Generating Uniform Random Variables from Atmospheric Noise

Student ID: 15012534
Supervisor: Dr Terry Soo
Word Count (excl. appendix): 10,984

Department of Statistical Science
University College London

September 6, 2021

Contents

Abstract

Modern programming languages have built-in libraries for randomly sampling data from various distributions, but all of these are based on so called *pseudorandom* number generators. They rely on completely deterministic procedures such as the Mersenne Twister. Although great for quickly generating data that is reproducible, to alleviate any concerns surrounding their fundamentally deterministic nature, we require *true* random number generators. We test a method of using atmospheric noise to generate random bits, which are then used to simulate uniform random variables. We then use the National Institute of Standards and Technology testing suite to assess the validity of this method and attempt to estimate the entropy in the generated data. We find that that our samples are at least as random as both the default Python PRNG and a well-known industry standard of true random numbers. A documented implementation of this has been publicly made available on Github and it is very simple to replace the underlying generator in Python's random library with our own. We therefore argue that users should utilise true random number generators such as our method in order to prevent issues with pseudorandomness.

1 Introduction

Many aspects of computational data science, and indeed any application that requires simulation, are underpinned by the theory of randomness. Machine learning algorithms such as Gradient Boosting Machines require random sampling in order to take subsets of the training data and Neural Networks need a way to initialise a set of starting weights, for example. To this end, most programming languages provide a callable method that algorithms can utilise; but by using software alone, it is impossible to generate an actual random sequence. Termed pseudorandom number generators (PRNGs), instead, classes of algorithms have been developed in order to approximate true randomness. In particular, they aim to output a sequence of numbers that have no trivial statistical relationship. By far the most popular of these is the Mersenne Twister algorithm (Matsumoto and Nishimura, 1998) and Python uses a version with a very large period of $2^{19937}-1$. In fact, this implementation is ported from C and therefore makes the execution time for random draws very low, making it useful for when we need reproducible and quick sampling procedures. However, since the randomness of the output is fully deterministic and dependent on the randomness of the input (or seed), some concerns exist about their use in certain applications. For example, the Metropolis–Hastings Monte-Carlo algorithm uses draws from a uniform distribution to complete the rejection sampling procedure (Hastings, 1970). A PRNG may not be sufficient to ensure convergence to the probability distribution we are trying to simulate. In other words, the critical assumption of randomness is violated and therefore theoretical results may not be attained.

1.1 Motivation

There are many documented cases of when a PRNG is the source of weakness in a system (Li, 2013). A lot of these arise from the fact that they are susceptible to adversarial attacks; a counter algorithm can be used to decipher the seed and give an attacker full knowledge of current and future values. In high stakes scenarios such as an online casino, algorithmic trading or even cryptographic applications, this is not good enough. Common protocols such as Secure Shell Protocol (SSH) use RNGs to generate public and private keys for communication over a network. Most cryptocurrencies also use RNGs in their hash functions, like Ethereum

with KECCAK-256 (Wood, 2018). It becomes especially apparent that we need the ability to generate true random number generators (TRNGs) when you consider that the security industry follows the Kerckhoff Principle. A concept widely embraced in most modern systems, it roughly states that "the security of a system must depend solely on the key and not on the design of the system itself" (Petitcolas, 2011). Simply put, a good cryptographic algorithm must have an original key that is very hard to guess by a would-be attacker - even if they had thorough knowledge of every other part of the system. Analysis on whether or not our RNG has cryptographic-level security is beyond the scope of this paper, however.

By employing the use of hardware, TRNGs use the intrinsic entropy present in a physical system or a natural process to generate sequences of numbers in a completely non-deterministic way. The key to good TRNGs is a chaotic source of entropy, such as radioactive decay; a well-known unpredictable physical phenomenon. Although connecting some sort of Geiger–Müller tube to a computer is impractical and too slow for normal users, there are companies that provide this service for a fee. Hotbits, for example, uses Cæsium-137 decay to generate random numbers and have built an API to which users can subscribe to (Walker, 2006). Further, modern processors have been designed with circuitry that allow an operating system to harness entropy from the fluctuations in thermal noise from the silicon (Intel, 2014). Unfortunately, this system has the common drawback of most TRNGs in that it takes far too long to gather enough entropy to generate lots of numbers. In fact, (Route, 2017) shows that it can be nearly 20 times slower than a Mersenne Twister in a Monte-Carlo simulation application. Therefore Intel, for example, suggests that instead we use this method to seed a PRNG. Conversely, the random bits can be combined using an XOR operation with other sources of entropy to create an even more secure generator. Despite the variety of options available to users, and the fact this method is compliant with the U.S. National Institute of Standards and Technology (NIST) standards, there is still some paranoia around these pre-built accumulators of entropy.

Inevitably, this leads to the auditing of such RNGs. Various testing methods exist solely for the purpose of trying to establish a statistical relationship between the outputs of such a generator. The most commonly used software suite is from the NIST and it involves a battery of increasingly stringent tests. We will use this to examine how well our method performs. In particular, we will use noisy radio signals. We will then try to isolate and examine the atmospheric noise present; and check to see if the entropy gathered is enough to build a practicable TRNG. First, we explain some concepts below.

1.2 Mersenne Twister

A Mersenne prime is a prime number that is exactly one less than a power of two. For example, both 7 (i.e. 2^3-1) and 31 (i.e. 2^5-1) are Mersenne primes. As the name suggests, the Mersenne Twister is an algorithm that generates a number in the range $[0, 2^n - 1]$ for some $n \in \mathbb{N}$ in a pseudorandom fashion, using one of the aforementioned primes as the period length. The most commonly used variant is one that uses a period of length $2^{19937} - 1$ and has subsequently been named MT19937. Algorithm 1 from (Tan, 2016) describes an implementation of MT19937 in more detail. Importantly, this Mersenne Twister is *not* cryptographically secure as observing just 624 occurrences allows a counter-algorithm to perfectly predict future values by exactly replicating the initial conditions (Kopp, 2020). This is not a fallacy that is unique to the Mersenne Twister as many other PRNGs also use a similar state mechanism that is used to generate the next value in the sequence (Shema, 2012). However, this makes such algorithms much faster to compute on modern processors than TRNGs.

1.3 Entropy

(Shannon, 1948) first defined the notion of *information* entropy as the amount of "uncertainty" in observing the possible outcomes of a given random variable. More formally; where $I(X) = -\log_b P(x)$ is the information content of a discrete random variable X, with possible outcomes $x_1, ..., x_n$, the Shannon entropy H is given as as the expectation

$$H(X) = \mathbb{E}\left[I(X)\right] = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i), \tag{1.1}$$

where $b = 2$ is commonly used to give entropy in bits. To illustrate this, consider a coin flip, where the outcome is either "heads" or "tails" with equal probability. The amount of entropy in each flip can therefore be calculated as

$$H(X) = -\sum_{i=1}^{2} \frac{1}{2} \log_2 \frac{1}{2} = -\frac{1}{2}(-1) - \frac{1}{2}(-1) = 1.$$

The equal probability of either occurrence means we have the most uncertainty in this scenario. If we know more information about the random variable, the amount of entropy in the system will decrease. For example, consider now a biased coin where the probability of landing on "heads" is 0.75. The amount of entropy in each flip has now decreased to

$$H(X) = -P(H) \log_2 P(H) - P(T) \log_2 P(T) = -\frac{3}{4}(-0.415) - \frac{1}{4}(-2) = 0.81125.$$

Shannon entropy has been used extensively in previous literature as a measure of uncertainty in many different types of systems. Although, since then, there have been papers that define other measures such as min-entropy,

$$H_\infty(X) = \min_x \{-\log_2 P(x)\}, \tag{1.2}$$

which one can think of as the probability that the most likely value will occur. There is also the Rényi entropy,

$$H_\alpha(X) = \frac{\alpha}{1-\alpha} \log_2 \mathbb{E}\left[P(x^\alpha)\right], \alpha \geqslant 0 \text{ and } \alpha \neq 1, \tag{1.3}$$

which is a generalisation upon Shannon's work. Note that in the limit as $\alpha \to 1$, we get Shannon entropy. However, we will stick to Shannon's definition for our calculations. This is because it can be shown that, in the long run over many samples, the min-entropy and Shannon entropy are roughly the same and the distinction between them becomes less significant (Vadhan, 2012). Moreover, Shannon entropy is much easier to calculate and work with than Rényi entropy.

1.4 Radio Signals

To understand the data that is being collected by our radio receiver, we must understand how radio signals propagate through the atmosphere. For simplicity, consider that the information being sent is an audio signal representing music (but the following explanation can be extended

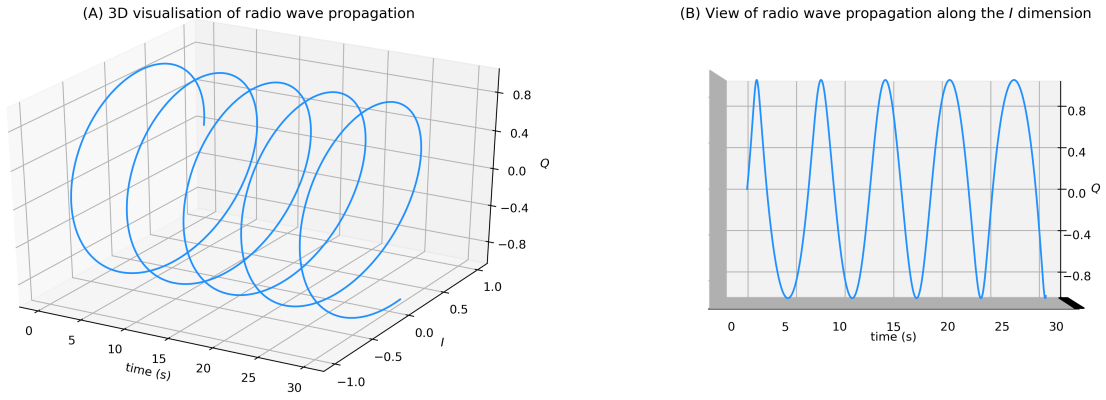

Figure 1: *(A) Visualisation of radio wave propagation through the atmosphere; note the the helical motion. (B) The resulting sinusoidal waveform when orthogonal to one of the dimensions. For simplicity, here we have* $I = \cos(t)$ *and* $Q = \sin(t)$ *for* $t > 0$ *representing time in seconds.*

to other sources such as digital TV). A transmitter is first set to oscillate at a predetermined frequency, termed the *carrier wave*. A modulator is then used to modify some aspect of the carrier wave to impart the audio signal. For example, this may be the amplitude of the wave leading to AM transmission or the frequency of the wave leading to FM transmission (Lapidoth, 2016). Although it varies by region, FM radio in Europe uses a carrier wave with a frequency between 87.5 to 108 MHz (ITU, 1984).

