Post 1: Random Number Generators and Their Applications

Jesse Gao

October 28, 2017

Sections:

Introduction: RNG and simulations

TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
10/25/01 © 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

Random numbers have a wide variety of applications in the modern age. They play a big part in field of computer simulation statistical model testing. The concept of random numbers is simple: a number that is drawn from a set of possible values, each of which is equally probable. However, making random generators and making sure the generated numbers are random involves surprisingly deep mathematical research. First of all, how would one generate a random number on a computer, if the programming is predetermined? How do we know an RNG produces statistically random numbers if the numbers, by definition, cannot be predicted? Producing proper random numbers is key to running simulations, especially when uniform distributions are integral. Using RNGs for input values is an easy way to simulate events that seem random, such as a dice toss or cards drawn from a deck.

Generating a random number

There are numerous ways of generating random numbers. Flipping a coin can be thought of as generating a random 0 or 1. The Linux OS generates random numbers using the user's key presses and mouse movements for entropy, or "randomness," in other words. However both are not truly random, in the sense that given enough information, the outcomes can be predicted. A coin flip's outcome is a function of how one flicks the coin, how long its in the air, etc. The Linux RNG can be predicted by knowing the internal state of the machine. For the purposes of statistical analysis and simulation, a standard pseudo random number generator is good enough. The pRNG used in R generates output based on a seed, so the same seed will produce the same output. The outputs of the pRNG are statistically indistinguishable from true randomness, so each possible output is equally likely. We can verify this with a sampling example.

```
# Just make 100000 samples and see if each number (1-10) appears about the same amount of times.
# Run this for yourself if you want.
samples = sample(x = 1:10, 100000, replace = TRUE)
table(samples)
```

```
## samples
##     1     2     3     4     5     6     7     8     9    10 
## 10084  9917  9908 10121  9939 10043  9939 10089  9966  9994
```

Making sure an RNG performs accordingly requires a series of statistical tests which include:

- Frequency Test: Monobit
- Frequency Test: Block
- Runs Test
- Test for the Longest Runs of Ones in a Block
- Binary Matrix Rank Test
- Discrete Fourier Transform (Spectral Test) and many more, which are recommended in this article which discusses testing a random number generator service RNGs used for statistical analysis we do probably don't need to satisfy all of the rigorous tests because sequences only need to be statistically random, which means it contains no recognizable patterns or regularities. Essentially, the random numbers we require just need to be more or less unpredictable with only the previously generated numbers as given information. Pi is considered to be a good example of a statistically random number generator according to this article because there are no currently provable patterns or regularities amongst its digits. The article also mentions pi performing well on all the randomness tests performed. More elaborate pRNGs exist for the purpose of providing very unpredictable random numbers for cryptographic purposes, which is a little irrelevant but I researched out of curiousity. The ones used by security focused companies and the NSA contain much larger sources of entropy than whatever our laptops can handle. One particularly interesting pRNG was Cloudflare's, which had a literal wall of lava lamps as a source of entropy.

Simulating with random numbers

The simulations we run almost always have to do with using a large number of samples to provide statistical information on certain repeatable events. This type of random sampling falls under the Monte Carlo methods, where random sampling is used to find a probability distribution for a factor that has inherent uncertainty. Essentially, the method boils down to these steps:

1. Set the number and domain of possible outputs
2. Generate random inputs over the domain
3. Apply functions
4. Analyze the aggregated results

Here's a simple example for using the Monte Carlo method to approximate pi:

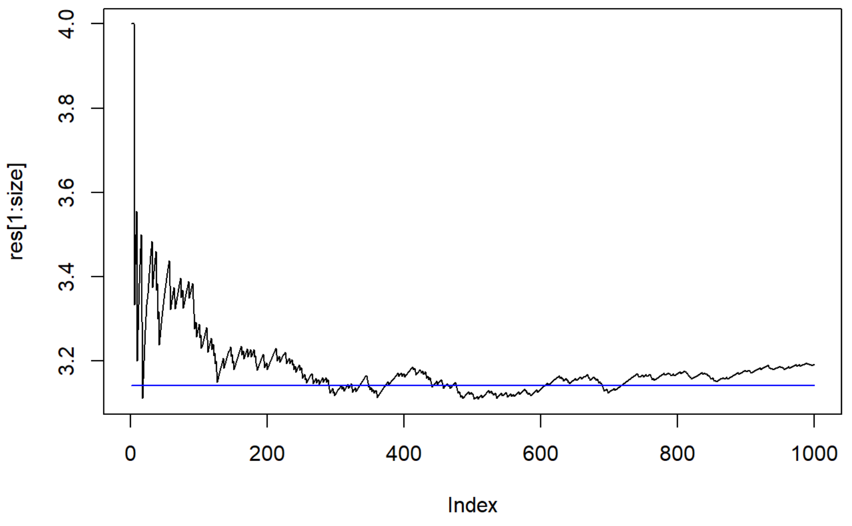
```
# I wrote this code myself but I used the link above for reference

# First we set domain
minimum = 0
maximum = 1
size = 1000

# Next we have a function for generating random points and record how many points are within a certain radius
estim = function(s){
  results = rep(0,s)
  numberIn = 0
  for(i in 1:s){
    # Generates a vector with 2 random numbers between 0 and 1
    x = runif(n = 2, min = minimum, max = maximum)
    # Checks whether or not the point x above is 1 away from (0,0)
    if(sqrt(x[1]*x[1] + x[2]*x[2]) <= 1){
      numberIn = numberIn + 1
    }
    # Records the proportion of dots within a radius to total dots (for the plot)
    prop = numberIn / i
    results[i] = prop * 4
  }
  return(results)
}
res = estim(size)
res[size]
```

```
## [1] 3.192
```

```
plot(res[1:size], type = 'l')
lines(rep(pi, size)[1:size], col = 'blue')
```



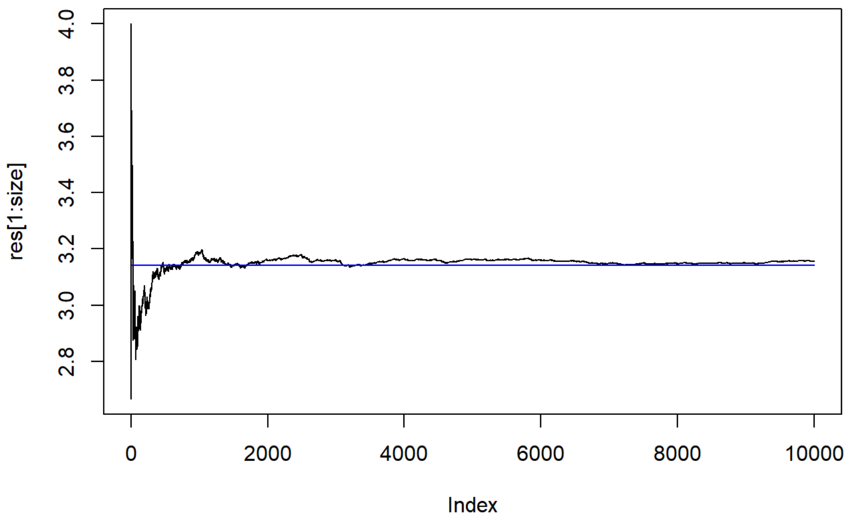
The estimation looks good but still seems a bit off compared to the actual value of pi, which is marked by a blue line. By the law of large numbers, the sample generated average from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. So all we have to do is do more trials for a closer approximation of pi:

```
# Changed the size of
size = 10000

res = estim(size)
res[size]
```

```
## [1] 3.1572
```

