

# Lecture: Randomness

Alonso Peña, PhD, CQF  
Quantum Computing in Finance

In this second lecture we will:

- review the main concepts about **randomness** in finance, including pseudo random number generators (PRNG) and quantum random number generators (QRNG)
- develop a **Python** computer programs to generate random numbers using classical computers and PRNG
- develop Python computer programs based on the **Qiskit** module in order to generate random numbers using quantum computers

## The Problem: Tossing a Coin

In finance **randomness** is everywhere. There are many reasons for this. One is that we have incomplete information about the financial markets and some of the key parameters or variables are unknown or uncertain. Also, we do not know the future and we ought to make assumptions about the future behavior of key financial variables.



So how can we obtain random numbers?

One way to be tossing a coin, but that would be very inefficient and time-consuming.

So how can we get computers to help use generate random numbers?

# A MILLION Random Digits

WITH  
100,000 Normal Deviates

RAND

TABLE OF RANDOM DIGITS															
326															
16250	21005	44897	60205	70944	51207	29121	54411	54853	96629	11803					
16251	92365	29694	85008	15516	26039	31181	58792	82312	21068	88461					
16252	99325	65160	14179	36763	63136	84869	03827	44844	86712	17978					
16253	16390	34573	34198	84111	20018	50973	70786	01243	37148	89163					
16254	05078	09218	42238	18782	06309	69699	53319	57222	07878	84698					
16255	88891	34003	43513	41884	57626	71579	07896	90066	81407	23695					
16256	96458	73295	65733	61214	35258	13896	06004	37780	66697	34917					
16257	72551	85022	04036	21526	94234	09358	81626	36111	82083	43239					
16258	12752	79041	90480	73199	14145	93051	95900	63992	56576	61593					
16259	72959	07091	45353	12836	57641	60873	95310	23936	85882	96457					
16260	73774	65729	82896	19854	96333	37983	48628	69844	07759	00591					
16261	43451	92113	65982	83462	86340	27236	82005	73949	46753	54308					
16262	72503	62561	14957	05129	79071	98382	24214	64819	01178	99575					
16263	80472	62755	45275	78268	46345	49001	09747	83086	07493	02683					
16264	08908	05557	78757	41215	20316	05965	11479	81722	86102	77796					
16265	63209	37288	97150	23163	28200	07613	67140	28649	62638	67121					
16266	00289	27466	40897	61417	31778	58502	42847	19044	91723	61177					
16267	19154	39424	91896	24210	79695	66463	16869	92545	88722	84558					
16268	98500	21421	46648	11808	23838	80927	16240	61934	92221	23858					
16269	33651	21850	46609	55791	71243	70743	40329	60828	48377	48140					
16270	26780	30455	25240	63403	49267	85837	55504	40128	40839	18167					
16271	28304	80697	64668	09834	61436	73836	90767	28310	77249	25962					
16272	56581	38265	36834	74252	43305	75124	31235	29246	55396	24939					
16273	30860	72439	30959	08662	18995	55329	19518	56338	61125	08922					
16274	57534	50828	93112	52539	29604	32138	32342	41018	33479	57380					
16275	35050	45309	00989	74035	29708	62583	80670	09202	37327	78695					
16276	40677	95993	89814	86065	05762	05655	22490	97615	11552	74720					
16277	76013	97577	73953	67550	76466	40985	88864	07328	94895	72288					
16278	07758	85786	01304	15105	84381	92964	52258	39910	33751	95567					
16279	96005	29274	93887	76858	36866	02982	84187	14581	82584	71295					
16280	97554	85603	72636	06046	34270	63868	53137	83946	20967	39459					
16281	95381	74175	92432	81274	62306	45036	90285	95089	55652	77214					
16282	07333	52306	75748	84592	16388	15891	06135	50773	70338	19266					
16283	85364	53411	96981	70087	58169	56535	56438	37553	08171	46782					
16284	56906	84239	09345	56042	75713	09699	63433	41653	26535	76536					
16285	62770	19023	18312	29427	00317	61935	68232	58890	01909	26180					
16286	30108	19041	96933	66717	27681	50286	34404	08981	75529	31385					
16287	68613	00698	07398	31913	19653	61394	48542	84657	21032	85319					
16288	36027	73569	65088	96563	65855	96119	41806	57468	39843	18332					
16289	95617	79992	82965	91313	34761	81679	43965	96057	38143	19025					
16290	79953	10532	24823	32959	26838	30590	45430	45192	74952	83719					
16291	52565	74291	60455	41555	41390	03981	77129	10727	45449	53359					
16292	36672	58413	79448	11687	32351	84045	06175	15088	88311	54131					
16293	71814	80667	74371	51630	81877	72098	84832	89965	10538	72710					
16294	17243	59932	49156	95685	54369	45992	03668	60449	16171	20528					
16295	24504	83085	91755	78783	36356	47517	75347	68855	38083	37142					
16296	65418	60502	89344	89471	72675	79957	46598	82607	19721	96042					
16297	71056	52544	20353	83280	85747	64373	61980	55358	69913	59647					
16298	33583	14957	96240	36172	65099	49411	43860	63841	27354	15591					
16299	94812	45025	30161	20247	43424	07643	45788	44162	20893	46612					

# **The Classical Solution**

PRNG is a highly-sophisticated field (Gentle 2013; Glasserman 2013). There are different types of random numbers. These can be classified depending on where they come from. The most common are pseudo-random number generators (PRNG), which generate numbers that appear random, but are actually deterministic, as they are produced following some complex algorithm. They require the current state of the system to start their sequence, which is called the seed. A popular PRNG algorithm is the Mersenne Twister algorithm developed by Makoto Matsumoto and Takuji Nishimura in 1997. Another types of random number generators include quantum random number generators (QRNG), which we will discuss later.

# Python Lab



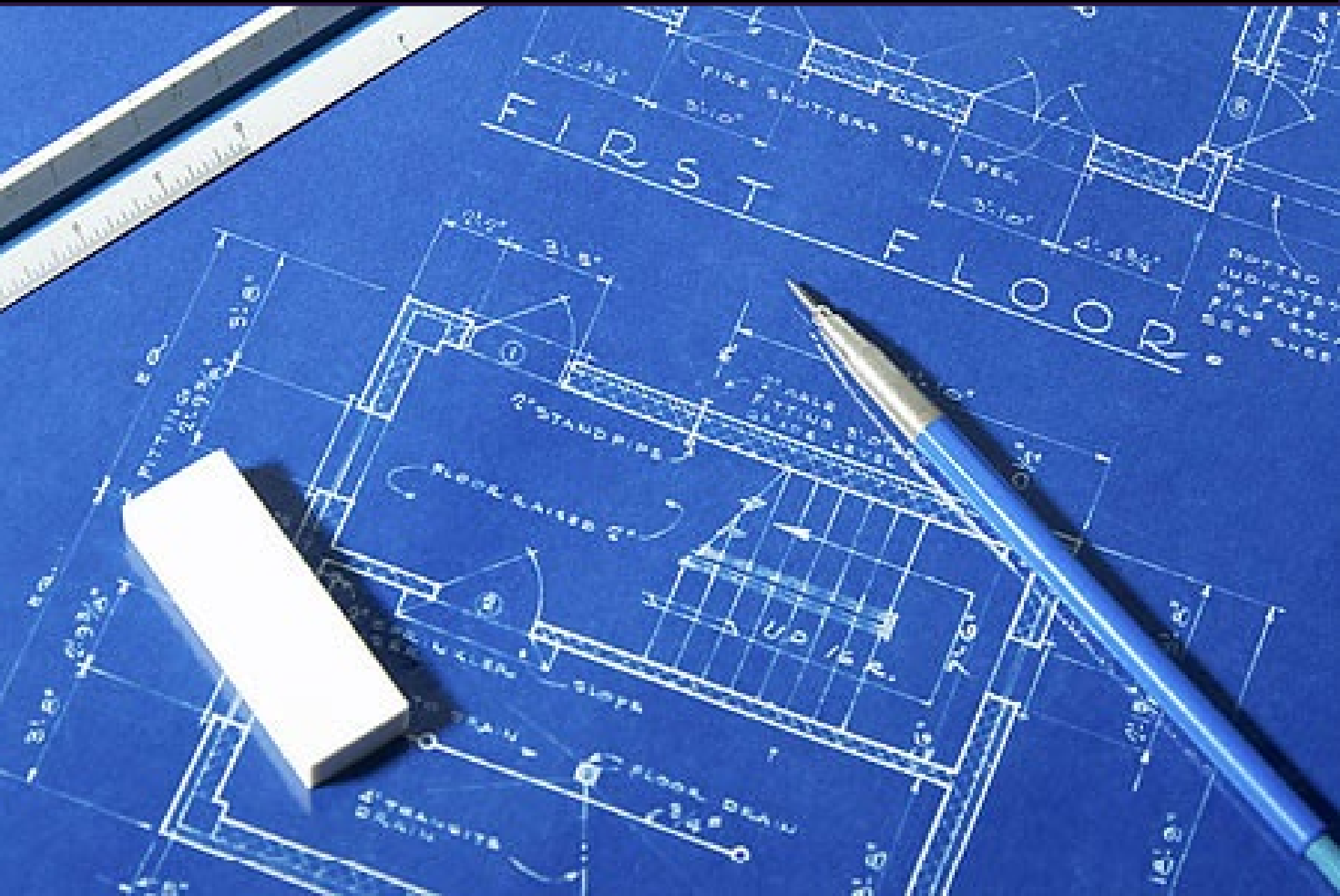
**Python** offers a number of ready-made solutions for random number generation. The module `random` is in fact designed specifically to offer high quality pseudo random numbers from different statistical distributions. In the following demonstrations we will use this module to generate random numbers in a classical computer.



