Thoughts on Random Number Generators

frank.jung@marlo.com.au

06 Dec 2018

One of the first exercises given to me as a mathematics student was to write a random number generator (RNG). This turned out to be not so easy. Test sequences cycled quickly or were too predictable or were not evenly distributed. Typically when we talk of RNG's we are describing *pseudorandom* number generators. Nowadays, we have a many programs that will generate *pseudorandom* numbers.

Where are random numbers used? As a developer they were rarely required. Recently, however we've seen them appear in more and more places - it seems they are everywhere!

In DevOps, I've used RNG's for creating message payloads of arbitrary size, and for file or directory names. These values are often created using scripts written in bash. This first article will explore three simple RNG's that can be run from bash. It is not an exhaustive list, as there are others such as jot that is also easy to use from bash. However, the three described here are likely to already be installed on your Linux box.

The three RNG's being evaluated here are:

- Bash `RANDOM` variable
- Awk `rand()` function
- `/dev/urandom` device

A word of caution: None of these tools are suitable for generating passwords or for cryptography.

Source and test data for this article can be found here.

In future articles we will take a closer look at random values used in testing code and how they can be used for model simulation in statistics.

Testing Numeric Sequences

This discussion will not test the programs for randomness. Instead, we are going to evaluate a short list of `1,000` values generated by each RNG. To ease comparison the values will be scaled to the range `[0,1)`. They are rounded to two decimal places and the list of values is formatted as `0.nn`.

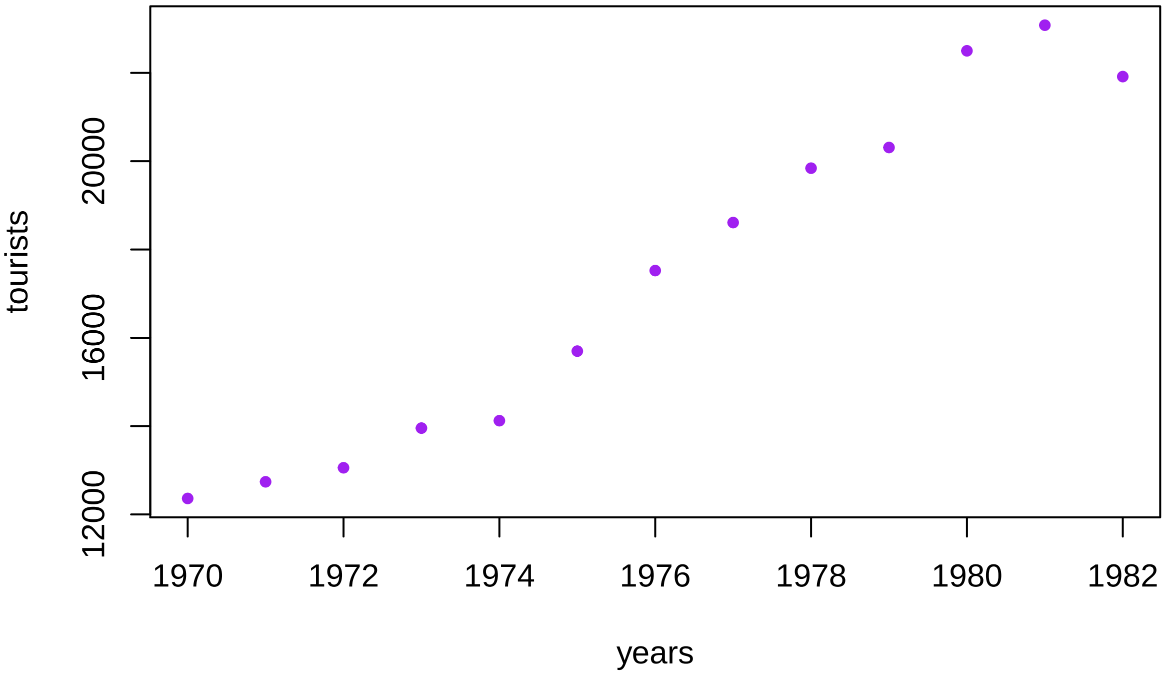
There are many tests that can be applied to sequences to check for randomness. Here we are only looking at one: the Bartels Rank test. It's limitations and those of other tests are described here. I've choosen this test as it is relatively easy to understand and interpret. Rather than comparing the magnitude of each observation with its preceding sample, Bartels Rank test, ranks all the samples from the smallest to the largest. The rank is the corresponding sequential number in the list of possibilities. Under the null hypothesis of randomness, any rank arrangement from all possibilities should be equiprobable. Bartels Rank test is also suitable for small samples.

