Quantum Galton Board

Julio Cesar Flores Molina[1], Mirian Ahuatzi[2], Ana Noguera[3]

[1]juliocesarfmz7@gmail.com [2]mirianahc3@gmail.com [3]ana.noguerav011@gmail.com

1 Introduction

The Galton board is a classical device that illustrates how a sequence of binary random events converges to a Gaussian distribution. Its quantum counterpart, known as the Quantum Galton Board (QGB), simulates this process using Hadamard and controlled-SWAP gates to place a quantum "ball" in superposition and propagate it through multiple paths simultaneously.
As part of the WISER 2025 challenge—developed in collaboration with the Naval Nuclear Laboratory (NNL)—this project implements and analyzes QGB circuits using the Qiskit framework. The four-level circuit code provided in [1] was implemented, and it was found to reproduce the expected Gaussian profile. These results confirm the ability of quantum circuits to emulate classical probabilistic dynamics.
This implementation provides a foundation for completing the remaining challenge deliverables, including generalizing the circuit for arbitrary depth, constructing biased versions to target exponential and Hadamard-walk distributions. The project will conclude by computing statistical distances to compare the measured and theoretical distributions, accounting for stochastic uncertainty.

2 The Quantum Galton Board

The Quantum Galton Board (QGB) models the classical Galton board's probabilistic behavior using quantum circuits based on superposition and entanglement. Instead of choosing a path at each peg, a quantum "ball" explores all possible trajectories simultaneously. Using Hadamard and controlled-SWAP gates, the QGB efficiently encodes 2^n classical outcomes with polynomial resources [1], contributing to broader applications of quantum hardware in statistical simulations [2].

2.1 Quantum peg

The core element of the QGB is a quantum peg. In the case of only one peg, it uses a control qubit q_0 placed in superposition with a Hadamard gate and three data qubits q_1, q_2, q_3, where $q_2 = 1$ represents the quantum "ball", and q_1 and q_2 are in zero state at the beginning. The circuit performs a controlled-SWAP between q_1 and q_2, a CNOT from q_2 to q_0, and a SWAP from q_2 to q_3, simulating one binary decision step in superposition:

$$q_3 q_2 q_1 q_0 = \frac{1}{\sqrt{2}}(0011 + 1001).$$

The single peg circuit can be seen in figure 1, note that it is composed of two swap gates and one control NOT gate, in addition to the X gate and Hadamard gate, it will be a little different in the generalization.

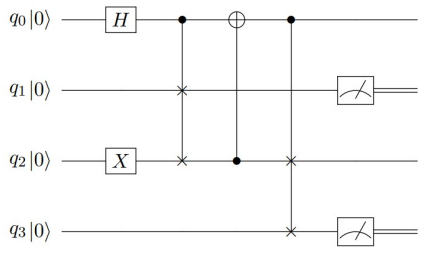

Figure 1: A Quantum circuit analogue of the physical peg. The input channel or 'ball' is indicated by q_2 and the output channels correspond to the measurement outcomes.

2.2 Multi-Level Construction

Multiple peg units can be chained to simulate deeper levels. The control qubit is reset and reused at each level to conserve resources, with additional corrections (such as inverted CNOTs) used to preserve balanced amplitudes. For n levels our scheme requires $2n+2$ qubits; for example, a 2-level QGB includes 3 pegs and 6 qubits, while a 4-level QGB comprises 10 pegs and 10 qubits. The modular design facilitates circuit scaling without exponential growth in gate depth [1].

2.3 Output Measurement

Only the data qubits are measured after each circuit run. For each shot, one qubit collapses to 1, indicating the final position of the quantum ball. Repeating this process over many shots produces a histogram of outcomes. In the unbiased case, the probability distribution closely follows a binomial law:

$$P(k) \approx \frac{1}{2^n}\binom{n}{k}.$$

This outcome approximates a discrete Gaussian distribution, consistent with classical expectations [2].