A receiver is used at the other end, and as the wave passes through the circuitry, it induces a current. This is a much weaker form of the original signal and so it must be amplified and then demodulated to extract the audio signal from the carrier wave. The more sophisticated the receiver, the more samples it can take per second and therefore the more accurately it can recreate the original audio signal. The Nyquist-Shannon sampling theorem states that "an analog signal waveform may be uniquely and precisely reconstructed from samples taken at equal time intervals, provided the sampling rate is at least twice the highest frequency in the original signal" (Shannon, 1949). Note that oversampling can also lead to unwanted artifacts and noise, which could be beneficial in our use case.

It is important to realise that the carrier waveform travels in a helical motion and the receiver actually captures the 2-dimensional "snapshot" of the signal at each time interval. These dimensions are termed in-phase (I) and quadrature (Q). Figure 1 shows that viewed in only the time and I dimensions, we would see this helix instead as a sinusoidal wave. Similarly if viewed in only the time and Q dimensions, but out of phase by 90 degrees. Note that if we propagate a modulated signal (i.e. different from the carrier wave), the IQ samples are independant of each other.

It is efficient to represent this using an array of complex numbers. For example, suppose the receiver samples the complex number $0.7 - 0.5i$. What does this mean? First, we note that the amplitude $A = \sqrt{0.7^2 + 0.5^2} = 0.860$ and the phase, $\phi = \tan\left(\frac{0.7}{0.5}\right)^{-1} = 0.951$. Then, by the following trigonometric identity $a\cos(\alpha) + b\sin(\alpha) = A\cos(\alpha - \phi)$, we have that our received signal $x(t)$ at time $t > 0$ is

$$x(t) = 0.7\cos(2\pi f_c t) - 0.5\sin(2\pi f_c t) = 0.86\cos(2\pi f_c t - 0.951),$$

where f_c is the carrier frequency. This allows us to alter both the amplitude and the frequency of the transmission signal (i.e. transmit a complicated waveform like a song) by just altering I and Q (Lichtman, 2021).

2 Literature Review

Much of the past research in this field has been aimed towards two streams; identifying valid sources of entropy or finding practical methods of extracting it. Combined with the copious amount of sensory input available to be examined and their inherent portable nature, cellular (mobile) phones have been identified as ideal candidates for both purposes.

2.1 Sensor-Based Entropy Source Analysis

(Krhovjak et al., 2007) found that the ability of the peripherals in mobile phones to generate random data was promising. Due to API restrictions at the time, in particular they focused on the camera. They found that the noise present in camera images from a Nokia N73 device was highly dependant on the ambient temperature, with the level of noise decreasing as the temperature dropped. Although a known property of optical sensors, this showed that the source of entropy was indeed from the chip and not due to the software processing of the image. Further, they found that after normalising pixel values (an attempt to remove any remaining systematic correction effects due to post-processing beyond their control), the blue color provided more Shannon entropy than red or green. By taking a continuous stream of 12 images per second and after verifying that the autocorrelation of pixels between subsequent frames was not significantly different to a white noise process, they estimated that a single independant blue pixel held 4.7608 bits of Shannon entropy. Similarly, we will also check the autocorrelation of our input as we don't want them to be time-dependant.

Current mobile phone models are able to record in much more detail and at much faster frame-rates. However, as phones became more complex, they also included access to even more sensory devices and therefore more potential sources of entropy. Further investigations have been carried out on the use of the built-in Global Positioning Service (GPS), the accelerometer, the magnetic field sensor (magnetometer), and the orientation sensor (Suciu et al., 2011). They extracted randomness from drastically different sources and then were able to combine them in a way that preserved and enhanced the amount of entropy that was gathered. This combination of data was found to pass 7 tests from the NIST test battery. The authors conclude that the advantage of this method is that it leverages the mobility and portability of a mobile phone to provide extra levels of unpredictability, based on environmental and human-device interaction. Further, the technique allows for new streams of data to be added as more sensors become available. As suggested, we will also use an input source that allows for changes in the environment to affect the data collected. In particular, we identified that using radio signals would allow for this.

2.2 Atmospheric Noise

By sampling the current induced by a radio wave in a radio receiver's inner circuitry, we are able to recreate the original transmission. However, the signal is subject to erroneous and unwanted interference that we term collectively as "noise". The sources of said noise can be either synthetic or natural. Synthetic noise could arise from nearby sources of radiation from appliances, radio signals accidentally diffracted from much further away and even the power supply of the receiving device being improperly tuned. Through careful engineering efforts, these sources can be eliminated or at least minimised (Hum, 2018). Modern radio receivers do as much of this as possible automatically. On the other hand, naturally occurring sources of noise are much harder to remove and therefore of more interest to us. From extraterrestrial particles interacting with the Earth's magnetic field to lighting discharges in thunderstorms,

(Bianchi and Meloni, 2007) found that both cosmic and atmospheric noise sources were the main culprits in high frequency (30 - 300 MHz) radio transmission interference. In information theory, these are collectively known as additive white Gaussian noise (AWGN). This is because by the central limit theorem, we know that the summation of many random processes will tend towards a Gaussian distribution.

(Haahr, 1998) utilises this to provide TRNGs on their website; often used for competitive draws and random sampling in medical studies (where it is vital to remove any potential deterministic effects from the grouping process). The service has radio antennae situated around the world and then a proprietary and confidential method is used to generate bits of entropy using the data gathered. We use the service to generate 1,048,576 binary samples for $5. It takes around 25 minutes for these numbers to be generated and we will compare our methods with this industry standard. We hope to generate samples that are at least as "good" as this, in terms of randomness.

2.3 Random Bit Extraction

A random bit extractor (RBE), or a randomness extractor, is a function that receives input from a source of entropy and generates a "more" random output. To formalise this definition, we first introduce some notation.

Notation: $\{0,1\}^n$ is a sequence of 1's and 0's of length n.

Formally, an RBE is defined as a mapping from $\{0,1\}^m \times \{0,1\}^q \to \{0,1\}^n$, such that $\{0,1\}^n$ is an n-bit sample that has been removed of any bias that may be present in either $\{0,1\}^m$ or $\{0,1\}^q$. (Trevisan and Vadhan, 2000) define a **strong** extractor as one that will attempt to output a stream of bits that is both independent from the source and, most importantly, uniformly distributed.

Perhaps the simplest and one of the earliest example of an RBE is credited to (von Neumann, 1951). Again, to illustrate this, suppose we flip a coin that has probability p of landing on heads and $1-p$ of landing on tails. Note here that p is not necessarily 0.5 and therefore some bias may exist. We repeat this n times, and as we do, we note down '$x_i = 1$' if heads and '$x_i = 0$' if tails, arbitrarily. After n flips, we would have a n-bit sample from Bernoulli(p), i.e. $\{0,1\}^n$. Von Neumann's algorithm considers each consecutive pairs of bits and defines the mapping:

$$00 \to \Lambda, \quad 01 \to 0, \quad 10 \to 1, \quad 11 \to \Lambda,$$

where Λ is the null output (i.e. discard the pair). The resulting output bits are random variables from the Bernoulli(0.5) distribution. Thus, regardless of any bias in the coin, we can generate an unbiased sequence of bits, as long as each trial is independant. Von Neumann also defines the notion of *efficiency* in this context to mean the ratio between the expected number of output bits to input bits. The main drawback of this method is that it discards a lot of bits in the process. In other words, it is very inefficient. We use a stepsize of 2 (i.e. we look at x_i and x_{i+1} and then move onto x_{i+1} and x_{i+2}) leading to halving the output regardless of whether or not we discard any pairs. Further, we don't re-use any bits that are discarded. In fact, it can be shown that approximately, $m = np(1-p)$ (Peres, 1992). Thus, in the best-case scenario with $p = 0.5$, if we assume that the inputs are independant and not correlated at all, it is clear that we lose about three-quarters of the data we collected.

Since then, much research has been undertaken in order to improve upon this algorithm and realise gains in efficiency. (Hoeffding and Simons, 1970) investigated *delay*; defined as the mean

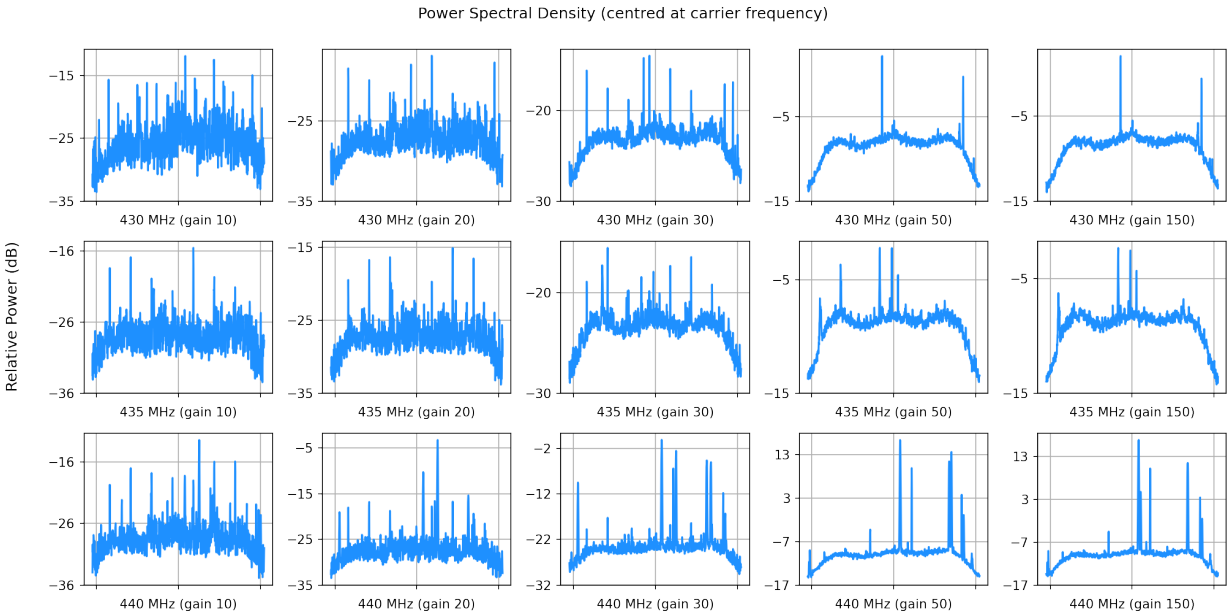

Figure 2: *A grid of power spectral density plots for each frequency and gain. Increasing the gain seems to reduce the noise present in the signal and it also amplifies the signal. We see that the first two frequencies do not seem to differ much, but that there is a strong (high decibel) signal present at 440 MHz.*

