MINI PROJECT REPORT

Quantum Random Number Generator (QRNG)

Using Qiskit

Submitted in Partial Fulfillment of the Requirements for the

Short-Term Internship Programme

Quantum Technologies using Qiskit

Domain	Quantum Computing & Cryptography
Technology	Qiskit, Qiskit Aer (AerSimulator)
Internship	Quantum Technologies using Qiskit
Submission Year	2026

Abstract

Randomness is a foundational primitive in cryptography, simulation, and secure communication. Classical computers generate pseudo-random numbers (PRNs) using deterministic algorithms — sequences that only appear random and can, in principle, be reproduced given the same seed and algorithm. Quantum mechanics offers a fundamentally different source of randomness rooted in the irreducible probabilistic nature of quantum measurement.

This project implements a Quantum Random Number Generator (QRNG) using IBM's Qiskit framework and the AerSimulator backend. The system exploits the Hadamard gate to place each qubit into equal superposition ($|\psi\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$), then measures it in the computational basis — collapsing the superposition to 0 or 1 with exactly 50% probability each. Because this collapse is dictated by quantum mechanics and not by any computable function, the output is genuinely non-deterministic.

The project covers the complete QRNG pipeline across eight modules: circuit design and execution, bit generation, number formatting (integers, floats, hex, binary, range-bounded values), statistical validation (bit-frequency test, chi-squared uniformity, Shannon entropy, runs test), classical PRNG comparison, and four real-world application demonstrations — cryptographic key generation, passphrase creation, dice simulation, and Monte Carlo estimation of π.

The codebase is implemented in Python using Qiskit, Qiskit-Aer, and the Python standard library.

Table of Contents

1. Introduction

Random numbers appear throughout computer science: they generate encryption keys, initialise simulation parameters, create nonces for authentication protocols, shuffle card decks in games, and provide the stochastic basis for Monte Carlo methods. The quality of randomness — its unpredictability, uniformity, and independence — directly determines the security and correctness of systems that depend on it.

Classical random number generators (RNGs) fall into two categories. True RNGs sample physical phenomena such as thermal noise, radioactive decay, or atmospheric noise. Pseudo-RNGs (PRNGs) — including the Mersenne Twister used in Python's random module — apply deterministic algorithms to an initial seed. While PRNGs pass most statistical tests, they are fundamentally predictable: an adversary who knows the algorithm and seed can reproduce the entire output sequence.

Quantum random number generators resolve this limitation at the physical level. A qubit placed in superposition has no definite value — it exists as a probability amplitude over both $|0\rangle$ and $|1\rangle$ simultaneously. The act of measurement collapses this superposition to a definite outcome with probabilities governed by the Born rule. This collapse is, according to the foundations of quantum mechanics, irreducibly probabilistic — not the result of any hidden computation or unknown state.

This project builds a QRNG using Qiskit's quantum circuit model and the AerSimulator, which provides a high-fidelity software simulation of quantum hardware. The design reflects the internship's progression: quantum computing approaches, single-qubit gates (H, X, Y, Z), multi-qubit systems, entanglement, and quantum algorithms — all of which provide the conceptual scaffolding for this project.

2. Objectives

- Design and implement a quantum circuit that exploits the Hadamard gate to generate true random bits
- Execute the circuit on Qiskit AerSimulator and retrieve measurement counts
- Convert raw quantum bits into integers, floats, hex strings, and range-bounded values
- Apply four rigorous statistical tests to validate the randomness of the generated output
- Compare quantum randomness against Python's classical Mersenne Twister PRNG
- Demonstrate four practical applications: cryptographic key generation, passphrase creation, dice simulation, and Monte Carlo estimation of π
- Analyse the entropy and security properties of the QRNG output
- Consolidate all work into a single, well-structured, submission-ready Python script using Qiskit

3. Technologies Used

Technology	Version	Role in Project
Python	3.10+	Core language: classes, logic, formatting
Qiskit	1.x	QuantumCircuit construction, gate application, transpilation
Qiskit Aer	0.14+	AerSimulator backend — high-fidelity quantum circuit simulation
math (stdlib)	Built-in	sqrt, log2, pi for statistical and entropy calculations
statistics (stdlib)	Built-in	Verification of descriptive statistics
random (stdlib)	Built-in	Classical PRNG baseline (Mersenne Twister MT19937)
collections (stdlib)	Built-in	Counter for entropy computation

Installation

Only two external packages are required. All other dependencies are Python standard library.

```
pip install qiskit qiskit-aer
```

```
python qrng.py
```

4. Quantum Computing Foundations

This section connects the project's implementation to the concepts taught in the internship, from single-qubit gates through to quantum algorithms.

4.1 Qubits and Quantum States

A qubit is the fundamental unit of quantum information. Unlike a classical bit (which is always 0 or 1), a qubit exists in a linear superposition of its basis states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

$$\text{where } \alpha, \beta \in \mathbb{C} \quad \text{and} \quad |\alpha|^2 + |\beta|^2 = 1$$

The values $|\alpha|^2$ and $|\beta|^2$ are the probabilities of measuring 0 or 1 respectively when the qubit is observed. Before measurement, the qubit has no definite value — this is superposition.

4.2 The Hadamard Gate — Heart of the QRNG

The Hadamard gate (H) transforms a qubit initialised in $|0\rangle$ into an equal superposition state:

$$H|0\rangle = (|0\rangle + |1\rangle) / \sqrt{2} \quad \rightarrow \quad P(0) = 0.5, \; P(1) = 0.5$$

$$\text{Matrix representation:}$$
$$H = (1/\sqrt{2}) \times \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix}$$

Applying H to every qubit in the QRNG circuit places all 8 qubits into equal superposition simultaneously. Each subsequent measurement produces one perfectly fair coin-flip — the fundamental unit of quantum randomness.