3 Implementation of a QGB

The basic quantum peg, showed in Fig 1, is composed of a control qubit (q_0) and three working qubits (q_1, q_2, q_3). All qubits are in initial state $|0\rangle$, with the exception of q_2, to which an X gate is applied to represent that the ball is in that channel.
Next, a *Hadamard* gate is applied to the control qubit q_0, thus generating a superposition.
Then to emulate the first two-path split point for the ball, a gate *CSWAP* is applied between the q_1 and q_2 qubits, which exchanges the value between the two depending on the value in q_0. Giving the state as

$$|q_2 q_1 q_0\rangle = \frac{1}{\sqrt{2}}(|011\rangle + |100\rangle) \tag{1}$$

Subsequently, an inverse *CNOT* is applied from q_2 to q_0, stabilizing the overlap with respect to the traveled path, becoming the state as

$$|q_2 q_1 q_0\rangle = \frac{1}{\sqrt{2}}(|011\rangle + |101\rangle) \tag{2}$$

Finally other gate *CSWAP* is applied between q_2 and q_3 to give the desired final state

$$|q_3 q_2 q_1 q_0\rangle = \frac{1}{\sqrt{2}}(|0011\rangle + |1001\rangle) \tag{3}$$

obtaining the final result as a state of superposition of the trajectories.
Now, for implement the peg at 2 levels, i.e. 3 pegs, as we can see in fig 2. The circuit implements 6 qubits, with 3 classical bits and can be divided into three sections. The first section is identical to the last circuit of one peg, in the second section the control cubit $|q_0\rangle$ is reset; and it is used to implement two new parallel pegs on the previous output paths, q_2 and q_4. Additionally, the same quantum peg logic is applied using the *Hadamard*, *CSWAPs* and *CNOTs* gates, adding an additional inverted CNOT, which return 50% probability qubit control, allowing to us to implement the quantum peg to the other half of the circuit.

Finally, the qubits $|q_1\rangle$, $|q_3\rangle$, and $|q_5\rangle$ evolve into the following state:

$$|q_5 q_3 q_1\rangle = \frac{1}{\sqrt{4}}\left(|001\rangle + 2|010\rangle + |100\rangle\right), \tag{4}$$

This is the quantum state on which the measurement is subsequently performed, which represents the ball's final trajectory and, after repeating the simulation many times, a probability distribution is obtained.
To scale QGB to 4 levels, (corresponding to 9 pegs), we build on the code presented in [1], using a total of 10 qubits: one control qubit q_0 and nine data qubits q_1 to q_9.

4 Scaling the QGB to n levels

Now, the most interesting part is to generalize QGB to n levels, that is, to create a program that has as input the number of levels n and as output the distribution corresponding to n levels.
A peg is basically built with three gates, two *CSWAP* and a *CNOT*, so we can make a quiskit function that implements a single peg with these three gates. However, by observing Figure 2, it can be seen that, thanks to the additional CNOT between the pegs, a pattern is created: a CSWAP followed by a CNOT. We can then exploit this pattern to define a function that partially implements a peg, and build an entire level by repeatedly applying this function. The function that implements a peg partially and the one that implements a complete level are shown below.

```
def apply_peg(circ: QuantumCircuit, coin: int, i: int, j:
      int):
    circ.cswap(coin, i, j)
    circ.cx(j, coin)

def build_level(circ: QuantumCircuit, coin: int, k: int):
    circ.reset(coin)
    circ.h(coin)

    for i in range(mid - k, mid + k):
        apply_peg(circ, coin, i, i + 1)
```

Listing 1: 1) Function that implements a single (partially) "peg". 2) Function that generates a level with k pegs.

As in the previous cases, $2n + 2$ qubits are needed to implement an n-level or n-layer QGB. In the second function, the parameter k refers to the layer number, which is equal to the number of pegs in that layer. The number of layers is a variable that can be modified within the code and determines the total number of qubits.

```
# Number of levels
n = 6
# Total qubits: 1 ancilla + (2*n+1) bins
total_qubits = 2*n + 2
# index of the central bin
mid = n + 1
```

Listing 2: Defining number of levels.

