Random Number Generation And Testing

Prepared by:-

Aditya Shashikant Nandgaokar(11012434)

Email Id:- aditya.nandgaokar2196@gmail.com

Mohammad Taha Siddiqui (11012463)

Email Id: tahasiddiqi0807@gmail.com

Introduction

A **random number generator** (**RNG**) is a device that generates a sequence of numbers or symbols that cannot be reasonably predicted better than by a random chance. Random number generators can be true **hardware random-number generators** (HRNG), which generate genuinely random numbers, or **pseudo-random number generators** (PRNG) which generate numbers which look random, but are actually deterministic, and can be reproduced if the state of the PRNG is known. Various applications of randomness have led to the development of several different methods for generating random data, of which some have existed since ancient times, among whose ranks are well-known "classic" examples, including the rolling of dice, coin flipping, the shuffling of playing cards, the use of yarrow stalks (for divination) in the I Ching, as well as countless other techniques. Because of the mechanical nature of these techniques, generating large numbers of sufficiently random numbers (important in statistics) required a lot of work and/or time. Thus, results would sometimes be collected and distributed as random number tables.

Several computational methods for pseudo-random number generation exist. All fall short of the goal of true randomness, although they may meet, with varying success, some of the statistical tests for randomness intended to measure how unpredictable their results are (that is, to what degree their patterns are discernible). This generally makes them unusable for applications such as cryptography. However, carefully designed **cryptographically secure pseudo-random number generators** (CSPRNG) also exist, with special features specifically designed for use in cryptography.

Practical Application And Uses

Random number generators have applications in gambling, statistical sampling, computer simulation, cryptography, completely randomized design, and other areas where producing an unpredictable result is desirable. Generally, in applications having unpredictability as the paramount, such as in security applications, hardware generators are generally preferred over pseudo-random algorithms, where feasible. Random number generators are very useful in developing Monte Carlo-method simulations, as debugging is facilitated by the ability to run the same sequence of random numbers again by starting from the same *random seed*. They are also used in cryptography – so long as the seed is secret. Sender and receiver can generate the same set of numbers automatically to use as keys. The generation of pseudo-random numbers is an important and common task In computer programming. While cryptography and certain numerical algorithms require a very high degree of *apparent* randomness, many other operations only need a modest amount of unpredictability. Some simple examples might be presenting a user with a "Random Quote of the Day", or determining which way a computer-controlled adversary might move in a computer game. Weaker forms of randomness are used in hash algorithms and in creating amortized searching and sorting algorithms.

Some applications which appear at first sight to be suitable for randomization are in fact not quite so simple. For instance, a system that "randomly" selects music tracks for a background music system must only appear random, and may even have ways to control the selection of music: a true random system would have no restriction on the same item appearing two or three times in succession.

Generation Methods

Physical Methods

The earliest methods for generating random numbers, such as dice, coin flipping and roulette wheels, are still used today, mainly in games and gambling as they tend to be too slow for most applications in statistics and cryptography.

A physical random number generator can be based on an essentially random atomic or subatomic physical phenomenon whose unpredictability can be traced to the laws of quantum mechanics. Sources of entropy include radioactive decay, thermal noise, shot noise, avalanche noise in Zener diodes, clock drift, the timing of actual movements of a hard disk read/write head, and radio noise. However, physical phenomena and tools used to measure them generally feature asymmetries and systematic biases that make their outcomes not uniformly random. A randomness extractor, such as a cryptographic hash function, can be used to approach a uniform distribution of bits from a non-uniformly random source, though at a lower bit rate.

The appearance of wideband photonic entropy sources, such as optical chaos and amplified spontaneous emission noise, greatly aid the development of the physical random number generator. Among them, optical chaos has a high potential to physically produce high-speed random numbers due to its high bandwidth and large amplitude. A prototype of a high speed, real-time physical random bit generator based on a chaotic laser was built in 2013.

Various imaginative ways of collecting this entropic information have been devised. One technique is to run a hash function against a frame of a video stream from an unpredictable source. Lavarand used this technique with images of a number of lava lamps. HotBits measures radioactive decay with Geiger–Muller tubes, while Random.org uses variations in the amplitude of atmospheric noise recorded with a normal radio.

Another common entropy source is the behavior of human users of the system. While people are not considered good randomness generators upon request, they generate random behavior quite well in the context of playing mixed strategy games. Some security-related computer software requires the user to make a lengthy series of mouse movements or keyboard inputs to create sufficient entropy needed to generate random keys or to initialize pseudorandom number generators.

Computational Methods

Most computer generated random numbers use pseudorandom number generators (PRNGs) which are algorithms that can automatically create long runs of numbers with good random properties but eventually the sequence repeats (or the memory usage grows without bound). These random numbers are fine in many situations but are not as random as numbers generated from electromagnetic atmospheric noise used as a source of entropy.[9] The series of values generated by such algorithms is generally determined by a fixed number called a **seed.** One of the most common PRNG is the linear congruential generator, which uses the recurrence.

$$X_{n+1} = (a X_n + b) \bmod m$$

to generate numbers, where *a*, *b* and *m* are large integers, and X_{n+1} is the next in *X* as a series of pseudo-random numbers. The maximum number of numbers the formula can produce is one less than the modulus, *m*-1. The recurrence relation can be extended to matrices to have much longer periods and better statistical properties. To avoid certain non-random properties of a single linear congruential generator, several such random number generators with slightly different values of the multiplier coefficient, *a*, can be used in parallel, with a "master" random number generator that selects from among the several different generators.

A simple pen-and-paper method for generating random numbers is the so-called middle square method suggested by John von Neumann. While simple to implement, its output is of poor quality. It has a very short period and severe weaknesses, such as the output sequence almost always converging to zero. A recent innovation is to combine the middle square with a Weyl sequence. This method produces high quality output through a long period. See Middle Square Weyl Sequence PRNG.