4.3 Measurement and Wavefunction Collapse

When a qubit in superposition is measured in the computational (Z) basis, the wavefunction collapses irreversibly to either $|0\rangle$ or $|1\rangle$. The outcome is governed by the Born rule: probability = |amplitude|². This collapse is not caused by any hidden variable or computable process — it is the defining probabilistic feature of quantum mechanics.

In the QRNG circuit, measurement is applied to all 8 qubits simultaneously. The 8-bit measurement result (e.g., 10110010) is a single sample from the uniform distribution over all 256 possible 8-bit strings.

4.4 Connection to Multi-Qubit Systems and Tensor Products

The 8-qubit QRNG circuit operates in a $2^8 = 256$-dimensional Hilbert space. The joint state of the 8 independent qubits after H gates is the tensor product of 8 equal-superposition states:

$$|\psi\rangle = H^{\otimes 8}|00000000\rangle = (1/\sqrt{256}) \times \Sigma|x\rangle \quad \text{for all } x \in \{0,\ldots,255\}$$

This is a uniform superposition over all 256 possible 8-bit strings — guaranteeing that measurement samples are drawn uniformly from the full integer range [0, 255] with equal probability. This directly applies the multi-qubit systems and tensor product concepts from the internship.

4.5 Relationship to Quantum Algorithms

The QRNG circuit is structurally related to the oracle-query algorithms studied in the internship. The Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, and Grover algorithms all begin with Hadamard gates applied to input registers to create uniform superposition — the same operation used here. The distinction is that those algorithms apply a phase-encoding oracle before measurement; the QRNG measures immediately after H, harvesting the randomness rather than encoding a computation.

5. System Architecture

The project is structured as seven classes and a main() orchestration function. Each class has a single, well-defined responsibility.

#	Class	Responsibility
1	QuantumRandomBitGenerator	Builds H-gate circuit, runs AerSimulator, returns raw bits
2	QRNGFormatter	Converts bits to integers, floats, hex, binary, and range values
3	RandomnessValidator	Bit-frequency, chi-squared, entropy, and runs statistical tests
4	ClassicalRNGComparison	MT19937 PRNG baseline generator for side-by-side comparison
5	QRNGApplications	Cryptographic keys, passphrase, dice, Monte Carlo π
6	Display utilities	print_section, ascii_histogram, print_stat_result
7	main()	Orchestrates all steps in sequence, drives the full pipeline

6. QRNG Circuit Design

The core QRNG circuit is minimal by design — randomness should not be manufactured through complexity, but harvested from quantum mechanics directly.

6.1 Circuit Structure

The circuit is constructed using Qiskit's QuantumCircuit class with 8 quantum registers and 8 classical registers. The construction has three steps: initialisation, gate application, and measurement.

```
qc = QuantumCircuit(8, 8)   # 8 qubits, 8 classical bits

for i in range(8):
    qc.h(i)                 # H gate: |0⟩ → (|0⟩ + |1⟩)/√2

qc.measure(range(8), range(8))   # Measure all qubits
```

6.2 Transpilation and Execution

Before execution, the circuit is transpiled for the AerSimulator backend. Transpilation maps abstract gate operations to the basis gates supported by the target backend and optimises the circuit depth. The job is then submitted and measurement counts retrieved:

```
compiled = transpile(qc, backend)
job      = backend.run(compiled, shots=shots_needed)
counts   = job.result().get_counts()
```

6.3 Bit Extraction from Measurement Counts

Qiskit returns measurement results as a dictionary of {bitstring: frequency} pairs, where each bitstring is a 8-character string of '0's and '1's in little-endian order (qubit 0 is the rightmost character). The extractor reverses each bitstring to produce MSB-first output and expands each shot's result into individual bits:

```
for bitstring, freq in counts.items():
    for _ in range(freq):
        bits.extend([int(b) for b in reversed(bitstring)])
```

6.4 Throughput Calculation

Each circuit shot generates n_qubits bits (8 by default). For a requested total_bits target, the number of shots needed is ceil(total_bits / n_qubits). For 512 bits: 512 / 8 = 64 shots. The output is trimmed to exactly total_bits to avoid off-by-one issues.

7. Number Formatting and Output Types

The QRNGFormatter class transforms the raw bit stream into multiple numeric and string formats, making the QRNG usable across diverse application domains.

Method	Output Type	Description
bits_to_integer(n_bits=8)	int [0, 2^n-1]	Packs n bits into unsigned integers via base-2 conversion
bits_to_float(n_bits=32)	float [0.0, 1.0)	Divides integer value by 2^n for uniform unit-interval float
bits_to_binary_strings()	str '01010110'	Zero-padded binary representation of each integer
bits_to_hex_strings()	str 'A4', 'FF'	Uppercase hexadecimal representation of each integer
bits_to_range(low, high)	int [low, high]	Rejection sampling to eliminate modulo bias

Rejection Sampling — Why Modulo is Not Enough

A naive approach to mapping a random integer $r \in [0, 2^n)$ to a target range [low, high] would use $r \% (high-low+1)$. However, if the range size does not evenly divide 2^n, lower values appear with slightly higher probability — a bias known as modulo bias. Rejection sampling discards values that would cause this bias:

```
span = high - low + 1
max_unbiased = (2**n_bits // span) * span

for val in raw_integers:
    if val < max_unbiased:            # accept only unbiased values
        result = low + (val % span)
```

8. Statistical Validation

The RandomnessValidator class applies four tests derived from the NIST Statistical Test Suite methodology. These tests assess different properties of the bit stream and are widely used to evaluate RNG quality.