To implement the circuit, a for loop is used to create the levels using the `build_level` function. Then, all the qubits are measured except the control qubit. This could be done more efficiently by measuring only the qubits that are known to be potentially activated depending on the number of levels

```
# 0) Initialize the ball in the center bin
circuit.x(q[mid])
# 1) Create the n levesl
for level in range(1, n + 1):
    build_level(circuit, coin=0, k=level)
    circuit.barrier()
# 2) Measurements
for i in range(1, total_qubits):
    circuit.measure(q[i], c[i-1])
```

Listing 3: Creating the QGB.

The AerSimulator is instantiated using the 'matrix product state' (MPS) method, which employs tensor networks to efficiently simulate quantum states with limited entanglement. This technique optimizes computational resources by representing the system's state in a compressed form, making it especially useful for quantum circuits with low entanglement like this case. The circuit is transpiled for the AerSimulator with optimization level 3, which maximizes gate fusion, cancels redundant gates, and minimizes circuit depth.

```
simulator = AerSimulator(method='matrix_product_state')

circuit_optimized = transpile(circuit, simulator,
    optimization_level=3)
```

Listing 4: Optimized application of the circuit.

With this, we can run the simulation, where we obtain the count of the measurement results.

```
def run_quantum_mps(circ: QuantumCircuit, shots: int =
     10000):
    job = simulator.run(circ, shots=shots)
    result = job.result()
    counts = result.get_counts()
    print("Counting results:", counts)
    _plot_counts(counts, title=f"Distribution of results
         (levels={n})")
```

Listing 5: Simulation.

The simulation for $n = 30$ was performed using Qiskit's AerSimulator, executed on the Google Colab platform with NVIDIA Tesla T4 GPU acceleration. This setup provided an accessible environment with parallel processing capabilities. The total execution time was approximately 30 minutes. The code was executed 10,000 times, the obtained distribution is presented in Fig. 3.
By fitting the histogram to a Gaussian distribution, we obtained a good approximation, with values of $\mu = 9.0$ and $\sigma = 2.7$. To achieve this, we sorted the labels (...00000010000...) in ascending order and assigned them natural numbers (0, 1, 2, 3,...) and then we made the usual adjustment; the resulting curve can be seen in Fig.4.
It is interesting to note that in Fig. 3 there are not n+1 (31) outputs as expected in a GBQ; this is because no "balls" were measured in the states furthest from the center. However, the possibility of measuring those states was non-zero. To reach n+1 outputs, the circuit must be executed several times.
Now, due to the way the pegs are constructed in the circuit, we have an extra CNOT gate at the end of each level. This gate doesn't affect the circuit, since it affects the control qubit, which is reset immediately afterward. We can see this difference by comparing figures 2 and 5.

5 Other distributions

5.1 Exponential distribution

In place of the typical Gaussian distribution that emerges from a normal Galton board, one can progressively bias each level to approximate an exponential distribution over the final positions. To achieve this, it suffices to tune the rotation angle of the "coin" (e.g. via RY gates) so that the bias toward one side increases geometrically or exponentially at each level. In this way, the quantum amplitudes propagating through the board no longer split evenly but instead decay by a constant factor, and the probability of finding the "ball" in a bin at distance k from the center behaves approximately as

$$P(k) \;\propto\; e^{-\lambda\, k}\,,$$

where λ controls the decay rate. By choosing the level-dependent angles, for instance

$$\theta_k \;=\; \theta_0\, e^{-\alpha k}\,,$$

one obtains a final frequency distribution exhibiting an exponential profile.

```
def build_level(circ: QuantumCircuit, coin: int, k: int,
     theta0=pi/4, alpha= 0.1):
    circ.reset(coin)
    # Compute rotation angle that decays exponentially
         with level index k
    theta_k = theta0 * np.exp(-alpha * k)
    # Apply an RY rotation to bias the coin
    circ.ry(theta_k, coin)

    start, end = mid - k, mid + k
    for target in range(start, end):
        apply_peg(circ, coin, target, target + 1)
```

Listing 6: Function that builds one level with an exponentially decaying bias.