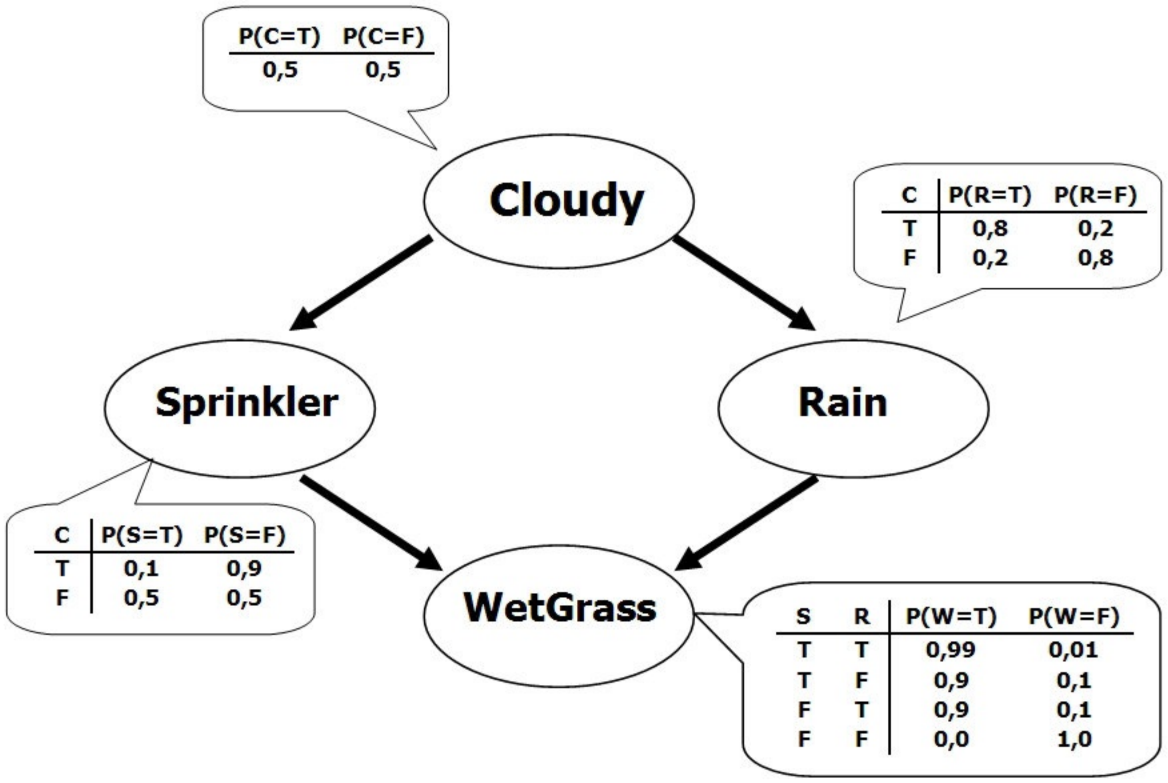
```
plot(res[1:size], type = 'l')
lines(rep(pi, size)[1:size], col = 'blue')
```



As you can see, the approximated values is much closer now with 10,000 trials. For a visual interpretation of what is happening with this approximation, check out this gif.

Simulation: An advanced example

Because the Monte Carlo methods can be used to create good estimates of probability distributions, it can be used in all kinds of fields. One interesting usage is approximating probabilities of nodes in a Bayes net. A Bayes net is a representation of the probabilities of a set of events, where the probabilities of each event is expressed given their parents' value. One example is shown below:



The probability of the grass being wet is clearly shown to be dependent on whether or not the sprinklers are on. However we might want to know the probability of the grass being wet given the sky being cloudy. We can get this by taking a large amount of random samples starting at the cloudy node and working our way down. Then we count how many times the grass is wet given the sky being cloudy or not in order to find the probability distribution of the grass being wet given cloudiness. This directly computes the probability of the event we are looking for, which can substitute the mathematical way of summing together probabilities. Practically, these methods mean we can take observed distributions and run many tests on models in order to find new distributions.

References

1. https://www.random.org/analysis/
2. http://www.purdue.edu/uns/html4ever/2005/050426.Fischbach.pi.html
3. https://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/
4. https://www.fastcodesign.com/90137157/the-hardest-working-office-design-in-america-encrypts-your-data-with-lava-lamps
5. http://rstudio-pubs-static.s3.amazonaws.com/115850_cee3fc32511f49458ed13c8c959244b6.html
6. http://www.palisade.com/risk/monte_carlo_simulation.asp
7. https://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif
8. https://www.r-bloggers.com/estimation-of-the-number-pi-a-monte-carlo-simulation/