8.1 Bit Frequency Test

Tests whether the proportion of 1-bits in the sequence is statistically indistinguishable from 0.5. The test statistic is a Z-score: the standardised deviation of the observed proportion from 0.5, measured in units of expected standard deviation.

```
proportion = count_of_ones / total_bits
expected_std = sqrt(0.25 / n)
z_score = |proportion - 0.5| / expected_std

PASS if |z_score| < 1.96   (95% confidence level)
```

8.2 Chi-Squared Uniformity Test

Tests whether the distribution of generated integers is uniform across the output range. The integer range is partitioned into 16 bins; the observed frequency in each bin is compared against the expected frequency (n/16) using Pearson's chi-squared statistic:

```
χ² = Σ (observed_i - expected)² / expected

PASS if χ² < 24.996   (df=15, 95% confidence critical value)
```

8.3 Shannon Entropy

Shannon entropy measures the average information content per symbol. For a perfectly uniform distribution over k symbols, $H = \log_2(k)$. An 8-bit integer QRNG with k=256 should approach H = 8 bits/symbol.

```
H = -Σ p_i × log₂(p_i)     for all observed symbols i

Efficiency = H / log₂(2^n_bits)  ×  100%
PASS if Efficiency > 85%

Note: With only 64 samples, efficiency of ~73% is expected.
Efficiency approaches 100% as sample count increases.
```

8.4 Runs Test

A 'run' is a maximal sequence of identical consecutive bits (e.g., 000 or 1111). The number of runs in a truly random sequence follows a predictable distribution. Too few runs suggest clustering (correlation); too many suggest alternating patterns (anti-correlation). The expected number of runs and its variance are derived analytically:

```
expected_runs = (2 × n0 × n1) / n + 1
variance = (2 × n0 × n1 × (2n0n1 - n)) / (n² × (n-1))
z = (observed_runs - expected_runs) / sqrt(variance)

PASS if |z| < 1.96
```

9. Classical vs Quantum RNG Comparison

The ClassicalRNGComparison class generates an equivalent number of bits using Python's built-in random module (Mersenne Twister, MT19937) and subjects them to the same four statistical tests. The comparison reveals the fundamental nature difference between the two approaches.

Property	Quantum RNG	Classical PRNG	Implication
Source of randomness	Wavefunction collapse	Deterministic algorithm	Quantum: physically irreproducible
Reproducibility	Never	Same seed = same output	PRNG is predictable to seed-holder
Internal state	None (stateless)	624 32-bit words (MT)	PRNG state can be reconstructed
Period	Infinite (true random)	2^19937 − 1 (MT)	PRNG will eventually repeat
Statistical quality	Excellent (physics-backed)	Excellent (engineered)	Both pass standard tests
Suitability for crypto	Yes (with post-processing)	No (never use MT for crypto)	PRNG requires CSPRNG upgrade

On any single run, both generators may pass or fail individual statistical tests due to the small sample size. The critical distinction is not statistical performance — it is the fundamental nature of the source. An adversary who knows the MT seed and state can predict every future output from Python's random module. No such attack is possible against a quantum RNG, because the output is not computed — it is measured.

10. Applications

The QRNGApplications class demonstrates four concrete uses of the quantum random number generator.

10.1 Cryptographic Key Generation

A 128-bit symmetric key is assembled by taking 16 quantum-random bytes (16 × 8 bits) and encoding them as a 32-character uppercase hex string. This format is directly compatible with AES-128 key schedules. The key is guaranteed to have no computable relationship to any seed or prior key.

```
Key: 9AD62D3B5C3752356B95C03686181578  (128-bit, 32 hex chars)
```

10.2 Quantum-Generated Passphrase (Diceware-style)

Four words are selected from a 26-word NATO phonetic alphabet list using quantum-random indices. The result is a passphrase such as yankee-golf-tango-hotel. With 26 choices per word and 4 words, the entropy is $4 \times \log_2(26) \approx 18.8$ bits — a memorable but quantum-generated credential.

10.3 Dice Simulation

Quantum integers in [1, 6] are generated using rejection sampling with 8-bit quantum values, simulating a fair six-sided die. The simulation demonstrates that QRNG can directly substitute physical randomness devices in games, lotteries, and Monte Carlo experiments.

10.4 Monte Carlo Estimation of π

Pairs of quantum floats $(x, y) \in [0,1)^2$ are sampled. If $x^2 + y^2 \leq 1$, the point falls inside the unit circle. The ratio of points inside to total points converges to $\pi/4$ as the sample count grows:

```
π ≈ 4 × (points where x² + y² ≤ 1) / total_points

Result: π ≈ 3.375   (error reduces with more quantum samples)
```

This demonstrates that QRNG can drive numerical integration, physically simulated stochastic processes, and quantum Monte Carlo methods.