The histogram in the figure was obtained using $n = 10$ levels and a parameter $\alpha = 0.1$. Initially, for small k, the coin bias is strong and the resulting probability distribution decays rapidly, resembling a discrete exponential. However, as more levels are added, the number of quantum paths increases, and their amplitudes interfere and superpose, producing an effect analogous to a quantum central limit theorem. The final distribution, while still asymmetric, develops a central peak and smoothed tails, resembling a skewed

Gaussian. To preserve a more clearly exponential profile, one may reduce the number of levels n or increase the bias decay parameter α, so that the bias dominates before interference effects become significant.

5.2 Hadamard quantum walk

In order to obtain a distribution of "Hadamard Walk", we adapted the QGB code for the Gaussian distribution. In the first code, `apply_peg` contains the sequence `circ.cswap(coin, i, j)` followed by `circ.cx(j, coin)`. But for Hadamard walk code, `apply_peg` contains only `circ.cswap(coin, i, j)`. The CNOT was removed because in the original quantum Galton Board, it altered the state of the coin at each peg, breaking part of the coherence and bringing the behavior closer to classical. In the Hadamard walk, the coin must maintain its coherence so that quantum interference accumulates throughout all the steps.

```
def apply_peg(circ: QuantumCircuit, coin: int, i: int, j:
      int):
    circ.cswap(coin, i, j)

def build_level(circ: QuantumCircuit, coin: int, k: int):
    # Hadamard Coin
    circ.h(coin)
    start, end = mid - k, mid + k
    for target in range(start, end):
        apply_peg(circ, coin, target, target + 1)
```

Listing 7: Function "Appy peg" and "build level" for Hadamard quantum walk.

In this code there is no `reset(coin)` inside `build_level`, only `circ.h(coin)` is present. In the original implementation, the coin qubit was explicitly reinitialized to `|0>` at the start of each level and then placed into superposition, but for the Hadamard Walk no per-level reset occurs. The coin's state is therefore preserved across levels rather than being reset by the code before each Hadamard.
Another difference is that this code applies `circuit.h(0)` and `circuit.s(0)` before constructing the levels. This produces a single initialization of the coin qubit to a superposition state with a relative phase (Hadamard followed by phase `S`) prior to all levels.

```
circuit.h(0)
circuit.s(0)
# Ball in the central bin
circuit.x(q[mid])
```

Listing 8: Inicialization of the Hadamard quantum walk circuit.

These modifications not only reproduce the "Hadamard quantum walk" distribution shown in Fig. 7 for $n = 30$ and 100,000 shots, but also substantially reduce the simulation time relative to the original QGB implementation. The simulations were performed with `Qiskit AerSimulator` on a Google Colab instance with NVIDIA Tesla T4 GPU acceleration (the same execution environment used for the QGB runs). On our Colab instance the Hadamard-walk simulation for $n = 30$ and 100,000 shots completed in approximately 30 s, whereas the original QGB circuit required significantly longer runtimes.
The principal reasons for the observed speed-up are the removal of an additional CNOT (which reduces the total gate count per peg), reduced entanglement growth due to preservation of coin coherence, elimination of mid-circuit resets, and improved transpiler optimization (the simpler and more regular circuit structure permits more effective gate fusion and cancellation, yielding lower circuit depth and fewer elementary operations). Collectively, these factors reduce the computational complexity for simulator and account for the substantially shorter execution time of the Hadamard-walk implementation.

References

[1] Mark Carney and Ben Varcoe. "Universal Statistical Simulator". In: *arXiv preprint arXiv:2202.01735* (2022). URL: https://arxiv.org/abs/2202.01735.

[2] Arthur G Rattew et al. "The Efficient Preparation of Normal Distributions in Quantum Registers". In: *Quantum* 5 (2021), p. 609. URL: https://doi.org/10.22331/q-2021-12-23-609.

