TESTING ‘RANDOMNESS’ OF RANDOM NUMBER GENERATORS

MA-202 2022-23

Arihant Jain 21110032
Gaurav Shah 21110064
Husain Malwat 21110117
Parth Shah 21110152
Shreyansh Murathia 21110201
Vikash Beniwal 21110238

Advised By: Prof. Joycee Mekie

Abstract:

Randomness occurs when events or series of events happen without any pattern, and we cannot predict them in advance. Natural examples of randomness include nuclear decay, and true random processes can only be generated using hardware random generators, which generate random numbers based on physical processes such as quantum-mechanical phenomena. In contrast, everyday computing systems, algorithmic settings, and cryptography rely on pseudo-random numbers. Various methods of PRNGS are used in software, such as Linear Congruential Generator, Mersenne Twister, and Pseudo Random Number Generator, which use mathematical formulas to approximate random numbers. They typically start with a seed value and perform operations to generate sequences of seemingly random numbers. The article then discusses four sources that use different methods to generate random numbers, including the Python-random function, PUF-based code using hardware, Python-secrets, and a true random generator that uses atmospheric noise.

Random number generators find wide application in fields where security is a key requirement. Cybersecurity, cryptography, and IOT are such crucial places where we require a non-predictable key for protection.

1. Introduction:

1.1 True Vs. Pseudo-Random Number Generators

Random Number Generators(RNGs) are sequence generators that use a nondeterministic source (i.e., the entropy source), along with some processing function (i.e., the entropy distillation process) to produce randomness. The use of a distillation process is needed to overcome any weakness in the entropy source that results in the production of non-random numbers (e.g., the occurrence of long strings of zeros or ones). The noise in an electrical circuit, the timing of user processes (e.g keystrokes or mouse movements), or the quantum effects in a semiconductor are usually used as entropy sources. This is the working principle of RNGs.

However, in our realization, most RNGs are not truly random. The output of any RNG needs to satisfy strict randomness criteria as measured by statistical tests in order to determine that the physical sources of the RNG inputs appear random. For example, a physical source such as electronic noise may contain a superposition of regular structures, such as waves or other periodic phenomena, which may appear to be random, yet are determined to be non-random using statistical tests. This is a major flaw existing in the world of computer science.

The primary criteria for a sequence to be qualified as ‘Truly Random’ is-

All elements of the sequence are generated independently of each other, and the value of the next element in the sequence cannot be predicted, regardless of how many elements have already been produced.

Such an expectation is not practically realizable and thus RNGs can only get close to becoming truly random but it is impossible to be truly random.

The main objective of RNGs is to fulfill the property of Forward Unpredictability. It implies that the next output number in the sequence should be unpredictable in spite of any knowledge of previous random numbers in the sequence. The biggest application of RNGs is for cryptographic applications. For example, common cryptosystems employ keys that must be generated in a random fashion. Many cryptographic protocols also require random or pseudo random inputs at various points, e.g., for auxiliary quantities used in generating digital signatures, or for generating challenges in authentication protocols.

Other applications include science experiments, art (such as synthesizers for music production), statistical analysis of data, cryptography, gaming, gambling, and other fields.

1.2 RNGs used:

(a) Random module from Python:

This module implements a pseudo-random number generator by using the Mersenne Twister algorithm. Mersenne twister has a long period of 2^{19937}-1 before it repeats itself.
The code to generate a random sequence of 0 and 1 bits uses `random.choice()` which selects either 0 or 1 and repeats this process 1.5 million times to generate a sequence.[1]

(b) Secrets module from Python:

The Mersenne Twister is not cryptographically secure [2]. It keeps track of 624 32-bit values, and if an attacker can gather 624 sequential values, then the entire sequence can be predicted. Therefore, python produces a 'secrets' module which provides cryptographically secure sequences.
Similar to the random module, we use `secrets.choice()` to generate a sequence.

(c) PUF-based random number generator:

PUF(Physical Unclonable Functions), are hardware random number generators that extract randomness directly from complex physical systems.[3]

(d) Random number database from Random.org:

The bytes generated are from atmospheric noise. The bytes are then concatenated to form a bitstream.

1.3 Testing methodology to find 'randomness' of RNGs:

Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability. A statistical test is formulated to test a specific null hypothesis (H0), which is that the sequence being tested is random. The alternate hypothesis (Ha) would be that the sequence is not random.

Each test tries to accept or reject the null hypothesis. To do that, each test has a randomness statistic, which under an assumption, gives a distribution of likely values. A critical value is then determined (at 99% significance level for the test). If the test statistic exceeds the critical value, then the null hypothesis is rejected, and otherwise accepted.

1.4 Most commonly used terminologies in the tests

(a) Entropy: A measure of the disorder or randomness in a closed system. The entropy of uncertainty of a random variable X with probabilities p_1, …, p_n is defined to be

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i$$

(b) Erfc: The complementary error function erfc(z)

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} e^{-u^2} du$$

(c) Incomplete Gamma Function: The incomplete gamma function Q(a,x)

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} = \frac{1}{\Gamma(a)} \int_{0}^{\infty} t^{z-1} e^{-t} dt$$

(d) Level of Significance (α): The probability of falsely rejecting the null hypothesis, i.e., the probability of concluding that the null hypothesis is false when the hypothesis is, in fact, true.

(e) P-value: The probability (under the null hypothesis of randomness) that the chosen test statistic will assume values equal to or worse than the observed test statistic value when considering the null hypothesis.

2. Test Implementations and Results:

Frequency (Monobit) Test:

This test checks to see if a sequence has roughly the same amount of ones and zeros as would be anticipated for a really random sequence. The test determines whether the fraction of ones is near to 0.5, meaning that there should be roughly equal numbers of ones and zeros in a sequence. All subsequent tests are based on the result of this test.

The variables used in the Test:

n - The length of the bit string

Decision Rule: P-Value. If P-value < 0.01, the sequence is classified as non-random, and the test FAILS.

Test Implementation:

1. Conversion to ±1: The zeros and ones of the input sequence are converted to values of –1 and +1 and are added together to produce Sn (‘difference’ as per the code).

```
   def __init__(self):
        # Generate base Test class
        super(MonobitTest, self).__init__("Monobit", 0.01)

    def _execute(self,
                 bits: numpy.ndarray):
        """
        Overridden method of Test class: check its docstring for further
information.
        """
        # Compute ones and zeroes
        ones: int = numpy.count_nonzero(bits)
        zeroes: int = bits.size - ones
        # Compute difference
        difference: int = abs(ones - zeros)
```

2. Compute the test statistic,

$$s_{obs} = \frac{|S_n|}{\sqrt{n}}$$

3. Calculate the P-value,

$$P\text{-}value = \mathbf{erfc}\left(\frac{s_{obs}}{\sqrt{2}}\right)$$

```
        # Compute score
        score: float = math.erfc(float(difference) /
(math.sqrt(float(bits.size)) * math.sqrt(2.0)))
        # Return result
        if score >= self.significance_value:
            return Result(self.name, True, numpy.array(score))
        return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

After executing the test for the four random number datasets, the results were :-

PUF Random Dataset - *P-value = 0.376, Pass*

Python Random Dataset - *P-value = 0.407, Pass*

Python Secrets Dataset - *P-value = 0.02, Pass*

True Random Dataset - *P-value = 0.417, Pass*

Thus, we conclude that all selected datasets passed the Monobit test. All selected datasets can be declared 'Random' according to this test. The proportion of zeros to ones in all datasets is close to 1. In case of a Fail result, this test is a clear indication that the dataset is not random and further tests need not be performed.

Frequency Test within a Block:

The focus of this test is the proportion of ones within M-bit blocks. The purpose of this test is to determine whether the frequency of ones in an M-bit block is approximately M/2, as would be expected under an assumption of randomness.

The variables used in the Test:

M - The length of each block
n - The length of the bit string

Decision Rule: P-Value. If P-value < 0.01, the sequence is classified as non-random, and the test FAILS.

Test Implementation:

1. Partition the input sequence into N = $\left\lfloor \frac{n}{M} \right\rfloor$ n-overlapping blocks. Discard any unused bits.

   Here, we have used 2 functions, one is for defining the size of the block and other is for executing the test. Extremum limits have been applied such that the test always remains within the range of functionality.