number of steps before a pair of bits are found that don't lead to a discard. They found that von Neumann's method has a delay of 4 whereas they find one with a delay of just 3.1. This measure would be useful if we were to instead create a streaming application that generates random bits live. (Elias, 1972) further improved on this work by developing a method that first splits the input bits into blocks $x^N = x_1, ..., x_N$ and outputs blocks $z^K = z_1, ..., z_K$. He proposed that taking the average of K/N is a better alternative definition of *efficiency*. Using this, he was able to develop an RBE that had a significantly higher efficiency. For $N = 4$, Elias' algorithm uses the following mapping (Ryabko and Matchikina, 2000):

$$\begin{array}{llll} 0000 \to \Lambda, & 0001 \to 11, & 1111 \to \Lambda, & 1110 \to 00, \\ 1000 \to 00, & 1100 \to 1, & 0111 \to 01, & 0011 \to 01, \\ 0100 \to 01, & 0101 \to 0, & 1011 \to 10, & 1010 \to 10, \\ 0010 \to 10, & 0110 \to 00, & 1101 \to 11, & 1001 \to 11. \end{array}$$

We explore both von Neumann's and Elias' algorithms; where we find that the latter preserves almost 60% more of the input data (Table 1).

3 Methods & Experiment

Initially, we were interested in investigating the use of the radio receivers that were often included as part of a mobile phone's sensor array. However, most modern mobile phones do not have this receiver (in particular, we checked the Apple iPhone 12 range and the Samsung Galaxy S10 range). It's likely that the advent of the internet has lead to this function being made redundant as users could listen to the radio via online streaming instead. Therefore we turn to a computer-based radio scanner system to receive live radio signals. We use a

Realtek RTL2832U (RTL) analogue receiver to turn radio signals into electrical signals. The communications mechanism is via USB to get IQ data back into a software package, which in our case is Python 3. Often, a software defined radio (SDR) setup such as this will need processing to transform the time-domain data to the frequency-domain so the user can listen to the audio signals. This can be done via Discrete Fourier Transforms, for example (Collins et al., 2018). However, since we are interested in the pure IQ signals, we forego this step. Furthermore, without knowing the data that is transmitted, we cannot exactly separate it from the noise on the receiving end. Thus we must design a method for collecting the entropy from the samples. But first, we perform some exploratory analysis. Note that this paper has a complementary Jupyter Notebook that provides an implementation of the following analyses in Python.

The RTL-SDR device we employ has several settings that must be set. In particular, we care about the following:

- `center_freq` - this is the carrier frequency we wish to sample,
- `sample_rate` - how many samples we wish to take per second; default is `2.097e6`,
- `sample_size` - how many samples do we want to take overall; default is `1.048e6`,
- `freq_correction` - if the receiver is out of tune, we add a correction; default is `60`,
- `gain` - the level of amplification we apply to the input signal; default is `auto`.

We leave everything at default except the centre frequency f_c and the gain g. Initial investigations show that these settings are what govern most of the variation in the data sampled. Furthermore, we are unable to vary other settings much due to a limitation in the hardware (such as the sample rate). To help us decide, we sample from a wide range of f_c and g. Note that the range we can try is limited by the minimum (25 MHz) and maximum (1750 MHz) resolution of the device. Further, the (Ofcom, 2010) frequency allocation table states that the frequency band between 430 MHz and 440 MHz is reserved for amateur radio-location in the UK. To avoid any industrial level signals (FM radio, maritime radio-navigation, emergency services, etc), we sample from this band only. The results are visualised in Figure 2. The Power Spectral Density (PSD) diagram is a representation of the frequency-domain and is a distribution of the signals which compose the sample (Hunter, 2007). This allows us to see if there are any strong underlying transmissions and see how noisy the sample is. In fact, we find through examination of the y-axis that a strong signal is present at 440 MHz. Using this information, we arbitrarily pick $f_c = 435$ MHz as our carrier frequency of choice. Furthermore, it appears that lower gain values lead to more noisy signals. To analyse this, we split the complex samples into their real and imaginary components and plot the resulting distribution of values. Figure 3 shows, firstly, that the shape of the imaginary distributions are very similar to their real counterparts (in fact, we find that all of the results for I are analogous for Q; so we will not explore Q in isolation further in this paper for brevity). We also see that the lower the gain the lower the variance in the sampled data. On the other hand, it seems that increasing the gain too far will result in samples with a variance large enough such that some values are truncated. This limitation of being inside the unit circle seems to be a shortcoming of this particular RTL-SDR device. Since we found that low gain samples are noisier, but high gain samples have more variance, we conclude the initial analysis by selecting $g = 40$ (the median value we tested).

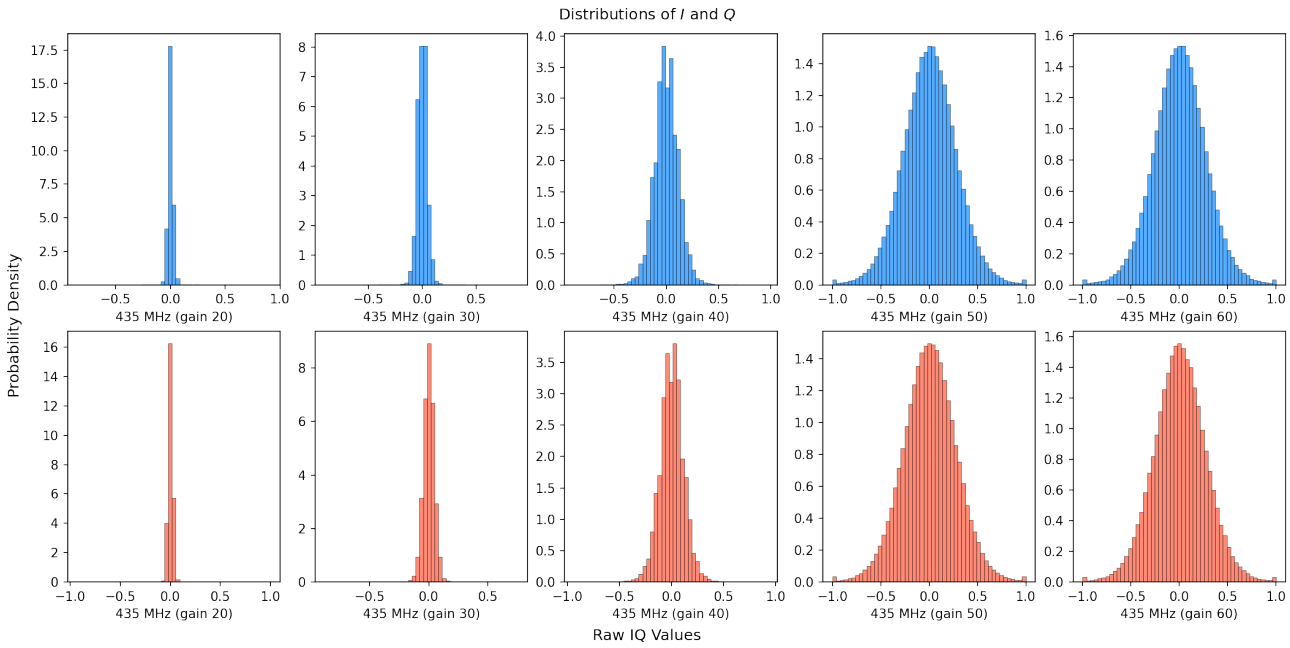

Figure 3: *Histograms of the samples' real component I in blue and the imaginary component Q in orange. The low gain samples (30 and below) do not appear to deviate much from the mean and are centred around 0. The high gain samples (50 and above) seem to add variance to the IQ values. However, we note that they suffer from truncation; likely due to limitation of hardware.*

3.1 Autocorrelation

Autocorrelation is the term given to the correlation between values of a time-series at two different points in time (termed the *lag*). This can tell us whether or not there are significant patterns present in a sample. Therefore we need to design a sampling process of radio signals that will also minimise autocorrelation across as many lags as possible. To achieve this, we use Algorithm 2; henceforth known as "sliced sampling" - which simply attempts to prevent any autocorrelation by only taking, say, every kth data point in a sample and discarding the rest. Note that higher values of k would lead to much longer waiting times for the samples to be gathered, but would likely lead to less autocorrelation. Since speed is one of the factors we are considering, we choose a relatively small value of $k = 1024$.

To illustrate this process, suppose we take a complex-valued sample z_i of length $n = 1,048,576$ using the RTL-SDR device. This takes approximately 2 seconds. Then, we take a complex-valued sub-sample x_j, where j is every kth multiple of i. In other words,

$$x_j = \{z_{1024}, z_{2048}, ..., z_{1,048,576}\}.$$

Note that x_j is of length $\frac{n}{k} = 1024$. We repeat this procedure, each time sub-sampling from a different sample z_i and appending to x_j until we have enough values. If we want x_j to also be of length n, this would take 2048 seconds (or approximately 34 minutes). We also make note of the weather at the location during the sampling period. This could be important as it has been shown that a large component of atmospheric noise is lightning strikes. AccuWeather reports that the cloud cover was at 72% and the thunderstorm likelihood was at 11%.