Most computer programming languages include functions or library routines that provide random number generators. They are often designed to provide a random byte or word, or a floating point number uniformly distributed between 0 and 1.

The quality i.e. randomness of such library functions varies widely from completely predictable output, to cryptographically secure. The default random number generator in many languages, including Python, Ruby, R, IDL and PHP is based on the Mersenne Twister algorithm and is *not* sufficient for cryptography purposes, as is explicitly stated in the language documentation. Such library functions often have poor statistical properties and some will repeat patterns after only tens of thousands of trials. They are often initialized using a computer's real time clock as the seed, since such a clock generally measures in milliseconds, far beyond the person's precision. These functions may provide enough randomness for certain tasks (for example video games) but are unsuitable where high-quality randomness is required, such as in cryptography applications, statistics or numerical analysis.

Much higher quality random number sources are available on most operating systems; for example /dev/random on various BSD flavors, Linux, Mac OS X, IRIX, and Solaris, or CryptGenRandom for Microsoft Windows. Most programming languages, including those mentioned above, provide a means to access these higher quality sources.

Generation from a probability distribution

Generating From Probability distribution

There are a couple of methods to generate a random number based on a probability density function. These methods involve transforming a uniform random number in some way. Because of this, these methods work equally well in generating both pseudo-random and true random numbers. One method, called the inversion method, involves integrating up to an area greater than or equal to the random number (which should be generated between 0 and 1 for proper distributions). A second method, called the acceptance-rejection method, involves choosing an x and y value and testing whether the function of x is greater than the y value. If it is, the x value is accepted. Otherwise, the x value is rejected and the algorithm tries again.

By Humans

Random number generation may also be performed by humans, in the form of collecting various inputs from end users and using them as a randomization source. However, most studies find that human subjects have some degree of non-randomness when attempting to produce a random sequence of e.g. digits or letters. They may alternate too much between choices when compared to a good random generator; thus, this approach is not widely used.

True Vs Pseudo-Random Numbers

There are two principal methods used to generate random numbers. The first method measures some physical phenomenon that is expected to be random and then compensates for possible biases in the measurement process. Example sources include measuring atmospheric noise, thermal noise, and other external electromagnetic and quantum phenomena. For example, cosmic background radiation or radioactive decay as measured over short timescales represent sources of natural entropy.

The speed at which entropy can be harvested from natural sources is dependent on the underlying physical phenomena being measured. Thus, sources of naturally occurring "true" entropy are said to be blocking– they are rate-limited until enough entropy is harvested to meet the demand. On some Unix-like systems, including most Linux distributions, the pseudo device file /dev/random will block until sufficient entropy is harvested from the environment.[1] Due to this blocking behavior, large bulk reads from /dev/random, such as filling a hard disk drive with random bits, can often be slow on systems that use this type of entropy source.

The second method uses computational algorithms that can produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a seed value or key. As a result, the entire seemingly random sequence can be reproduced if the seed value is known. This type of random number generator is often called a pseudorandom number generator. This type of generator typically does not rely on sources of naturally occurring entropy, though it may be periodically seeded by natural sources. This generator type is non-blocking, so they are not rate-limited by an external event, making large bulk reads a possibility.

Some systems take a hybrid approach, providing randomness harvested from natural sources when available, and falling back to periodically re-seeded software-based cryptographically secure pseudorandom number generators (CSPRNGs). The fallback occurs when the desired read rate of randomness exceeds the ability of the natural harvesting approach to keep up with the demand. This approach avoids the rate-limited blocking behavior of random number generators based on slower and purely environmental methods.

While a pseudorandom number generator based solely on deterministic logic can never be regarded as a "true" random number source in the purest sense of the word, in practice they are generally sufficient even for demanding security-critical applications. Indeed, carefully designed and implemented pseudo-random number generators can be certified for security-critical cryptographic purposes, as is the case with the yarrow algorithm and fortuna. The former is the basis of the /dev /random source of entropy on FreeBSD, AIX, OS X , NetBSD, and others. OpenBSD also uses a pseudo-random number algorithm based on ChaCha20 known as arc4random.

Post –Processing And Statistical Checks

Even given a source of plausible random numbers (perhaps from a quantum mechanically based hardware generator), obtaining numbers which are completely unbiased takes care. In addition, behavior of these generators often changes with temperature, power supply voltage, the age of the device, or other outside interference. And a software bug in a pseudo-random

number routine, or a hardware bug in the hardware it runs on, may be similarly difficult to detect.

Generated random numbers are sometimes subjected to statistical tests before use to ensure that the underlying source is still working, and then post-processed to improve their statistical properties. An example would be the TRNG9803 hardware random number generator, which uses an entropy measurement as a hardware test, and then post-processes the random sequence with a shift register stream cipher. It is generally hard to use statistical tests to validate the generated random numbers. Wang and Nicol proposed a distance-based statistical testing technique that is used to identify the weaknesses of several random generators. Li and Wang proposed a method of testing random numbers based on laser chaotic entropy sources using Brownian motion properties.

Other Consideration

Random numbers uniformly distributed between 0 and 1 can be used to generate random numbers of any desired distribution by passing them through the inverse cumulative distribution function (CDF) of the desired distribution (see Inverse transform sampling). Inverse CDFs are also called quantile functions. To generate a pair of statistically independent standard normally distributed random numbers (x, y), one may first generate the polar coordinates(r, ϑ), where $r \sim \chi_2^2$ and $\vartheta \sim \text{UNIFORM}(0,2\pi)$ (see Box–Muller transform).

Some 0 to 1 RNGs include 0 but exclude 1, while others include or exclude both.The outputs of multiple independent RNGs can be combined (for example, using a bit-wise XOR operation) to provide a combined RNG at least as good as the best RNG used. This is referred to as software whitening.