```
def __init__(self):
        # Define specific test attributes
        self._sequence_size_min: int = 100
        self._default_block_size: int = 20
        self._blocks_number_max: int = 100
        # Define cache attributes
        self._last_bits_size: int = -1
        self._block_size: int = -1
        self._blocks_number: int = -1
        # Generate base Test class
        super(FrequencyWithinBlockTest, self).__init__("Frequency Within
Block", 0.01)

    def _execute(self,
                 bits: numpy.ndarray) -> Result:
        """
        Overridden method of Test class: check its docstring for further
information.
        """
        # Reload values is cache is empty or no longer up-to-date
        # Otherwise, use cache
        if self._last_bits_size == -1 or self._last_bits_size !=
bits.size:
            # Get the number of blocks (N) with the default minimum block
size (M)
            block_size: int = self._default_block_size
            blocks_number: int = int(bits.size // block_size)
            # Get the block size (M) if the number of blocks (N) exceed
the allowed max
```

```
            if blocks_number >= self._blocks_number_max:
                blocks_number = self._blocks_number_max - 1
                block_size = int(bits.size // blocks_number)
            # Save in the cache
            self._last_bits_size = bits.size
            self._block_size = block_size
            self._blocks_number = blocks_number
        else:
            block_size: int = self._block_size
            blocks_number: int = self._blocks_number
```

2. Determine the proportion π_i of ones in each M-bit block using the equation

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)M+j}}{M}, \quad \text{for } 1 \le i \le N.$$

```
  # Initialize a list of fractions
        block_fractions: numpy.ndarray =
numpy.zeros(blocks_number, dtype=float)
        for i in range(blocks_number):
            # Get the bits in the current block
            block: numpy.ndarray = bits[i * block_size:((i + 1) *
block_size)]
            # Compute ones and save the fraction in the array
            block_fractions[i] = numpy.count_nonzero(block) /
block_size
```

3. Compute the χ^2 tatistic: $\chi^2(obs) = 4\,M \sum_{i=1}^{N} (\pi_i - ½)^2.$

```
   # Compute Chi-square
        chi_square: float = numpy.sum(4.0 * block_size *
((block_fractions[:] - 0.5) ** 2))
```

$$(N/2,\ \chi^2(obs)/2)$$

4. Compute P-value = igame

```
      # Compute score (P-value) applying the lower incomplete
gamma function
        score: float = scipy.special.gammaincc((blocks_number /
2.0), chi_square / 2.0)
        # Return result
        if score >= self.significance_value:
            return Result(self.name, True, numpy.array(score))
        return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

After executing the test for the four random number datasets, the results were :-

PUF Random Dataset - *P-value = 0.0, Fail*

Python Random Dataset - *P-value = 0.472, Pass*

Python Secrets Dataset - *P-value = 0.851, Pass*

True Random Dataset - *P-value = 0.843, Pass*

Thus, we conclude that except for the PUF Random Dataset, all tests passed the test. This reflects that there were large deviations from equal proportions of ones and zeros in many blocks of the PUF Random Dataset. On the other hand, the rest of the datasets do not have any large variations in blocks thus they have passed this test. Thus, we can be sure that except the PUF Random Dataset, all datasets are ‘Random’ according to this test.

Runs Test:

The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits. A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence. In particular, this test determines whether the oscillation between such zeros and ones is too fast or too slow

The variables used in the Test:

n - The length of the bit string

Decision Rule: P-Value. If P-value < 0.01, the sequence is classified as non-random, and the test FAILS.

Test Implementation:

1. Compute the pre-test proportion π of ones in the input sequence: $\pi = \frac{\sum_j \varepsilon_j}{n}$.

```
            def __init__(self):
        # Generate base Test class
        super(RunsTest, self).__init__("Runs", 0.01)

    def _execute(self,
                 bits: numpy.ndarray) -> Result:
        """
        Overridden method of Test class: check its docstring for
further information.
        """
        proportion: float = numpy.count_nonzero(bits) / bits.size
```

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1 ,$$

2. Compute the test statistic, where r(k)=0 if $\varepsilon_k=\varepsilon k+1$, and r(k)=1 otherwise.

```
 # Count the observed runs (list of adjacent equal bits)
        observed_runs: float = 1.0
        for i in range(bits.size - 1):
            if bits[i] != bits[i + 1]:
                observed_runs += 1.0
```

3. Compute P-value = $erfc\left(\frac{|V_n(obs)-2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$.

```
        # Compute score (P-value)
        score: float = math.erfc(abs(observed_runs - (2.0 *
bits.size * proportion * (1.0 - proportion))) / (2.0 *
math.sqrt(2.0 * bits.size) * proportion * (1 - proportion)))
        # Return result
        if score >= self.significance_value:
            return Result(self.name, True, numpy.array(score))
        return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

After executing the test for the four random number datasets, the results were :-

PUF Random Dataset - P-value = 0.0, Fail

Python Random Dataset - P-value = 0.201, Pass

Python Secrets Dataset - P-value = 0.64, Pass

True Random Dataset - P-value = 0.738, Pass

Thus, we conclude that except for the PUF Random Dataset, all tests passed the test. This reflects that there were too few oscillations (changes from a string of zeros to string of ones or vice versa) in the PUF Random Dataset. On the other hand, the rest of the datasets have very high oscillations leading them to becoming more ‘random’ in comparison with the PUF Random Dataset. Thus, except for the PUF Random Dataset, all datasets are ‘Random’ according to this test.

A major real life application of this test is in the field of Trading where Technical traders can use a runs test to analyze statistical trends and help spot profitable trading opportunities.

Linear Complexity Test:

The test divides the bitstream (n) into N blocks of M bits (n=MN). Then, it tries to find the length of the fixed LSFR (Linear Fixed Shift Register), L_i (i=1,2…N). The randomness of the stream is characterized by a longer LFSR.

The variables used in the Test:

M- length of bits in a block
n- length of the bitstream
ε - Sequence of bits generated by the random number generator(RNG)
K - Degrees of freedom (K=6 for the test)

Test Statistic: χ^2 (obs), Measure how well the observed occurrences match the expected number under an assumption of randomness.

Decision Rule: P-Value (computed using χ and K). If P-value < 0.01, the sequence is classified as non-random, and the test FAILS.

Test Implementation:

1. Partition the n-bit sequence into N-independent blocks of M bits.

```
def __init__(self):
        # Define specific test attributes
        self._sequence_size_min: int = 1000000
        self._pattern_length: int = 512
        self._freedom_degrees: int = 6
        self._probabilities: numpy.ndarray = numpy.array([0.010417,
0.03125,
        0.125,0.5, 0.25, 0.0625, 0.020833])
```

Initialisation and parameters are set [1]

2. Use the Berlekamp-Massey algorithm and determine the linear complexity L_i (length of the shortest LFSR sequence that generates all bits in the block i) for each N block. Within any L_i bit sequence, some combination of the bits, when added together modulo 2, produces the next bit in the sequence (bit Ly + 1)

```
def _berlekamp_massey(sequence: numpy.ndarray) -> int:
        """
```

```
        Compute the linear complexity of a sequence of bits by the means of the
Berlekamp Massey algorithm.
        :param sequence: the sequence of bits to compute the linear complexity
for
        :return: the int value of the linear complexity
        """
        # Initialize b and c to all zeros with first element one
        b: numpy.ndarray = numpy.zeros(sequence.size, dtype=int)
        c: numpy.ndarray = numpy.zeros(sequence.size, dtype=int)
        b[0] = 1
        c[0] = 1
        # Initialize the generator length
        generator_length: int = 0
        # Initialize variables
        m: int = -1
        n: int = 0
        while n < sequence.size:
            # Compute discrepancy
            discrepancy = sequence[n]
            for j in range(1, generator_length + 1):
                discrepancy: int = discrepancy ^ (c[j] & sequence[n - j])
            # If discrepancy is not zero, adjust polynomial
            if discrepancy != 0:
                t = c[:]
                for j in range(0, sequence.size - n + m):
                    c[n - m + j] = c[n - m + j] ^ b[j]
                if generator_length <= n / 2:
                    generator_length = n + 1 - generator_length
                    m = n
                    b = t
            n = n + 1
        # Return the length of generator
        return generator_length
```

3. Under an assumption of randomness, calculate the theoretical mean μ:

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

Also, For each substring, calculate a value of T_i.

$$T_i = (-1)^M \cdot (L_i - \mu) + \frac{2}{9}$$

```
 # Compute mean
self._mu: float = (self._pattern_length / 2.0) + (((-1) **
(self._pattern_length + 1)) + 9.0) / 36.0 - ((self._pattern_length / 3.0) +
(2.0 / 9.0)) / (2 ** self._pattern_length)
```

```
 # Compute the linear complexity of the blocks
        blocks_linear_complexity: numpy.ndarray = numpy.zeros(blocks_number,
dtype=int)
        for i in range(blocks_number):
            blocks_linear_complexity[i] = self._berlekamp_massey(bits[(i *
self._pattern_length):((i + 1) * self._pattern_length)])
```

4. Record the Ti values in v0,…, v6 as follows:

$Ti \leq -2.5$	Increment v_0 by 1
$-2.5 < Ti \leq -1.5$	Increment v_1 by 1
$-1.5 < Ti \leq -0.5$	Increment v_2 by 1
$-0.5 < Ti \leq 0.5$	Increment v_3 by 1
$0.5 < Ti \leq 1.5$	Increment v_4 by 1
$1.5 < Ti \leq 2.5$	Increment v_5 by 1
$Ti > 2.5$	Increment v_6 by 1

```
 # Count the distribution over tickets
        tickets: numpy.ndarray = ((-1.0) ** self._pattern_length) *
(blocks_linear_complexity[:] - self._mu) + (2.0 / 9.0)
        # Compute frequencies depending on tickets
        frequencies: numpy.ndarray =
numpy.zeros(self._freedom_degrees + 1, dtype=int)
```

5. Compute χ^2 (obs) using,

$$\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

Where, π_0=0.010417, π_1=0.03125, π_2=0.125, π_3=0.5, π_4=0.25, π_5=0.0625, π_6=0.020833

And calculate P-Value using,

$$P\text{-}value = \textbf{igamc}\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$$

```
 chi_square: float = float(numpy.sum(((frequencies[:] -
(blocks_number * self._probabilities[:])) ** 2.0) / (blocks_number *
self._probabilities[:])))
        # Compute the score (P-value)
        score: float = scipy.special.gammaincc((self._freedom_degrees
/ 2.0), (chi_square / 2.0))
        # Return result
        if score >= self.significance_value:
            return Result(self.name, True, numpy.array(score))
        return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

Python RNG - FAILED - score: 0.0
Python RNG (Secrets)

The result implies that there was a degree of randomness below the accepted level of significance (P>0.01). Also, the test requires *n>1,000,000* and *500<M<5000* (M=512 in the test code). Failing the test and *score*=0 would imply that the observed frequency counts of T_i stored in the ν_I values were different from the expected values. If *score >0.01*, then it is expected that the distribution of the frequency of the Ti should be proportional to the computed π_i.