To get a feel of the test: consider two of the vignettes provided by the R package, randtests.

Example 5.1 in Gibbons and Chakraborti (2003), p.98.

Annual data on total number of tourists to the United States for 1970-1982.

```
years <- 1970:1982
tourists <- c(12362, 12739, 13057, 13955, 14123, 15698, 17523, 18610, 19842,
20310, 22500, 23080, 21916)
plot(years, tourists, pch = 20, col = 'purple')
```



```r
bartels.rank.test(tourists, alternative = "left.sided", pvalue = "beta")
```

```
##
##  Bartels Ratio Test
##
## data:  tourists
## statistic = -3.6453, n = 13, p-value = 1.21e-08
## alternative hypothesis: trend
```

What this tells us about the sample data is there is strong evidence against the null hypothesis of randomness. It does favour the alternative hypothesis of a trend.

Example in Bartels (1982)

Changes in stock levels for 1968-1969 to 1977-1978 (in $A million), deflated by the Australian gross domestic product (GDP) price index (base 1966-1967).

```r
x <- c(528, 348, 264, -20, -167, 575, 410, -4, 430, -122)
bartels.rank.test(x, pvalue = 'beta')
```

```
##
##  Bartels Ratio Test
##
## data:  x
## statistic = 0.083357, n = 10, p-value = 0.9379
## alternative hypothesis: nonrandomness
```

Here, the sample data provides weak evidence against the null hypothesis of randomness. Which does not support alternative hypothesis of non-random data.

(For a simple guide on how to interpret the p-value, see this)

Random Number Generators that can be used in Bash scripts

The following sections describe 3 RNG's. It includes a small description of typical use of the RNG. Then a list of 1,000 values is produced and analysed using Bartels Rank test.

bash RANDOM variable

Bash, provides the shell variable $RANDOM. It will generate a pseudorandom signed 16-bit integer between 0 and 32767.

`RANDOM` is easy to use is bash:

```
$ RANDOM=123; echo $RANDOM
2877
```

Here, we have seeded `RANDOM` with a value. Seeding a random variable will return the same sequence of numbers. This is required for results to be reproducible.

To generate a random integer between `START` and `END` use:

```
RANGE=$(( END - START + 1))
echo $(( (RANDOM % RANGE) + START ))
```

Where `START` < `END` are non-negative numbers.

For example, to simulate 10 rolls of a 6 sided dice:

```
RANDOM=123 # an optional seed
START=1
END=6
RANGE=$(( END - START + 1 ))

for (( i=0; i<10; i++))
do
   echo -n $(( (RANDOM % RANGE) + START )) " "
done
```

Which yields:

```
  4  4  4  5  2  3  2  6  3  2
```

Checking Sequences from RANDOM

Prepare the test data:

```
RANDOM=123
for i in $(seq 1000); do
   printf "%0.2f\n" $(bc <<< "scale=2; $RANDOM/32768");
done > ./bash.random
```

```
bashText <- readLines(con <- file(paste0(getwd(), '/bash.random')))
close(con)
bashRandom <- as.numeric(bashText)
```