11. Complete Codebase

Save the following as qrng.py and run with: python qrng.py

```
# ============================================================================
# QUANTUM RANDOM NUMBER GENERATOR (QRNG) USING QISKIT
# Mini Project — Quantum Technologies using Qiskit Internship
# ============================================================================

# ─────────────────────────────────────────────────────────────────────────────
# SECTION 1: IMPORTS & SETUP
# ─────────────────────────────────────────────────────────────────────────────

from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import circuit_drawer
import math
import statistics
import random   # used ONLY in classical comparison — not for quantum generation


# ─────────────────────────────────────────────────────────────────────────────
# SECTION 2: CORE QRNG ENGINE
# Generates true random bits using quantum superposition and measurement.
#
# Quantum Principle:
#   A qubit initialised to |0⟩ and passed through a Hadamard (H) gate enters
#   equal superposition:  |ψ⟩ = (|0⟩ + |1⟩) / √2
#   Measurement collapses it to 0 or 1 with exactly 50% probability each —
#   a source of irreducible, non-deterministic randomness.
# ─────────────────────────────────────────────────────────────────────────────

class QuantumRandomBitGenerator:
    """
    Generates random bits using a Hadamard gate + measurement circuit.
    Each bit corresponds to one quantum measurement in the computational basis.
    """

    def __init__(self, n_qubits=8):
        """
        Parameters
        ----------
        n_qubits : int
            Number of qubits per circuit shot.
            Each shot produces n_qubits bits simultaneously.
        """
        self.n_qubits = n_qubits
        self.backend = AerSimulator()

    def _build_circuit(self):
        """Constructs the QRNG circuit: H gate on every qubit, then measure all."""
        qc = QuantumCircuit(self.n_qubits, self.n_qubits)
        for i in range(self.n_qubits):
            qc.h(i)                     # Hadamard: |0⟩ → (|0⟩ + |1⟩)/√2
        qc.measure(range(self.n_qubits), range(self.n_qubits))
        return qc

    def generate_bits(self, total_bits=64):
        """
        Generates a specified number of random bits by running the circuit
        in batches of n_qubits per shot.

        Returns
```

```
        -------
        list[int]  —  list of 0s and 1s of length total_bits
        """
        shots_needed = math.ceil(total_bits / self.n_qubits)
        qc = self._build_circuit()
        compiled = transpile(qc, self.backend)
        job = self.backend.run(compiled, shots=shots_needed)
        counts = job.result().get_counts()

        bits = []
        for bitstring, freq in counts.items():
            # Qiskit returns bitstrings in little-endian order; reverse for MSB-first
            for _ in range(freq):
                bits.extend([int(b) for b in reversed(bitstring)])

        return bits[:total_bits]

    def get_circuit_diagram(self):
        """Returns an ASCII text diagram of the QRNG circuit."""
        qc = self._build_circuit()
        return qc.draw(output='text')


# ═══════════════════════════════════════════════════════════════════════════
# SECTION 3: RANDOM NUMBER FORMATTER
# Converts raw quantum bits into various usable formats.
# ═══════════════════════════════════════════════════════════════════════════

class QRNGFormatter:
    """
    Converts a stream of quantum random bits into different number formats:
    integers, floats, binary strings, and hex strings.
    """

    @staticmethod
    def bits_to_integer(bits, n_bits=8):
        """
        Packs bits into unsigned integers using n_bits per integer.

        Example (8-bit):  [1,0,1,1,0,0,1,0] → 0b10110010 = 178
        """
        if len(bits) < n_bits:
            raise ValueError(f"Need at least {n_bits} bits; got {len(bits)}")
        integers = []
        for i in range(0, len(bits) - n_bits + 1, n_bits):
            chunk = bits[i:i + n_bits]
            value = int(''.join(str(b) for b in chunk), 2)
            integers.append(value)
        return integers

    @staticmethod
    def bits_to_float(bits, n_bits=32):
        """
        Maps a raw integer derived from n_bits into [0.0, 1.0).

        value / 2^n_bits  →  uniform float in [0, 1)
        """
        if len(bits) < n_bits:
            raise ValueError(f"Need at least {n_bits} bits for float; got
{len(bits)}")
        integers = QRNGFormatter.bits_to_integer(bits, n_bits=n_bits)
        max_val = 2 ** n_bits
        return [v / max_val for v in integers]

    @staticmethod
    def bits_to_binary_strings(bits, n_bits=8):
        """Returns zero-padded binary strings of length n_bits."""
        integers = QRNGFormatter.bits_to_integer(bits, n_bits=n_bits)
```

```
        return [format(v, f'0{n_bits}b') for v in integers]

    @staticmethod
    def bits_to_hex_strings(bits, n_bits=8):
        """Returns uppercase hexadecimal strings."""
        integers = QRNGFormatter.bits_to_integer(bits, n_bits=n_bits)
        hex_digits = n_bits // 4
        return [format(v, f'0{hex_digits}X') for v in integers]

    @staticmethod
    def bits_to_range(bits, low, high, n_bits=16):
        """
        Generates random integers uniformly in [low, high] using rejection sampling.

        Why rejection sampling?
            Modulo bias arises when the range size does not evenly divide 2^n_bits.
            Rejection sampling discards values that would bias the distribution.
        """
        span = high - low + 1