PUF RNG - Not eligible for the test
True RNG

For PUF RNG, *n=30,000* and for True RNG, *n=131,076.* As the test requires a minimum of *n=1,000,000*, they are not eligible for the test.

Serial Test:

The test finds the frequency of all overlapping m-bit patterns across n-bit sequences and determines whether the occurrence of $2^m m$ *bit* overlapping patterns is the same as what would be expected for the true random sequence. A random sequence would have uniformity. Every *m-bit* pattern would have the same chance of occurring as any other *m-bit* pattern. If m=1, the test becomes the *Monobit test*.

The variables used in the Test:

m- length of bits in a block
n- length of the bitstream
ε - Sequence of bits generated by the random number generator(RNG)

Test Statistic: $\nabla\psi^2_m(obs)$ *and* $\nabla^2\psi^2_m$, A measure of how well the observed frequencies of m-bit patterns match he expected frequencies of the m-bit patterns.

Decision Rule: If the *P-value<0.01, then the sequence is classified as non-random.*

Test Implementation:

1. Form an augmented sequence, ε', by padding first *m-1* bits of ε at the end.

```
 # Pad the sequence
padded_bits: numpy.ndarray =
numpy.concatenate((bits,bits[0:self._pattern_length - 1]))
```

2. Determine the frequency of all possible overlapping *m-bit* blocks, all possible overlapping *(m-1)-bit* blocks and all possible overlapping *(m-2)-bit* blocks.
   Let, $V_{i1..i(m)}$ denote the frequency of the *m-bit* pattern $i_1 \ldots i_m$;
   $V_{i1..i(m-1)}$ denote the frequency of the (m-1)-bit pattern $i_1 \ldots i_{m-1}$;
   $V_{i1..i(m-2)}$ denotes the frequency of the (m-2)-bit pattern $i_1 \ldots i_{m-2}$.

```
 def _count_pattern(pattern: numpy.ndarray, padded_sequence:
numpy.ndarray, sequence_size: int) -> int:
        """
        Count the matches in the padded sequence of the given size
```

```
with the given pattern.
        :param pattern: the pattern to match against
        :param padded_sequence: the sequence of bits once padded
        :param sequence_size: the size of the original sequence of
bits
        :return: the integer value of the count
        """
        count: int = 0
        for i in range(sequence_size):
            match: bool = True
            for j in range(len(pattern)):
                if pattern[j] != padded_sequence[i + j]:
                    match = False
            if match:
                count += 1
        return count
```

3. Calculate:

$$\psi_m^2 = \frac{2^m}{n} \sum_{i_1 \ldots i_m} \left(v_{i_1 \ldots i_m} - \frac{n}{2^m} \right)^2 = \frac{2^m}{n} \sum_{i_1 \ldots i_m} v_{i_1 \ldots i_m}^2 - n$$

$$\psi_{m-1}^2 = \frac{2^{m-1}}{n} \sum_{i_1 \ldots i_{m-1}} \left(v_{i_1 \ldots i_{m-1}} - \frac{n}{2^{m-1}} \right)^2 = \frac{2^{m-1}}{n} \sum_{i_1 \ldots i_{m-1}} v_{i_1 \ldots i_{m-1}}^2 - n$$

$$\psi_{m-2}^2 = \frac{2^{m-2}}{n} \sum_{i_1 \ldots i_{m-2}} \left(v_{i_1 \ldots i_{m-2}} - \frac{n}{2^{m-2}} \right)^2 = \frac{2^{m-2}}{n} \sum_{i_1 \ldots i_{m-2}} v_{i_1 \ldots i_{m-2}}^2 - n$$

```
def _psi_sq_mv1(block_size: int, sequence_size: int, padded_sequence:
numpy.ndarray) -> float:
"""
Compute the Psi-Squared statistics from the NIST paper.
:param block_size: the size of the block
:param sequence_size: the size of the sequence of bits
:param padded_sequence: the original sequence once padded
:return: the float value of Psi-Squared statistics
"""
# Count the patterns
counts: numpy.ndarray = numpy.zeros(2 ** block_size, dtype=int)
for i in range(2 ** block_size):
    pattern: numpy.ndarray = (i >> numpy.arange(block_size, dtype=int)) &
1
```

```
counts[i] = SerialTest._count_pattern(pattern, padded_sequence,
sequence_size)
# Compute Psi-Squared statistics and return it
psi_sq_m: float = numpy.sum(counts[:] ** 2)
psi_sq_m *= (2 ** block_size) / sequence_size
psi_sq_m -= sequence_size
return psi_sq_m
```

```
 # Compute Psi-Squared statistics
psi_sq_m_0: float = self._psi_sq_mv1(self._pattern_length, bits.size,
padded_bits)
psi_sq_m_1: float = self._psi_sq_mv1(self._pattern_length - 1,
bits.size, padded_bits)
psi_sq_m_2: float = self._psi_sq_mv1(self._pattern_length - 2,
bits.size, padded_bits)
```

4. Calculate:

$$\nabla\psi_m^2 = \psi_m^2 - \psi_{m-1}^2, \text{ and}$$
$$\nabla^2\psi_m^2 = \psi_m^2 - 2\psi_{m-1}^2 + \psi_{m-2}^2.$$

$$P\text{-}value1 = \mathbf{igamc}\left(2^{m-2}, \nabla\psi_m^2\right) \text{ and}$$
$$P\text{-}value2 = \mathbf{igamc}\left(2^{m-3}, \nabla^2\psi_m^2\right).$$

```
delta_1: float = psi_sq_m_0 - psi_sq_m_1
delta_2: float = psi_sq_m_0 - (2 * psi_sq_m_1) + psi_sq_m_2
# Compute the scores (P-values)
score_1: float = scipy.special.gammaincc(2 ** (self._pattern_length - 2),
delta_1 / 2.0)
score_2: float = scipy.special.gammaincc(2 ** (self._pattern_length - 3),
delta_2 / 2.0)
# Return result
if score_1 >= self.significance_value and score_2 >= self.significance_value:
   return Result(self.name, True, numpy.array([score_1, score_2]))
return Result(self.name, False, numpy.array([score_1, score_2]))
```

Results and Inferences:

Python RNG
Python RNG (Secrets) - *FAILED - score: 0.0*

PUF RNG - `FAILED - score: 0.0`
True RNG

The result implies that there was a degree of randomness below the accepted level of significance (P>0.01). As we have selected *m=4* in the test, the minimum input size *n* becomes $floor(log_2^{128})-2 = 5$. As all inputs are greater than 128, all of them are eligible for the test. Failing the test and *score*=0, would imply that uniformity present in the sequence is not as expected in a random sequence. The *score=0* does not imply that *P-value1=0 and P-value2=0.* In *step-4* of implementation, result is TRUE iff both *P-value > 0.01.* If one or both *P-values* are less than 0.01, result=FALSE and *score=0.*
P-value would be less if the $\nabla\psi^2_m$ *and* $\nabla^2\psi^2_m$, had been large. [2]

Approximate Entropy Test:

Similar to the *Serial Test,* this test finds the frequency of all possible overlapping *m-bit* patterns across the entire sequence. Instead of comparing occurrence of $2^m m$ *bit* overlapping patterns to random sequence, this test compares the frequency of overlapping blocks of consecutive lengths *(m & m+1 bits)* with the expected result of random sequence.

The variables used in the Test:

m- length of bits in the first block. m+1 is the length of a consecutive block.
n- length of the bitstream
ε - Sequence of bits generated by the random number generator(RNG)

Test Statistic: χ^2 (obs), calculates how well the value of *ApEn(m)* (observed) matches the value of the expected value. [$ApEn(m) = \phi^{(m)} - \phi^{(m+1)}$, where $\phi^{(m)} = \sum \pi_i \log \pi_i$]

Decision Rule: If the *P-value<0.01, then the sequence is classified as non-random*

Test Implementation:

1. Form an augmented sequence, ε', by padding first *m-1* bits of ε at the end.
2. A frequency count is made of the *n* overlapping blocks (e.g., if a block containing ε_j to ε_{j+m-1} is examined at time j, then the block containing ε_{j+1} to ε_{j+m} is examined at time j+1). Let C^i_m represent the count of the possible *m-bit ((m+1)-bit)* values, where *i* is the *m-bit* value.