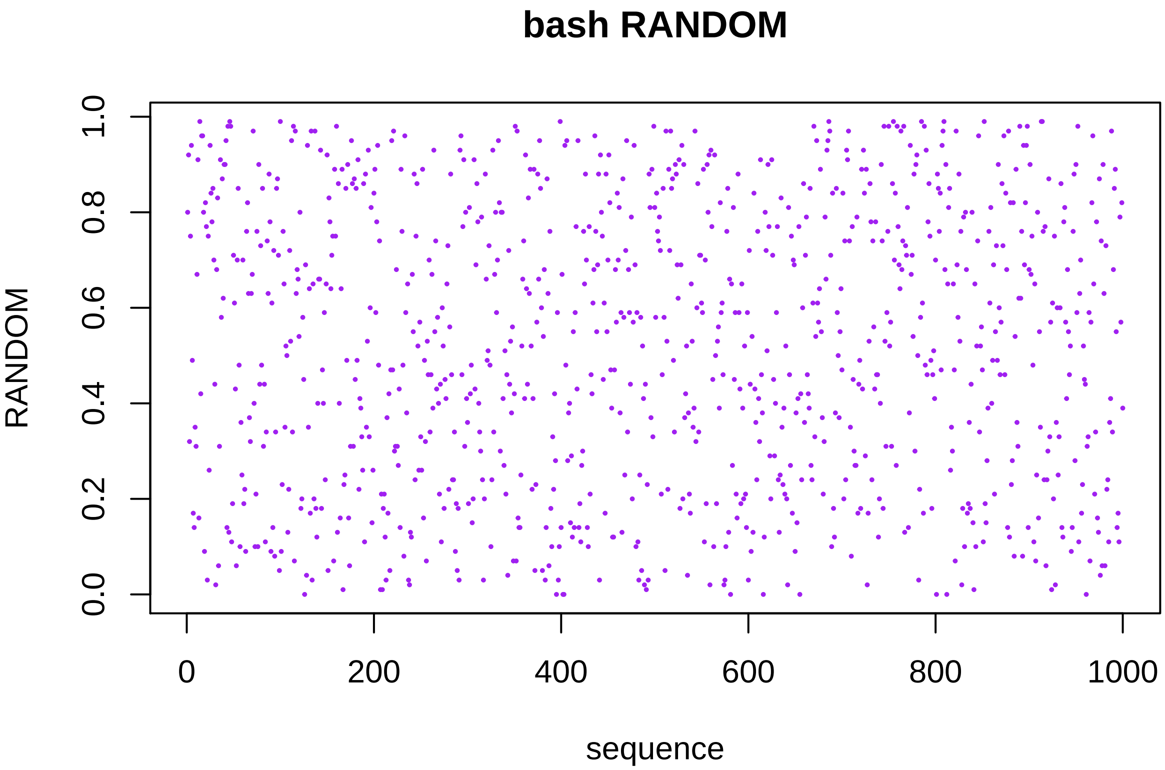
Show first 10 values:

```
head(bashRandom, n = 10)
```

```
##  [1] 0.80 0.92 0.32 0.75 0.94 0.49 0.17 0.14 0.35 0.31
```

Plot the sequence vs value from RNG:

```
plot(x = seq(1,1000), y = bashRandom,
     xlab = 'sequence', ylab = 'RANDOM',
     title('bash RANDOM'),
     pch = 20, cex = 0.3, col = 'purple')
```



Run Bartels Rank test:

```
bartels.rank.test(bashRandom, 'two.sided', pvalue = 'beta')
```

```
##
##  Bartels Ratio Test
##
## data:  bashRandom
## statistic = 0.51645, n = 1000, p-value = 0.6058
## alternative hypothesis: nonrandomness
```

Result

With a p-value > 0.05 there is weak evidence against the null hypothesis of randomness.

awk rand()

Here we are using `rand` function from GNU Awk. This function generates random numbers between 0 and 1.

```
# seeded with 123
$ echo | awk -e 'BEGIN {srand(123)} {print rand();}'
0.517322
```

If you don't specify a seed in `srand()` it will return the same results.

You can also generate random integers in a range.