An autocorrelation plot can then be used to visualise the autocorrelations at each time lag. If a time-series is found to be non-random, then one or more of the autocorrelations will be significantly non-zero. It is important to note, however, that the reverse is not true; having

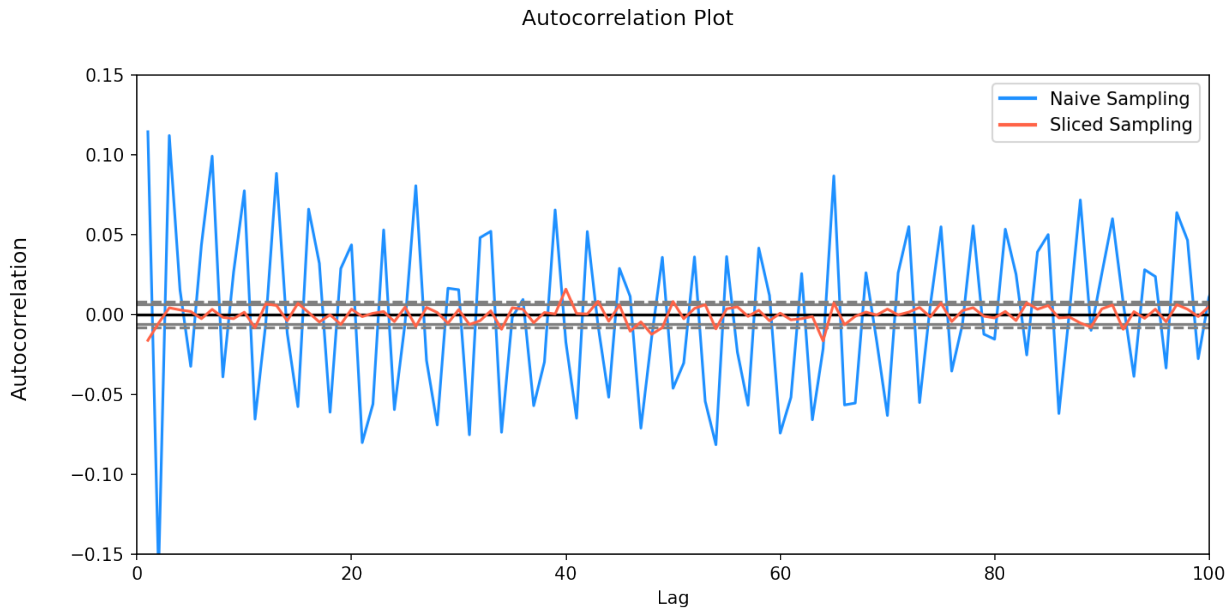

Figure 4: *Autocorrelation plot for both sampling methods, up to the first 100 lags. A hypothesis test is completed at each lag, and the the dashed line is the 99% confidence band. We observe that the naive sampling method has almost all autocorrelations outside this range whereas the reverse is true for sliced sampling.*

no autocorrelations does not imply that a time series is random (Box and Jenkins, 1976). In the case of RNGs, more rigorous tests must also be undertaken (which we explore in section 3.4). But first we note that Figure 4 shows clearly the benefit of sliced sampling over using a naive sampling process. This is likely because taking a sample without any post-processing may include a signal that has a pattern (i.e. a song or a repeating message) and so only taking every kth value from the sample breaks this up. There are very few significant autocorrelations present when checking the first 100 lags; in fact, almost all of the autocorrelations fall within the 99% confidence limits (by definition, we also expect approximately one out of every one hundred lags to be outside this limit anyway). We therefore conclude that by using sliced sampling, we can be fairly confident of the fact that adjacent observations do not correlate.

3.2 Uniform Distribution

An important characteristic of RNGs for computational applications is that they are able to draw from a standard uniform distribution. Both the data from random.org and Python's Mersenne Twister are able to drawn from $U[0, 1]$. Thus, to be able to compare properly, we now explore methods to do the same. In particular, we explore two methods of generating uniform random variables from independant normal pairs (i.e. from the *standardised* raw IQ values) and two methods of generating uniform random variables from Bernoulli(p) samples. First, though, we test these assumptions of normality and independence. We standardize the samples and apply the Kolmogorov-Smirnov test for normality (Indra Mohan Chakravarti, 1967). By standardise, we mean to calculate the z-values:

$$z_i = \frac{x_i - \bar{x}}{\bar{\sigma}},$$

where $\bar{x}$ is the sample mean and $\bar{\sigma}$ is the sample standard deviation; as opposed to using the data

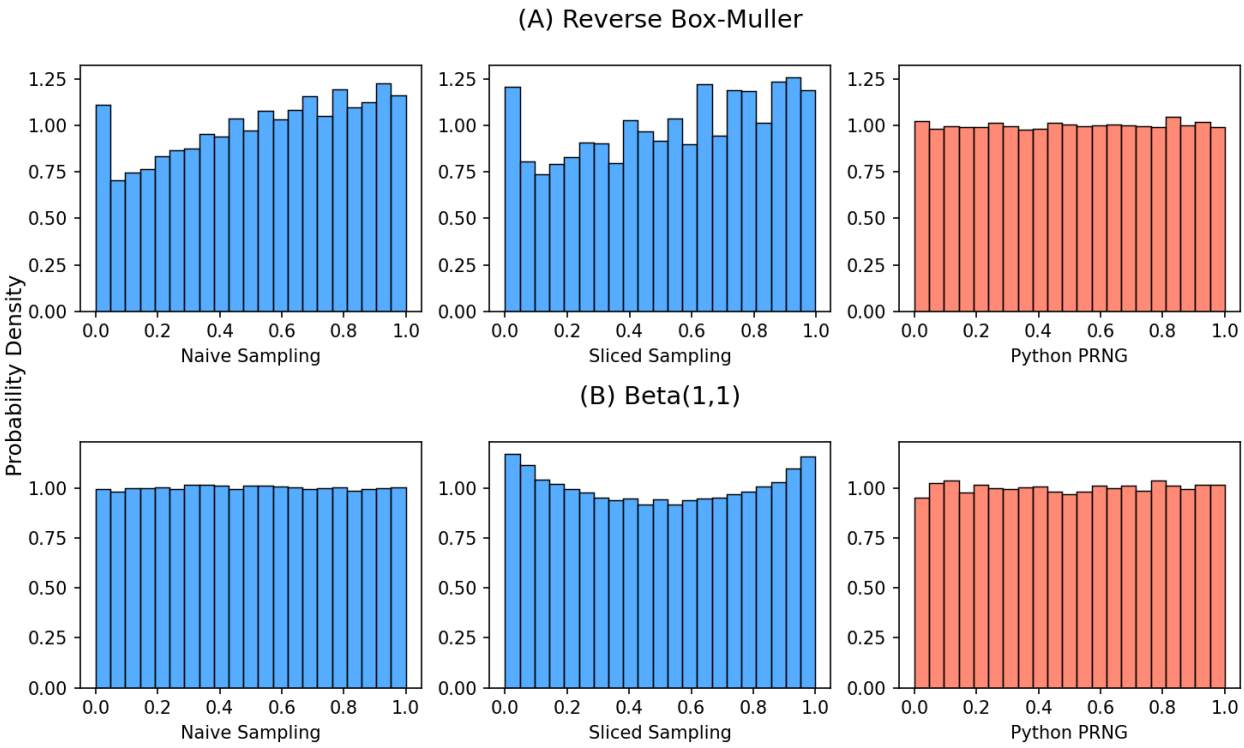

Figure 5: *Histograms of the result after applying Algorithm 3 (A) and Algorithm 4 (B) to standardised IQ data and to Python PRNG generated draws from* $N(0, 1)$.

as is. This test is non-parametric and will compare a histogram of ranked I and Q z-values to the normal cumulative distribution function. Unfortunately, even after standardising the samples, it fails the Kolmogorov-Smirnov test for normality (with p-value $\ll 0.01$). Thus, the first two methods used below to generate uniform random variables from independant normal pairs may fail as this assumption is clearly violated.

We also look for pairwise correlation between I and Q. In particular, we use Spearman's rank correlation as it does not assume normality. We use Pandas, which is a Python library for data wrangling and statistical analysis (Reback et al., 2020). The overall pairwise correlation between I and Q is reported as $\rho = -0.000237$. This is evidence of an extremely weak negative monotonic relationship, but close enough to 0 to conclude that the assumption of independence does indeed hold. We now briefly explain these 4 methods before looking at their results.

3.2.1 Reverse Box-Muller

The Box–Muller transform is a method of generating pairs of independent, standard, normally distributed random numbers from a uniform random source (Box and Muller, 1958). We will reverse the procedure. We also compare it to Python's PRNG samples put through the same procedure, which serves as a benchmark of sorts when visually comparing the methods. This is useful as, although more rigorous, some common tests for uniformity such as the χ^2 test are not very high powered. Furthermore, the results are pretty clear by observation, anyway. Termed "RBM", an implementation of this in Python is given in the appendix as Algorithm 3.