3. Compute $C_i^m = \frac{\#i}{n}$ *for each value of i.*
4. Calculate $\phi^{(m)} = \sum_{i=0}^{2^m-1} \pi_i \log \pi_i$, where, $\pi_i = C_j^3$ *and* $j=log_2 i$
5. Repeat steps 1-4 and by replacing *m* to *m+1*.

```
# Define Phi-m statistics list
        phi_m: [] = []
        for iteration in range(blocks_length, blocks_length + 2):
            # Compute the padded sequence of bits
            padded_bits: numpy.ndarray = numpy.concatenate((bits,
            bits[0:iteration - 1]))
            # Compute the frequency count
            counts: numpy.ndarray = numpy.zeros(2 ** iteration, dtype=int)
            for i in range(2 ** iteration):
               count: int = 0
               for j in range(bits.size):
                if self._pattern_to_int(padded_bits[j:j + iteration]) ==i:
                        count += 1
                counts[i] = count
            # Compute C-i as the average of counts on the number of bits
            c_i: numpy.ndarray = counts[:] / float(bits.size)
            # Compute Phi-m based on C-i
            phi_m.append(numpy.sum(c_i[c_i > 0.0] * numpy.log((c_i[c_i >
```

6. Compute the test statistic:

$$\chi^2 = 2n[log\ 2 - ApEn(m)],\ where\ ApEn(m) = \phi^{(m)} - \phi^{(m+1)}$$

And, compute *P-value* using,

$$P\text{-}value = \textbf{igamc}(2^{m-1}, \frac{\chi^2}{2})$$

```
# Compute Chi-Square from the computed statistics
chi_square: float = 2 * bits.size * (math.log(2) - (phi_m[0] -
phi_m[1]))
# Compute the score (P-value)
score: float = scipy.special.gammaincc(2 ** (blocks_length - 1),
(chi_square / 2.0))
# Return result
```

```
if score >= self.significance_value:
       return Result(self.name, True, numpy.array(score))
return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

Python RNG
Python RNG (Secrets) - *FAILED - score: 0.0*
PUF RNG
True RNG

For the test to run, minimum input size is given by $m < floor(log_2 n) - 5$. We have set the m=4 for this test. Therefore $n > 1024$ which is true for all RNGs. The reason for failing is that the $P\text{-}values < 0.01$ for all tests which can be caused by large values of *ApEn(m)* as smaller values of *ApEn(m)* imply strong regularity and thus more random nature [2].

Cumulative Sums (Cusum) Test

The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. The test is based on the principle of random walks and looks for deviations in the cumulative sum of the binary sequence from what would be expected for a random sequence.

The variables used in the Test:

n = The length of the bit string.

ε = The sequence of bits as generated by the RNG or PRNG being tested; this exists as a global structure at the time of the function call; ε = ε1, ε2, … ,εn.

Mode = A switch for applying the test either forward through the input sequence (mode = 0) or backward through the sequence (mode = 1).

Test Statistic: (Z) The largest excursion from the origin of the cumulative sums in the corresponding (-1, +1) sequence

Decision Rule: If the P-value obtained is ≥ 0.01 (P-value = 0.615, 0, 0, 0), the conclusion is that the sequence is random and the test passes

Test Implementation:

1. Form a normalized sequence: The zeros and ones of the input sequence (ε) are converted to values Xi of –1 and +1 using Xi = 2εi – 1.

   For example, if ε = 1011010111, then X = 1, (-1), 1, 1, (-1), 1, (-1), 1, 1, 1

```
def __init__(self):
        # Generate base Test class
         super(CumulativeSumsTest, self).__init__("Cumulative Sums",
0.01)
def _execute(self,
   bits: numpy.ndarray) -> Result:
        """
Overridden method of Test class: check its docstring for further
information.
        """
        # Copy the bits to a new array
        bits_copy: numpy.ndarray = bits.copy()
        # Convert all the zeros in the array to -1
        bits_copy[bits_copy == 0] = -1
```

2. Compute partial sums Si of successively larger subsequences, each starting with X1 (if mode = 0) or Xn (if mode = 1).

Mode = 0 (forward)	Mode = 1 (backward)
S1 = X1	S1 = Xn
S2 = X1 + X2	S2 = Xn + Xn-1
S3 = X1 + X2 + X3	S3 = Xn + Xn-1 + Xn-2
.	.
.	.
Sk = X1 + X2 + X3 + … + Xk	Sk = Xn + Xn-1 + Xn-2 + … + Xn-k+1
.	.
.	.
Sn = X1 + X2 + X3 + … + Xk + …+ Xn	Sn = Xn + Xn-1 + Xn-2 + … + Xk-1 + …+ X1

That is, Sk = Sk-1 + Xk for mode 0, and Sk = Sk-1 + Xn-k+1 for mode 1.

```
 # Compute the partial sum with forward (mode 0) and backward
(mode 1) modes and record the largest excursion
forward_sum: int = 0
 backward_sum: int = 0
 backward_sum: int = 0
 backward_max: int = 0
for i in range(bits_copy.size):
forward_sum += bits_copy[i]
backward_sum += bits_copy[bits_copy.size - 1 - i]
forward_max = max(abs(forward_sum), forward_max)
backward_max = max(abs(backward_sum), backward_max)
```

3. Compute the test statistic z =max1≤k≤n|Sk|, where max1≤k≤n|Sk| is the largest of the absolute values of the partial sums Sk.

4. $$\text{Compute } P\text{-value} = 1 - \sum_{k=\left(\frac{-n}{z}+1\right)/4}^{\left(\frac{n}{z}-1\right)/4}\left[\Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k-1)z}{\sqrt{n}}\right)\right] + \sum_{k=\left(\frac{-n}{z}-3\right)/4}^{\left(\frac{n}{z}-1\right)/4}\left[\Phi\left(\frac{(4k+3)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)\right]$$

If the computed P-value is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

```
# Compute the scores (P-Values)

  score_1: float = self._compute_p_value(bits_copy.size,
forward_max)
  score_2: float = self._compute_p_value(bits_copy.size,
backward_max)

# Return result
 if score_1 >= self.significance_value and score_2 >=
self.significance_value:
       return Result(self.name, True, numpy.array([score_1,
score_2]))
return Result(self.name, False, numpy.array([score_1, score_2]))
```

```
def is_eligible(self,
                    bits: numpy.ndarray) -> bool:
"""
Overridden method of Test class: check its docstring for further
information.

"""
```

```
@staticmethod
def _compute_p_value(sequence_size: int, max_excursion: int) ->
float:
 """
Compute P-Value given the sequence size and the max excursion.
:param sequence_size: the length of the sequence of bits
:param max_excursion: the max excursion backward or forward
:return: the computed float P-Value
"""
```

5. Input Size Recommendation

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$). Those random no. sequences have less than 100 bits then the test will be failed.

```
# Execute first sum
sum_a: float = 0.0
start_k: int = int(math.floor((((float(-sequence_size) /
max_excursion) + 1.0) / 4.0)))
end_k: int = int(math.floor((((float(sequence_size) /
max_excursion) - 1.0) / 4.0)))
for k in range(start_k, end_k + 1):
c: float = 0.5 * math.erfc(-(((4.0 * k) + 1.0) * max_excursion) /
math.sqrt(sequence_size) * math.sqrt(0.5))
d: float = 0.5 * math.erfc(-(((4.0 * k) - 1.0) * max_excursion) /
math.sqrt(sequence_size) * math.sqrt(0.5))
sum_a = sum_a + c - d
 # Execute second sum
sum_b: float = 0.0
start_k = int(math.floor((((float(-sequence_size) / max_excursion)
```

```
- 3.0) / 4.0)))
 end_k = int(math.floor((((float(sequence_size) / max_excursion) -
1.0) / 4.0)))
for k in range(start_k, end_k + 1):
c: float = 0.5 * math.erfc(-(((4.0 * k) + 3.0) * max_excursion) /
math.sqrt(sequence_size) * math.sqrt(0.5))
d: float = 0.5 * math.erfc(-(((4.0 * k) + 1.0) * max_excursion) /
math.sqrt(sequence_size) * math.sqrt(0.5))
sum_b = sum_b + c - d
# Return value
return 1.0 - sum_a + sum_b
```

Results and Inferences:

Python RNG - *FAILED - score: 0.0*
Python RNG (Secrets)

The result implies that there was a degree of randomness below the accepted level of significance (P>0.01). Also, the test requires n ≥ 100 (in the test code).

The excursions of the random walk from zero are near zero, indicating that the tested sequence is likely random. For certain types of non-random sequences, the excursions of this random walk from zero will be large and the test will be failed.

If the Cusum Test fails, it indicates that the binary sequence may be non-random and further investigation is necessary to identify the source of the non-randomness.

PUF RNG - *PASSED - score: 0.615*

True RNG - *FAILED - score: 0.0*

For PUF RNG, n=30,000 and for True RNG, n=131,076. As the test requires a minimum n=1,00 they are eligible for the test and P value should be P>0.01 to pass the test.

Random Excursions Test

The focus of this test is the number of cycles having exactly K visits in a cumulative sum random walk. The cumulative sum random walk is derived from partial sums after the (0,1) sequence is transferred to the appropriate (-1, +1) sequence.
A cycle of a random walk consists of a sequence of steps of unit length taken at random that begin at and return to the origin. The purpose of this test is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence.

The variables used in the Test:

n = The length of the bit string.

ε = The sequence of bits as generated by the RNG or PRNG being tested; this exists as a global structure at the time of the function call; ε = ε1, ε2, … ,εn.

Test Statistic: χ 2 (obs), For a given state x, a measure of how well the observed number of state visits within a cycle match the expected number of state visits within a cycle, under an assumption of randomness.

The reference distribution for the test statistic is the χ 2 distribution.

Decision Rule: If the computed P-value is < 0.01 (P-value =0.005, 0.683, 0.683, 0.63), then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random and in a random case then the sequence of no. will pass the test.

Test Implementation:

1. Form a normalized (-1, +1) sequence X: The zeros and ones of the input sequence (ε) are changed to values of –1 and +1 via Xi = 2εi – 1.

   For example, if ε = 0110110101, then n = 10 and X = -1, 1, 1, -1, 1, 1, -1, 1, -1, 1.