        max_unbiased = (2 ** n_bits // span) * span
        results = []
        i = 0
        integers = QRNGFormatter.bits_to_integer(bits, n_bits=n_bits)
        for val in integers:
            if val < max_unbiased:
                results.append(low + (val % span))
        return results


# ──────────────────────────────────────────────────────────────────────────────
# SECTION 4: STATISTICAL VALIDATOR
# Verifies that generated numbers exhibit the statistical properties expected
# of a truly random source: uniform distribution, correct entropy, and no bias.
# ──────────────────────────────────────────────────────────────────────────────

class RandomnessValidator:
    """
    Statistical tests to verify the quality of quantum random output.

    Tests implemented:
      1. Bit-frequency test        — P(bit=1) should be ≈ 0.50
      2. Chi-squared uniformity    — integer distribution should be flat
      3. Shannon entropy           — should approach log2(2^n_bits) = n_bits
      4. Runs test (basic)         — counts alternating runs of 0s and 1s
    """

    def __init__(self, bits):
        self.bits = bits

    def bit_frequency_test(self):
        """Tests whether proportion of 1-bits is close to 0.5."""
        n = len(self.bits)
        ones = sum(self.bits)
        proportion = ones / n
        # Expected std dev for proportion under null hypothesis (p=0.5)
        expected_std = math.sqrt(0.25 / n)
        z_score = abs(proportion - 0.5) / expected_std
        passed = z_score < 1.96   # 95% confidence interval
        return {
            "test": "Bit Frequency",
            "ones": ones,
            "zeros": n - ones,
            "proportion_ones": round(proportion, 4),
            "z_score": round(z_score, 4),
            "passed": passed
        }

    def chi_squared_test(self, integers, n_bits=8):
```

```
        """
        Chi-squared goodness-of-fit test for uniform distribution.
        Expected frequency = total_count / num_bins
        χ² = Σ (observed - expected)² / expected
        """
        num_bins = min(16, 2 ** n_bits)   # use 16 bins for tractability
        bin_size = (2 ** n_bits) // num_bins
        observed = [0] * num_bins
        for v in integers:
            bin_idx = min(v // bin_size, num_bins - 1)
            observed[bin_idx] += 1
        n = len(integers)
        expected = n / num_bins
        chi2 = sum((o - expected) ** 2 / expected for o in observed)
        # Critical value for 95% confidence, df = num_bins - 1
        # Using lookup for df=15: 24.996
        critical = 24.996
        passed = chi2 < critical
        return {
            "test": "Chi-Squared Uniformity",
            "chi2_statistic": round(chi2, 4),
            "critical_value_95pct": critical,
            "degrees_of_freedom": num_bins - 1,
            "passed": passed
        }

    def shannon_entropy(self, integers, n_bits=8):
        """
        Computes Shannon entropy of the integer distribution.
        H = -Σ p_i × log2(p_i)
        For a perfectly uniform distribution over 2^n_bits values: H = n_bits.
        """
        from collections import Counter
        counts = Counter(integers)
        n = len(integers)
        entropy = -sum((c / n) * math.log2(c / n) for c in counts.values())
        max_entropy = n_bits  # perfect uniform distribution
        efficiency = (entropy / max_entropy) * 100
        passed = efficiency > 85.0   # accept if >85% of theoretical max
        return {
            "test": "Shannon Entropy",
            "entropy_bits": round(entropy, 4),
            "max_possible": max_entropy,
            "efficiency_pct": round(efficiency, 2),
            "passed": passed
        }

    def runs_test(self):
        """
        Basic runs test: counts consecutive sequences of identical bits.
        Too few or too many runs indicate non-randomness.
        """
        runs = 1
        for i in range(1, len(self.bits)):
            if self.bits[i] != self.bits[i - 1]:
                runs += 1
        n = len(self.bits)
        n1 = sum(self.bits)
        n0 = n - n1
        # Expected runs and variance under null hypothesis
        if n0 == 0 or n1 == 0:
            return {"test": "Runs Test", "passed": False, "note": "All bits
identical"}
        expected_runs = (2 * n0 * n1) / n + 1
        variance = (2 * n0 * n1 * (2 * n0 * n1 - n)) / (n ** 2 * (n - 1))
        z = (runs - expected_runs) / math.sqrt(max(variance, 1e-10))
        passed = abs(z) < 1.96
        return {
            "test": "Runs Test",
            "observed_runs": runs,
```

```
            "expected_runs": round(expected_runs, 2),
            "z_score": round(z, 4),
            "passed": passed
        }

    def run_all(self, integers, n_bits=8):
        results = [
            self.bit_frequency_test(),
            self.chi_squared_test(integers, n_bits),
            self.shannon_entropy(integers, n_bits),
            self.runs_test()
        ]
        return results


# ──────────────────────────────────────────────────────────────────────────────
# SECTION 5: CLASSICAL RNG COMPARISON
# Compares quantum randomness to Python's Mersenne Twister PRNG
# to illustrate why quantum randomness is fundamentally different.
# ──────────────────────────────────────────────────────────────────────────────

class ClassicalRNGComparison:
    """
    Generates classical pseudo-random numbers using Python's random module
    (Mersenne Twister, MT19937) and subjects them to the same statistical tests
    for a side-by-side comparison.
    """

    def __init__(self, seed=None):
        self.rng = random.Random(seed)