Computational and hardware random number generators are sometimes combined to reflect the benefits of both kinds. Computational random number generators can typically generate pseudo-random numbers much faster than physical generators, while physical generators can generate "true randomness."

Low-Discrepancy Sequences As An Alternative

Some computations making use of a random number generator can be summarized as the computation of a total or average value, such as the computation of integrals by the Monte Carlo method. For such problems, it may be possible to find a more accurate solution by the use of so-called low-discrepancy sequences, also called quasirandom numbers. Such sequences have a definite pattern that fills in gaps evenly, qualitatively speaking; a truly random sequence may, and usually does, leave larger gaps.

Backdoors

Since much cryptography depends on a cryptographically secure random number generator for key and cryptographic nonce generation, if a random number generator can be made predictable, it can be used as backdoor by an attacker to break the encryption.

The NSA is reported to have inserted a backdoor into the NIST certified cryptographically secure pseudorandom number generator Dual EC DRBG. If for example an SSL connection is created using this random number generator, then according to Matthew Green it would allow NSA to determine the state of the random number generator, and thereby eventually be able to read all data sent over the SSL connection. Even though it was apparent that Dual EC DRBG was a very poor and possibly backdoored pseudorandom number generator long before the NSA backdoor was confirmed in 2013, it had seen significant usage in practice until 2013, for example by the prominent security company RSA Security. There have subsequently been accusations that RSA Security knowingly inserted a NSA backdoor into its products, possibly as part of the Bullrun program. RSA has denied knowingly inserting a backdoor into its products.

It has also been theorized that hardware RNGs could be secretly modified to have less entropy than stated, which would make encryption using the hardware RNG susceptible to attack. One such method which has been published works by modifying the dopant mask of the chip, which would be undetectable to optical reverse-engineering. For example, for random number generation in Linux, it is seen as unacceptable to use Intel's RdRand hardware RNG without mixing in the RdRand output with other sources of entropy to counteract any backdoors in the hardware RNG, especially after the revelation of the NSA Bullrun program.

In 2010, a U.S. lottery draw was rigged by the information security director of the Multi-State Lottery Association (MUSL), who surreptitiously installed backdoor malware on the MUSL's secure RNG computer during routine maintenance. During the hacks the man won a total amount of $16,500,000 by predicting the numbers correct a few times in year.

ASLR or Address Space Layout Randomization, a mitigation against rowhammer and related attacks on the physical hardware of memory chips has been found to be inadequate as of early 2017 by VUSec. The random number algorithm if based on a shift register implemented in hardware is predictable at sufficiently large values of p and can be reverse engineered with enough processing power. This also indirectly means that malware using this method can run on both GPUs and CPUs if coded to do so, even using GPU to break ASLR on the CPU itself.

Activities And Demonstrations

The following sites make available Random Number samples:

1. The SOCR resource pages contain a number of hands-on interactive activities and demonstrations of random number generation using Java applets.
2. The Quantum Optics Group at the ANU generates random numbers sourced from quantum vacuum. You can download a sample of random numbers by visiting their quantum random number generator research page.
3. Random.org makes available random numbers that are sourced from the randomness of atmospheric noise.
4. The Quantum Random Bit Generator Service at the Ruđer Bošković Institute harvests randomness from the quantum process of photonic emission in semiconductors. They supply a variety of ways of fetching the data, including libraries for several programming languages.
5. The Group at the Taiyuan University of technology generates random numbers sourced from chaotic laser. You can obtain a sample of random number by visiting their Physical Random Number Generator Service.

Introduction to Chi-Square testing

A chi-squared test, also written as χ^2 test, is any statistical hypothesis test where the sampling distribution of the test statistic is a chi squared distribution when the null hypothesis is true.

Chi-squared tests are often constructed from a sum of squared errors, or through the sample variance. Test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data.

$$\chi_c^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Random Number Generation to be used for testing:

We are using a list of up to about 3000 random numbers between 0-255. The 256 numbers are randomly distributed throughout the list and will be tested using various testing techniques.

Testing Techniques

Uniform Distribution

The Chi-Square statistic is most commonly used to evaluate tests of independence when using a crosstabulation. Crosstabulation presents the distributions of two categorical variables simultaneously, with the intersections of the categories of the variables appearing in the cells of the table. The test of independence assesses whether an association exists between the two variables by comparing the observed pattern of responses in the cells to the pattern that would be expected if the variables were truly independent of each other. Calculating the Chi-Square statistic and comparing it against a critical value from the Chi-Square distribution allows the researcher to assess whether the observed cell counts are significantly different from the expected cell counts.