For more details see:

**<https://docs.python.org/3/library/random.html>**

# LABORATORY 1



## **LABORATORY 1: Classical INTEGER Random Number Generation**

In this laboratory we present two codes to generate integer random numbers. We make use of the method `randint()` to be found in module `random` from Python.

```
random.randint()
```

## **Code 2.1 Integer Random Numbers: Single Sample**

In the following code, we present the use of method `randint()` from module `random` to generate integer random numbers. In the example below, we generate integers between 1 and 128. Note that we have used the method `seed()` to set the seed of the pseudorandom number generator.

```
#  
CODE_2_1_PRNG_INTEGER  
# single integer sample  
# between 1 and 128  
# import the random module  
import random as rn  
rn.seed(123) # set seed of  
# random sequence to 123  
# using the randint()  
# method obtain five samples  
print(rn.randint(1,128))  
print(rn.randint(1,128))  
print(rn.randint(1,128))  
print(rn.randint(1,128))  
print(rn.randint(1,128))
```

Running the code above in Jupyter Lab produces the following results:

14  
69  
23  
105  
69

Output of code 2.1: five integers between 1 and 128.

## **Code 2.2 Integer Random Numbers: Multiple Samples**

An elementary extension of the previous code, would generate multiple samples of the integer random numbers at the same time, i.e. as a vector of random numbers. In the example below we generate 1000 integer random numbers between 1 and 128 using multiple times the method `randint()`. As for all PRNG, we have set the seed of the random sequence to an specific value, in this case 123.

```
# CODE_2_2_PRNG_INTEGER_MUTIPLE
# vector integer samples
# import the required libraries
import random as rn
import matplotlib.pyplot as plt
import numpy as np

rn.seed(123) # set seed of random sequence to 123
num_samples = 1000
out_integer = np.zeros(num_samples)

# generate vector of random numbers
for k in range(num_samples):
    out_integer[k]=rn.randint(1,128)

# output individual samples (first 100)
print(out_integer[0:100])

# generate PLOT
plt.plot(out_integer)
plt.show()

# generate HISTOGRAM
plt.hist(out_integer, bins = 10)
plt.show()
```

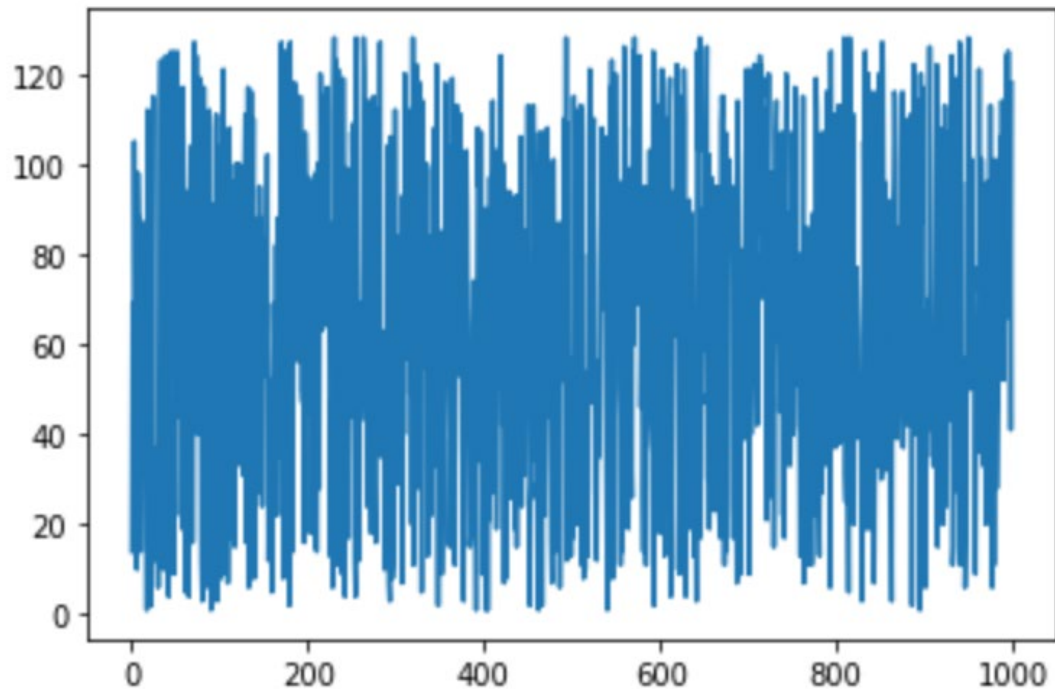
Running the code above in Jupyter Lab generates the results illustrated below. Here we see the first 100 integer random numbers from the set generated.

```
[ 14.  69.  23. 105.  69.  28.  10.  98.  86.  88.  14.  41.  35.  87.
 86.  63.  42.   1. 112.  23.  97.  18.   2.  81. 115.  27.  12.  24.
 37.  33.   6.  75. 111. 123.  68. 121.  10.  79.  88. 124.  53.  81.
  4. 102. 112. 125. 108.  96.   9.  47.  22. 125.  68.  44.  86.  87.
101.  19. 117. 100.  87.   5.  49.  23.  94.   4.  91.  59. 104.  16.
 74. 127.  60. 124.  46.  40. 114.  97. 119.  12.   3.  35. 117.  50.
  6.  66.  91.  70. 112. 108.   1.  28.  17.  91.  23.  22. 111.   3.
 97.  19.]
```

**OUTPUT code 2.2: First 100 integers between 1 and 128 of the 1,000 integers generated by the code.**

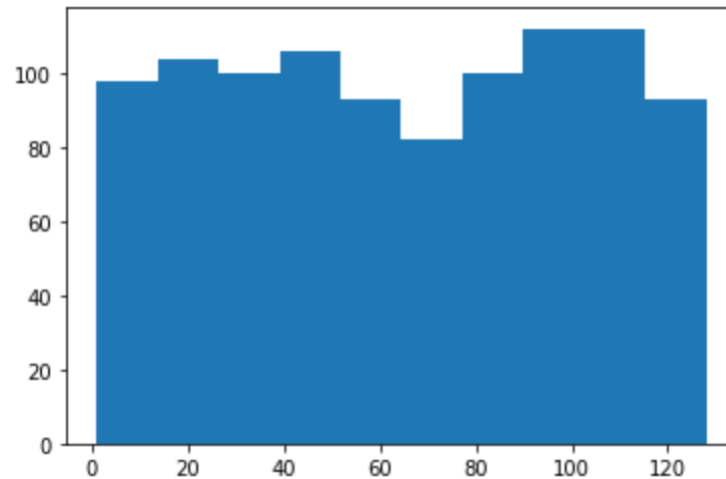


The numerical results can be also visualized in terms of a plot graph. In the next plot we see the 1000 simulated integers as a line graph.