```
# Copy the bits to a new array
 bits_copy: numpy.ndarray = bits.copy()
# Convert all the zeros in the array to -1
 bits_copy[bits_copy == 0] = -1
```

2. Compute the partial sums Si of successively larger subsequences, each starting with X1. Form the set S = {Si}.

S1 = X1

S2 = X1 + X2

S3 = X1 + X2 + X3

.

.

Sk = X1 + X2 + X3 + … + Xk

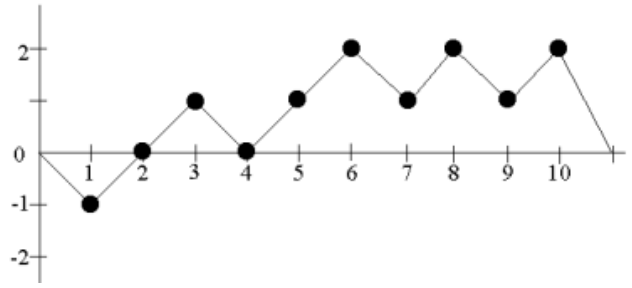
.

.

Sn = X1 + X2 + X3 + … + Xk + …+ Xn

Form a new sequence S' by attaching zeros before and after the set S. That is, S' = 0, s1, s2, … , sn, 0

For the example in this section, S' = 0, -1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 0. The resulting random walk is shown below.



Example Random Walk (S')

3. Let J = the total number of zero crossings in S', where a zero crossing is a value of zero in S ' that occurs after the starting zero. J is also the number of cycles in S′, where a cycle of S′ is a subsequence of S′consisting of an occurrence of zero, followed by no-zero values, and ending with another zero. The ending zero in one cycle may be the beginning zero in another cycle. The number of cycles in S ' is the number of zero crossings. If J < 500, discontinue the test6 .

For example in this section, if S' = {0, –1, 0 1, 0, 1, 2, 1, 2, 1, 2, 0}, then J = 3 (there are zeros in positions 3, 5 and 12 of S'). The zero crossings are easily observed in the above plot. Since J = 3, there are 3 cycles, consisting of {0, -1, 0}, {0, 1, 0} and {0, 1, 2, 1, 2, 1, 2, 0}.

4. For each cycle and for each non-zero state value x having values $-4 \leq x \leq -1$ and $1 \leq x \leq 4$, compute the frequency of each x within each cycle.

For the example in this section, the first cycle has one occurrence of –1, the second cycle has one occurrence of 1, and the third cycle has three occurrences each of 1 and 2. This can be visualized.

5. For each of the eight states of x, compute νk(x) = the total number of cycles in which state x occurs exactly k times among all cycles, for k = 0, 1, …, 5 (for k = 5, all frequencies ≥ 5 are 5 stored in ν5(x)). Note that $\sum \nu_k(x) = J$.

```
# Generate the padded cumulative sum of the array of -1, 1
   sum_prime: numpy.ndarray = numpy.concatenate((numpy.array([0]),
numpy.cumsum(bits_copy), numpy.array([0]))).astype(int)
 # Compute the cycles iterating over each position of S'
(sum_prime) and define the first cycle
 cycles: [] = []
  cycle: [] = [0]
for index, _ in enumerate(sum_prime[1:]):
 # Once a zero crossing is found add all the non zero elements of
S' to the cycle
# Else wrap up the cycle and start a new cycle
if sum_prime[index] != 0:
cycle += [sum_prime[index]]
else:
cycle += [0]
cycles.append(cycle)
cycle: [] = [0]
 # Append the last cycle
cycles.append(cycle)
    # Compute the size of the cycles list
 cycles_size: int = len(cycles)
# Setup frequencies table (Vk(x))
frequencies_table: dict = {
 -4: numpy.zeros(6, dtype=int),
-3: numpy.zeros(6, dtype=int),
-2: numpy.zeros(6, dtype=int),
-1: numpy.zeros(6, dtype=int),
1: numpy.zeros(6, dtype=int),
2: numpy.zeros(6, dtype=int),
3: numpy.zeros(6, dtype=int),
4: numpy.zeros(6, dtype=int),
}
# Count occurrences
for value in frequencies_table.keys():
for k in range(frequencies_table[value].size):
  count: int = 0
# Count how many cycles in which x occurs k times
```

```
for cycle in cycles:
# Count how many times the value used as key of the table occurs
in the current cycle
occurrences: int = numpy.count_nonzero(numpy.array(cycle) ==
value)
# If the value occurs k times, increment the cycle count
    if 5 > k == occurrences:
count += 1
elif occurrences >= 5:
count += 1
frequencies_table[value][k] = count
```

6. For each of the eight states of x, compute the test statistic

$$\chi^2(obs) = \sum_{k=0}^{5} \frac{(\nu_k(x) - J\pi_k(x))^2}{J\pi_k(x)}$$

7. For each state of x, compute P-value = igamc(5/2, χ2 (obs) 2) . Eight P-values will be produced. 5 4.333033 For the example when x = 1,

$$\text{P-value} = \text{igamc}\left(\frac{5}{2}, \frac{4.333033}{2}\right) = 0.502529.$$

Since the P-value obtained is ≥ 0.01 (P-value = 0.502529), the conclusion is that the sequence is random.

Note that if χ 2 (obs) were too large, then the sequence would have displayed a deviation from the theoretical distribution for a given state across all cycles.

8. It is recommended that each sequence to be tested consist of a minimum of 1,000,000 bits (i.e., $n \geq 10^6$). Those random no. sequences have less than 1,000,000 bits then the test will be failed.

```
# Compute the scores (P-values)
 scores: [] = []
```

```
for value in frequencies_table.keys():
   # Compute Chi-Square for this value
 chi_square: float = numpy.sum(((frequencies_table[value][:] -
(cycles_size * (self._probabilities_xk[abs(value) - 1][:]))) ** 2)
/ (cycles_size * self._probabilities_xk[abs(value) - 1][:]))
# Compute the P-value for this value
score: float = scipy.special.gammaincc(5.0 / 2.0, chi_square /
2.0)
scores.append(score)
 # Return result
if all(score >= self.significance_value for score in scores):
return Result(self.name, True, numpy.array(scores))
def is_eligible(self,
bits: numpy.ndarray) -> bool:
"""
Overridden method of Test class: check its docstring for further
information.
 """
# This test is always eligible for any sequence
return True
```

Results and Inferences:

Python RNG - *FAILED - score: 0.683*
Python RNG (Secrets) - *FAILED - score: 0.683*

The result implies that there was a degree of randomness below the accepted level of significance (P>0.01). Also, the test requires n ≥ 100 (in the test code).

Random Excursions Test checks whether the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence. Passing the test suggests that the sequence is consistent with being random and exhibits expected behavior of a random walk.

PUF RNG - *FAILED - score: 0.005*
True RNG - *FAILED - score: 0.63*

For PUF RNG, n=30,000 and for True RNG, n=131,076. As the test requires a minimum n ≥ 10^6 they are eligible for the test and P value should be P>0.01 to pass the test.

Random Excursions Variant Test

The focus of this test is the total number of times that a particular state is visited (i.e., occurs) in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of visits to various states in the random walk. This test is actually a series of eighteen tests (and conclusions), one test and conclusion for each of the states: -9, -8, …, -1 and +1, +2, …, +9.

The variables used in the Test:

n = The length of the bit string.

ε = The sequence of bits as generated by the RNG or PRNG being tested; this exists as a global structure at the time of the function call; ε = ε1, ε2, … ,εn.

Test Statistic: (ξ) For a given state x, the total number of times that the given state is visited during the entire random walk as determined.

If ξ is distributed as normal, then |ξ| is distributed as half normal.) If the sequence is random, then the test statistic will be about 0. If there are too many ones or too many zeroes, then the test statistic will be large

Decision Rule: If the computed P-value is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

Test Implementation:

1. Form the normalized (-1, +1) sequence X in which the zeros and ones of the input sequence (ε) are converted to values of –1 and +1 via X = X1, X2, …, Xn, where Xi = 2εi – 1.

For example, if ε = 0110110101, then n = 10 and X = -1, 1, 1, -1, 1, 1, -1, 1, -1, 1.

2. Compute partial sums Si of successively larger subsequences, each starting with x1. Form the set S = {Si}

S1 = X1

S2 = X1 + X2

S3 = X1 + X2 + X3

.

.

Sk = X1 + X2 + X3 + … + Xk

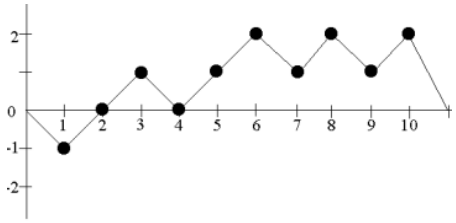
.

.

Sn = X1 + X2 + X3 + … + Xk + …+ Xn

3. Form a new sequence S' by attaching zeros before and after the set S. That is, S' = 0, s1, s2, … , sn, 0. For example,

S' = 0, -1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 0. The resulting random walk is shown below



Example Random Walk (S')

```
def _execute(self,
bits: numpy.ndarray) -> Result:
        """
Overridden method of Test class: check its docstring for further
information.
"""
# Copy the bits to a new array
bits_copy: numpy.ndarray = bits.copy()
# Convert all the zeros in the array to -1
bits_copy[bits_copy == 0] = -1
# Generate the padded cumulative sum of the array of -1, 1
sum_prime: numpy.ndarray = numpy.concatenate((numpy.array([0]),
numpy.cumsum(bits_copy), numpy.array([0]))).astype(int)
        # Count the number of cycles in S' (sum_prime)
 cycles_size: int = numpy.count_nonzero(sum_prime[1:] == 0)
# Generate the counts of offsets
  unique, counts = numpy.unique(sum_prime[abs(sum_prime) < 10],
return_counts=True)
  # Compute the scores (P-values)
scores: [] = []
  for key, value in zip(unique, counts):
```

$$; P\text{-}value = \mathit{erfc}\left(\frac{|\xi(x) - J|}{\sqrt{2J(4|x| - 2)}}\right).$$

4. For each $\xi(x)$, compute P-value . Eighteen P-values are computed.

   If the computed P-value is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

5. It is recommended that each sequence to be tested consist of a minimum of 1,000,000 bits (i.e., $n \geq 106$)