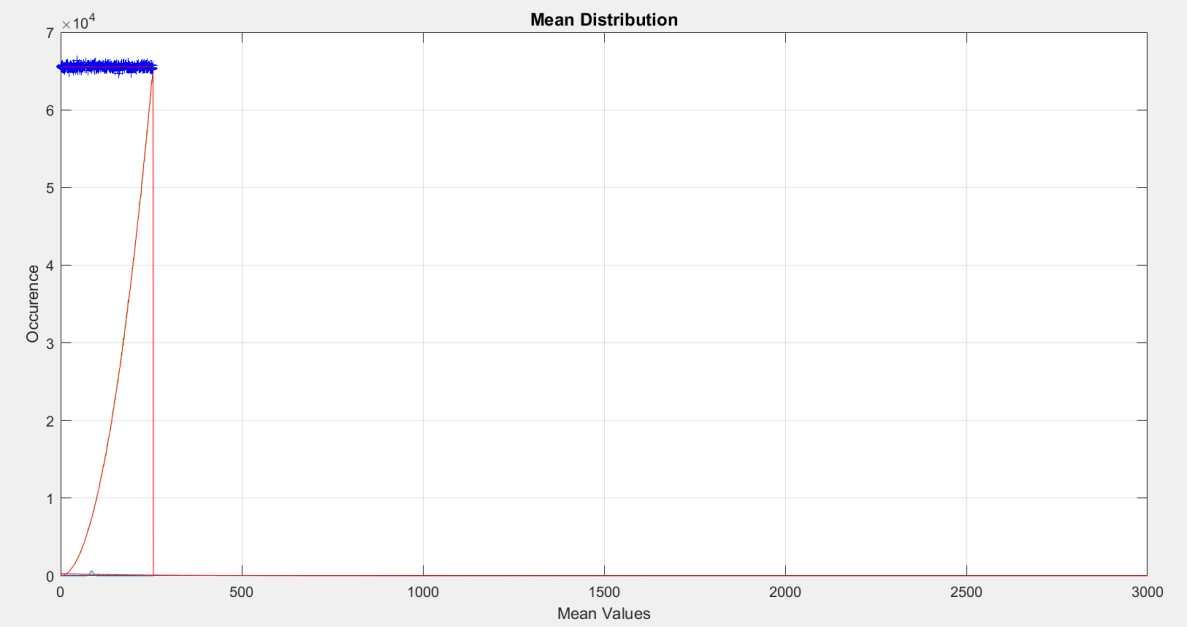
Running MATLAB code:

```
format longE;

clc
clear

X = load ('RandomNumbersLarge_1.txt');


count_0 = 0;
count_255 = 0;
count = zeros(256,1);
%count(256) = 0;
count_c(256) = 0;  %Initialization only with 0 or 1

%Get Distribution
for n=1:length(X)
```

```
   count(X(n) + 1) = count(X(n) + 1) + 1; %Array starts with 1 and not 0

end

%Expected Values for uniform distribution
for n2 = 1:256
   count_c(n2) = 256*256;
end


figure(1)
n2 = 1:256;
plot(n2,count(n2),'-b+');
hold on
plot(n2,count_c(n2),'-r');

%Calculating Sigma of Uniform Distribution
Sigma = 0;
for i=1:1:length(count)
   Sigma = Sigma + (count(i) - 256*256)^2;
end

Sigma = Sigma/length(count);
Sigma = sqrt(Sigma)       % ~sqrt(256*256)



Chi_Square = 0;
for n=1:1:length(count)
   Chi_Square = Chi_Square + ((count(n) - 256*256)^2)/(256*256);
end

Chi_Square = Chi_Square/256


%--------------------------------------------------------------------------
Chi_Square_S1 = 0;
for n=1:1:length(count)
   Chi_Square_S1 = Chi_Square_S1 + ((count(n) - (256*256 + Sigma))^2)/(256*256);
end

Chi_Square_S1 = Chi_Square_S1/256

%--------------------------------------------------------------------------
Chi_Square_S2 = 0;
for n=1:1:length(count)
   Chi_Square_S2 = Chi_Square_S2 + ((count(n) - (256*256 + 2*Sigma))^2)/(256*256);
end
```

```
Chi_Square_S2 = Chi_Square_S2/256

%-------------------------------------------------------------------------
Chi_Square_S3 = 0;
for n=1:1:length(count)
   Chi_Square_S3 = Chi_Square_S3 + ((count(n) - (256*256 + 3*Sigma))^2)/(256*256);
end

Chi_Square_S3 = Chi_Square_S3/256
```

Result



(Graphic taken from MATLAB)

Gap Test

Each random number is an independent sample drawn from a continuous uniform distribution between 0 and 1.

For this test we are using a sample of random number presented in two groups with about 3000 random numbers in each.

The *Gap Test* measures the number of digits between successive occurrences of the same digit

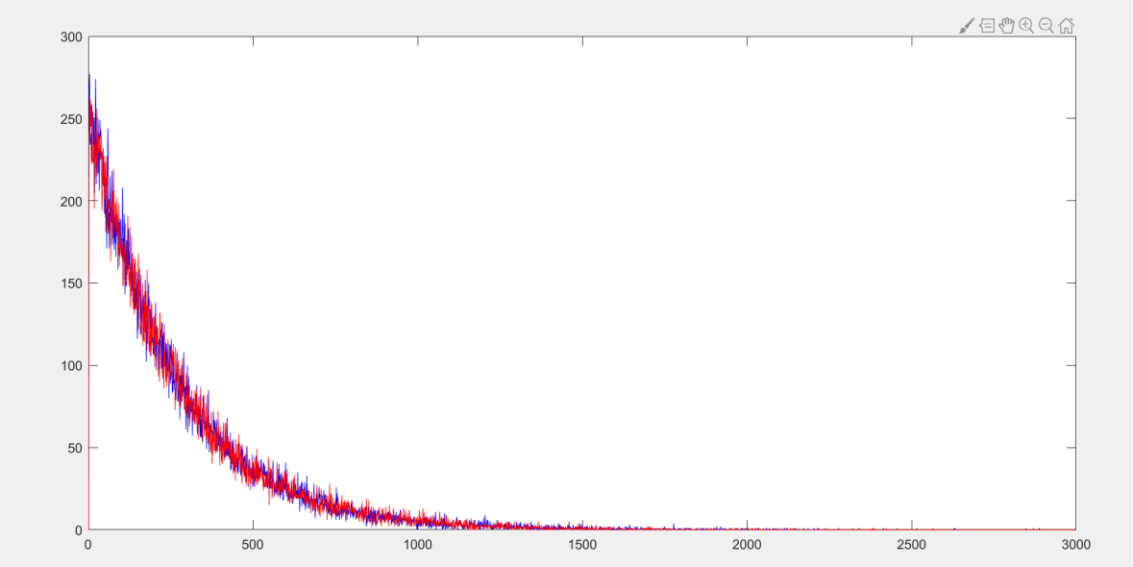
Counts the number of digits that appear between repetitions of a particular digit and then uses the Kolmogorov-Smirnov test to compare with the expected number of gaps.