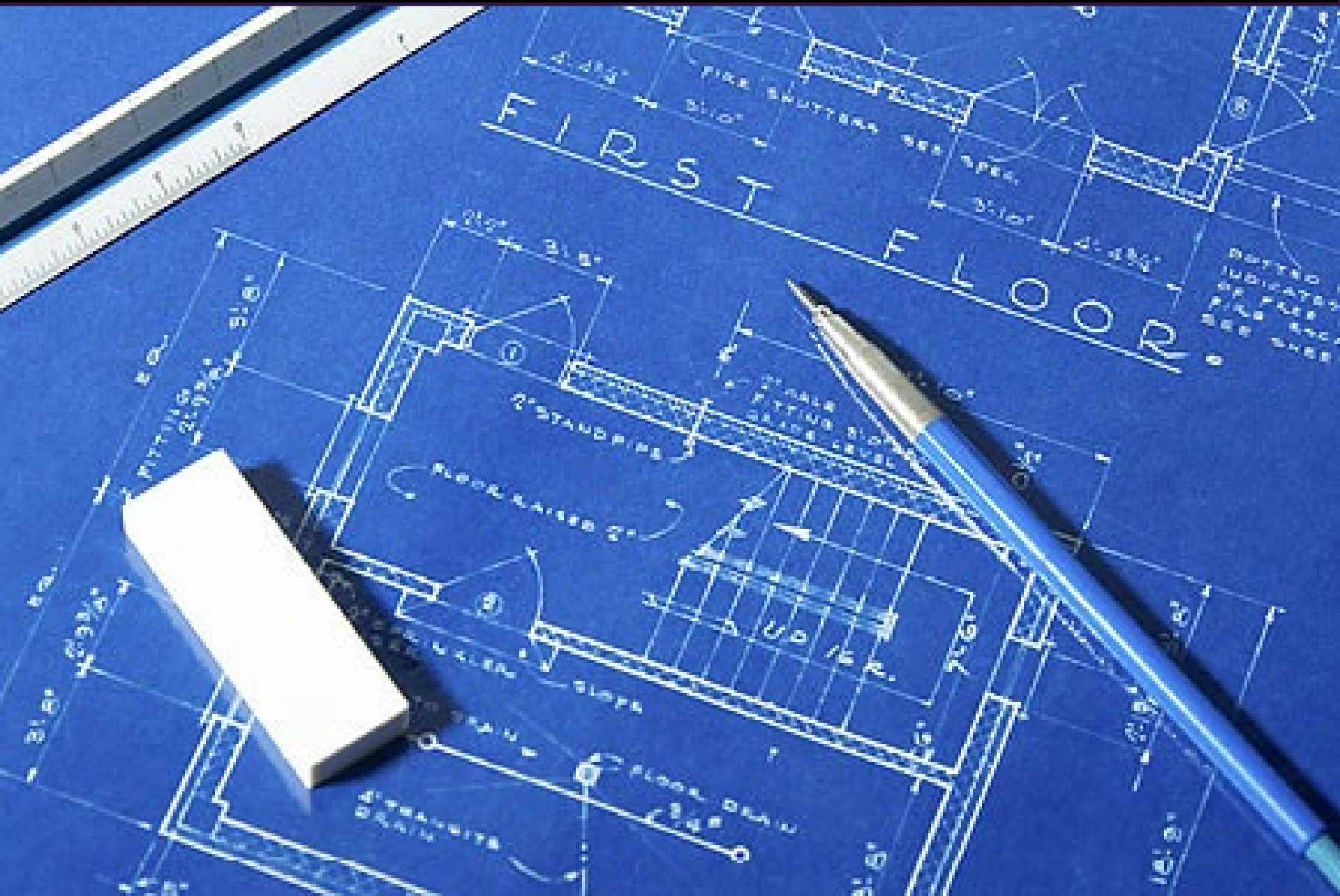
**OUTPUT code 2.2: Plot of the previous 1000 integers.**

We can finally summarize the 1000 generated data points above in terms of a histogram. We confirm that we can observe as expected the bins to be approximately similar in height indicating a flat or uniform distribution between 1 and 128.



**OUTPUT code 2.2: Histogram of the 1000 integers.**

# LABORATORY 2



## LABORATORY 2: Classical UNIFORM Random Number Generation

In many applications random numbers from a uniform distribution are particularly useful. In the following two codes we illustrate how these can be obtained using the method `random()` of the module `random` in Python. As in LABORATORY 1, we obtain first five individual observations, as single samples, and then a set of 1,000 observations.

```
random.random()
```

## **Code 2.3 Uniform Random Numbers: Single Sample**

The necessary code is very similar to Code 2.1, but we substitute the method `randint()` and method `random()`. The full code with this substitution is given as Code 2.3. The result of running this code in Jupyter Lab, gives the following numerical results.

```
0.41661987254534116  
0.010169169457068361  
0.8252065092537432  
0.2986398551995928  
0.3684116894884757
```

**OUTPUT Code 2.3: Five samples from the  
uniform distribution**

## **Code 2.4 Uniform Random Numbers: Multiple Samples**

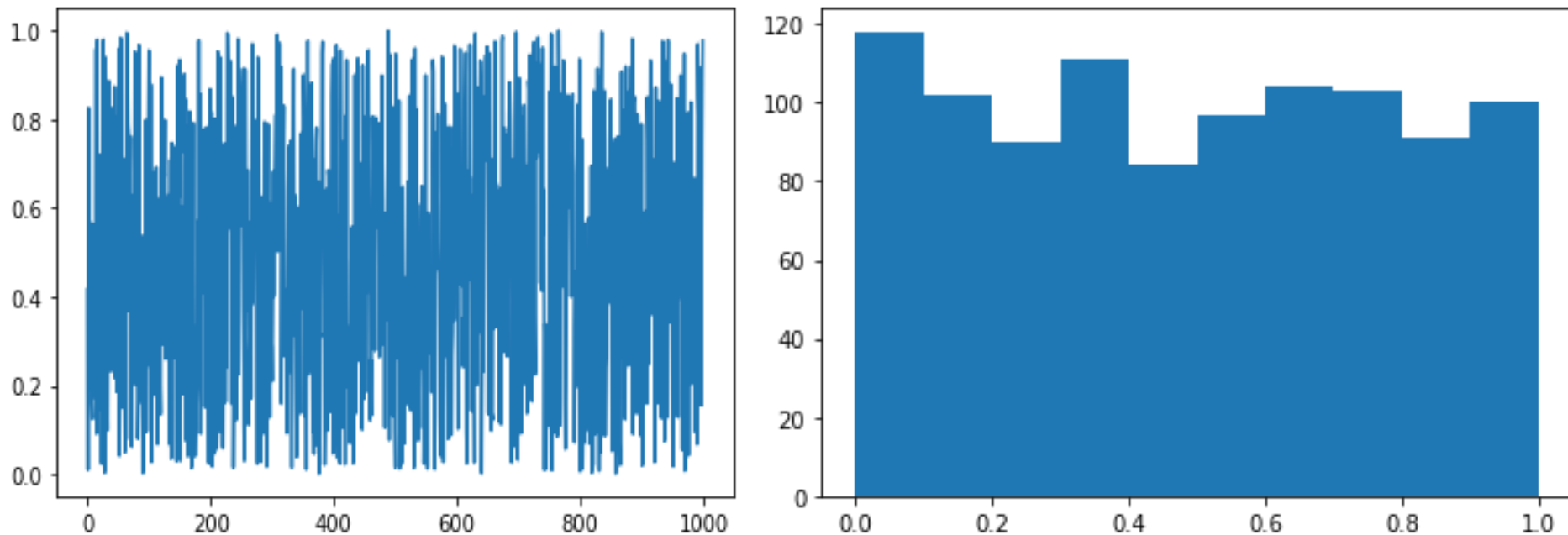
The extension of the code for a single sample to multiple samples is easy. One needs only to follow code 2.2 and substitute the method `randint()` and method `random()`. By running the code we obtain the following numerical results:

```
[0.41661987 0.01016917 0.82520651 0.29863986 0.36841169 0.19366135
0.56600817 0.16168782 0.12426688 0.43293627 0.56207849 0.17434356
0.55322109 0.35490139 0.95806479 0.0912941 0.97864 0.41211939
0.50393537 0.14814617 0.71896714 0.18997138 0.34156043 0.02352122
0.33951777 0.96748246 0.97879845 0.74453004 0.00345461 0.9402385
0.87076697 0.77083434 0.17887376 0.09949963 0.41453247 0.88553647
0.57808603 0.73658221 0.2326211 0.52359758 0.70938644 0.82483395
0.80712453 0.23230811 0.87318997 0.21638043 0.80189924 0.5550853
0.18582836 0.58860862 0.51823939 0.9586625 0.04153922 0.16413828
0.98329265 0.83220489 0.15027247 0.22911297 0.53913827 0.15698083
0.32376073 0.04931922 0.71167729 0.07856275 0.99460425 0.92056984
0.78882887 0.77240589 0.3677523 0.67870617 0.76125877 0.48077625
0.06152505 0.62610395 0.32576083 0.61008907 0.42082846 0.95253078
```