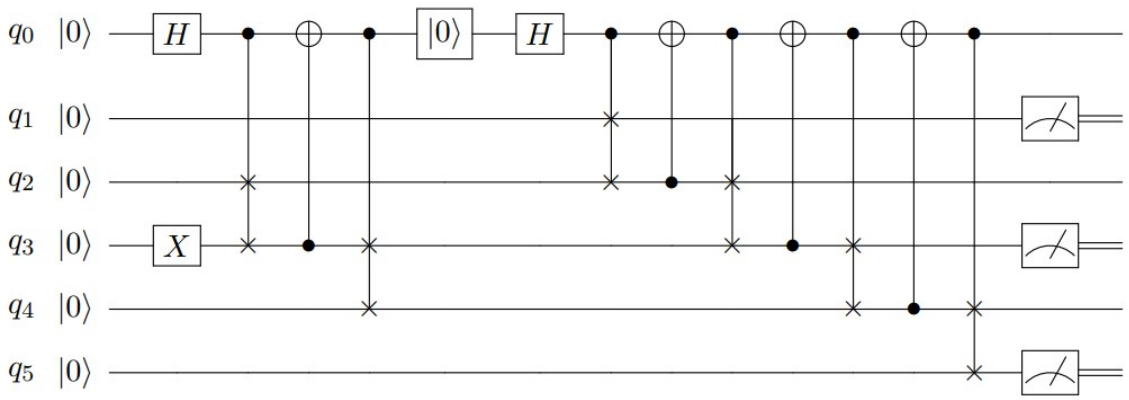

Figure 2: Quantum circuit implementation of a 2-level Quantum Galton Board composed of 3 quantum pegs. The circuit uses one recycled control qubit q_0 to induce superpositions at each level, and five data qubits to route the "quantum ball" initialized in q_3. After passing through three peg modules, the state is measured in q_1, q_3, and q_5, encoding the final position of the ball.

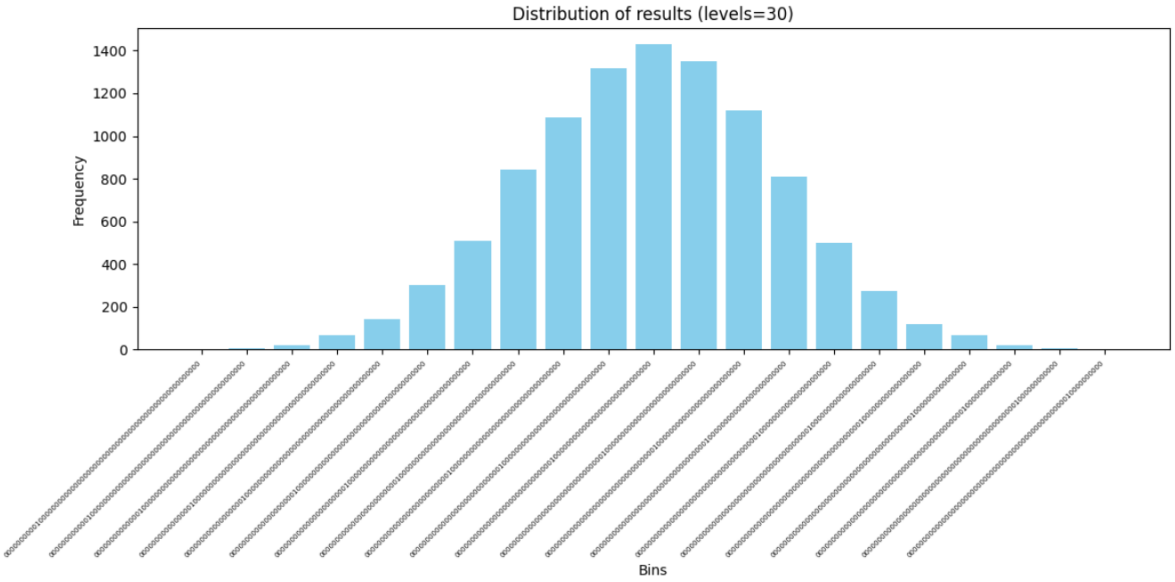

Figure 3: Output results from the simulation of the 30 levels of QGB circuit, with 10,000 shots.