Running MATLAB code

```
clc
clear


X = load ('RandomNumbersLarge_1.txt');

figure(1);
GapArray(10000) = 0;
GapVar = 7;
Gap = 1;

for n=1:1:length(GapArray)
   GapArray(n) = 0;
end

for n=1:1:length(X)

   if(GapVar ~= X(n))
      Gap = Gap + 1;
   else
      GapArray(Gap + 1) = GapArray(Gap + 1) + 1;
      Gap = 1;
   end
end
```

```
n=1:1:3000;
plot(n,GapArray(n),'-b');
hold on

for n=1:1:length(GapArray)
   GapArray(n) = 0;
end

GapVar = 11;
Gap = 1;
for n=1:1:length(X)

   if(GapVar ~= X(n))
      Gap = Gap + 1;
   else
      GapArray(Gap + 1) = GapArray(Gap + 1) + 1;
      Gap = 1;
   end
end

n=1:1:3000;
plot(n,GapArray(n),'-r');
```

Result



(Graphic taken from MATLAB)

Random check using Chi Square

This code is used to generate a series of graphs where a number of testing methods have been used on the same array of numbers.

Running MATLAB code:

```
format longE;

clc
clear



X = load ('C:\MATLAB\RandomNumbersLarge_1.txt');



count_0 = 0;
count_255 = 0;
count = zeros(256,1);
count_c(256) = 0;  %Initialization only with 0 or 1

%Get Distribution
for n=1:length(X)

   count(X(n) + 1) = count(X(n) + 1) + 1; %Array starts with 1 and not 0

   %Checking
   if X(n) == 0
      count_0 = count_0 + 1;
   end
   if X(n) == 255
      count_255 = count_255 + 1;
   end

end

%Expected Values for uniform distribution
for n2 = 1:256
```

```
   count_c(n2) = 256*256;
end


figure(1)
n2 = 1:256;
plot(n2,count(n2),'-b+');
hold on
plot(n2,count_c(n2),'-r');

%Get the number of occurences of a selected number in all blocks
a = 13;
count_a = zeros(256,1);
ct = 0;
for n=1:256:length(X)-256 %Analizing Blocks of 256
   for k=n:1:(n+256)
      if X(k) == a
         ct = ct + 1;
      end
   end
   count_a(ct + 1) = count_a(ct + 1) + 1;
   ct = 0;
end

figure(2)
n2 = 1:256;
ymax = (count_a(1)*1.2);
plot(n2,count_a(n2),'-b+');
axis([0 12 0 ymax]);


figure(3)
count_kol = 1;
count_kk = zeros(256,1);
count_ktest = zeros(256,1);

for n=1:256:length(X)-256

   for k=n:1:(n+256)
      count_ktest(X(k) + 1) = count_ktest(X(k) + 1) + 1;
   end

   for k2=1:256
      if(count_ktest(k2) > 0)
        count_kol = count_kol + 1;
      end
   end
```

```
   count_kk(count_kol) = count_kk(count_kol) + 1;

   count_kol = 1;
   for k2=1:256
      count_ktest(k2) = 0;
   end

end

k = 1:256;
plot(k,count_kk(k),'-g+');
grid on

figure(4);
GapArray(10000) = 0;
GapVar = 11;
Gap = 1;
for n=1:1:length(X)

   if(GapVar ~= X(n))
      Gap = Gap + 1;
   else
      GapArray(Gap + 1) = GapArray(Gap + 1) + 1;
      Gap = 1;
   end
end

n=1:1:3000;
plot(n,GapArray(n),'-g');

fid = fopen('Gap.txt','wt');
fprintf(fid,'%i     %i\n',[n;GapArray(n)]);
fclose(fid);


figure(5);
Max = 1;
countMax = zeros(256,1);

for n=1:3:length(X)-3

   Max = max(X(n),X(n+1));
   Max = max(Max,X(n+2));

   countMax(Max) = countMax(Max) + 1;

   Max = 1;
```

```
end

n=1:1:256;
plot(n,countMax(n),'-g');

Count_Uni = zeros(256,1);
for n=1:1:length(count)

   Count_Uni = count - 256*256;

end

X_0_1(256 * 256 * 32) = 0;
Div = 256^8;
OffSet = 1;


for n=1:1:length(X_0_1)

   X_0_1(n) = X(OffSet)* 256^7;
   X_0_1(n) = X_0_1(n) + X(OffSet+1)* 256^6;
   X_0_1(n) = X_0_1(n) + X(OffSet+2)* 256^5;
   X_0_1(n) = X_0_1(n) + X(OffSet+3)* 256^4;
   X_0_1(n) = X_0_1(n) + X(OffSet+4)* 256^3;
   X_0_1(n) = X_0_1(n) + X(OffSet+5)* 256^2;
   X_0_1(n) = X_0_1(n) + X(OffSet+6)* 256;
   X_0_1(n) = X_0_1(n) + X(OffSet+7);
   X_0_1(n) = X_0_1(n)/Div;

   OffSet = OffSet + 8;
end



X_0_1_S = sort(X_0_1);
h_ks = kstest(X_0_1_S);

% v=-3:0.01:3; %values set
% mu=0; sigma=1; %random variable parameters
% ypdf=normpdf(v,mu,sigma); %normal PDF
%h_norm = kstest(ypdf);

h_run = runstest(X);

Chi_Square = 0;
for n=1:1:length(count)

   Chi_Square = Chi_Square + ((count(n) - 256*256)^2)/(256*256);
```