**OUTPUT Code 2.4: First 100 uniform samples of the 1,000 samples generated by the code.**



As well as the following visual representation of the 1,000 random samples from the uniform distribution, and its associated histogram.



**OUTPUT Code 2.4: Plot of the previous 1000 uniform samples (left) and histogram (right).**

# **The Quantum Solution**

## **How can we generate random numbers using quantum computers?**

The most direct way is to take advantage of the intrinsic nature of qubits.

Remember, in contrast to bits who can only take two states (i.e. 0 or 1), qubits can have a multiplicity of states.

The state of the qubit can be represented as a point in the Bloch Sphere.

So we can setup qubits to be in a state precisely in between  $|0\rangle$  and  $|1\rangle$  using a Hadamard gate (the H-Gate) we saw in Lecture 1). In other words, we put the qubit in a superposition of the two states  $|0\rangle$  and  $|1\rangle$  .

And afterwards by measuring the actual location of the qubit, we force it to collapse to either the South Pole (0) or the North Pole (1).

Because the likelihood of finding them in either is equal, we can regard this as Tossing a Coin, and would find the qubit sometimes in state 0, and sometimes in state 1.

The probability of each is 50%. No need for PRNG or seeds, just the observed effects of quantum mechanics!



**How do we do this in practice?** We are going to start by generating random numbers from a single qubit that could be either pointing down (0) or pointing up (1). We are then going to take this as our building block and construct from this single measurement of a qubit a single bit of information. We will do this first in the IBM Quantum Experience using the visual tools from the website and then we will do it by write a computer program in Python with the help of the Qiskit module. Our goal is to generate various types of random numbers using quantum computing as follows:

## **Quantum Binary Random Numbers**

Here is we directly observe a single quabit and after measuring it we transform it into a bit. We can repeat this process multiple times, say 8, and from the 8 measurements obtain 8 bits that would be useful to form a byte.

## Quantum Integer Random Numbers

With the methodology above to generate bits and bytes from qubits, we can then transform the binary representations obtained into integers. For example, applying a H-Gate to the same qubit 8 times produces 8 states of information, say 0 1 0 0 1 1 0 1 that if we join them can be regarded as a byte (or 8-bit binary number) as 01001101 and transformed into the decimal integer 77.

## Quantum Uniform Random Numbers

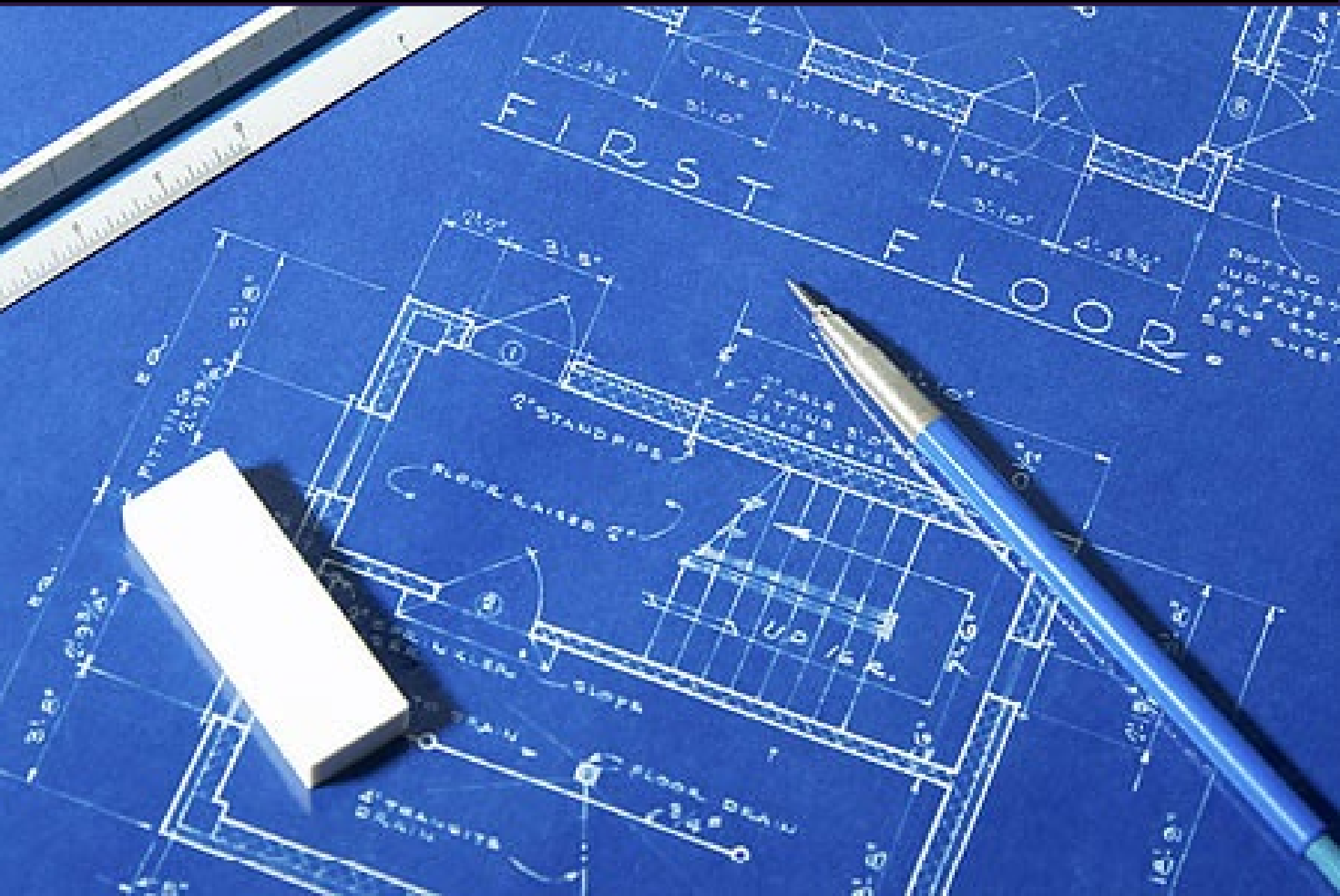
Then with a large enough set of integer numbers we can cover the domain  $[0,1]$  and interpret these as samples from a uniform distribution. Of course too few would mean very few samples of the domain, like in the case of 8-bits representing 128 equidistant points between 0 and 1. Using 32 bits then the number increases significantly to  $2^{32} = 4,294,967,296$ .