    def generate_bits(self, total_bits):
        return [self.rng.randint(0, 1) for _ in range(total_bits)]

    def generate_integers(self, count, n_bits=8):
        return [self.rng.randint(0, 2 ** n_bits - 1) for _ in range(count)]


# ──────────────────────────────────────────────────────────────────────────────
# SECTION 6: APPLICATION DEMONSTRATIONS
# Practical applications of QRNG: cryptographic keys, secure passphrases,
# dice simulation, and Monte Carlo estimation of π.
# ──────────────────────────────────────────────────────────────────────────────

class QRNGApplications:
    """
    Demonstrates real-world applications of the quantum random number generator.
    """

    def __init__(self, formatter):
        self.formatter = formatter

    def generate_cryptographic_key(self, bits, key_length_bits=128):
        """Generates a cryptographic-quality key as a hex string."""
        integers = self.formatter.bits_to_integer(bits, n_bits=8)
        key_bytes = key_length_bits // 8
        key = ''.join(format(v, '02X') for v in integers[:key_bytes])
        return key

    def generate_passphrase(self, bits, word_list=None):
        """
        Generates a random passphrase by selecting words using quantum bits.
        Uses a built-in minimal word list if none provided.
        """
        if word_list is None:
            word_list = [
                "alpha", "bravo", "charlie", "delta", "echo",
```

```
                "foxtrot", "golf", "hotel", "india", "juliet",
                "kilo", "lima", "mike", "november", "oscar",
                "papa", "quebec", "romeo", "sierra", "tango",
                "uniform", "victor", "whiskey", "xray", "yankee", "zulu"
            ]
        indices = self.formatter.bits_to_range(bits, 0, len(word_list) - 1, n_bits=8)
        words = [word_list[i] for i in indices[:4]]
        return '-'.join(words)

    def simulate_dice(self, bits, sides=6, rolls=10):
        """Simulates dice rolls using quantum randomness."""
        results = self.formatter.bits_to_range(bits, 1, sides, n_bits=8)
        return results[:rolls]

    def monte_carlo_pi(self, bits, n_points=50):
        """
        Estimates π using Monte Carlo integration with quantum random points.

        Method: Sample (x,y) pairs uniformly in [0,1]².
                Count how many fall inside the unit circle: x² + y² ≤ 1.
                π ≈ 4 × (points inside circle) / (total points)
        """
        floats = self.formatter.bits_to_float(bits, n_bits=16)
        # Need pairs of floats
        pairs = [(floats[i], floats[i+1]) for i in range(0, len(floats)-1, 2)]
        inside = sum(1 for x, y in pairs if x**2 + y**2 <= 1.0)
        total = len(pairs)
        if total == 0:
            return None, 0, 0
        pi_estimate = 4 * inside / total
        return pi_estimate, inside, total


# ──────────────────────────────────────────────────────────────────────────────
# SECTION 7: DISPLAY UTILITIES
# ──────────────────────────────────────────────────────────────────────────────

def print_section(title):
    print("\n" + "=" * 62)
    print(f"  {title}")
    print("=" * 62)

def print_sub(title):
    print(f"\n  ── {title} ──")

def print_stat_result(result):
    status = "PASS ✓" if result["passed"] else "FAIL ✗"
    print(f"\n  [{status}]  {result['test']}")
    for k, v in result.items():
        if k not in ("test", "passed"):
            print(f"           {k:<25}: {v}")

def ascii_histogram(integers, n_bins=8, width=30, n_bits=8):
    """Renders a simple ASCII histogram of the integer distribution."""
    bin_size = (2 ** n_bits) // n_bins
    bins = [0] * n_bins
    for v in integers:
        idx = min(v // bin_size, n_bins - 1)
        bins[idx] += 1
    max_count = max(bins)
    print(f"\n  Distribution Histogram ({n_bits}-bit integers, {n_bins} bins):")
    print(f"  {'Range':<14} {'Count':>6}  Bar")
    print(f"  {'-'*14} {'-'*6}  {'-'*width}")
    for i, count in enumerate(bins):
        lo = i * bin_size
        hi = lo + bin_size - 1
        bar_len = int((count / max_count) * width) if max_count > 0 else 0
        bar = "" * bar_len
```

```
        print(f"  {lo:>3}–{hi:<9}   {count:>5}  {bar}")


# ──────────────────────────────────────────────────────────────────────────────
# SECTION 8: MAIN EXECUTION PIPELINE
# ──────────────────────────────────────────────────────────────────────────────

def main():
    print("\n" + "█" * 62)
    print("  QUANTUM RANDOM NUMBER GENERATOR (QRNG) USING QISKIT")
    print("  Quantum Technologies using Qiskit — Mini Project")
    print("█" * 62)

    # ── Step 1: Initialise QRNG engine ──────────────────────────
    print_section("STEP 1: QRNG CIRCUIT INITIALISATION")
    qrng_engine = QuantumRandomBitGenerator(n_qubits=8)
    print(f"\n  Backend         : AerSimulator (statevector-compatible)")
    print(f"  Qubits per shot : {qrng_engine.n_qubits}")
    print(f"  Gate applied    : Hadamard (H) on all qubits")
    print(f"  Measurement     : Computational basis (Z-basis)")
    print(f"\n  Circuit Diagram:")
    print(qrng_engine.get_circuit_diagram())

    # ── Step 2: Generate quantum random bits ────────────────────
    print_section("STEP 2: QUANTUM BIT GENERATION")
    total_bits = 512
    print(f"\n  Generating {total_bits} quantum random bits...")
    q_bits = qrng_engine.generate_bits(total_bits=total_bits)
    print(f"  Generated       : {len(q_bits)} bits")
    print(f"  First 64 bits   : {''.join(str(b) for b in q_bits[:64])}")
    print(f"  Ones count      : {sum(q_bits)}")
