# Randomness: Comparing a Trig Function vs. awk's Built-in rand() Function

Rande Shern[*]

March 16, 2021

## Abstract

A simple pseudorandom number generating function is created using the $sin()$ function. The results of this function is compared to awk's built in $rand()$ function. Both functions are designed to return a random number in [0, 1.0). Their outputs will be compared to Uniform(0,1) using a Chi-square test. And the 'rate' at which the two sequences "cover" the range of integers {0,1,2,3,4,5,6,7,8,9} will be haphazardly investigated. Plus, lets compute their Cross Entropy and pontificate blindly about what it means.

## Introduction

Generating random numbers with a computer is an intriguing area of computer science. True randomness cannot be achieved but the process of psuedorandom number generation can yield a useful approximation. Most modern day programming languages have a library to handle random number generation.

The language *awk* is universal across all flavors of Linux. awk is generally used to process text files, but it has a built in random number generator (RNG) named *rand()*. There is also an associated function named *srand(seed)* which is used to set a seed and create a new starting point for random number generation each time a program is run.

Over the decades research has been done to find robust ways of implementing RNGs. From days pasts, one such method used functions from trigonometry, namely the sin() function. A simple awk function can be written using sin() as the basis for attempting random number generation.

In this paper, the results of using the sin() function as the basis for a RNG versus the more 'advanced' algorithm of awk's built in rand() function is investigated. Below we'll investigate how close each function is to Uniform(0,1) using a Chi-square test. Plus we'll curate an analysis of variance in an attempt measure the rate (or even how well) the two function "cover" the range of integers 1 through 10. And finally, we'll figure out how to compute Cross Entropy with these two functions, and wave our hands wildly with speculation about what this means.

## 1 Using Sin() for pseudorandomness

The trig functions date back to antiquity. It is easy to imagine the sin() function being used by Egyptian engineers to build the pyramids. Later, mathematicians derived the trig functions as infinite sums of series. Now, we will use the result of the sin() function as a way to *create* randomness (approximately.)

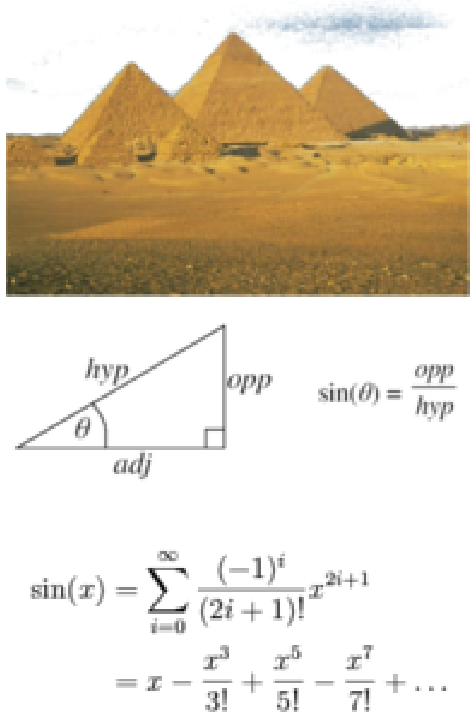
hyp
opp
θ
adj
sin(θ) = opp/hyp
sin(x) = Σ_{i=0}^∞ ((−1)^i/(2i+1)!) x^{2i+1}
= x − x³/3! + x⁵/5! − x⁷/7! + ...

Figure 1: Ancient Pyramids and sin() function.

### The sin() based Random Number Generator

The sin() function from trigonometry has the domain $(-\infty, +\infty)$ for Real numbers and codomain $[-1, +1]$ for Real numbers. The range of a standard RNG is in [0, 1.0). Sin() has negative values in its range, so the

[*] He is approximately a PhD student in AI.

negative values can be curtailed or made positive with ABS(). A procedure that feeds the decimals of draw $i$ into draw $i+1$ will be used in an attempt to approximate randomness. The defined function is named erratic().

```
Global seed = 0.5; # To avoid the 0.0

function erratic()
{  x = sin(seed) * 1000;
   result = abs(x - int(x)); # grab the
      decimals
   seed = result; # set seed for the next
      result
   return result;
}
```

To use this "erratic" function, first a seed value has to be set. This is done globally. When the function erratic() is called the built in sine function is computed and stored in *result*. In this case, we first move the decimal three places to the right and then the integer part of the result is subtracted and the absolute value is used. Each time the function is called the remaining decimals are assigned to the global seed variable. This is used for the next input value of the sine function.

