

Q12
The arrival times are 0.23346, 1.31174 and 2.0973.

Q13
The service times are 0.0306, 0.1224 and 1.4250.

Return the simulation

#### Redefine process functions with streams

```r
myArr1<-function(){vexp(1, 1, stream = 1)}
myArr2<-function(){vexp(1, 11/10, stream = 1)}
mySvc1<-function(){vexp(1, 10/9, stream = 2)}
```

Simulation again

```r
#simulation with different arrival processes
output11<-ssq(maxArrivals = 20, interarrivalFcn = myArr1, serviceFcn = mySvc1, seed = 8675309, saveAllS

output12<-ssq(maxArrivals = 20, interarrivalFcn = myArr2, serviceFcn = mySvc1, seed = 8675309, saveAllS
```

Interarrival and service time

```r
output11$interarrivalTimes
```

```
##  [1] 0.23345951 1.07828020 0.78556111 0.35311660 0.90501100 2.42985756
##  [7] 0.47442318 0.01652069 1.50555953 0.49710849 0.74338402 0.35126435
## [13] 2.16909893 2.85546077 0.02080979 0.70771366 1.22571701 0.34982305
## [19] 0.64498544 0.20104739
```

```r
output11$serviceTimes
```

```
##  [1] 0.03059150 0.12241584 1.42498986 4.55083122 2.49443812 0.21221607
##  [7] 0.09939047 0.52513160 1.35447248 0.14203867 0.53753282 0.65148306
## [13] 2.43953310 0.42059015 0.80551067 0.25625162 0.04951228 0.02394590
## [19] 1.96568156 0.84102916
```

```r
output12$interarrivalTimes
```

```
##  [1] 0.21223592 0.98025473 0.71414646 0.32101509 0.82273727 2.20896142
##  [7] 0.43129380 0.01501881 1.36869049 0.45191681 0.67580365 0.31933123
## [13] 1.97190812 2.59587343 0.01891799 0.64337606 1.11428819 0.31802095
## [19] 0.58635040 0.18277035
```

```
output12$serviceTimes
```

```
##  [1] 0.03059150 0.12241584 1.42498986 4.55083122 2.49443812 0.21221607
##  [7] 0.09939047 0.52513160 1.35447248 0.14203867 0.53753282 0.65148306
## [13] 2.43953310 0.42059015 0.80551067 0.25625162 0.04951228 0.02394590
## [19] 1.96568156 0.84102916
```

Q14

```
sum(output11$interarrivalTimes-output12$interarrivalTimes)
```

```
## [1] 1.595291
```

Q15

```
sum(output11$serviceTimes-output12$serviceTimess)
```

```
## [1] 0
```

Rerun simulation

```
mySvc2<-function(){vexp(1, 11/10, stream = 2)}
#simulation with different service processes
output11<-ssq(maxArrivals = 20, interarrivalFcn = myArr1, serviceFcn = mySvc1, seed = 8675309, saveAllS

output12<-ssq(maxArrivals = 20, interarrivalFcn = myArr1, serviceFcn = mySvc2, seed = 8675309, saveAllS
```

Same interarrival and different service time processes

```
output11$interarrivalTimes
```

```
##  [1] 0.23345951 1.07828020 0.78556111 0.35311660 0.90501100 2.42985756
##  [7] 0.47442318 0.01652069 1.50555953 0.49710849 0.74338402 0.35126435
## [13] 2.16909893 2.85546077 0.02080979 0.70771366 1.22571701 0.34982305
## [19] 0.64498544 0.20104739
```

```
output11$serviceTimes
```

```
##  [1] 0.03059150 0.12241584 1.42498986 4.55083122 2.49443812 0.21221607
##  [7] 0.09939047 0.52513160 1.35447248 0.14203867 0.53753282 0.65148306
## [13] 2.43953310 0.42059015 0.80551067 0.25625162 0.04951228 0.02394590
## [19] 1.96568156 0.84102916
```

```
output12$interarrivalTimes
```

```
##  [1] 0.23345951 1.07828020 0.78556111 0.35311660 0.90501100 2.42985756
##  [7] 0.47442318 0.01652069 1.50555953 0.49710849 0.74338402 0.35126435
## [13] 2.16909893 2.85546077 0.02080979 0.70771366 1.22571701 0.34982305
## [19] 0.64498544 0.20104739
```

```
output12$serviceTimes
```

```
##  [1] 0.03090051 0.12365236 1.43938370 4.59679921 2.51963447 0.21435967
##  [7] 0.10039441 0.53043596 1.36815402 0.14347341 0.54296244 0.65806369
## [13] 2.46417485 0.42483854 0.81364714 0.25884002 0.05001240 0.02418777
## [19] 1.98553692 0.84952440
```

Reflection / Investigation:

- **What is the default RNG used by Python? What is its period? List your source(s).**

The basic function random(), generates a random float uniformly in the half-open range $0.0 \leq X < 1.0$. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937} - 1$. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

Source:https://docs.python.org/3/library/random.html#

- **What is the default RNG used by R? What is its period? List your source(s).**

Same as Python, the default RNG used by R is also "Mersenne-Twister" which has a period of $2^{19937} - 1$.

Source:https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html

- **What additional RNGs are available in Python, and how do you set up code to use them? List your source(s).**

In the NumPy package, there are more additional RNGs available to use. The random values produced by generators originate in so-called BitGenerators. The default in NumPy is actually PCG-64 and other options include an updated version PCG-64 DXSM, MT19937(which is the python standard RNG), Philox, and SFC64.

Sources: https://numpy.org/doc/stable/reference/random/generator.html https://numpy.org/doc/stable/reference/random/bit_generators/index.html https://www.pcg-random.org/

- **What additional RNGs are available in R, and how do you set up code to use them? List your source(s).**

The other available RNGs includes–"Wichmann-Hill", "Marsaglia-Multicarry", "Super-Duper", "Knuth-TAOCP-2002", "Knuth-TAOCP", "L'Ecuyer-CMRG" and "user-supplied". You can choose to use these additional RNGs by setting argument `kind= NULL` in the function. If kind is a character string, set the R's RNG to the kind desired otherwise it use default "Mersenne-Twister" generator.

Source:https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html

- **Given the article posted to Lyceum last class, which of the RNGs you mention in 1-4 above are presented in the article, and how does the article describe and/or rank their relative "goodness"? Include pointers to specific passages in the article.**

In the article, the following RNGs are presented: "Mersenne-Twister"(P17), PCG(P11), "Wichmann-Hill"(P8), "Marsaglia-Multicarry"(P8). However, only MT19937 and PCG-32 were actually tested and ranked.

The final ranking of all RNGs are based on rankings from both blind/statistical tests and graphical tests. For the blind test, The target is to find evidence against a specific null hypothesis–""the sequence to be tested is random".They take the count of the number of tests for which the null hypothesis is not rejected for a PRNG as its merit for randomness.(P24-26) While the graphical test, the numbers are plotted in a graph to see whether any visible pattern exists or not. As period length a PRNG is expected to be very large, so, for every graphical test also, all numbers of a period can not be used; rather a set of numbers need to be generated based on some seed.(P29-30)

Based on test results, MT19937-64 holds 4th place, PCG-32 holds 5th place and MT19937-32 holds 6th place(P41-46).