**Qiskit Lab**



We start with a single qubit. This might seem too little, but as we will see, it will be the building block upon which we can construct much larger techniques to generate random numbers the quantum way.

# LABORATORY 4

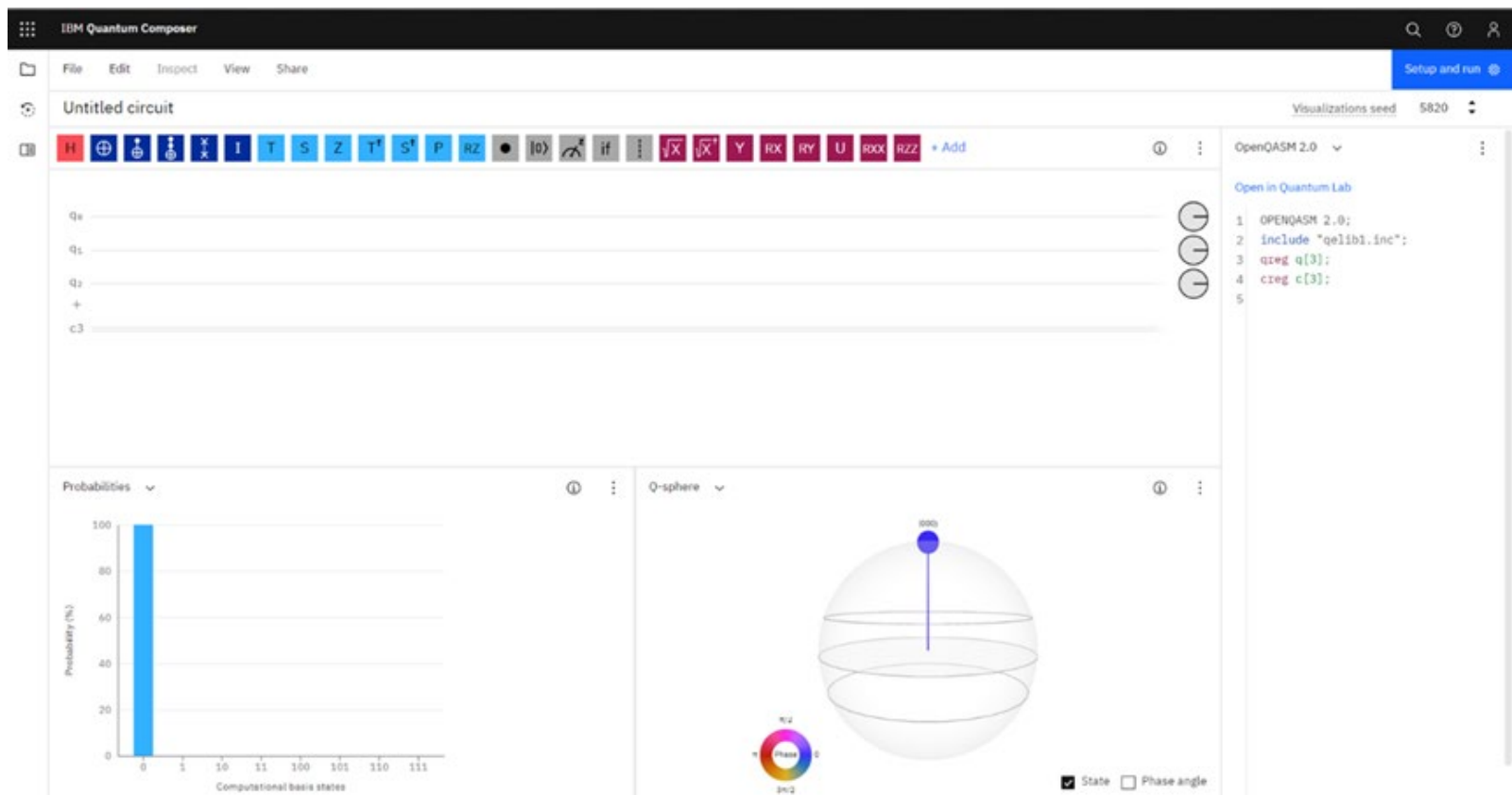


## **LABORATORY 4: IBM Quantum**

In this lab, we will explore how we can generate quantum random numbers using the internet. We will do it first using the QASM simulator (which simulates the behaviour of an ideal quantum computer) and subsequently we will do it on an actual quantum computer.

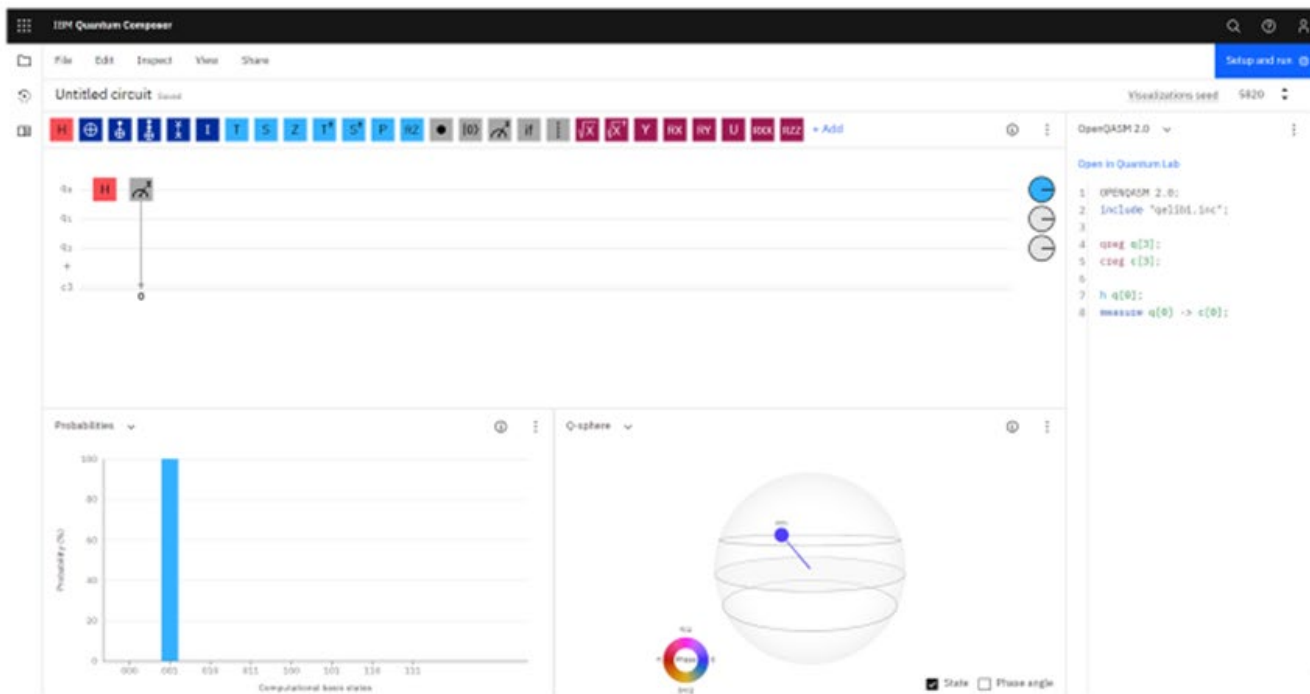
### **1-bit QRNG Using QASM Simulator**

We start by going to the website of the IBM Quantum, do the login with our credentials and then go to the Circuit Composer as illustrated below .