Sending small decimals into the math sin() function and using the resulting decimals in the next round is how an approximation of randomness is achieved. Now let us investigate this approximation?

Table 1: Sample output of the erractic() function. seed = 0.5.

| | result |
|---|---|
| 1 | 0.470985 |
| 2 | 0.764089 |
| 3 | 0.879613 |
| 4 | 0.492355 |
| 5 | 0.702670 |
| 6 | 0.257353 |
| 7 | 0.521975 |
| 8 | 0.593438 |
| 9 | 0.214905 |
| 10 | 0.254255 |

## A Chi-square Test of the erratic() and rand() Output

The Chi-Square test is a statistical test and can be used to compare two probability distribution and assign a confidence as to how *similar* they are. This is done with a simple formula and a pre-constructed table of values encoding powerful theory.

Nominally a programming language's RNG function should be Uniform(0,1). With one adaption this will be the 'target' distribution, as claimed in the null hypothesis of the Chi-Square test. The simple adaption is, by use of a case statement, will be modifying the output of *erratic* and binning it onto the range of integers 1,2,3,4,5,6,7,8,9,10. This is done for convenience when writing the program, and making graphs, but the main goal of the Chi-square is still captured. A decision will be made if the output using a sin() function can be confidently assumed 'close' to what a true Uniform distribution would yield.

Here is the formula of the Chi-square. We have 9 $(10-1)$ degrees of freedom because the integers 1..10 make up the search space.

$$\chi^2 = \sum_{d \,\in\, Data}^{n} \frac{(O_d - E_d)^2}{E_d}$$

```
function chisquare(observed,expected,n)
{  score = 0;
   for (i=1; i<=n; i++)
   {  score += (observed[i] -
         expected)*(observed[i] -
         expected)/expected;
   }
   return score;
}
```

## Experimental Design

This Chi-square test has 9 df and the test value from the table is 16.919. Here is summary of our hypothesis:

| | |
|---|---|
| **null** | Our "Function" based method is not dissimilar from Uniform(0,1). |
| **alternative** | Something is up, the game is blown my friend. |

There are two "Functions" in play: 1. erratic, 2. rand().

Remembering back to the description of awk's built in rand() function it was brought up there is a seed value that can be set each time a program is run to ensure sequences of random numbers are not repeated. Here we want to do two experiments. One in which the seed is purposefully the same initial value every time, so we get the same test sequence each test trial; and Two an experiment with the seed value different each test trail. Our experiment will thus produce two tables of output, one for each handling of the seed variable. The reason for this is due to a hunch.

Data sets of size 50, 100, 200, 500, 1000, 2000, 5000 will be created. This is the number of random numbers generated by both the erratic function and awk's built in rand() function. At each step, the Chi-square statistic will scored and a result of either 'Pass' or 'Fail' reported. By Pass and Fail, what is meant is an answer to the question do we *accept the null* (e.g. Pass or Fail)?

## Analysis of Chi-square Test Results

The results are somewhat surprising and a little fascinating. Each table displays the results of a Chi-square test run over seven separate trials of differing sample size. On each row, we see the result of both the erratic function and awk's built in rand() function. Plus, the statistic's actual score is listed in parenthesis.