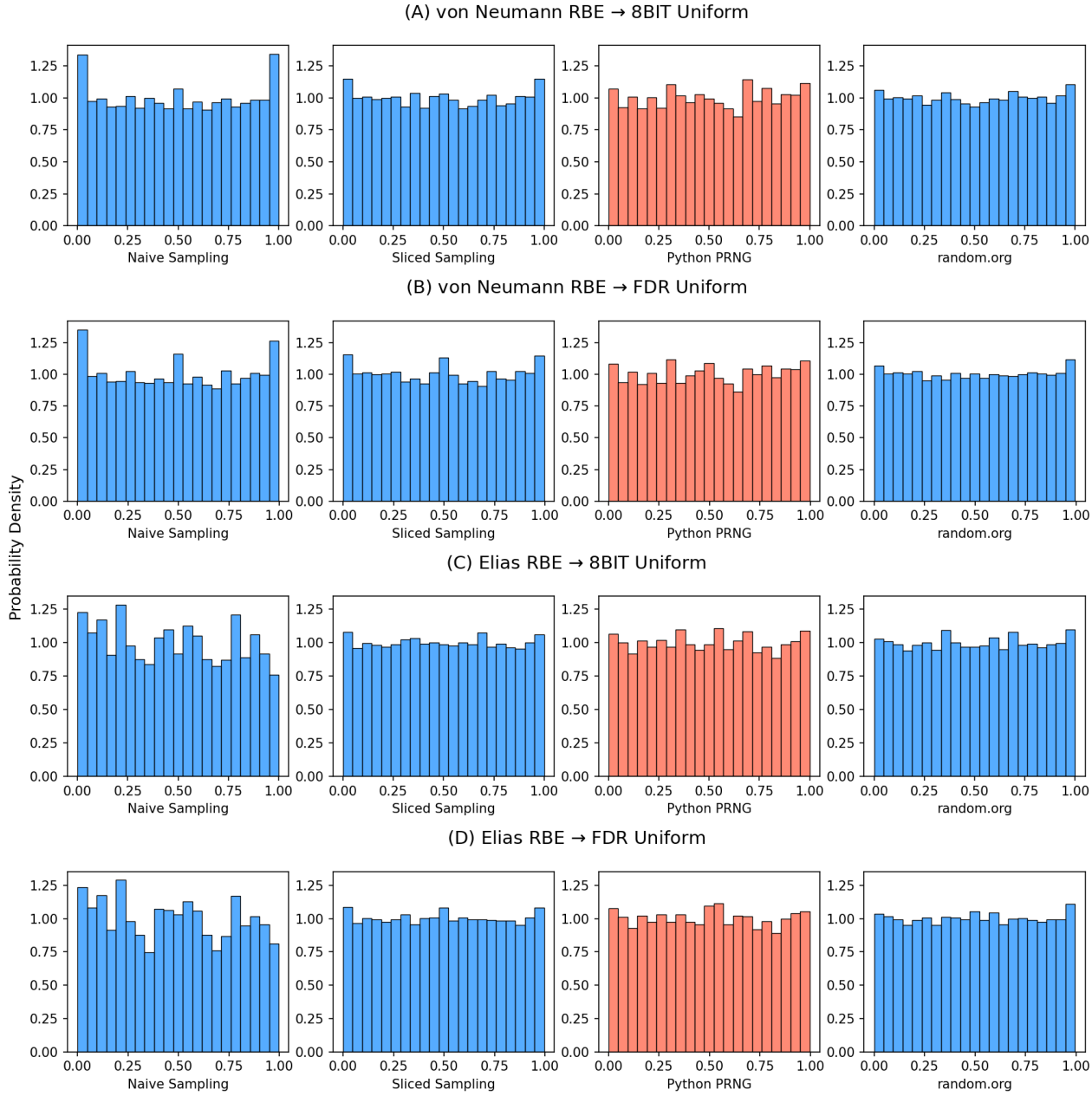

Figure 6: *Histograms of the result of applying Algorithm 5 (A) and Algorithm 6 (B) on randomly extracted bits via the von Neumann RBE. Histograms of the result of applying Algorithm 5 (C) and Algorithm 6 (D) on randomly extracted bits via the Elias RBE.*

3.2.2 Approximating Beta(1,1)

The uniform distribution $U[0,1]$ is a special case of the Beta distribution. To see this, let the random variable $X \sim \text{Beta}(\alpha, \beta)$. For $0 < x < 1$, the probability density function of the Beta distribution is defined as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)},$$

where $\Gamma(n) = (n-1)!$ is the gamma function.

Note that setting $\alpha = \beta = 1$ yields

$$f(x) = \frac{x^0(1-x)^0}{\Gamma(1)\Gamma(1)} = \frac{1(1)}{1(1)} = 1.$$

This is exactly the probability density function of a standard uniform random variable. We leverage this fact to generate uniform random variables. Termed "BETA", an implementation of this in Python is given in the appendix as Algorithm 4. Note that this method was inspired by a post from (whuber (https://stats.stackexchange.com/users/919/whuber), 2014).

3.2.3 Decimal Representation of Bytes

One algorithm we employ is simple but makes use of nearly all of the data. It takes 8-bit (or byte) blocks of the input bit stream and calculates the decimal number from the binary representation. The argument being that if every permutation from `00000000` to `11111111` is equally probable, then we should end up with a sample of length $n/8$ that is uniform in $[0, 255]$. We can then simply scale this down to the standard uniform by dividing each draw by 255. Termed "8BIT", Algorithm 5 is an implementation of this in Python and is given in the appendix. From 847,878 input bits (i.e the output of applying the Elias mapping on 1,048,576 pairs of IQ values), we found that it outputs 105,985 draws from the uniform in 0.7 seconds.

3.2.4 Fast Dice Roller

(Lumbroso, 2013) introduces a more advanced algorithm to draw from the uniform distribution, again, given a stream of random bits. Termed "FDR", Algorithm 6 is an implementation of this in Python and is given in the appendix. From the same 847,878 input bits, we found that it outputs 105,586 draws from the uniform (i.e. pretty much the same). This is because the rejection step on line 15 is used approximately 7 out of 8 times (leading to a sample of length around $n/8$). However, we find that it executes in just 0.09 seconds, making it over 7 times as fast as Algorithm 5. Thus, it is much more suited to applications such as live streaming random bits.

3.3 Process

We now detail the methodology from start to end to better clarify the whole process.

1. Sample atmospheric noise using an RTL-SDR device. The resulting IQ data is stored in complex-valued arrays. Both the real and imaginary components take values between -1 and 1.

2. Upon initial inspection, the standardised IQ values appear to be normally distributed around 0. Thus we employ methods that transform independant pairs of normally distributed random values to the standard uniform distribution. In particular, these are the Reverse Box-Muller and the Approximating Beta(1,1) methods described above. We find that, actually, these values are *not* normally distributed. The consequences of this violation are illustrated in Figure 5 and a further discussion is presented in section 4.

3. Therefore, we turn to methods that transform random bits (i.e. 1's and 0's) to the standard uniform distribution. To do this, we first take our standardised IQ values and turn both the real and imaginary parts into separate sequences of 1's and 0's by saying that:

$$\begin{cases} 1, \text{if } x > 0, \\ 0, \text{otherwise}, \end{cases}$$

where x is the value of some data point in the sample.

4. We then *de-bias* this sequence by applying a random bit extractor (we test both the von Neumann RBE and the Elias RBE). Table 1 shows that the efficiency of the Elias mapping is much greater than that of von Neumann's.

5. The de-biased random bits can then be used to produce draws from the standard uniform distribution by applying either the "8BIT" or "FDR" algorithms described above. Neither of these depend on an assumption of normality and therefore give better results. This is illustrated in Figure 6 and a further discussion is also presented in section 4.

Algorithm	input	output	efficiency
von Neumann	2,097,152	524,369	25.0%
Elias ($N = 4$)	2,097,152	847,878	40.4%

Table 1: The results of applying the von Neumann RBE and the Elias RBE on I and Q values.

3.4 NIST Test Suite

How do we know if the de-biased bits are random enough? To test the validity of the binary data output from an RBE, we can run the results through a battery of tests. One such example is the popular testing suite from the NIST (Bassham et al., 2010). For each test, we first define a test statistic calculated on the bit stream and then a hypothesis test is performed with the null hypothesis being that the data is indeed random. The p-value is reported and we say that can reject the null hypothesis if the value is less than 0.01 (i.e. a significance level of 99%). Note that this stricter than the typical case of using a 95% significance level in the sense that we require *more* evidence before we reject the null. This is because we are using a battery of 15 tests; meaning we care less about the sensitivity of each test but we care more about decreasing the chances of a false positive. Table 2 details the results.

The Python implementation of the test suite has been built by (Ang, 2017) and was imported from GitHub. As the work was released under the open source MIT licence, some adaptations were able to be made for it to work with our data and versions of various libraries. We briefly describe each test below and what it sets out to accomplish. Although this is covered in much greater detail in the paper by Bassham et al, we use a common notation between tests and

simplify the explanations somewhat. This encompasses describing the null hypothesis, the test statistic and the reference distribution from which we calculate the p-value.

Notation: ϵ refers to the n-bit stream of 1's and 0's that we are testing for randomness, i.e. $\{0,1\}^n$. η refers to a transformation of ϵ, where the 0's are replaced by -1's via $\eta_i = 2\epsilon_i - 1$, i.e. $\{-1,+1\}^n$. T refers to the test statistic. π_i refers to *expected probabilities* under the null hypothesis.

1. **Frequency (Monobit) Test**
   This is a simple test that checks to see if the proportion of 1's to 0's are the same. That is, the null hypothesis is that the number of 1's divided by the total number of bits is 0.5. To do this, we use η. Take the sum over η, say $S_n = \sum_{i=1}^{n} \eta_i$. Then the test statistic
   $$T = \frac{|S_n|}{\sqrt{n}}$$
   should be close to 0. The reference distribution for the test statistic is the half-normal (since we are looking at $|S_n|$ as opposed to S_n).

2. **Frequency Test within a Block**
   A more generalised version of the monobit test, this splits the bit stream into N lots of non-overlapping M-bit blocks (discarding any bits leftover at the end). The null hypothesis states that, if ϵ is random, then the frequency of 1's in each block is $M/2$. Note that for $M = 1$, it is the degenerate monobit case. To calculate the test statistic, first calculate the proportions
   $$\pi_i = \frac{\sum_{j=1}^{M} \epsilon_{(i-1)M+j}}{M},$$
   for $1 \leqslant i \leqslant N$. Then the test statistic
   $$T = 4M \sum_{i=1}^{N} (\pi_i - 0.5)^2$$
   should be compared against a $\chi^2(N)$ distribution to see if all the π_i's are close to 0.5.

3. **Runs Test**
   This test checks to see whether or not oscillations between 1's and 0's happen at the correct frequency. To illustrate this, suppose $\epsilon =$ `1 00 11 0 1 0 11`. By oscillation we mean a change from 0 to 1 or from 1 to 0. In this case, there are 7 oscillations. The null hypothesis states that, if ϵ is random, for a given ϵ of length n the frequency of oscillations is $n/2$. The test statistic can be formalised as
   $$T = \sum_{k=1}^{n-1} r(k) + 1 \qquad \text{where } r(k) = \begin{cases} 0, & \text{if } \epsilon_k = \epsilon_{k+1}, \\ 1, & \text{otherwise.} \end{cases}$$
   T being larger than expected implies that ϵ oscillates too fast to be random (and vice versa). The reference distribution for the test statistic is $\chi^2(n-1)$.

4. **Test for the Longest Run of Ones in a Block**
   This test first splits the bit stream into N lots of M-bit blocks and then checks for the longest run of 1's within each block. Given that our ϵ has over 750,000 bits, the authors suggest we use $M = 10,000$. The authors also describe that under the null hypothesis, we should see
   $$v_0 \leqslant 10, v_1 = 11, v_2 = 12, v_3 = 13, v_4 = 14, v_5 = 15, v_6 \geqslant 16,$$

where v_i is the longest run of ones (i.e v_3 refers to a block where 111 is the longest consecutive run of ones we see). Simply tally the v_i's for ϵ and calculate a p-value using a $\chi^2(6)$ test.

5. **Binary Matrix Rank Test**
   A test that originally appeared in the DIEHARD (Marsaglia, 1995) battery of tests, this splits ϵ into non-overlapping MQ-bit blocks and then forms an $M \times Q$ matrix from each block. The rank, say R_i, of each matrix is calculated and stored. In our case, we set $M = Q = 32$. The null hypothesis states that, if ϵ is random, we should find that the proportion of ranks to be approximately

$$\begin{cases} \pi_0 = 0.29, \text{ for } R = 32, \\ \pi_1 = 0.58, \text{ for } R = 31, \\ \pi_2 = 0.13, \text{ for } R \leqslant 30. \end{cases}$$

   The test statistic is then

$$T = \sum_{i=0}^{2} \frac{(v_i - N\pi_i)^2}{N\pi_i},$$

   where v_i is the total number of matrices which have rank R_i. The reference distribution for the test statistic is $\chi^2(3)$.