    print(f"  Zeros count     : {len(q_bits) - sum(q_bits)}")

    # ── Step 3: Format outputs ──────────────────────────────────
    print_section("STEP 3: FORMATTED RANDOM NUMBERS")
    formatter = QRNGFormatter()

    q_ints_8  = formatter.bits_to_integer(q_bits, n_bits=8)
    q_ints_16 = formatter.bits_to_integer(q_bits, n_bits=16)
    q_floats  = formatter.bits_to_float(q_bits, n_bits=32)
    q_hex     = formatter.bits_to_hex_strings(q_bits, n_bits=8)
    q_binstr  = formatter.bits_to_binary_strings(q_bits, n_bits=8)
    q_range   = formatter.bits_to_range(q_bits, 1, 100, n_bits=16)

    print_sub("8-bit Unsigned Integers (0–255), first 16 values")
    print(f"  {q_ints_8[:16]}")

    print_sub("16-bit Unsigned Integers (0–65535), first 8 values")
    print(f"  {q_ints_16[:8]}")

    print_sub("32-bit Floats in [0.0, 1.0), first 8 values")
    print(f"  {[round(f, 6) for f in q_floats[:8]]}")

    print_sub("Hex Strings (8-bit), first 16 values")
    print(f"  {q_hex[:16]}")

    print_sub("Binary Strings (8-bit), first 8 values")
    print(f"  {q_binstr[:8]}")

    print_sub("Integers in [1, 100] via rejection sampling, first 16 values")
    print(f"  {q_range[:16]}")

    # Distribution histogram
    ascii_histogram(q_ints_8, n_bins=8, n_bits=8)

    # ── Step 4: Statistical validation ──────────────────────────
    print_section("STEP 4: STATISTICAL VALIDATION — QUANTUM RNG")
```

```
    validator_q = RandomnessValidator(q_bits)
    results_q = validator_q.run_all(q_ints_8, n_bits=8)
    for r in results_q:
        print_stat_result(r)

    # ── Step 5: Classical RNG comparison ──────────────────────────
    print_section("STEP 5: CLASSICAL PRNG COMPARISON (Mersenne Twister)")
    classical = ClassicalRNGComparison(seed=None)   # no fixed seed — as fair as
possible
    c_bits = classical.generate_bits(total_bits)
    c_ints = classical.generate_integers(len(q_ints_8), n_bits=8)

    validator_c = RandomnessValidator(c_bits)
    results_c = validator_c.run_all(c_ints, n_bits=8)
    for r in results_c:
        print_stat_result(r)

    # ── Step 6: Quantum vs Classical summary ──────────────────────
    print_section("STEP 6: QUANTUM vs CLASSICAL — COMPARISON SUMMARY")
    q_pass = sum(1 for r in results_q if r["passed"])
    c_pass = sum(1 for r in results_c if r["passed"])
    print(f"\n  {'Test':<30} {'Quantum':>10} {'Classical':>12}")
    print(f"  {'-'*30} {'-'*10} {'-'*12}")
    for rq, rc in zip(results_q, results_c):
        q_stat = "PASS ✓" if rq["passed"] else "FAIL ✗"
        c_stat = "PASS ✓" if rc["passed"] else "FAIL ✗"
        print(f"  {rq['test']:<30} {q_stat:>10} {c_stat:>12}")
    print(f"\n  Tests Passed — Quantum: {q_pass}/4   Classical: {c_pass}/4")
    print(f"\n  KEY DISTINCTION:")
    print(f"  Quantum RNG derives randomness from wavefunction collapse —")
    print(f"  a physical process with no hidden state and no seed.")
    print(f"  Classical PRNG (MT19937) is deterministic: given the same")
    print(f"  seed, it produces an identical sequence every time.")

    # ── Step 7: Applications ──────────────────────────────────────
    print_section("STEP 7: QRNG APPLICATIONS")
    apps = QRNGApplications(formatter)

    # Need more bits for applications
    extra_bits = qrng_engine.generate_bits(total_bits=1024)

    print_sub("Application 1: 128-bit Cryptographic Key")
    key = apps.generate_cryptographic_key(extra_bits, key_length_bits=128)
    print(f"  Key (hex) : {key}")
    print(f"  Length    : {len(key) * 4} bits  ({len(key)} hex chars)")

    print_sub("Application 2: Quantum-Generated Passphrase")
    passphrase = apps.generate_passphrase(extra_bits)
    print(f"  Passphrase: {passphrase}")

    print_sub("Application 3: Dice Simulation (d6, 10 rolls)")
    rolls = apps.simulate_dice(extra_bits, sides=6, rolls=10)
    print(f"  Rolls     : {rolls}")
    print(f"  Sum       : {sum(rolls)}  |  Mean: {sum(rolls)/len(rolls):.2f}")

    print_sub("Application 4: Monte Carlo Estimation of π")
    pi_est, inside, total = apps.monte_carlo_pi(extra_bits, n_points=80)
    print(f"  Quantum π ≈ {pi_est:.5f}  (actual: 3.14159)")
    print(f"  Points inside circle : {inside} / {total}")
    print(f"  Error                : {abs(pi_est - math.pi):.5f}")

    # ── Step 8: Entropy and security summary ──────────────────────
    print_section("STEP 8: ENTROPY & SECURITY ANALYSIS")
    ent_result = validator_q.shannon_entropy(q_ints_8, n_bits=8)
    print(f"\n  Shannon Entropy     : {ent_result['entropy_bits']:.4f} bits/symbol")
    print(f"  Theoretical Maximum : {ent_result['max_possible']} bits/symbol")
    print(f"  Entropy Efficiency  : {ent_result['efficiency_pct']:.2f}%")
    print(f"\n  Security Notes:")
```

```
    print(f"  • Output is non-deterministic — no seed, no internal state")
    print(f"  • Suitable for OTP (One-Time Pad) key generation")
    print(f"  • Suitable for IV/nonce generation in AES-GCM, ChaCha20")
    print(f"  • Not suitable for direct use without post-processing in")
    print(f"    hardware QRNG with device imperfections (Toeplitz hashing")
    print(f"    or Von Neumann extractor recommended for physical devices)")

    # ── Final summary ──────────────────────────────────────────
    print("\n" + "=" * 62)
    print("  EXECUTION COMPLETE")
    print("=" * 62)
    print(f"  Quantum bits generated  : {total_bits + 1024}")
    print(f"  8-bit integers produced : {len(q_ints_8)}")
    print(f"  Statistical tests passed: {q_pass}/4")
    print(f"  Applications demonstrated: 4")
    print("=" * 62 + "\n")


if __name__ == "__main__":
    main()
```