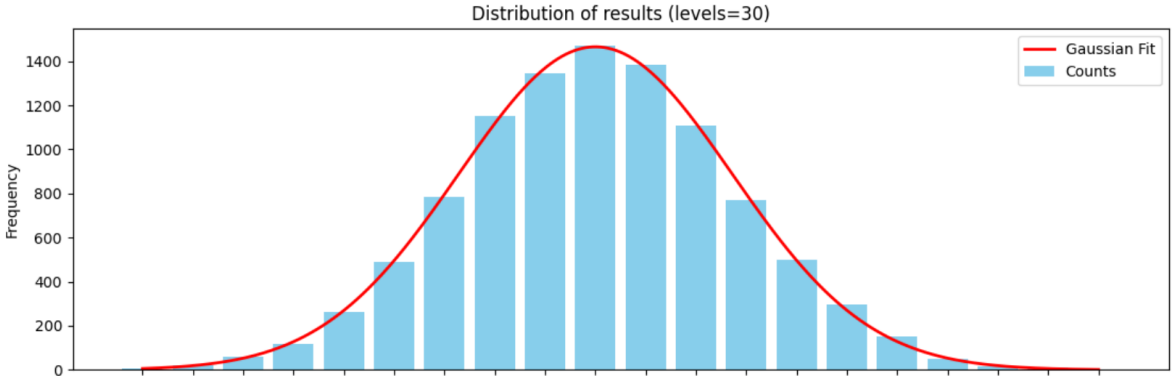

Figure 4: Gaussian distribution fitted to the histogram of Fig. 3 with values of $\mu = 9.0$ = and $\sigma = 2.7$.

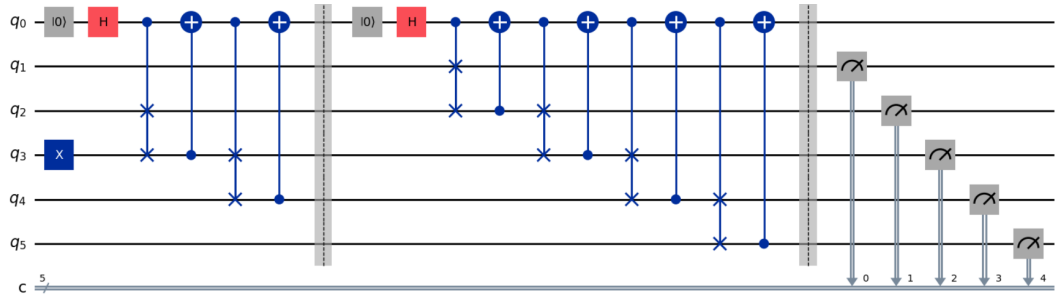

Figure 5: Quantum circuit implementation of a 2-level Quantum Galton Board composed of 3 quantum pegs. It was made with the code presented in the section "Scaling the QGB to n levels" for n = 2. We can see that it uses an extra CNOT at the end of each level.