6. **Discrete Fourier Transform (Spectral) Test**
   This test performs a Discrete Fourier Transform (DFT) on ϵ and looks at the peak heights to check for periodicity. To do this we use η and then apply DFT to get, say $S = DFT(\eta)$. The null hypothesis states that, if ϵ is random, the number of peaks S has does not deviate more than 5% from the expected threshold. The test statistic is

$$T = \frac{N_1 - N_0}{\sqrt{n \log(1/0.05)}},$$

   where N_1 is the observed number of peaks of S and $N_0 = \frac{0.95n}{2}$ is the expected number of peaks under the null. The reference distribution for the test statistic is the normal.

7. **Non-overlapping Template Matching Test**
   This test also first splits ϵ into N lots of M-bit blocks. It then uses a sliding window of size m-bits to search for occurrences of a given bit pattern, say B, also of size m. Note that the authors show that B must be aperiodic. If the pattern is not found, we move the window ahead by 1 bit. If the pattern *is* found, make a note of it (by incrementing a counter, say W_i) and move the window ahead by m-bits (hence the non-overlapping). At the end, there will be N lots of W_i. The null hypothesis states that, if ϵ is random, the number of matches does not deviate from the theoretical mean μ and variance σ^2:

$$\mu = \frac{M - m + 1}{2^m}, \qquad \sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right).$$

   The test statistic is then

$$T = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}.$$

   As default, $B = \texttt{000000001}$ so that $m = 9$, and $N = 8$ so that $M = n/8$. The reference distribution for the test statistic is $\chi^2(N)$.

8. **Overlapping Template Matching Test**
   Similar to the previous test with the exception that, if a pattern *is* found, move the window ahead by 1-bit as opposed to m-bits (hence the overlapping). The authors also use $B = \texttt{111111111}$ as default instead. Note that the calculation of the test statistic is also slightly different and a detailed explanation of the difference can be found in the paper.

9. **Maurer's "Universal Statistical" Test**
   This test applies a simple (lossless) compression algorithm to ϵ to see how much it can be compressed without loss of information. The null hypothesis states that, if ϵ is random, then ϵ will not be significantly compressible. To do this, first split ϵ into 2 parts, say of length Q and P. One can consider these to be similar to a training and testing split (except here we have $Q < P$ where typically a machine learning scenario would have the opposite). We also define M as the length of bits we are pattern matching. The authors recommend that for ϵ of length less than 904,960, $M = 6$ should be used. Thus, split both Q and P blocks into further sub-blocks of size 6.

   The algorithm is "trained" or initialised on the Q block and a lookup table, say L, is created, attributing an integer for each possible M-bit string. This integer is the most recent sub-block where the pattern was found. For example, if the last time we encounter `000000` is in the 15th sub-block, then $L_0 = 15$. Then move onto `000001`; and if the last time we encounter it is in the 343th sub-block, then $L_1 = 343$. This continues until we've checked for all possible M-bit string combinations in Q. Finally, use the P block to "test" or apply the algorithm using said lookup table, where for each sub-block replace the 6 binary digits with the integer attributed to it. The test statistic is then

$$T = \frac{1}{P} \sum_{i=Q+1}^{Q+P} \log_2 (i - L_j),$$

   where L_j is the tabulated integer representation of the ith block. The reference distribution for the test statistic is the half-normal.

10. **Linear Complexity Test**
    After again splitting ϵ into N lots of M-bit blocks, this test uses the Berlekamp-Massey algorithm (Menezes et al., 1997) to calculate the linear complexity, say L_i, of each block. The null hypothesis states that, if ϵ is random, it will have a large linear complexity. To quantify what "large" means, first calculate the theoretical mean as

$$\mu = \frac{M}{2} + \frac{9 + (-1)^{M+1}}{36} - \frac{M/3 + 2/9}{2^M}.$$

    Now calculate the metric $T_i = (-1)^M (L_i - \mu) + 2/9$, for each block $i = 0, ..., N$. Then initialise a vector $\{v_0, ..., v_6\}$ with zeroes and record the number of times each value of T_i occurs according to the following rule:

$$\begin{aligned}
T_i &\leqslant 2.5, && v_0 \mathrel{+}= 1, \\
-2.5 < T_i &\leqslant -1.5, && v_1 \mathrel{+}= 1, \\
-1.5 < T_i &\leqslant -0.5, && v_2 \mathrel{+}= 1, \\
-0.5 < T_i &\leqslant 0.5, && v_3 \mathrel{+}= 1, \\
0.5 < T_i &\leqslant 1.5, && v_4 \mathrel{+}= 1, \\
1.5 < T_i &\leqslant 2.5, && v_5 \mathrel{+}= 1, \\
T_i &\geqslant 2.5, && v_6 \mathrel{+}= 1.
\end{aligned}$$

Finally, compute the test statistic

$$T = \sum_{i=0}^{6} \frac{(v_i - N\pi_i)^2}{N\pi_i},$$

where π_i are pre-calculated proportions under the null and are given in the paper. The reference distribution for the test statistic is $\chi^2(7)$.

11. **Serial Test**
This is another pattern matching test, but the calculation of the test statistic is more complicated and it checks all possible overlapping m-bit patterns. The null hypothesis is that, if ϵ is random, then the frequency of of each m-bit pattern is the same (i.e. each m-bit block in ϵ exhibits uniformity). Note that for $m = 1$, it is the degenerate monobit case. First, extend ϵ by appending the first $m - 1$ bits onto the end (this is so we can check all overlapping bit sequences without running into array size issues). Then, count the occurrences of each combination of m-bit patterns, (m-1)-bit patterns and (m-2)-bit patterns. Store these in 3 separate vectors, say x, y and z. Compute the metrics

$$\Psi^2_m = \frac{2^m}{n} \sum_{i=1}^{2^m} {x_i}^2 - n,$$
$$\Psi^2_{m-1} = \frac{2^{m-1}}{n} \sum_{i=1}^{2^{m-1}} {y_i}^2 - n,$$
$$\Psi^2_{m-2} = \frac{2^{m-2}}{n} \sum_{i=1}^{2^{m-2}} {z_i}^2 - n.$$

There are two test statistics for this method; in particular they are the differences

$$T_1 = \Psi^2_m - \Psi^2_{m-1}, \text{ and,}$$
$$T_2 = \Psi^2_m - 2\Psi^2_{m-1} + \Psi^2_{m-2}.$$

The reference distribution for both test statistics is χ^2. Large T_1 or T_2 would lead us to reject the null. Note that, for the purpose of this study, we report the smallest of the two p-values for consistency in output.

12. **Approximate Entropy Test**
The purpose of this test is to count the frequency of repeating patterns in ϵ, say of length m and $m+1$. Then, we can calculate the relative approximate Shannon entropy between the two. The null hypothesis is that, if ϵ is random, then the approximate entropy will be high. To do this, first tabulate all 2^m possible combinations and note down the proportion of times they occur in ϵ as π_i (by virtue of sliding a window of length m across bit by bit). For example, if $m = 3$; π_0 is the proportion of `000`, π_1 is the proportion of `001`, etc. Using this, compute

$$\phi_m = \sum_{i=0}^{2^m-1} \pi_i \log_e \pi_i.$$

Repeat for $m + 1$. The test statistic is then given by

$$T = 2n \left[\log_e 2 - (\phi_m - \phi_{m+1})\right].$$

The reference distribution for the test statistic is χ^2. Note that for ease of notation, the authors use indices that lead to $log_e(0)$, which of course is undefined. Here we take it to mean 0 in any of the sums it appears in.

13. **Cumulative Sums (Cusum) Test**
This is a simple test that calculates a random walk. To do this, we use η. The null hypothesis is that, if ϵ is random, the cumulative sum of η shouldn't deviate from 0 too much nor should it be too close to 0. To check this, compute the cumulative partial sums of η, say $S_n = \sum_{i=1}^{n} \eta_i$. For example; $S_1 = \eta_1$, $S_2 = \eta_1 + \eta_2$, etc. The test statistic is then

$$T = \max_{1 \leqslant k \leqslant n} |S_k|.$$

The reference distribution for the test statistic is the normal. It is interesting to note that very small values of T actually imply that the 1's and 0's are mixed in *too* evenly.

14. **Random Excursions Test**
This test builds upon the previous test. However, the procedure is a bit contrived so to better illustrate it, suppose

$$\eta = \{-1, 1, 1, -1, 1, 1, -1, 1, -1, 1\}.$$

Collate the partial cumulative sums into a list, say $S = [-1, 0, 1, 0, 1, 2, 1, 2, 1, 2]$, and then pad the list with 0 on either side, say $S' = [0, -1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 0]$. Define J as the number of total *zero crossings* in S'. By this, we mean the number of times the sequence S' starts at 0, deviates away, then comes back to 0. In our example, we have $J = 3$. Next, we define a *cycle* as a subset of S' with the zero crossings on each side; i.e. the three cycles in our example are: $\{0, -1, 0\}$, $\{0, 1, 0\}$ and $\{0, 1, 2, 1, 2, 1, 2, 0\}$. Then, tabulate the frequency of each value, say x, that is "visited" in each cycle; i.e. the number $x = 2$ is visited three times in cycle 3 but never in the other cycles. In particular, the authors suggest we restrict the search to just 8 values either side of 0, say $x_i = \{-4, -3, -2, -1, 1, 2, 3, 4\}$.

Finally, for each x_i, calculate the number of cycles in which x_i is visited exactly k times as $v_k(x_i)$. For example, $v_3(1)$ is the number of cycles in which the value $x = 1$ is visited exactly three times. In our example, it is just once (i.e. although cycle 2 also visits the value $x = 1$, it does so only once whereas we are looking for cycles where it is visited exactly 3 times). The null hypothesis is then, if ϵ is random, the proportion of visits to each x_i is the pre-calculated amount $\pi_k(x_i)$, where $\pi_k(x_i)$ is the probability that the number x_i is visited k times in a given cycle. In particular, there are 8 test statistics (for each x_i):

$$T_i = \sum_{k=0}^{5} \frac{(v_k(x_i) - J\pi_k(x_i))^2}{J\pi_k(x_i)},$$

Note that values for $k > 5$ are stored in $k = 5$ as they should be negligible. Thus, we get 8 different p-values from $\chi^2(6)$. As with the serial test, we report only the smallest.