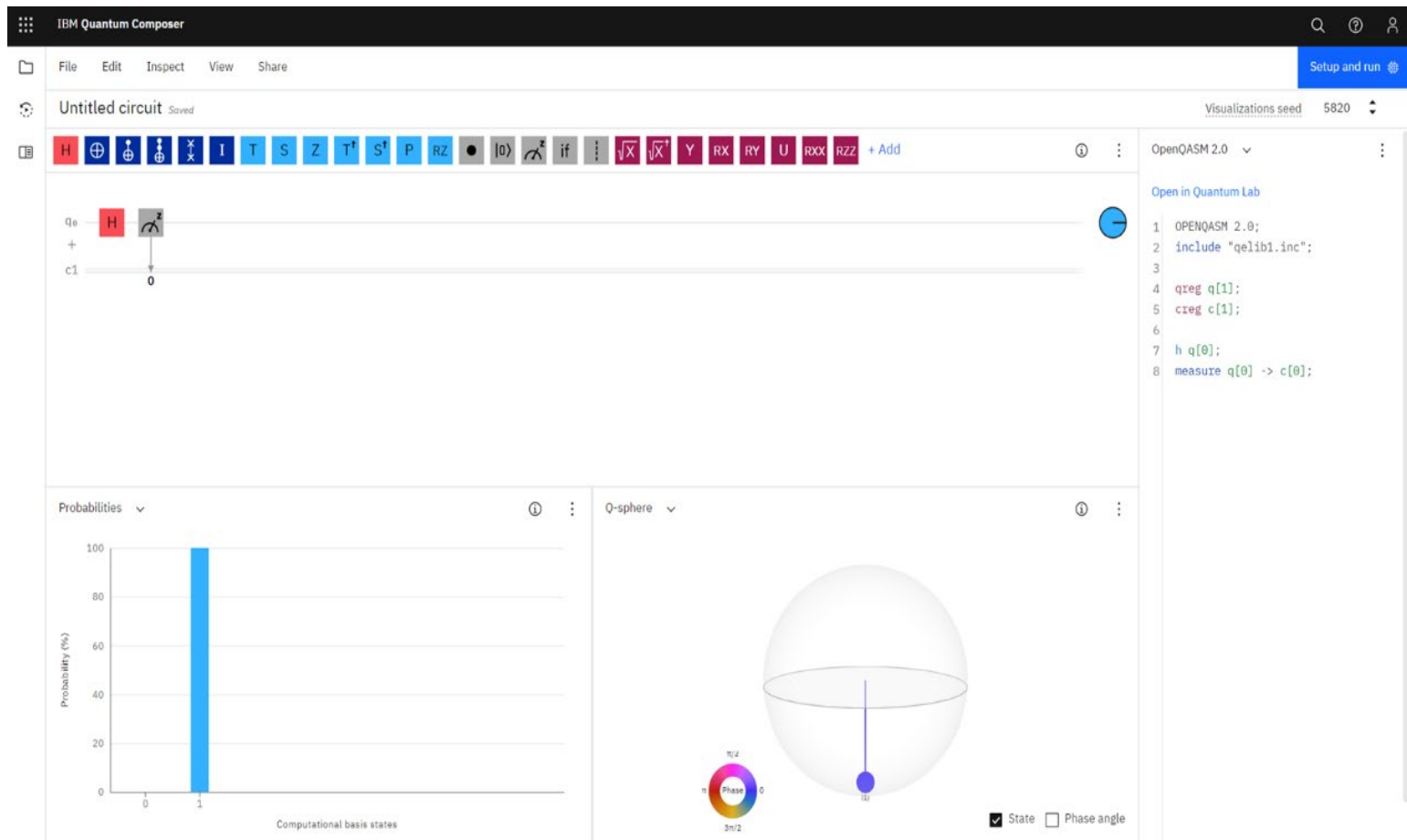
**STEP 1: Go to the IBM Quantum website Circuit Composer**

We then construct a very simple quantum circuit composed of 1 qubit and 1 classical bit. We drag and drop the Hadamard Gate (the H-Gate) to the first qubit in the line q0. We then drag and drop a measurement icon to the qubit after the application of the H-Gate, i.e. immediately to the right.



**STEP 2: Create a circuit with 1 qubit, apply a Hadamard Gate and a Measurement**

As before, we eliminate the unnecessary qubits q1 and q2 by simply hovering over them and clicking on their icons, resulting in

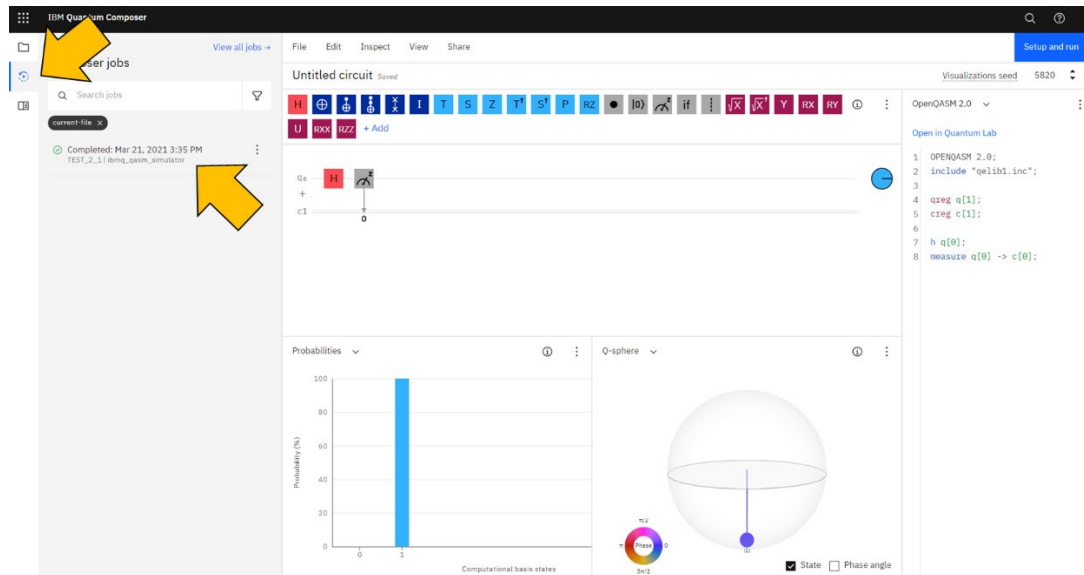


In the next step we go to the Job rung Setup and Run blue button (top right) settings and make sure that we have chosen the `ibmq_qasm_simulator` (which is the virtual or simulated ideal quantum computer) and that we have written 1,000 1000 shots (that is, the number of repetitions that the circuit is going to be run) and call this test `TEST_2_1`. We then click on the blue box box on the top bottom right corner (it's a button) and click `Run ibmq_qasm_simulator`. In the Jobs section, this request appears now as `QUEUED`.

The screenshot displays the IBM Quantum Composer interface. On the left, a quantum circuit is visible with two qubits, `q0` and `c1`. `q0` has an `H` gate, and `c1` has a `Z` gate. Below the circuit, a bar chart shows the probability distribution for computational basis states, with a single bar at state `1` reaching 100% probability. On the right, a panel titled "Set up and run your circuit" is open. It contains two steps: "Step 1: Choose a system or simulator" and "Step 2: Choose your settings". In Step 1, the `ibmq_qasm_simulator` is selected. In Step 2, the provider is set to `ibmq-q/open/main`, the number of shots is set to `1000`, and the job name is `TEST_2_1`. A blue button at the bottom right of the panel is labeled `Run on ibmq_qasm_simulator`. Yellow arrows highlight the selection of the simulator, the job name, and the `Run` button.