```
# Compute the scores (P-values)
 scores: [] = []
for key, value in zip(unique, counts):
# Compute the P-value for this value (if not zero)
if key != 0:
 scores.append(abs(value - cycles_size) / math.sqrt(2.0 *
cycles_size * ((4.0 * abs(key)) - 2.0)))
 # Return result
 if all(score >= self.significance_value for score in scores):
 return Result(self.name, True, numpy.array(scores))
return Result(self.name, False, numpy.array(scores))

def is_eligible(self,
      bits: numpy.ndarray) -> bool:
  """
Overridden method of Test class: check its docstring for further
information.
"""
# This test is always eligible for any sequence
return True
```

Results and Inferences:

Python RNG *- PASSED - score: 0.17*
Python RNG (Secrets) *- PASSED - score: 0.17*

The result implies that there was a degree of randomness below the accepted level of significance ($P>0.01$). Also, the test requires $n \geq 10^6$ (in the test code).

It is important to note that passing the Random Excursions Variant Test does not guarantee that the sequence is completely random, as no test can prove randomness with absolute certainty. Instead, it provides evidence that the sequence is likely to be random and can be used with some level of confidence in applications that require randomness.

PUF RNG - *PASSED - score: 1.208*
True RNG - *FAILED - score: 0.0*

For PUF RNG, n=30,000 and for True RNG, n=131,076. As the test requires a minimum $n \geq 10^6$ they are eligible for the test and If the computed P-value is < 0.01 (P-value = 1.208, 0.17, 0.17, 0.0), then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random and in a random case then the sequence of no. will pass the test.

Non-Overlapping Template Test:

We have derived the code from the GitHub repository; the essential parts include checking the eligibility of running the test based on the size of the dataset. The principle involves defining the templates to derive data points (namely, the occurrence of each sequence of m bits in a non-overlapping fashion. Due to this approach, we shift the pointer with m bits every time we find a matching sequence, and if not, then just by 1 bit. The time taken by code thus reduces as repeating patterns are not checked due to omission by m bits. For all the templates of a given length of m bits, we check the occurrences of that template and then code to get the p-value of the provided data.

The variables used in the Test:

n = The length of the bit string.

m =size of the template.

Step 1:

```
def __init__(self):
    # Define specific test attributes
    self._blocks_number: int = 8
    self._templates: [] = [
                           [[0, 1], [1, 0]],
                           [[0, 0, 1], [0, 1, 1], [1, 0, 0], [1, 1, 0]],
                           [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1], [1, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 0]],
                           [[0, 0, 0, 0, 1], [0, 0, 0, 1, 1], [0, 0, 1, 0, 1], [0, 1, 0, 1, 1], [0, 0, 1, 1, 1], [0, 1, 1, 1, 1], [1, 1, 1, 0, 0],
                           [[0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 1], [0, 0, 0, 1, 0, 1], [0, 0, 0, 1, 1, 1], [0, 0, 1, 0, 1, 1], [0, 0, 1, 1, 0, 1],
                           [[0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 1, 1, 1], [0, 0, 0, 1, 0, 0, 1], [0,
                           [[0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 1], [0, 0, 0, 0, 1,
                           ]
    # Define cache attributes
    self._last_bits_size: int = -1
    self._substring_bits_length: int = -1
```

Step 2: checking eligibility

```
def is_eligible(self,
                bits: numpy.ndarray) -> bool:
    """
    Overridden method of Test class: check its docstring for further information.
    """
```

Step 3:(declaring initial values and base case)

```
def _execute(self,
             bits: numpy.ndarray) -> Result:
    """
    Overridden method of Test class: check its docstring for further information.
    """
    # Choose the template B at random
    b_template: numpy.ndarray = numpy.array(random.choice(random.choice(self._templates)))
    # Reload values is cache is empty or no longer up-to-date
    # Otherwise, use cache
    if self._last_bits_size == -1 or self._last_bits_size != bits.size:
        # Split into N blocks of M bits
        substring_bits_length: int = int(bits.size // self._blocks_number)
        # Save in the cache
        self._last_bits_size = bits.size
        self._substring_bits_length = substring_bits_length
    else:
        substring_bits_length: int = self._substring_bits_length
```

Step 4:Checking for all templates throughout bitstring and incrementing as per the result:

```
    block: numpy.ndarray = bits[i * substring_bits_length:(i + 1) * substring_bits_length]
    # Define counting variables
    position: int = 0
    count: int = 0
    # Count the matches in the block with the chosen template
    while position < (substring_bits_length - b_template.size):
        if (block[position:position + b_template.size] == b_template).all():
            position += b_template.size
            count += 1
        else:
            position += 1
    matches[i] = count
```

Step 5: Statistical computation:

```
sigma: float = substring_bits_length * ((1.0 / float(2 ** b_template.size)) - (float((2 * b_template.size) - 1) / float(2 ** (2 * b_template.size))))
# Compute Chi-square
chi_square: float = float(numpy.sum(((matches[:] - mu) ** 2) / (sigma ** 2)))
# If Chi-square is zero, fail the test
if chi_square != 0:
    # Compute the score (P-value)
    score: float = scipy.special.gammaincc(self._blocks_number / 2.0, chi_square / 2.0)
    # Return result
    if score >= self.significance_value:
        return Result(self.name, True, numpy.array(score))
return Result(self.name, False, numpy.array(0.0))
```

Results And Inferences

PYTHON RANDOM- *Passed: score-0.063: p-value greater than 0.01.*

The test passed, indicating that for blocks of size 2 to 8, we can say with a 99% confidence interval that a particular sequence doesn't repeat often.

PYTHON-SECRETS- *Passed:score-0.355:*

It can be inferred that python secrets implements a much more random code then that provided by random function.

TRUE RANDOM- *Passed:-0.347:*

Using the non-overlapping test we can agree with the online source that it generates fairly randomised code.

PUF GENERATOR- *Passed: score-0.989: p-value greater than 0.01.*

Here it's evident that the PUF code is much more random than the python-random number generator as the score is much larger.

Overlapping Matching Template Test:

The non-overlapping test function has a very big demerit that it does'nt consider repeating sequences due to large omission but now it takes more time to analyse the dataset as we check all subsequences of m bits in the entire data.

The code involves checking eligibility of the dataset based on size, it consumes a large amount of data but generally we are constrained by availability of space and fast servers. The code involves defining the templates to be used to derive data points (namely occurrence of each sequence of m bits in an overlapping fashion). With this approach, we shift the pointer with 1 bit each and every time irrespective of the result of comparing. The accuracy of code is much better then previous approach as a result of more checks. For all the templates of given length of m bits we check the no of occurrences of that template and then code to get the p-value of the given data.The chi-squared random variable is used to compute the statistics.

Here only the step 4 changes primarily: the pointer

```
matches_distributions: numpy.ndarray = numpy.zeros(self._freedom_degrees + 1, dtype=int)
for i in range(self._blocks_number):
    # Define the block at the current index
    block: numpy.ndarray = bits[i * self._substring_bits_length:(i + 1) * self._substring_bits_length]
    # Define counting variable
    count: int = 0
    # Count the matches in the block with respect to the given template
    for position in range(self._substring_bits_length - self._template_bits_length):
        if (block[position:position + self._template_bits_length] == b_template).all():
            count += 1
    matches_distributions[min(count, self._freedom_degrees)] += 1
```

Results and inferences

PYTHON RANDOM - *Failed: score-0.00: p-value less than 0.01.*

We get to know that python generator is not random but surely a PRNG as it has overlapping sequences in a repeating fashion and so does not pass it.

PYTHON SECRETS - *Failed:-0.00: p-value less than 0.01.*

Python secrets implement a much more random code than that provided by random function but still it is not completely random as there are repeated sequences which are responsible for failure.

PUF-GENERATED ALGORITHM - *Not applicable*

Not applicable owing to insufficient data set and hardware.

TRUE RANDOM - *Not applicable*

Not applicable owing to insufficient data set and hardware.
Required data size for successful operation of test:

n	L	$Q = 10 \bullet 2^L$
≥ 387,840	6	640
≥ 904,960	7	1280
≥ 2,068,480	8	2560
≥ 4,654,080	9	5120
≥ 10,342,400	10	10240
≥ 22,753,280	11	20480
≥ 49,643,520	12	40960
≥ 107,560,960	13	81920
≥ 231,669,760	14	163840
≥ 496,435,200	15	327680
≥ 1,059,061,760	16	655360

Maurer's Universal Test:

The number of bits between matching patterns—a statistic related to the length of a compressed sequence—is the test's primary concern. A series that may be compressed significantly is regarded as non-random. One of the best tests follows a semi-log-normal distribution to study the cryptographic use of the device. The steps involved in performing the test are:
A test segment made up of K L-bit non-overlapping blocks, and an initialization segment made up of Q L-bit non-overlapping blocks make up the n-bit sequence. At the end of the row, bits that do not make up a whole L-bit block are discarded.
The test is initialized using the first Q blocks.
A table is generated using the initialization segment for each potential L-bit value.
Determine the number of blocks since the start of the test segment by looking at each of the K blocks.
last instance of the identical L-bit block. Substitute the location of the current block for the value in the table, and add the estimated distance between occurrences using the log2 factor. Finally, computing the p-value with the level of significance of 0.01.

Step 1: Initialising the test variables

```
ef __init__(self):
   # Define specific test attributes
   # Note: tables from https://static.aminer.org/pdf/PDF/000/120/333/a_universal_statistical_test_
   self._sequence_size_min: int = 387840
   self._default_pattern_size: int = 6
   self._freedom_degrees: int = 5
   self._substring_bits_length: int = 1062
   self._thresholds: [] = [904960, 2068480, 4654080, 10342400, 22753280, 49643520, 107560960, 231
   self._expected_value_table: [] = [0, 0.73264948, 1.5374383, 2.40160681, 3.31122472, 4.25342659,
   self._variance_table: [] = [0, 0.690, 1.338, 1.901, 2.358, 2.705, 2.954, 3.125, 3.238, 3.311, 3
   # Define cache attributes
   self._last_bits_size: int = -1
   self._pattern_length: int = -1
   self._blocks_number: int = -1
   self._q_blocks: int = -1
   self._k_blocks: int = -1
   # Generate base Test class
   super(MaurersUniversalTest, self).__init__("Maurers Universal", 0.01)
```

Step 2:

```
    if self._last_bits_size == -1 or self._last_bits_size != bits.size:
        # Compute the pattern size
        pattern_length: int = self._default_pattern_size
        for threshold in self._thresholds:
            if bits.size >= threshold:
                pattern_length += 1
        # Split the data into Q and K blocks
        blocks_number: int = int(bits.size // pattern_length)
        q_blocks: int = 10 * (2 ** pattern_length)
        k_blocks: int = blocks_number - q_blocks
        # Save in the cache
        self._last_bits_size = bits.size
        self._pattern_length = pattern_length
        self._blocks_number = blocks_number
        self._q_blocks = q_blocks
        self._k_blocks = k_blocks
    else:
        pattern_length: int = self._pattern_length
        blocks_number: int = self._blocks_number
        q_blocks: int = self._q_blocks
        k_blocks: int = self._k_blocks
```

Step 3: Computing statistic for all values of L

```
fn: float = computed_sum / k_blocks
# Compute magnitude
magnitude: float = abs((fn - self._expected_value_table[pattern_length]) / ((math.sqrt(self._variance_table[pattern_length])) * math.sqrt(2)))
# Compute the score (P-value)
score: float = math.erfc(magnitude)
# Return result
if score >= self.significance_value:
    return Result(self.name, True, numpy.array(score))
return Result(self.name, False, numpy.array(score))
```

Step 4: Checking eligibility with size

Results and inferences

PYTHON RANDOM - *Failed: score-0.01 {expected(f(n)> L}*

Failing the Maurer test implies that the code produces the repetition of its bit sequences at regular intervals and so can easily be compressed, that is the distance between repetitions is small enough and so not a good Random generator.

PYTHON SECRETS - *Failed:-0.01: p-value is equal to 0.01.*

Failing the Maurer test implies that the code produces the repetition of its bit sequences at regular intervals and so can easily be compressed, so we cannot rely on it as a strong RNG.

TRUE RANDOM - *Not applicable*

Not applicable owing to insufficient data set and hardware.

PUF GENERATED ALGORITH - *Not applicable*

Not applicable owing to insufficient data set and hardware.

Longest Run of Ones in a Block Test

A statistical test is used to measure the randomness of a number. It measures the longest consecutive subsequence of ones and compares it with the theoretically expected length of the longest run of ones in a random sequence. The true random sequence should have less deviation from expected results(i.e., larger value χ^2).

Test Description:

1. The sequence of length n is divided into blocks of length M, i.e., M blocks each of size N(n = N*M). The values of M and K are selected corresponding to the value of n.

```
        # Set the block size depending on the input sequence length
        block_size: int = 10000
        if bits.size < 6272:
            block_size: int = 8
        elif bits.size < 750000:
            block_size: int = 128
        # Set the block number and K depending on the block size
        k: int = 6
        blocks_number: int = 75
        if block_size == 8:
            k: int = 3
            blocks_number: int = 16
        elif block_size == 128:
            k: int = 5
            blocks_number: int = 49
        # Save in the cache
        self._last_bits_size = bits.size
        self._block_size = block_size
        self._blocks_number = blocks_number
        self._k = k
    else:
        block_size: int = self._block_size
        blocks_number: int = self._blocks_number
        k: int = self._k
```

2. We divide the blocks into k+1 classes based on the length of the longest run of ones and assign each block to a class.

```
    # Define the array of frequencies
    frequencies: numpy.ndarray = numpy.zeros(7, dtype=int)
    # Find longest run length in each block
    for i in range(blocks_number):
        block: numpy.ndarray = bits[i * block_size:((i + 1) * block_size)]
        run_length: int = 0
        longest_run_length: int = 0
        # Count the length of each adjacent bits group (runs) in the current block
        # and update the max length of them
        for j in range(block_size):
            if block[j] == 1:
                run_length += 1
                if run_length > longest_run_length:
                    longest_run_length = run_length
            else:
                run_length = 0
```

3. The length of the longest run of ones in observed data against expected data is measured by χ^2-distribution with K deg $X^2 = \sum_{i=0}^{K} \frac{(\nu_i - N\pi_i)^2}{N\pi_i}$

```
# Compute Chi-square
chi_square: float = 0.0
for i in range(k + 1):
    chi_square += ((frequencies[i] - blocks_number * self._probabilities(block_size, i)) ** 2)
    /
    (blocks_number * self._probabilities(block_size, i))
```

where π_i (i = 0,…, K) are the probabilities associated with K, and M and has been taken from [1].

The expected probability that $\nu \leq m$, given the number of ones in a block, r, is expressed as

$$P(\nu \leq m \mid r) = \frac{1}{\binom{M}{r}} \sum_{j=0}^{U} (-1)^j \binom{M-r+1}{j} \binom{M-j(m+1)}{M-r}$$

$$P(\nu \leq m) = \sum_{r=0}^{M} \binom{M}{r} P(\nu \leq m \mid r) \frac{1}{2^M}.$$

K=3, M=8

classes	probabilities
{ν≤1}	$\pi_0 = 0.2148$
{ν=2}	$\pi_1 = 0.3672$
{ν=3}	$\pi_2 = 0.2305$
{ν≥4}	$\pi_3 = 0.1875$

K=5, M=128

classes	probabilities
{ν≤4}	$\pi_0 = 0.1174$
{ν=5}	$\pi_1 = 0.2430$
{ν=6}	$\pi_2 = 0.2493$
{ν=7}	$\pi_3 = 0.1752$
{ν=8}	$\pi_4 = 0.1027$
{ν≥9}	$\pi_5 = 0.1124$

K=5, M=512

classes	probabilities
{ν≤6}	$\pi_0 = 0.1170$
{ν=7}	$\pi_1 = 0.2460$
{ν=8}	$\pi_2 = 0.2523$
{ν=9}	$\pi_3 = 0.1755$
{ν=10}	$\pi_4 = 0.1027$
{ν≥11}	$\pi_5 = 0.1124$

K=5, M=1000

classes	probabilities
{ν≤7}	$\pi_0 = 0.1307$
{ν=8}	$\pi_1 = 0.2437$
{ν=9}	$\pi_2 = 0.2452$
{ν=10}	$\pi_3 = 0.1714$
{ν=11}	$\pi_4 = 0.1002$
{ν≥12}	$\pi_5 = 0.1088$

4. Compute the p-value using the incomplete gamma function expressed as

$$\frac{\int_{X^2(obs)}^{\infty} e^{-u/2} u^{K/2-1} du}{\Gamma(K/2)2^{K/2}} = igamc\left(\frac{K}{2}, \frac{X^2(obs)}{2}\right)$$

```
# Compute score (P-value)
score: float = scipy.special.gammaincc(k / 2.0, chi_square / 2.0)
# Return result
if score >= self.significance_value:
    return Result(self.name, True, numpy.array(score))
return Result(self.name, False, numpy.array(score))
```

5. The result of the test is determined by the P-value, if the P-value is < 0.01, then the test is passed, or otherwise return failed.

```
if size_of_block == 8:
    return [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124][index]
elif size_of_block == 128:
    return [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124][index]
elif size_of_block == 512:
    return [0.1170, 0.2460, 0.2523, 0.1755, 0.1027, 0.1124][index]
elif size_of_block == 1000:
    return [0.1307, 0.2437, 0.2452, 0.1714, 0.1002, 0.1088][index]
else:
    return [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727][index]
```

Results and Inferences:

PUF_random_ test result - *FAILED - score: 0.0*

Failing the test and score=0 implies that the observed frequency ν_i values are different from the expected values. If the score>0.01, then it is expected that the distribution of the frequency π_i. It is possible for a true Random Number Generator to fail some tests due to random bias, but the ratio of successful tests should be considered to term any generator as true-random or pseudo-random.

Python_random_ test result - *PASSED - score: 0.475*

Python_secret_ Test result - *PASSED - score: 0.14*

True_random_Test result - *PASSED - score: 0.448*

The default_Python_Generator passed the test. Although it has passed the test, a higher P-value also suggests that the evidence is not strong enough. Thereby pointing to the fact that a single test only looks into a specific property of sequence. If a PRNG passes the given test, it cannot be concluded that the generator is truly random. It only implies that it has a specific property.

Binary Matrix Rank Test

The binary matrix rank test aims to determine the linear independence of the rows and columns of a binary matrix, i.e., it determines if the rows and columns of a binary matrix can be linearly combined to form all the other rows and columns in the matrix. This is important in many fields, including cryptography, coding theory, and communication systems.

Test Description:

1. Divide the binary sequence into Q-bit blocks, and collect M such blocks to form M*Q matrices. There will be N = n / (M*Q) such matrices.

```
def __init__(self, block: numpy.ndarray, rows_number: int, columns_number: int):
    self._rows = rows_number
    self._columns = columns_number
    self._matrix = block
    self._base_rank = min(self._rows, self._columns)
```

2. Before calculating the binary rank, we perform some transformations on the input binary matrix, including row operations to transform the input binary matrix into an upper triangular binary matrix and then into a reduced row-echelon form.