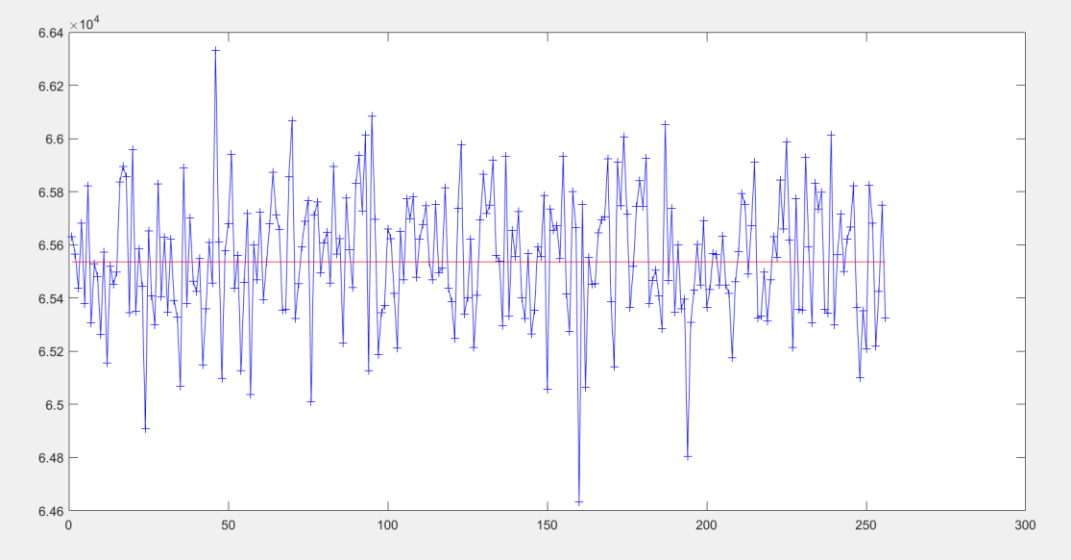
```
end

Chi_Square = Chi_Square/256;
```

Results

Figure one is representing Uniform Distribution:

```
n2 = 1:256;
plot(n2,count(n2),'-b+');
hold on
plot(n2,count_c(n2),'-r');
```

This plot can be observed as:

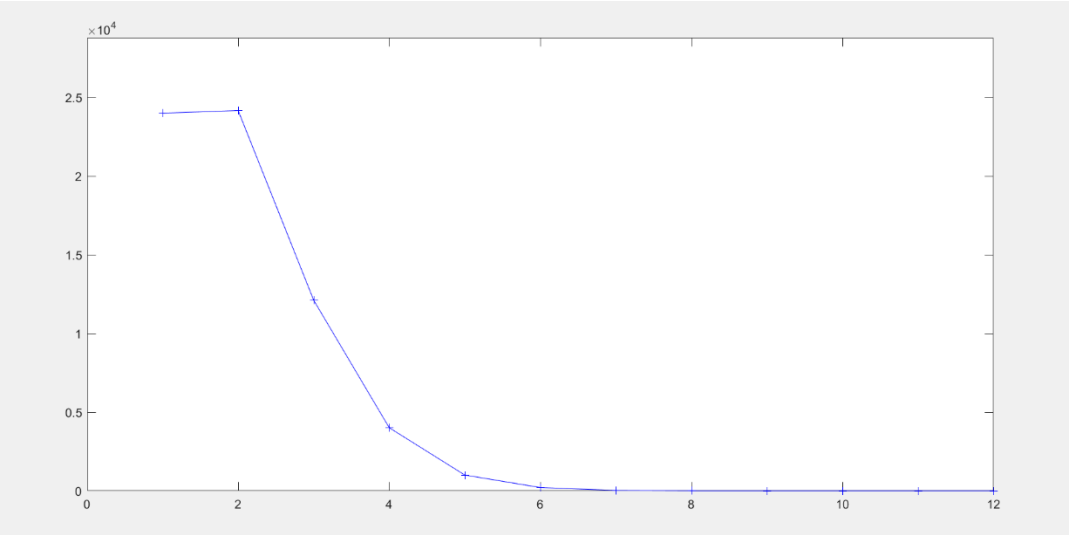


(Graphic taken from MATLAB)

Figure two is representing the maximum count:

```
n2 = 1:256;
ymax = (count_a(1)*1.2);
plot(n2,count_a(n2),'-b+');
axis([0 12 0 ymax]);
```

This plot can be observed as:

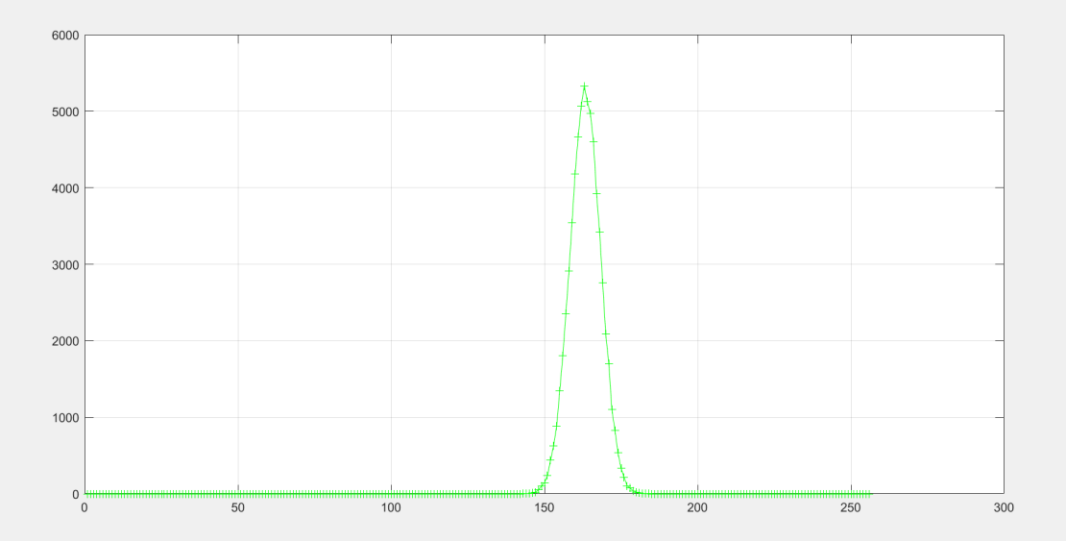


(Graphic taken from MATLAB)