**STEP 3: Run the circuit by clicking in bottom right blue box. Note that the job is now queued.**

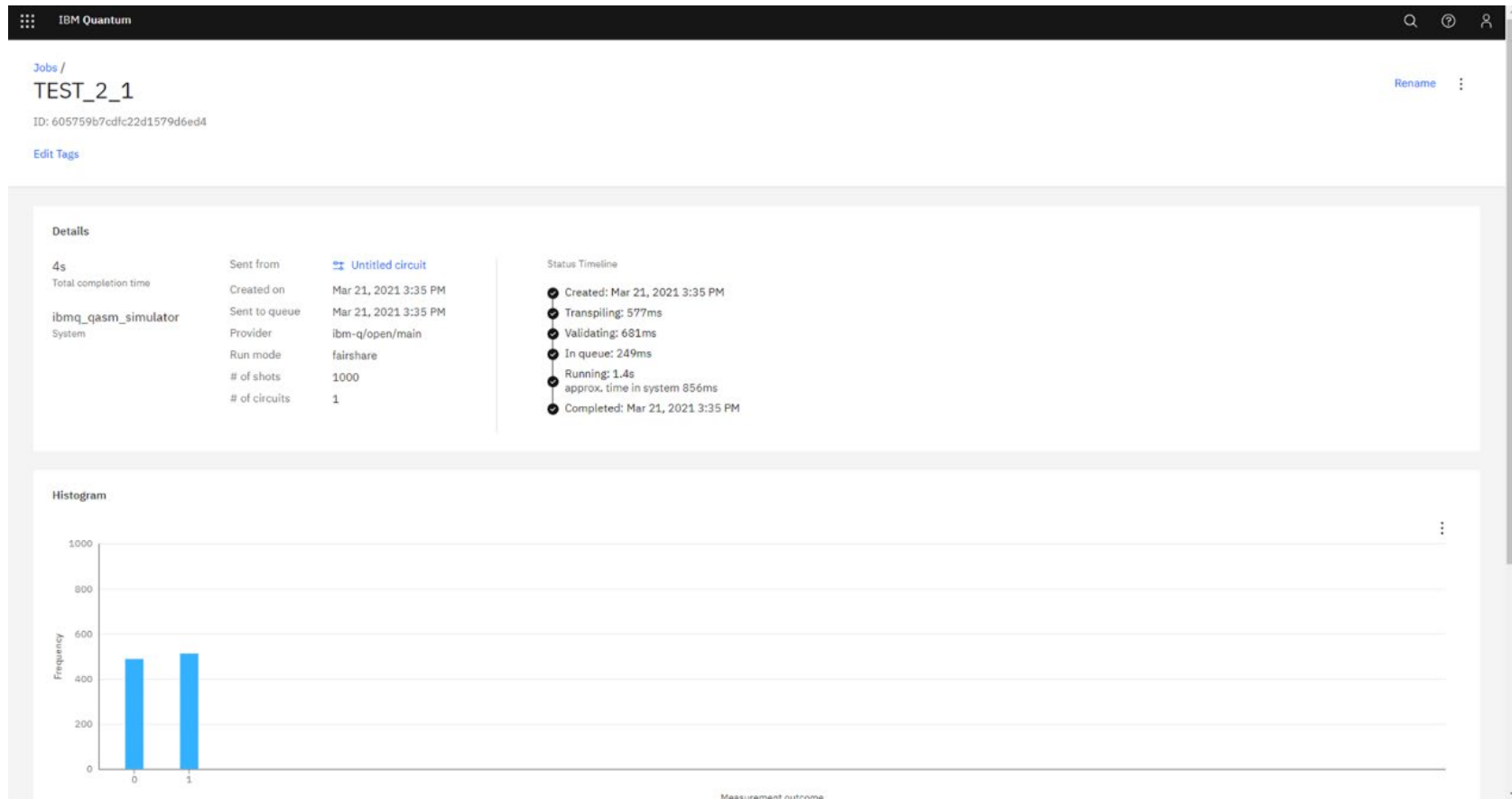
After a few seconds, but this may depend on how busy the system is, the job is finished. Now we can see the progress on the job by clicking on the icon Composer Jobs (left margin) where in the list the job says COMPLETED.



**STEP 4: In Composer Jobs the job appears as completed**



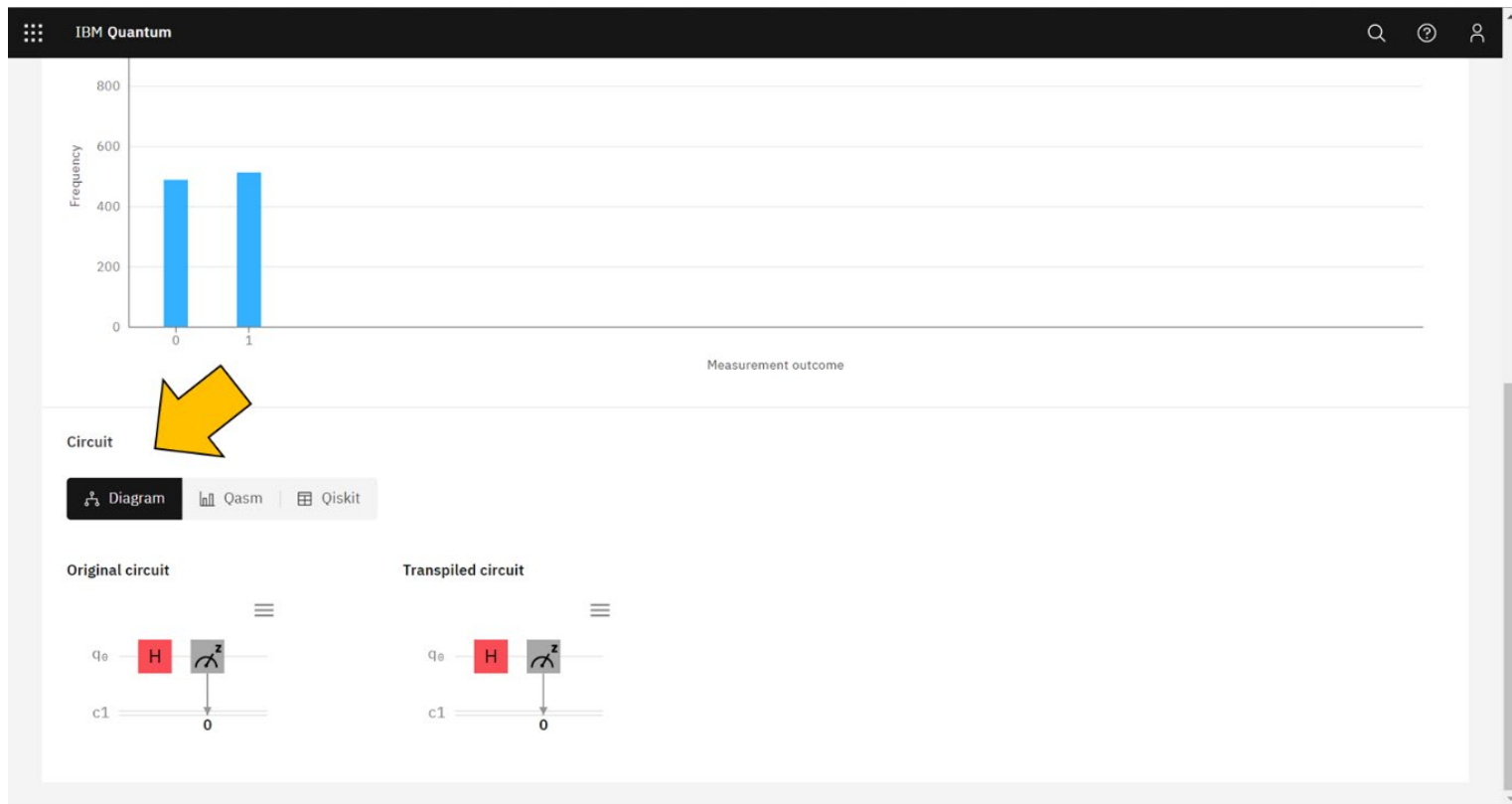
Clicking on the Job itself and the blue button **See More Details**, will take us to the results of the simulation in a new tab in the browser. Here we can see how long our circuit took to run, in this case 1.4 seconds, and as well as the number of shots (simulations) used and the time stamp of the end of the request.