For example, to simulate 10 rolls of a 6 sided dice:

```
$ echo | awk 'BEGIN {srand(123)} {for (i=0; i<10; i++) printf("%d ", int(6*rand()+1));}'
4 4 6 3 2 4 1 6 2 6
```

Checking Sequences from rand()

Prepare the test data:

```
seq 1000 | awk -e 'BEGIN {srand(123)} {printf("%0.2f\n",rand());}' > awk.random
```

```
awkText <- readLines(con <- file(paste0(getwd(), '/awk.random')))
close(con)
awkRandom <- as.numeric(awkText)
```

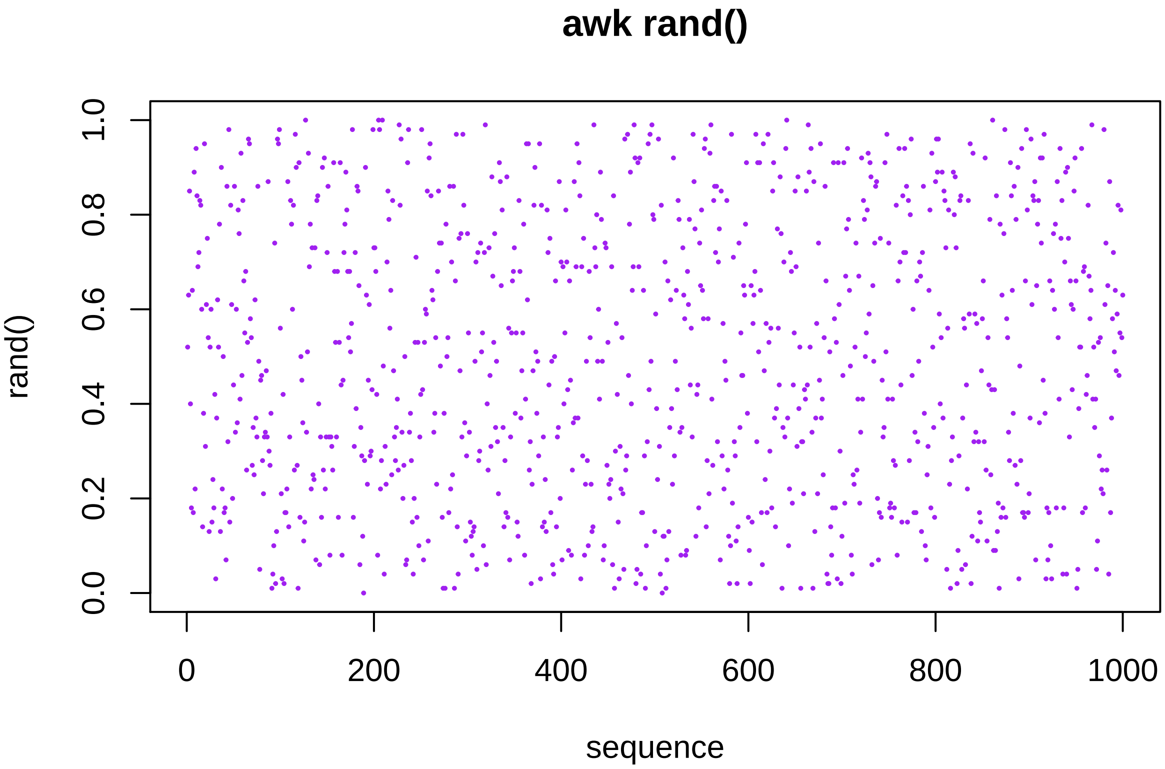
Show first 10 values:

```
head(awkRandom, n = 10)
```

```
##  [1] 0.52 0.63 0.85 0.40 0.18 0.64 0.17 0.89 0.22 0.94
```

Plot the sequence vs value from RNG:

```
plot(x = seq(1,1000), y = awkRandom,
     xlab = 'sequence', ylab = 'rand()',
     title('awk rand()'),
     pch = 20, cex = 0.3, col = 'purple')
```



Run Bartels Rank test:

```
bartels.rank.test(awkRandom, 'two.sided', pvalue = 'beta')
```

```
##
##  Bartels Ratio Test
##
```

```
## data:  awkRandom
## statistic = -1.8569, n = 1000, p-value = 0.06329
## alternative hypothesis: nonrandomness
```

Result

With a p-value > 0.05 there is weak evidence against the null hypothesis of randomness.

urandom device

The final tool we will look at is the /dev/urandom device. The device provides an interface to the kernels random number generator. This is a useful tool as it can generate a wide variety of data types.