Figure three is representing the count with zeros:

```
k = 1:256;
plot(k,count_kk(k),'-g+');
```

This plot can be observed as:

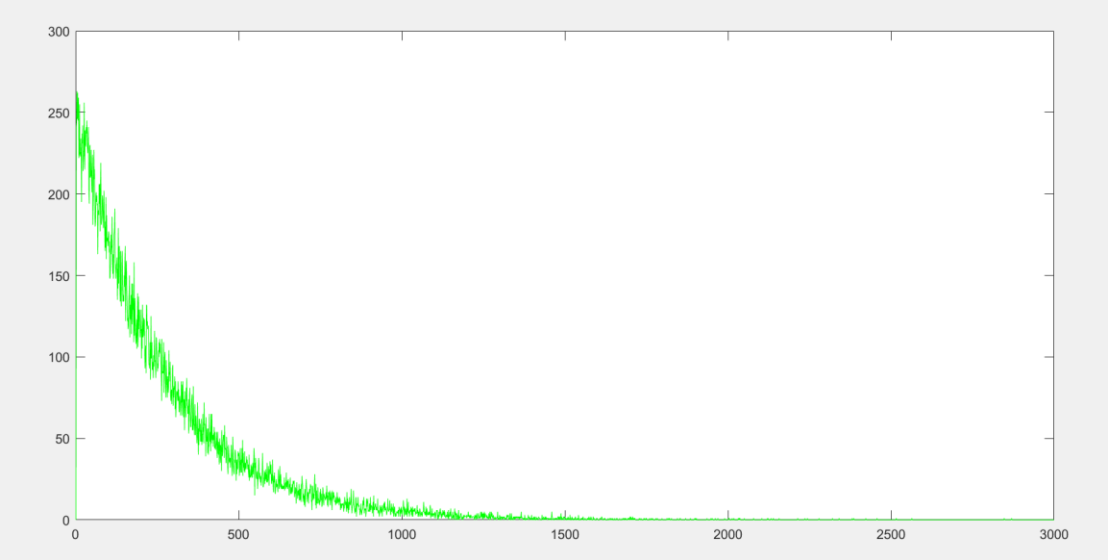


(Graphic taken from MATLAB)

Figure four is representing the Gap Array:

```
n=1:1:3000;
plot(n,GapArray(n),'-g');
```

This plot can be observed as:

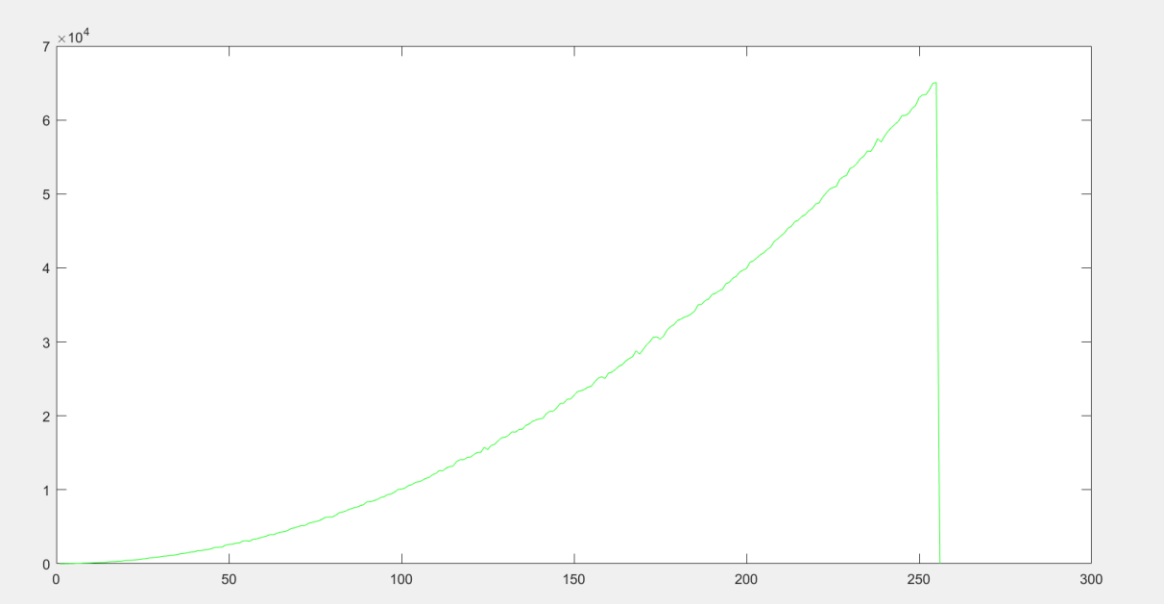


(Graphic taken from MATLAB)

Figure five is representing the plot using Max=1

```
n=1:1:256;
plot(n,countMax(n),'-g');
```

This plot can be observed as:



(Graphic taken from MATLAB)

