

QRNG Simulation

by Arif

This code benchmarks three different methods of random number generation: quantum simulation, classical cryptographic, and true quantum hardware.

Installation and Imports

```
!pip install qiskit qiskit-aer matplotlib
```

Installs required packages: Qiskit (quantum computing framework), Qiskit-Aer (quantum circuit simulator), and Matplotlib (plotting).

```
import time
import secrets
import requests
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
```

Imports necessary libraries for timing, cryptographic randomness, HTTP requests, plotting, and quantum circuit simulation.

Class 1: Quantum Simulator RNG

```
class QuantumSimulatorRNG:
    def __init__(self):
        self.backend = AerSimulator()
```

Creates a class for simulating quantum random number generation using Qiskit's local simulator backend.

```
def generate_bits(self, num_bits):
    qc = QuantumCircuit(1, 1)
```

Creates a quantum circuit with 1 qubit and 1 classical bit to store measurement results.

```
qc.h(0)
```

Applies a Hadamard gate to qubit 0, putting it in superposition (50% chance of measuring 0 or 1).

```
qc.measure(0, 0)
```

Measures qubit 0 and stores the result in classical bit 0.

```
job = self.backend.run(qc, shots=num_bits, memory=True)
```

Executes the circuit multiple times (shots=num_bits), with memory=True to record individual measurement outcomes.

```
result = job.result()
memory = result.get_memory()
```

Retrieves the job results and extracts the memory containing all individual measurement outcomes.

```
return "".join(memory)
```

Concatenates all the individual bit measurements into a single binary string.

Class 2: ANU Web API QRNG

```
class AnuWebQRNG:
    def __init__(self):
        self.url = "https://qrng.anu.edu.au/API/jsonI.php"
```

Creates a class that connects to Australian National University's real quantum random number generator API.

```
def generate_bits(self, num_bits):
    num_bytes = (num_bits // 8) + 1
```

Calculates how many bytes are needed (8 bits per byte), adding 1 to ensure enough bits.

```
params = {
    "length": num_bytes,
    "type": "uint8"
}
```

Prepares API parameters requesting unsigned 8-bit integers.

```
try:
    response = requests.get(self.url, params=params, timeout=10)
    response.raise_for_status()
```

Sends HTTP GET request to the API with 10-second timeout, then checks for HTTP errors.

```
data = response.json()["data"]
```

Extracts the random numbers from the JSON response.

```
binary_string = "".join(f"{x:08b}" for x in data)
```

Converts each number to 8-bit binary format and concatenates them.

```
return binary_string[:num_bits]
```

Returns exactly the requested number of bits by slicing the string.

```
except Exception as e:
    return f"Error: {e}"
```

Catches any errors and returns an error message.

Class 3: Classical RNG

```
class ClassicalRNG:  
    def generate_bits(self, num_bits):  
        num_bytes = (num_bits // 8) + 1
```

Creates a class using Python's cryptographically secure random number generator, calculating required bytes.

```
    random_bytes = secrets.token_bytes(num_bytes)
```

Generates cryptographically strong random bytes using OS entropy sources.

```
    binary_string = "".join(f"{x:08b}" for x in random_bytes)  
    return binary_string[:num_bits]
```

Converts bytes to binary string and returns the exact number of bits requested.

Benchmarking Function

```
def run_benchmark(target_bits=1024):  
    generators = {  
        "Local Qiskit Sim": QuantumSimulatorRNG(),  
        "Classical OS": ClassicalRNG(),  
        "Remote ANU API": AnuWebQRNG()  
    }
```

Creates instances of all three random number generators for comparison.

```
    results = {}  
    print(f"\n--- Benchmarking Generation of {target_bits} bits ---")
```

Initializes results dictionary and prints benchmark header.

```
for name, gen in generators.items():
    start_time = time.perf_counter()
    bits = gen.generate_bits(target_bits)
    end_time = time.perf_counter()
```

Iterates through each generator, measuring execution time using high-precision timer.

```
duration = end_time - start_time
if "Error" in bits:
    print(f"{name:<20} | FAILED      | {bits}")
    continue
```

Calculates duration and skips failed generators.

```
speed = target_bits / duration
results[name] = duration
print(f"{name:<20} | {duration:.5f} s | {speed:.0f}")
```

Calculates bits per second, stores duration, and prints formatted results.

Visualization Function

```
def plot_results(results):
    names = list(results.keys())
    times = list(results.values())
    plt.figure(figsize=(10, 6))
    bars = plt.bar(names, times, color=['#6929c4', '#009d9a', '#1192e8'])
```

Extracts names and times, creates bar chart with specified colors.

```
plt.ylabel('Time to Generate (Seconds)')
plt.title('QRNG Simulation Time Comparison')
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

Adds labels, title, and gridlines to the plot.

```
for bar in bars:  
    yval = bar.get_height()  
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.4f}s', va...)
```

Adds time labels on top of each bar.

Main Execution

```
if __name__ == "__main__":  
    data = run_benchmark(target_bits=2000)  
    plot_results(data)
```

Runs the benchmark with 2000 bits and displays the results chart.

The results show Classical OS is fastest (0.00014s), followed by Local Qiskit Sim (0.00897s), while Remote ANU API is slowest (3.02385s) due to network latency.