15. **Random Excursions Variant Test**
This is a much simpler version of the test above. It is simply the frequency of how many times each value x is visited across the whole sequence η. However, due to the simplicity, we can dial up the extent of the search and instead check $x_i = \{-9, -8, ..., -1, 1, ..., 8, 9\}$ for a total of 18 counts, say $C(x_i)$. The test statistics are then these counts (standardised):

$$T_i = \frac{|C(x_i) - J|}{\sqrt{2J(4|x_i| - 2)}}.$$

Thus we use a half-normal distribution to calculate the p-values. One again, we report only the smallest.

4 Results

The standard uniform distribution has the following cumulative distribution function: $F(x) = 1$. The histograms of the draws from this distribution should therefore be a rectangular shape. However, row A in Figure 5 shows that the violation of the normality assumption has had a tangible impact on the expected results of Algorithm 3. The histograms from both the naively and sliced sampled input data instead display a positive linear relationship between x and $F(x)$. The value 0 is also drawn too many times. We can conclude that this method does not work for our samples as they are *not* normally distributed. However, the PRNG generated normal variates show that, given the assumptions do hold, reversing the Box-Muller procedure is indeed a valid way to produce uniform random variables. This is seen in the orange histogram; which displays a model shape of the standard uniform distribution.

The second row of plots in Figure 5 shows that, perhaps surprisingly, Algorithm 4 works quite well on the naive sampling data. The histogram of the draws are very evenly spread out; more so than even the PRNG input. Unfortunately, this algorithm does not work for the sliced sampling data. The distribution exhibits a convex shape, with the extreme values of x more likely than others. In fact, the distribution is more akin to draws from Beta(0.5,0.5) than Beta(1,1). Since we found that the naive samples are not free from autocorrelations, we cannot recommend their use even if they *appear* to be uniform through this procedure. Thus, we conclude that this method also does not work. Note that it is however, a much faster implementation (0.9 seconds versus 2 seconds for approximately 2 million samples) when compared to Algorithm 3. This is due to the fact that we do not need to calculate any computationally expensive exponentials.

Now, we examine Figure 6. Note that all four sources started as $\epsilon = \{0,1\}^n$ where $n = 1,048,576$. As described in Table 1, the von Neumann RBE outputs a stream of bits $\{0,1\}^{n_1}$, where $n_1 = 524,369$ and the Elias RBE outputs a stream of bits $\{0,1\}^{n_2}$, where $n_2 = 847,878$. Both row A and row B in the figure illustrate how the naive samples lead to too many draws from the extreme values 0 and 1, regardless of which transformation is used. Similarly, both row C and row D illustrate how the naive samples lead to a multi-modal distribution that clearly has three peaks. Thus, we conclude that the naive samples should not be used.

On the other hand, Figure 6 shows that the bits-to-uniform procedures were more fruitful than the normal-to-uniform procedures for the sliced samples. Both the von Neumann RBE and Elias RBE output bits random enough that the transformation to uniform look very similar to the same transform applied to the PRNG bits. Further, row D shows that the sliced samples are indistinguishable from the random.org samples (in terms of uniformity); which we are using as a benchmark in this observation. We can therefore conclude that using the Elias RBE and then using an FDR transform on IQ data that is sliced sampled is likely better than using an Python PRNG in scenarios where uniform draws are required. It *is* valid to point out, however, that we have not thoroughly examined the uniformity of these methods through any rigorous tests. A more confident recommendation can only be made if such tests are carried out. Due to time constraints, we only carry out a rigorous battery of tests on the input bit stream itself. Thus, we can confidently say whether or not the input is *random enough*; but we cannot say if they can be mapped to a distribution *uniform enough*.

4.1 NIST

The previous section helped to narrow down which bit streams should be ran through the NIST Statistical Test Suite. In particular:

- ϵ_0: using sliced sampling to get IQ data and then de-biasing this by applying the Elias RBE to extract bits from Bernoulli(0.5).
- ϵ_1: using random.org to provide bits from Bernoulli(0.5).
- ϵ_2: using a Mersenne Twister implementation from Python's random package to extract pseudorandom draws from Bernoulli(0.5).

	ϵ_0		ϵ_1		ϵ_2	
	p-value	result	p-value	result	p-value	result
`monobit`	0.913519	True	0.638958	True	0.554662	True
`block_frequency`	0.981378	True	0.636752	True	0.328045	True
`runs`	0.537323	True	0.100181	True	0.751444	True
`longest_one_block`	0.293232	True	0.284535	True	0.75882	True
`binary_matrix_rank`	0.625951	True	0.460013	True	0.984521	True
`spectral`	0.497658	True	0.872918	True	0.151404	True
`non_overlapping_test`	0.0902411	True	0.851104	True	0.423234	True
`overlapping_patterns`	0.253469	True	0.739162	True	0.339093	True
`statistical`	0.0288939	True	0.236016	True	0.431989	True
`serial`	0.529415	True	0.461569	True	0.656086	True
`approximate_entropy`	0.0630514	True	0.625997	True	0.967476	True
`cumulative_sums`	0.355394	True	0.744536	True	0.945597	True
`random_excursions_test`	0.053688	True	0.0683624	True	0.570841	True
`variant_test`	0.0192774	True	0.0118	True	0.185021	True

Table 2: The results of the NIST Statistical Test Suite for three different input bit-streams ϵ_0, ϵ_1 and ϵ_2. All bit-streams passed (p-values > 0.01) every test from the suite.

4.2 Conclusion

Table 2 shows the results of running all three bit-streams through the NIST Statistical Test Suite. We also report the p-values as defined in subsection 3.4. Although, at first, the fact that the bit-streams passed every test thrown at them may seem underwhelming, this is actually a great result for our experiment. This means that our transformed IQ data is *as least* as random as the Python PRNG and indeed the industry standard that is random.org. Tests 9 (Maurer's "universal statistical") and 15 (random excursions variant) are the only ones that come close to failing ϵ_0; with p-values just above 0.01. The latter of which almost fails ϵ_1 also.

In the long run, we will be better off financially since the only cost of generating ϵ_0 is the initial investment into the RTL-SDR receiver (assuming you have already have a machine with USB ports and are able to run Python 3 on it). The particular device we used only cost £22.29. Since the data from random.org costs $5 per 1,048,576 bits, this means that at the current exchange rate (1.38:1), it would only take 7 uses before our method becomes cheaper. Furthermore, one could also invest in a more sophisticated device which allows more control over the settings. A slower sample rate, for example, would help greatly with gathering samples without any autocorrelations. RTL-SDR devices with higher sensitivities also allow the user to sample from greater range of frequencies and with different bandwidths (so one could theoretically find a band that had no powerful signals, i.e. only atmospheric noise). One could also sample from higher altitudes (closer to the ionosphere) or from locations that suffer from frequent storms (prevalent with lighting) since the setup is portable.

The argument against using the PRNG is even simpler; we know ϵ_0 *is* actually random and that ϵ_2 is not. Theoretical results and their applications (such as the popular Monte-Carlo simulation method) that depend on the assumption of randomness could fail if a PRNG is used. Even the main argument for using PRNGs (reproducibility) is moot since we can save the IQ values to a file in order to reproduce the same draws. This is illustrated in the Github documentation. If new numbers are needed, we can simply re-sample and produce more random draws. It is important to note that for many of the tests, the reported p-values for the PRNG are far larger than those of either TRNGs. This is likely because Python's `random` library has been purposefully engineered to pass such testing suites and the fact that our TRNG has also passed is a testament to the validity of the randomness. As (L'Ecuyer, 2010) states in the International Encyclopedia of Statistical Science, we shouldn't trust blindly that the default RNG in our software of choice will always lead to the result we desire. By sub-classing the method we provide, users can replace the underlying Mersenne Twister generator in `random` to instead get true random numbers in all of their applications.

5 Limitations and Improvements

The scope of this project was quite large and therefore there is plenty of room for improvement and indeed further testing. For example, the main limitation we faced was with the gathering of the samples themselves. Firstly, the hardware we used was fairly cheap and therefore a proof of concept rather than a complete end product. We would like to repeat the experiments with a device that allowed for much finer tuning of the settings. Perhaps the best way to improve the quality of the input bits is to use a transmitter in conjunction with the receiver. Then, we could transmit a signal ourselves (i.e. a user could just speak into a microphone) and receive it on the other end. This could allow us to completely remove the original signal and be left with just the atmospheric noise. Secondly, more research and understanding of digital signal processing would have helped with the software side of the sampling process. There may exist a methodology for isolating noise with the device we have already or theorems about which frequency to sample at, for example.

Also, as mentioned previously, the ideal end product would have been on a mobile phone and as some sort of downloadable application. This would vastly improve the ease of access to the end user. We envisioned a product where a researcher could just use their phone to gather IQ samples and send the random uniform draws as a file to their computer; making the process software independant. This could also be an application that runs in the background (with the consent of the user, of course) collecting entropy, allowing for the use of a much larger slicing index k. Thus, eliminating any autocorrelations (especially if the user travels around with their mobile phone over the period of several days or weeks) and then having a pre-recorded sample ready to go. However, as mobile phone manufacturers are unlikely to go back to including receivers that are easily addressable by API, an alternative would be to use a Raspberry Pi device. A researcher could mimic the setup and leave the device running to collect data over a long period of time. Unfortunately, this would then remove the benefit that comes with portability.

In terms of the experiment itself, it could have been useful to visualise the effect of different values of k. A plot of the number of autocorrelations present against the time taken to sample the IQ data would have allowed us to determine if there are diminishing returns to increasing k - and perhaps even find an optimal k. We also only explored Elias' RBE algorithm with $N = 4$. Different values of N may lead to even more efficient mappings when de-biasing the input bits. Furthermore, we could employ a test of uniformity on the final result to quantify

any differences between our method, Python's PRNG and random.org.

The NIST test suite itself has some tests (e.g. tests 7 and 8) that attempt to count the number of times a certain 8-bit pattern, say $B =$00000000, is observed. The current implementation does not check all possible combinations of B. It would have been useful to re-run those tests with all possible combinations of B and even with different lengths. Similarly to how we presented tests 14 and 15, we could have extended the code to calculate the test statistic for all B's and just report the smallest p-value from this set. Finally, there are other (albeit less well-documented) battery of tests out there that could provide more insight into the difference between the randomness of our bit-streams.

6 References

References

Ang, S. K. (2017). randomness_testsuite. https://github.com/stevenang/randomness_testsuite.