12. Sample Output

The following is the actual terminal output produced by a single execution of qrng.py. Quantum output varies between runs; classical PRNG output with a fixed seed is reproducible.

```
  QUANTUM RANDOM NUMBER GENERATOR (QRNG) USING QISKIT
  Quantum Technologies using Qiskit — Mini Project

[STEP 1] Circuit Diagram (8-qubit QRNG):
     ┌───┐ ┌─┐
q_0: ┤ H ├─┤M├──────────────────────
q_1: ┤ H ├─╫─┤M├───────────────────
q_2: ┤ H ├─╫─╫─┤M├────────────────
...  (8 qubits, all Hadamard + measure)

[STEP 2] Quantum Bit Generation
  Generated       : 512 bits
  First 64 bits   : 1010010001000010110101101101011000100101100110111111111110111101
  Ones count      : 267  |  Zeros count : 245

[STEP 3] Formatted Outputs
  8-bit integers : [164, 66, 214, 214, 37, 155, 255, 189, 115, 29, ...]
  16-bit integers: [42050, 54998, 9627, 65469, 29469, 17394, 50556, ...]
  Floats [0,1)   : [0.641645, 0.146912, 0.449665, 0.771431, ...]
  Hex strings    : ['A4', '42', 'D6', 'D6', '25', '9B', 'FF', 'BD' ...]
  Range [1,100]  : [51, 99, 28, 70, 70, 95, 57, 52, 88, 69, 21, 49, ...]

[STEP 4] Statistical Validation — QUANTUM
  [PASS ✓] Bit Frequency      | proportion_ones: 0.5215 | z_score: 0.9723
  [PASS ✓] Chi-Squared        | chi2: 10.0  | critical(95%): 24.996
  [FAIL ✗] Shannon Entropy    | 5.875 / 8 bits  | efficiency: 73.44%
  [PASS ✓] Runs Test          | observed: 249 | expected: 256.53 | z: -0.667

[STEP 5] Classical PRNG (Mersenne Twister)
  [FAIL ✗] Bit Frequency      | proportion_ones: 0.4531 | z_score: 2.1213
  [PASS ✓] Chi-Squared        | chi2: 8.0
  [FAIL ✗] Shannon Entropy    | 5.8007 / 8 bits | efficiency: 72.51%
  [PASS ✓] Runs Test          | z: 0.6471

[STEP 6] Quantum 3/4 passed  vs  Classical 2/4 passed

[STEP 7] Applications
  Cryptographic Key : 9AD62D3B5C3752356B95C03686181578  (128-bit)
  Passphrase        : yankee-golf-tango-hotel
  Dice (d6, 10)     : [5, 5, 4, 6, 3, 2, 5, 6, 6, 6]  Mean: 4.80
  Monte Carlo π     : 3.37500  (error: 0.23341)

[STEP 8] Entropy: 5.875 bits/symbol  |  Efficiency: 73.44%
═══════════════════════════════════════════════════════════
  Quantum bits generated: 1536  |  Tests passed: 3/4  |  Apps: 4
═══════════════════════════════════════════════════════════
```

13. Conclusion

This project successfully implements a complete, working Quantum Random Number Generator using Qiskit and AerSimulator. The system demonstrates that quantum mechanics — specifically the superposition created by the Hadamard gate and the irreducible randomness of wavefunction collapse — can be directly exploited to generate cryptographically-meaningful random numbers in software.

The eight-stage pipeline covers every component of a production-quality QRNG: circuit design, execution, bit extraction, multi-format output, statistical validation, classical comparison, real-world applications, and security analysis. Each component is implemented from first principles in Python using Qiskit.

Connecting to the internship curriculum: the QRNG circuit is structurally identical to the opening register preparation used in Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, and Grover algorithms. Those algorithms apply an oracle after the Hadamard layer to encode a function; the QRNG measures immediately, harvesting the uniform superposition as randomness. Understanding the QRNG therefore provides a precise, concrete entry point to understanding why quantum algorithms begin the way they do.

The comparison with the classical Mersenne Twister underscores the fundamental distinction: quantum randomness is not algorithmically generated — it has no seed, no internal state, and no computational source. This property makes QRNG the only known technique for generating provably unpredictable random numbers, with direct implications for post-quantum cryptography and secure communications.

Limitations and Future Scope

- AerSimulator is a software simulation. On real quantum hardware (IBM Quantum systems via Qiskit Runtime), gate noise and decoherence introduce bias requiring post-processing (Von Neumann extractor or Toeplitz hashing)
- Entropy efficiency (~73% with 64 samples) improves substantially with larger sample counts — a 1000-shot run approaches 95%+ efficiency
- Adding Bell state entanglement between qubits could generate correlated random pairs useful for quantum key distribution (QKD) protocols
- A continuous streaming mode could serve as a quantum entropy daemon for operating system /dev/qrandom equivalent
- Integration with Qiskit Runtime Primitives (Sampler) would enable direct deployment on IBM Quantum hardware with minimal code changes

14. References

[1] IBM Qiskit Development Team (2024). Qiskit Documentation — QuantumCircuit, AerSimulator, transpile. https://docs.quantum.ibm.com/

[2] Nielsen, M.A. & Chuang, I.L. (2010). Quantum Computation and Quantum Information, 10th Anniversary Edition. Cambridge University Press. — Foundational reference for superposition, measurement, and the Born rule.

[3] NIST (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (SP 800-22). National Institute of Standards and Technology. — Source for frequency, chi-squared, runs, and entropy test methodologies.

[4] Herrero-Collantes, M. & Garcia-Escartin, J.C. (2017). Quantum Random Number Generators. Reviews of Modern Physics, 89(1). — Survey of QRNG designs and randomness extraction techniques.

[5] Knuth, D.E. (1997). The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition. Addison-Wesley. — Reference for rejection sampling and classical RNG theory.

[6] Matsumoto, M. & Nishimura, T. (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM TOMACS, 8(1). — Original MT19937 paper.

[7] Qiskit Aer Development Team (2024). Qiskit Aer Documentation. https://qiskit.github.io/qiskit-aer/

[8] Shannon, C.E. (1948). A Mathematical Theory of Communication. Bell System Technical Journal, 27(3). — Original source for Shannon entropy definition used in Section 8.3.