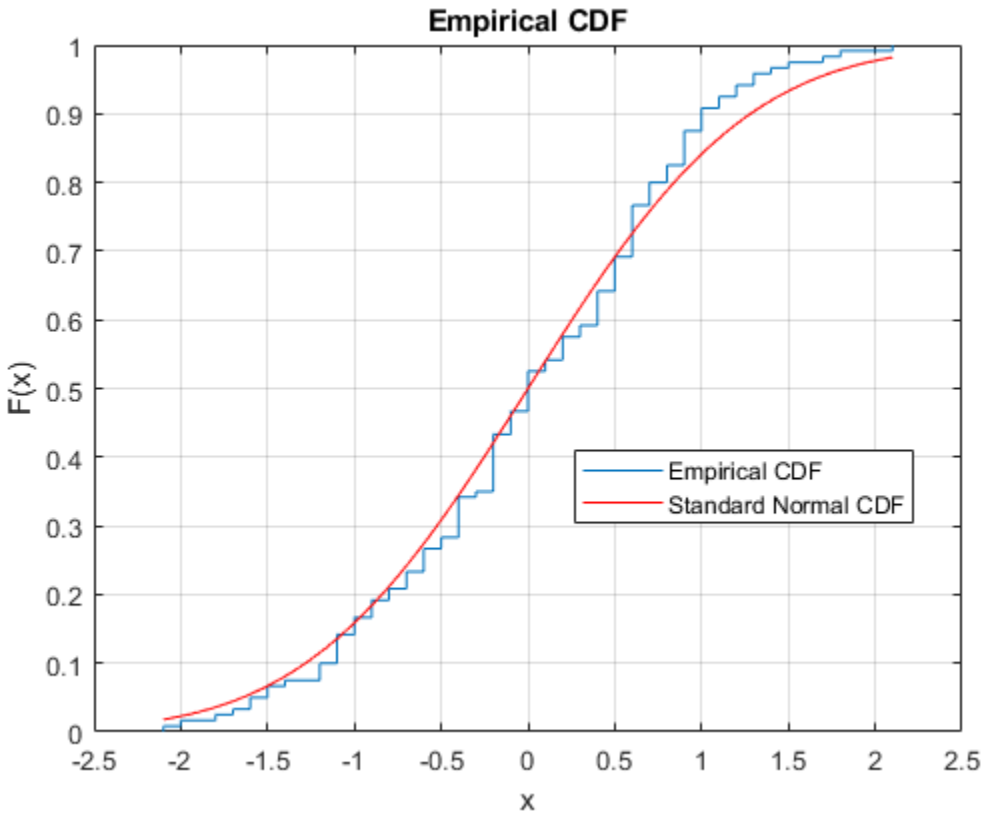
Other Examples include:

Kolmogorov-Smirnof test

Compares the continuous distribution function CDF, F(x), of the uniform distribution with the empirical CDF

The Kolmogorov-Smirnov test tries to determine if two datasets differ significantly. This test has the advantage of making no assumption about the distribution of data. Note however, that this generality comes at some cost: other tests may be more sensitive if the data meet the requirements of the test. In addition to calculating the *D* statistic, this page will report if the data seem normal or lognormal. It will enable you to view the data graphically which can help you understand how the data is distributed.

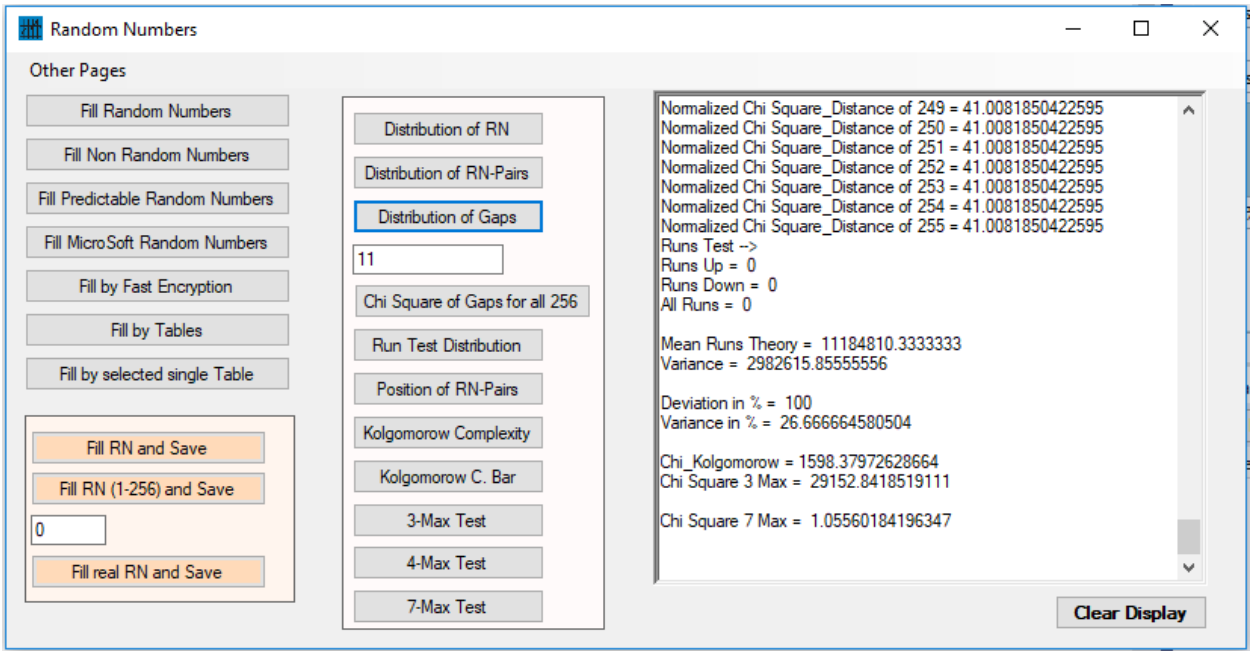
Here, we observe an example of this distribution and how the CDF is seen against ECDF using a similar array of random numbers.



Conclusion

Checking the results of different outcomes of a few testing techniques on visual studio, we get the following results.

According to the tests run on this platform, the resulting values show a variation in values. The range of our outcomes can be seen in the following figure.



References

- Walker, John. "HotBits: Genuine Random Numbers". Retrieved 2009-06-27.
- TrueCrypt Foundation. "TrueCrypt Beginner's Tutorial, Part 3". Retrieved 2009-06-27.
- The MathWorks. "Common generation methods". Retrieved 2011-10-13.
- Matthew Green (2013-09-20). "RSA warns developers not to use RSA products.
- Matthew Green (2013-09-20). "RSA warns developers not to use RSA products.
- MATLAB
- Visual Studio
- wikipedia.org