Bassham, L., A. Rukhin, J. Soto, J. Nechvatal, M. Smid, S. Leigh, M. Levenson, M. Vangel, N. Heckert, and D. Banks (2010, 2010-09-16). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.

Bianchi, C. and A. Meloni (2007, June). Natural and man-made terrestrial electromagnetic noise.

Box, G. and G. M. Jenkins (1976). Time Series Analysis: Forecasting and Control.

Box, G. E. P. and M. E. Muller (1958). A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics 29*(2), 610 – 611.

Collins, T. F., R. Getz, D. Pu, and A. M. Wyglinski (2018). Software Defined Radio for Engineers.

Elias, P. (1972). The Efficient Construction of an Unbiased Random Sequence. *The Annals of Mathematical Statistics 43*(3), 865 – 870.

Haahr, M. (1998). RANDOM.ORG: True Random Number Service. Accessed on 08.08.2021.

Hastings, W. K. (1970, 04). Monte Carlo sampling methods using Markov chains and their applications.

Hoeffding, W. and G. Simons (1970). Unbiased Coin Tossing with a Biased Coin. *The Annals of Mathematical Statistics 41*(2), 341 – 352.

Hum, S. V. (2018). Radio and Microwave Wireless Systems. Accessed on 08.08.2021.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment.

Indra Mohan Chakravarti, R. G. Laha, J. R. (1967). Handbook of Methods of Applied Statistics. Volume I.

Intel (2014). Intel Digital Random Number Generator (DRNG) Software Implementation Guide, Revision 2.0.

ITU (1984). European VHF/UHF Broadcasting Conference Geneva.

Kopp, H. (2020). Attacking a random number generator. Accessed on 08.08.2021.

Krhovjak, J., P. Svenda, and V. Matyas (2007, 01). The sources of randomness in mobile devices.

Lapidoth, A. (2016). A Foundation in Digital Communication.

L'Ecuyer, P. (2010). Uniform random number generators.

Li, A. (2013). Potential Weaknesses In Pseudorandom Number Generators.

Lichtman, M. (2021). IQ Sampling.

Lumbroso, J. (2013). Optimal Discrete Uniform Generation from Coin Flips, and Applications.

Marsaglia, G. (1995). The marsaglia random number cdrom: including the diehard battery of tests of randomness.

Matsumoto, M. and T. Nishimura (1998, January). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.

Menezes, A., P. van Oorschot, and S. Vanstone (1997). Handbook of Applied Cryptography.

Ofcom (2010). Frequency Allocation Table. Accessed on 19.08.2021.

Peres, Y. (1992, 03). Iterating Von Neumann's Procedure for Extracting Random Bits.

Petitcolas, F. A. P. (2011). Kerckhoffs' Principle.

Reback, J., W. McKinney, and jbrockmendel (2020, February). pandas-dev/pandas: Pandas.

Route, M. (2017, Aug). Radio-flaring Ultracool Dwarf Population Synthesis.

Ryabko, B. and E. Matchikina (2000). Fast and efficient construction of an unbiased random sequence.

Shannon, C. E. (1948). A mathematical theory of communication.

Shannon, C. E. (1949, jan). Communication in the Presence of Noise.

Shema, M. (2012). Chapter 7 - Leveraging Platform Weaknesses.

Suciu, A., D. Lebu, and K. Marton (2011, 08). Unpredictable random number generator based on mobile sensors.

Tan, Y. (2016). Chapter 10 - GPU-Based Random Number Generators.

Trevisan, L. and S. Vadhan (2000). Extracting Randomness from Samplable Distributions. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, USA, pp. 32. IEEE Computer Society.

Vadhan, S. (2012, December). Pseudorandomness.

von Neumann, J. (1951). Various Techniques Used in Connection with Random Digits. In A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.), *Monte Carlo Method*, Volume 12 of *National Bureau of Standards Applied Mathematics Series*, Chapter 13, pp. 36–38. US Government Printing Office.

Walker, J. (2006). Genuine random numbers, generated by radioactive decay. Accessed on 08.08.2021.

whuber (https://stats.stackexchange.com/users/919/whuber) (2014). Simulating draws from a Uniform Distribution using draws from a Normal Distribution. Cross Validated. Accessed on 29.08.2021.

Wood, G. (2018). Ethereum: A Secure decentralised generalised transaction ledger.

7 Appendix

Algorithm 1 Mersenne Twister adapted from "GPU-Based Random Number Generators, Tan 2016". This illustrates how the initial state variables govern the result of this deterministic algorithm. Figuring them out allows one to generate current and future values exactly.

1. Set r w-bit numbers $(x_i, i = 1, 2, ..., r)$ as initial values.
2. Set t_1, t_2, t_3, and t_4 as integers and m_1 and m_2 as bit-masks.
3. Let $\begin{pmatrix} 0 & I_{w-1} \\ a_w & a_{w-1}...a_1 \end{pmatrix}$, where I_{w-1} is the $(w-1)\times(w-1)$ identity matrix and $a_i, i = 1, ..., w$ take values of either 0 or 1.
4. Define $x_{i+r} = (x_{i+s} \oplus cA)$ where $c = x_i^{w:(l+1)} | x_{i+1}^{l:1}$ indicates the concatenation of the most significant (upper) $w - l$ bits of x_i and the least significant l bits of x_{i+1}.
5. Compute u_{i+r} as

$$\begin{aligned} z &= x_{i+1} \oplus (x_{i+1} \gg t_1), \\ z &= z \oplus ((z \ll t_2) \text{ AND } m_1), \\ z &= z \oplus ((z \ll t_3) \text{ AND } m_2), \\ z &= z \oplus (x \gg t_4), \\ u_{i+1} &= z/(2^w - 1), \end{aligned}$$

   where AND is the binary logical operator.
6. This generates pseudorandom variables $u_{i+r}, i = 1, 2, ...,$ from the standard uniform distribution

---

Algorithm 2 "Sliced Sampling"

1. Define f_c, the carrier frequency and k.
2. (a) Take a sample of length 2^{20} (approx 1 million).
   (b) Take every kth element and discard the rest.
3. Repeat step 2 until we have 2^{20} (approx 1 million) samples.

Algorithm 3 Reversing the Box-Muller method to generate draws from $U[0,1]$ given draws from $N(0,1) \times N(0,1)$. [Python v3.8.5, Numpy v1.21.2]

```
def uniforms_from_RBM(x,y):
    """
    Reverses the Box-Muller method to return uniform random variables from
    pairs of independent standard normals.
    Arguments:
     - x (array of floats, length same as y)
     - y (array of floats, length same as x)
    Returns:
     - uniforms (array of floats)
    """
    import numpy as np
    if len(x) != len(y):
        print("Error: len(x) not the same as len(y).")
    else:
        # scale to standard normal
        x_mu = np.mean(x)
        x_sigma = np.std(x)
        y_mu = np.mean(y)
        y_sigma = np.std(y)
        x_standard = (x-x_mu)/x_sigma
        y_standard = (y-y_mu)/y_sigma
        # reverse Box-Muller
        uniforms = []
        for i in range(len(x)):
            u = np.exp(-1*(x_standard[i]**2+y_standard[i]**2)/2)
            uniforms.append(u)
        return uniforms
```

Algorithm 4 Approximating Beta(1,1) to generate draws from $U[0,1]$ given draws from $N(0,1) \times N(0,1)$. [Python v3.8.5, Numpy v1.21.2]

```
def uniforms_from_BETA(x,y):
    """
    Uses the fact that the uniform distribution is a special case of the
    Beta distribution to generate uniform random variables.
    Arguments:
     - x (array of floats, length same as y)
     - y (array of floats, length same as x)
    Returns:
     - uniforms (array of floats)
    """
    import numpy as np
    if len(x) != len(y):
        print("Error: len(x) not the same as len(y).")
    else:
        # scale to standard normal
        x_mu = np.mean(x)
        x_sigma = np.std(x)
        y_mu = np.mean(y)
        y_sigma = np.std(y)
        x_standard = (x-x_mu)/x_sigma
        y_standard = (y-y_mu)/y_sigma
        # square to get chi^2
        chi2 = []
        for i in range(len(x)):
            z = x_standard[i]**2 + y_standard[i]**2
            chi2.append(z)
        # form beta RVs
        uniforms = []
        for i in range(len(chi2)):
            try:
                u = chi2[2*i]/(chi2[2*i-1] + chi2[2*i])
                uniforms.append(u)
            except:
                pass
        return uniforms
```

Algorithm 5 "8BIT": generate draws from $U[0,1]$ given draws from $\{0,1\}^n$. [Python v3.8.5, Numpy v1.21.2]

```
def uniforms_from_8BIT(bits):
    """
    This algorithm splits an input list into 8-bit blocks
    and converts each to the decimal representation.
    Arguments:
     - bits (a list of random bits, 1 or 0)
    Returns:
     - uniforms (a list of reals in Uniform[0,1])
    """
    import numpy as np
    # split into 8 bit chunks
    length = 8 * round(len(bits) / 8)
    # exclude the end if not exactly divisible by 8
    bits = bits[:length]
    bitlist = np.array_split(bits, length / 8)
    # generate uniforms
    uniforms = []
    for j in bitlist:
        uniforms.append(int(''.join([str(i) for i in j]),2) / 255)
    return uniforms
```

Algorithm 6 "FDR": generate draws from $U[0,1]$ given draws from $\{0,1\}^n$. [Python v3.8.5]

```
def fdr(n, flip):
    """
    The Fast Dice Roller algorithm adapted from the pseudocode
    given in Lumbroso, 2013.
    Arguments:
     - n (the range of uniform integers we wish to generate)
     - flip (an iterable that returns 1 or 0 when called)
    Returns:
     - c (an integer in Uniform[0,n])
    """
    v, c = 1, 0
    while True:
        v = 2 * v
        c = 2 * c + next(flip)
        if v >= n:
            if c < n:
                return c
            else:
                v = v - n
                c = c - n

def uniforms_from_FDR(bits):
    """
    Use the FDR algorithm to generate uniforms from a random bit source.
    Arguments:
     - bits (a list of random bits, 1 or 0)
    Returns:
     - uniforms (a list of reals in Uniform[0,1])
    """
    iterator = iter(bits)
    enough_bits = True
    uniforms = []
    while enough_bits:
        try:
            k = fdr(255, iterator)
            uniforms.append(k)
        except:
            enough_bits = False
    uniforms = [i / 255 for i in uniforms]
    return uniforms
```