For example print a list of unsigned decimal using od(1):

```
$ seq 5 | xargs -I -- od -vAn -N4 -tu4 /dev/urandom
 3287622437
 3753974622
 2103329150
 1495823055
 2218999858
```

It can also be used to source random hexadecimal values:

```
$ seq 5 | xargs -I -- od -vAn -N4 -tx4 /dev/urandom
  e480673d
  bc9f1aa8
  4ece5c41
  cba1e97c
  310001c4
```

Or it can generate a block of 64 random alphanumeric bytes using:

```
$ cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 64 | head -n 5
  6mBG7HU2KxwBmRR2CNkEwe2Cpx0zuKpGqfUK98GqrthpVQTVTMbSIFKWF6UbXWv2
  vx6IxPHiTW1VaJZgmzug6kmuXP9d0ZYMLSKKNYmLoZEc4gcZdoqUrxmIkoagjPcc
  FZHqk0RrNnUMVU0eX5IQGwtX6mfDRtUEAuBYFGIJi61RQ3ISdnta7Dk9gTymYpXo
  ggThurP1Q51k4K0Mi1YApgKYqb9qdNiSWQejMJBI00Pu0fQhJe0sUZQrNEekeCEN
  ETLFeRanuBBjVpr0Bm5lcjezn8eyuzNQjM9scgRK2BziIMlQwtkUA4T3MZAUZFk6
```

Checking Sequences from urandom

Prepare the test data:

```
cat /dev/urandom | tr -dc  '0-9' | fold -w2 | \
   awk '{printf("0.%02d\n",$1)}' | head -1000 > urandom.random
```

```
urText <- readLines(con <- file(paste0(getwd(), '/urandom.random')))
close(con)
urRandom <- as.numeric(urText)
```

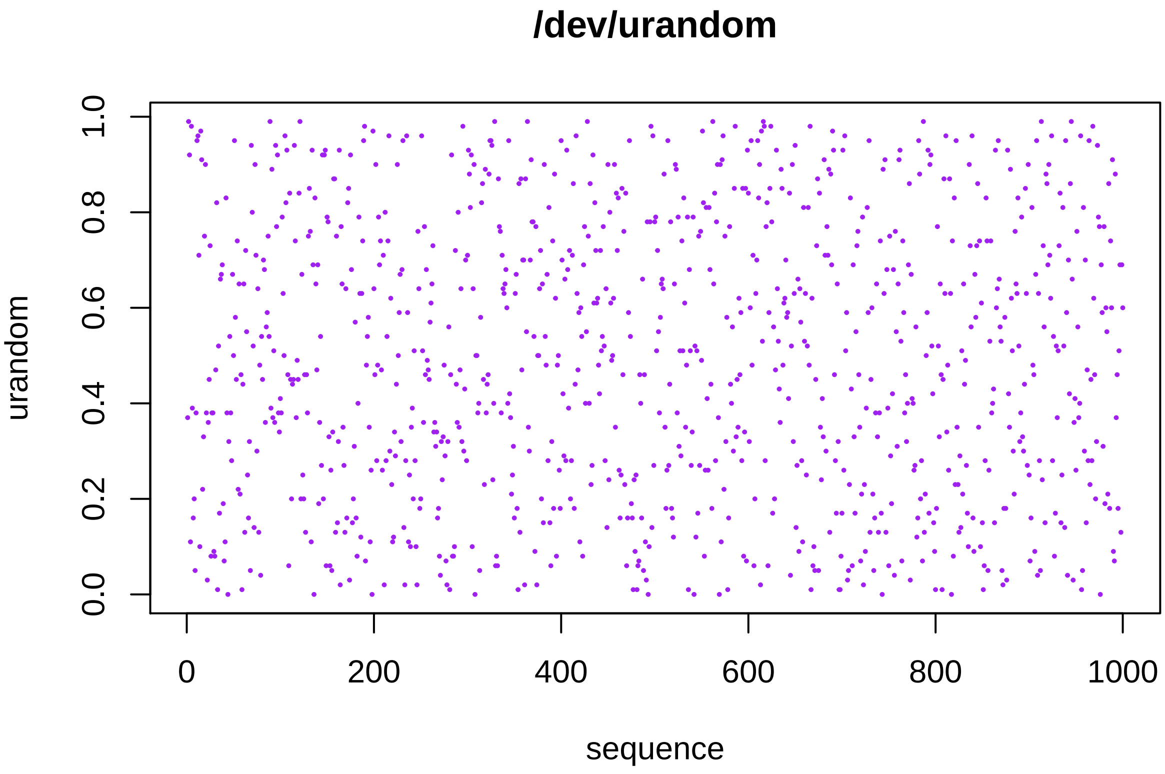
Show first 10 values:

```
head(urRandom, n = 10)
```

```
##  [1] 0.37 0.99 0.92 0.11 0.98 0.39 0.16 0.20 0.05 0.38
```