Table 2: Chi-square Test 1. seed = 0.5.

| Size | erratic | rand() |
|---|---|---|
| 50 | Accept (16.80) | Accept (11.6) |
| 100 | Accept (7.0) | Accept (14.6) |
| 200 | Accept (8.5) | Accept (7.5) |
| 500 | Accept (7.56) | **Reject** (22.44) |
| 1000 | **Reject** (18.52) | Accept (7.06) |
| 2000 | Accept (5.29) | Accept (3.97) |
| 5000 | **Reject** (21.29) | Accept (7.42) |

Table 3: Chi-square Test 2. seed = $RANDOM (12180).

| Size | erratic | rand() |
|---|---|---|
| 50 | Accept (12.8) | Accept (16.4) |
| 100 | Accept (4.6) | Accept (7.6) |
| 200 | Accept (10.5) | Accept (16.4) |
| 500 | Accept (5.4) | **Reject** (18.0) |
| 1000 | Accept (14.12) | Accept (10.36) |
| 2000 | Accept (8.41) | Accept (5.97) |
| 5000 | **Reject** (20.46) | Accept (6.80) |

In the top table (Table 2), the seed is set to .5. So, if we re-ran that table the sample score and result would occur every time. Both functions appear to be achieve the goal of pseudo randomness some of the time, but something happens every now and again. We know from theory RNGs have cycles.

A cycle is a point in a RNG at which point it begins the output sequence again, from the beginning. Designers of RNG work to make these cycle lengths as long a possible (or to make them start over as far apart as possible) while at the same time maintaining proper coverage of [0,1.0) so it appears to manifest "true" randomness. This experiment is reporting a failure here.

Both function fail a Chi-square test at different sample sizes. Why? Have they hit their cycle length and begun to repeat and destroyed their appearance of true randomness. But then why do they pass at longer run lengths? I don't really know.

Next, look at the second table (Table 3). In this experimental run, the seed value is set to a random value using Linux's built in RANDOM environmental variable. Here, again we se intermittent rejection of the null. This happens to both the erratic function and the rand() function. Again, I don't know why this happened except to say we know *a priori* we are not dealing with true randomness here, these methods were programmed. The explanation of taking a sample too early in the cycle could make sense.

## Cross Entropy

In this final section, an attempt at measuring cross entropy is done. This test needs the output of erratic() and that of rand() to be governed by probability distributions. Here, assume this is the case.

The cross entropy is the average number of bits needed to encode data coming from a source with distribution P when we use model Q. Whereas entropy is calculated on a single distribution, the cross entropy has two distributions.

We can deduce if distribution P is different from distribution Q by using the cross entropy of P and Q, because its value will be different than the entropy of Q alone. That is, if we want to test if the erratic function's distribution is close to Uniform(0,1), then we can use the entropy of Uniform(0,1) as a target. And the closer the cross entropy of the pair is to this value, we'll say that is better *output behavior*.

$$H(P,Q) = \sum_{x \in Digits} P(x) \cdot log(Q(x))$$

```
function crossentropy(P,Q,sampleSize,n)
{  score = 0;
   for (i=1; i<=n; i++)
   {  score += (P[i]/sampleSize)*log(Q);
   }
   return -1*score;
}
```

However, for this experiment it is not required to compute the entropy of Uniform(0,1), because if the cross entropy decreases as the sample size increases, then we know the two distribution are getting "closer" (look at my hands wave).

### Results

Table 4: Cross Entropy. seed = .5

| n | erratic |
|---|---|
| 50 | 2.30259 |
| 100 | 2.30259 |
| 200 | 2.30259 |
| 500 | 2.30259 |
| 1000 | 2.30259 |
| 2000 | 2.30259 |
| 5000 | 2.30259 |

That is exactly right.

## 2 Conclusion

In conclusion, for small jobs (e.g. setting the initial weights of a simple neural net) using the sine function as the basis for generating random number works ok. Don't push it too far, because we know the cycles of RNGs cause pseudo randomness to vanish if we draw too many samples.

Future Work: The cross entroy sections needs to explored further to make sure all the parts were in alignment and the values being computed are meaningful.

[ ... And it was good.]

## 3 References

"Probability and Statistics, 2nd", Book, by DeGroot
"Machine Learning Mastery", Blog, by Brownlee
"Neural Networks with JavaScript Succinctly", e-Book, by McCaffrey