```python
def compute_rank(self) -> int:
    """
    Computes the **binary rank** of the matrix.
    :return: an integer defining binary rank of the matrix.
    """
    # Perform row operations with forward elimination
    i: int = 0
    while i < self._base_rank - 1:
        if self._matrix[i][i] == 1:
            self._perform_row_operations(i, True)
        else:
            found = self._find_unit_element_swap(i, True)
            if found == 1:
                self._perform_row_operations(i, True)
        i += 1
    # Perform row operations without forward elimination
    i = self._base_rank - 1
    while i > 0:
        if self._matrix[i][i] == 1:
            self._perform_row_operations(i, False)
        else:
            if self._find_unit_element_swap(i, False) == 1:
                self._perform_row_operations(i, False)
        i -= 1
    # Compute the rank of the transformed matrix
    return self._compute_rank()
```

3. Next, calculate the binary rank (Rℓ: ℓ=1, 2, …, N) for each transformed matrix. This step performs only the rank computation for the transformed matrix already in row-echelon form.

```python
def _compute_rank(self) -> int:
    """
    Computes the rank of the transformed matrix. It must be called after the matrix is correctly prepared.
    :return: the rank of the transformed matrix
    """
    # Rank start from the minimum value of rows and columns
    rank: int = self._base_rank
    i: int = 0
    # Process all the rows
    while i < self._rows:
        all_zeros: bool = True
        # Process all the columns (check if there is at least a non-zero element)
        for j in range(self._columns):
            if self._matrix[i][j] == 1:
                all_zeros = False
        # If a row is of all zeros, it's not counted towards the rank
        if all_zeros:
            rank -= 1
        i += 1
    return rank
```

4. Classify matrices in three classes, full_rank_matrices, minus_rank_matrices and remainder_matrices with frequencies:

   F_M = the number of matrices with rank R_ℓ = M,

   F_{M-1} = the number of matrices with R = M–1,

   $N - F_M - F_{M-1}$ = the number of matrices remaining.

```
# Compute the number of full rank, minus rank and remained
rank matrices
        full_rank_matrices: int = 0
        minus_rank_matrices: int = 0
        remainder: int = 0
        for i in range(blocks_number):
             # Get the bits in the block and reshape them in a
2D array (the matrix)
             block: numpy.ndarray = bits[i * (self._rows_number
*   self._cols_number):(i   +   1)   *   (self._rows_number   *
self._cols_number)].reshape((self._rows_number,
self._cols_number))
            # Compute rank of the block matrix
                    matrix:  BinaryMatrix  =  BinaryMatrix(block,
self._rows_number, self._cols_number)
            rank: int = matrix.compute_rank()
            # Count the result
            if rank == self._rows_number:
                full_rank_matrices += 1
            elif rank == self._rows_number - 1:
                minus_rank_matrices += 1
            else:
                remainder += 1
```

5. Compute χ^2 and the P-value of the observations using the expression. If the P-value<0.01, we conclude that the given binary sequence is not random else, return random.

$$\chi^2(obs) = \frac{(F_M - 0.2888N)^2}{0.2888N} + \frac{(F_{M-1} - 0.5776N)^2}{0.5776N} + \frac{(N - F_M - F_{M-1} - 0.1336N)^2}{0.1336N}$$

$$P - value = e^{-\chi^2(obs)/2} \qquad P - value = \; igamc\left(1, \frac{\chi^2(obs)}{2}\right)$$

```
# Compute Chi-square

                chi_square: float = (((full_rank_matrices -
(self._full_rank_probability * blocks_number)) ** 2) /
(self._full_rank_probability * blocks_number)) +
(((minus_rank_matrices - (self._minus_rank_probability *
blocks_number)) ** 2) / (self._minus_rank_probability *
blocks_number)) + (((remainder -
(self._remained_rank_probability * blocks_number)) ** 2) /
(self._remained_rank_probability * blocks_number))

# Compute the score (P-value)
        score: float = math.e ** (-chi_square / 2.0)

# Return result
        if score >= self.significance_value:
            return Result(self.name, True, numpy.array(score))
        return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

PUF_random_ test result - *Not eligible for the test.*

For PUF RNG, n=30,000, but as the test requires a minimum of n=38,000, they are not eligible for the test.

Python_random_ test result - *PASSED - score: 0.30*

Python_secret_ Test result - *PASSED - score: 0.208*

True_random_Test result - *PASSED - score: 0.578*

The test to randomness passes, suggesting that the data appears to be random, and there is no strong evidence of the presence of any underlying pattern or structure in the data.

It is important to note that passing a single test does not guarantee that the data is completely random. It is always possible for patterns or structures to exist in the data that are not detectable by the particular statistical test being used. Therefore, passing a statistical test for randomness should be viewed as a preliminary indication of randomness rather than definitive proof of randomness.

Discrete Fourier Transform (Spectral) Test

The Discrete Fourier Transform Test detects any pattern in the input sequence that deviates from the expected theoretical result. We assume that a truly random sequence will have a flat power spectral density while a non-random sequence will have peaks at certain frequencies.

Test Description:

1. Make the sequence even in length, and convert all 0's and 1's to -1 and +1

```
bits_copy: numpy.ndarray = bits.copy()
if (bits_copy.size % 2) == 1:
    bits_copy = bits_copy[:-1]
# Convert all the zeros in the array to -1
bits_copy[bits_copy == 0] = -1
```

2. Compute the DFT of the sequence of -1 and +1; the output of DFT will be frequency domain sequence in complex variables.

$$f_j = \sum_{k=1}^{n} x_k \exp\left(2\pi_i (k-1) j / n\right),$$

where, x_k is the k^{th} bit (k = 1, ..., n) of the sequence.

```
# Compute DFT
discrete_fourier_transform = numpy.fft.fft(bits_copy)
```

3. As there is symmetry in the complex numbers, we find the modulus of first-half numbers only. The threshold value is 0.95; thus, 95% of values of mod_j should be less than h,

$$h = \sqrt{\left(\log\frac{1}{0.05}\right)n}$$

where,

mod_j = modulus of the complex number f_j

h = peak height threshold value

```
# Compute magnitudes of first half of sequence depending on the system type
if sys.version_info > (3, 0):
    magnitudes = abs(discrete_fourier_transform)[:bits_copy.size // 2]
else:
    magnitudes = abs(discrete_fourier_transform)[:bits_copy.size / 2]
# Compute upper threshold
threshold: float = math.sqrt(math.log(1.0 / 0.05) * bits_copy.size)
```

4. Find N_0, the expected theoretical (95 %) number of peaks, and N_1, the actual observed number of peaks in M that are less than T.

$N_0 = (0.95)^* n/2$

```
# Compute the expected number of peaks (N0)
expected_peaks: float = 0.95 * bits_copy.size / 2.0
# Count the peaks above the upper threshold (N1)
counted_peaks: float = float(len(magnitudes[magnitudes < threshold]))
```

5. Compute the value of d, the normalized difference, find the P-value, and conclude the results.

$$d = \frac{(N_1 - N_0)}{\sqrt{n(.95)(.05)/4}} \qquad P\text{-}value = erfc\left(\frac{|d|}{\sqrt{2}}\right)$$

```
# Compute the score (P-value) using the normalized difference
normalized_difference: float = (counted_peaks - expected_peaks) / math.sqrt((bits_copy.size * 0.95 * 0.05) / 4)
score: float = math.erfc(abs(normalized_difference) / math.sqrt(2))
```

```
# Return result
if score >= self.significance_value:
    return Result(self.name, True, numpy.array(score))
return Result(self.name, False, numpy.array(score))
```

Results and Inferences:

PUF_random_ test result - *FAILED - score: 0.0*

Python_random_ test result - *FAILED - score: 0.0*

Python_secret_ Test result - *FAILED - score: 0.0*

True_random_Test result - *FAILED - score: 0.0*

The result implies a degree of randomness below the accepted significance level ($P<0.01$). The test fails, indicating that the binary sequence may be non-random and further investigation is necessary to identify the source of the non-randomness. There could be several reasons why the statistical test of randomness fails. For example, there may be a systematic bias in the data collection process, or the data may have been intentionally manipulated or fabricated. Alternatively, there may be some underlying pattern or structure in the data that is not immediately apparent and requires further investigation.

It is possible for a true Random Number Generator to fail some tests due to random bias, but the ratio of successful tests should be considered to term any generator as true-random or pseudo-random. Here we know that the dataset used is truly random, but it still fails the test. It can be concluded that some bias may be there, even in truly random sequences.

4. Conclusion

After performing tests with the 4 generators used in analysis we have come up with the following results:

PUF generator passed 3 tests, true random generator passed 6 tests each whereas both python-secrets and random successfully cleared 7 tests.But this result is not sufficient because we see that we both python generators were eligible for 15 tests owing to large data size whereas we could not perform the 3 tests namely:

1. Overlapping-template matching,
2. Maurer's universal test,
3. Binary matrix rank test.

So, if we see the scores:

PUF had a score of 3/12=0.25
Python secret and random test cleared 7/15=0.46
Whereas the true random passed 6/12=0.5.

Although we are aware of the fact that true random generators must as such pass all the tests but then due to the design of tests in a particular fashion and depending on the constraints of source producing it we don't get a 100% score. If we analyze the test outcomes both the non-overlapping and monobit test pass with all 4 datasets and so they are not of much utility to comment on randomness.

5. References:

[1] https://docs.python.org/3/library/random.html

[2] https://peps.python.org/pep-0506/

[3] PUF-based random number generation, CW O'Donnell, GE Suh, S Devadas
In MIT CSAIL CSG Technical Memo 481, 2004

[4] For all tests, the following publication was used.

NIST Special Publication 800-22. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Information Technology Laboratory of the National Institute of Standards and Technology, May 2000.

[5] The Python package used to implement the tests is as follows:

https://pypi.org/project/nistrng/