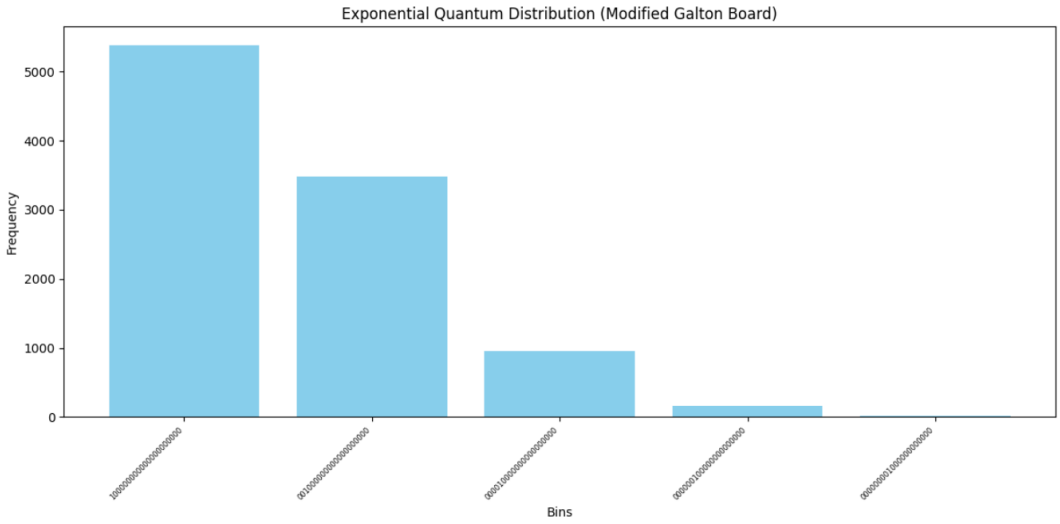

Figure 6: Output results from the simulation of the 10 levels of exponential quantum distribution, with 10,000 shots.

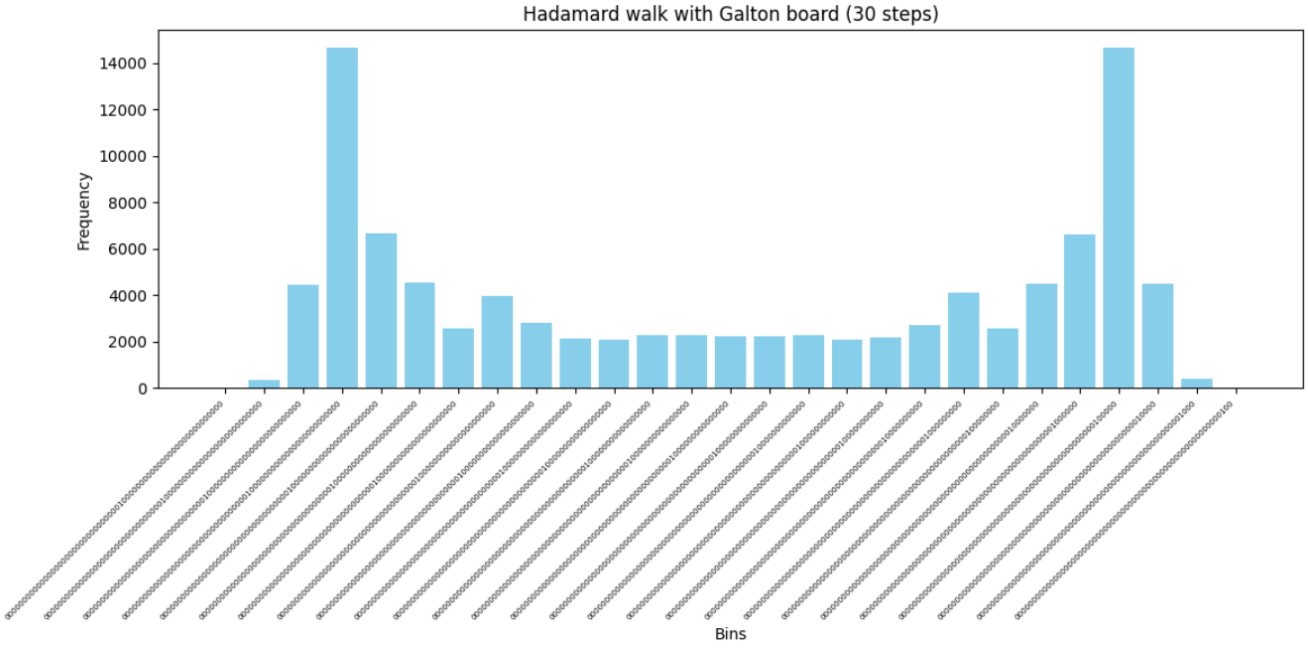

Figure 7: Output from the Hadamard quantum walk simulation after 30 levels with 100,000 shots.