Plot the sequence vs value from RNG:

```
plot(x = seq(1,1000), y = urRandom,
     xlab = 'sequence', ylab = 'urandom',
     title('/dev/urandom'),
     pch = 20, cex = 0.3, col = 'purple')
```



Run Bartels Rank test:

```
bartels.rank.test(urRandom, 'two.sided', pvalue = 'beta')
```

```
##
##  Bartels Ratio Test
##
## data:  urRandom
## statistic = 0.61185, n = 1000, p-value = 0.5409
## alternative hypothesis: nonrandomness
```

Result

With a p-value > 0.05 there is weak evidence against the null hypothesis of randomness.

Some final thoughts

Of the tools explored, `urandom` is the most versatile. So it has broader application. The down side is, its results are not easily reproducible and issues have been identified by a study by Gutterman, Zvi; Pinkas, Benny; Reinman, Tzachy (2006-03-06) for the Linux kernel version 2.6.10.

Personally, this has been a useful learning exercise. For one it showed the limitations in generating and testing for (psuedo)random sequences. Indeed, Aaron Roth, has suggested:

> As others have mentioned, a fixed sequence is a deterministic object, but you can still meaningfully talk about how "random" it is using Kolmogorov Complexity: (Kolmogorov complexity).

> Intuitively, a Kolmogorov random object is one that cannot be compressed. Sequences that are drawn from truly random sources are Kolmogorov random with extremely high probability.
>
> Unfortunately, it is not possible to compute the Kolmogorov complexity of sequences in general (it is an undecidable property of strings). However, you can still estimate it simply by trying to compress the sequence. Run it through a Zip compression engine, or anything else. If the algorithm succeeds in achieving significant compression, then this certifies that the sequence is -not- Kolmogorov random, and hence very likely was not drawn from a random source. If the compression fails, of course, it doesn't prove that the sequence has high Kolmogorov complexity (since you are just using a heuristic algorithm, not the optimal (undecidable) compression). But at least you can certify the answer in one direction.

In light of this knowledge, lets run the compression tests for the sequences above:

```
$ ls -l *.random
-rw-r--r-- 1 frank frank 5000 2018-12-06 14:57 awk.random
-rw-r--r-- 1 frank frank 5000 2018-12-06 14:57 bash.random
-rw-r--r-- 1 frank frank 5000 2018-12-06 14:59 urandom.random
```

Compress using zip:

```
$ for z in 'awk' 'bash' 'urandom'; do zip $z $z.random; done
  adding: awk.random (deflated 72%)
  adding: bash.random (deflated 72%)
  adding: urandom.random (deflated 72%)
```

Compare this to non-random (trend) data:

```
$ for i in $(seq 1000); do printf "0.%02d\n" $(( i % 100 )) ; done > test.nonrandom

$ zip test test.nonrandom
  adding: test.nonrandom (deflated 96%)
```

Or just constant data:

```
$ for i in $(seq 1000); do echo 0.00 ; done > test.constant

$ zip constant test.constant
  adding: test.constant (deflated 99%)
```

So zipping is a good, if rough, proxy for a measure of randomness.

Stay tuned for part two to discover how random data can be used in testing code.