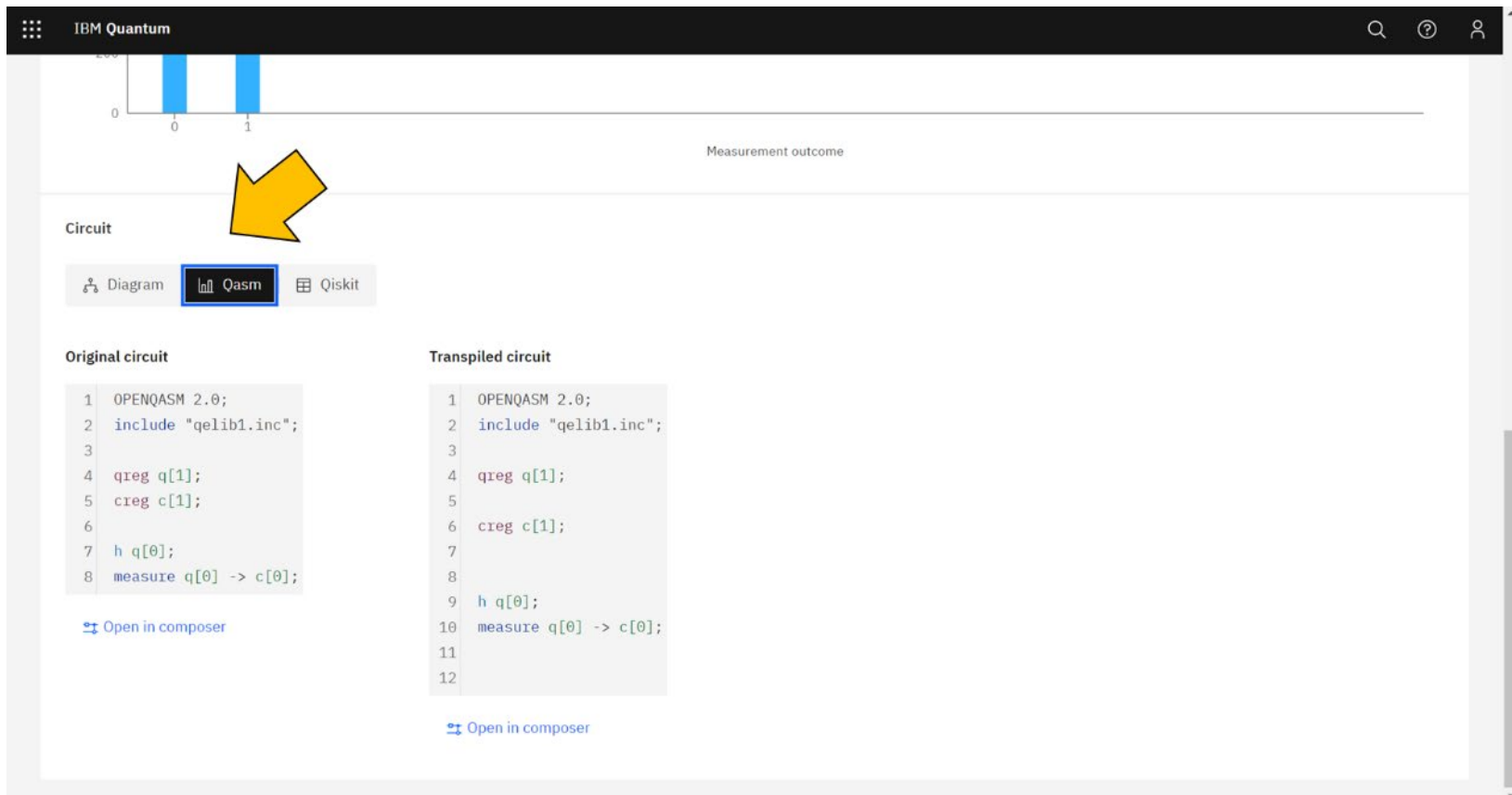
**STEP 5: Job details in a new tab.**

Scrolling down the page takes us to the histogram, that summarizes how many times we obtained a 0 and how many times we obtained a 1. As we can see, 0 appeared 48.8% of the shots used, i.e. 488 zeroes and 512 ones.

Scrolling further presents a block named Circuit, which contains three tabs: Diagram, Qasm and Qiskit. On Diagram we find a visual representation of our circuit



In Qasm we have the transcription of the circuit into the low-level language OPENQASM 2.0.



The screenshot displays the IBM Quantum Qiskit web interface. At the top, a black header bar contains the 'IBM Quantum' logo on the left and search, help, and user icons on the right. Below the header, a 'Measurement outcome' plot shows two blue bars at positions 0 and 1 on the x-axis. A large yellow arrow points from this plot down to the 'Circuit' section. The 'Circuit' section has three tabs: 'Diagram', 'Qasm' (which is selected and highlighted with a blue border), and 'Qiskit'. Below the tabs, there are two panels: 'Original circuit' and 'Transpiled circuit'. Both panels display the same OPENQASM 2.0 code. The 'Original circuit' panel includes a link 'Open in composer' at the bottom. The 'Transpiled circuit' panel also includes a link 'Open in composer' at the bottom.

IBM Quantum

Measurement outcome

Circuit

Diagram Qasm Qiskit

Original circuit

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3
4 qreg q[1];
5 creg c[1];
6
7 h q[0];
8 measure q[0] -> c[0];
```

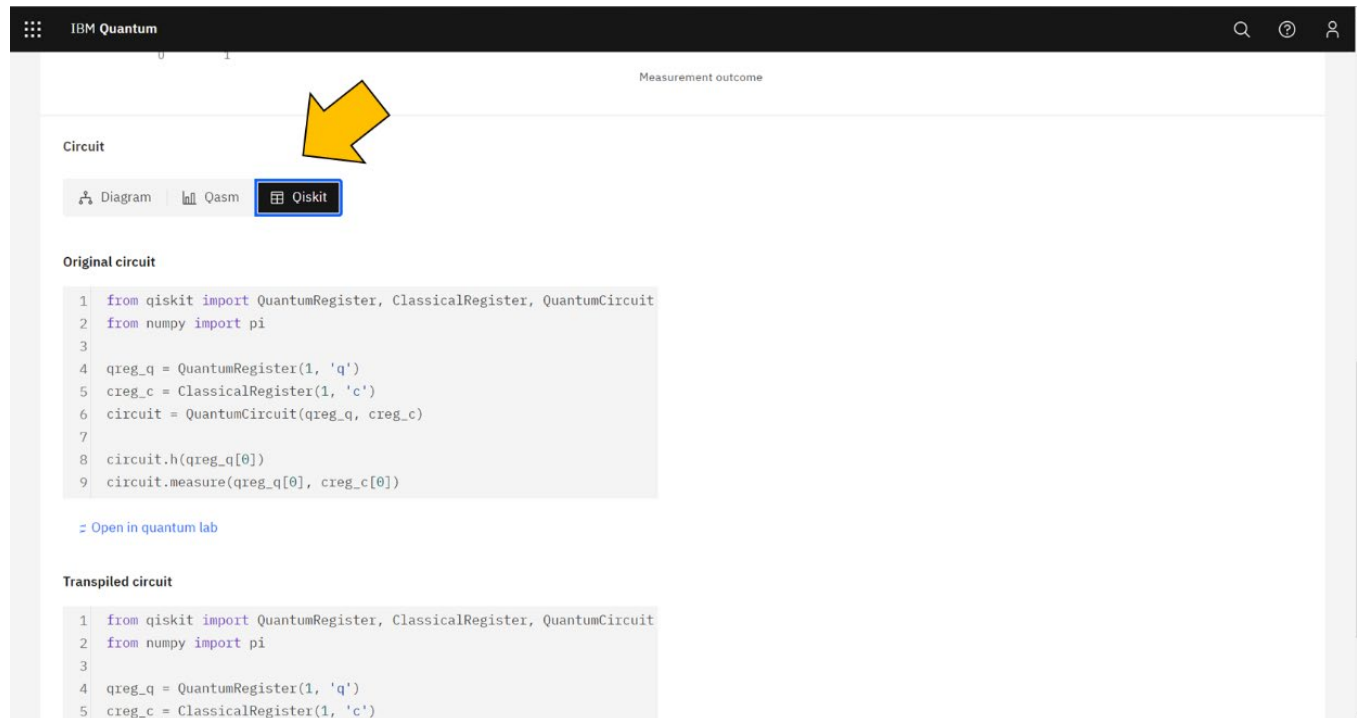
[Open in composer](#)

Transpiled circuit

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3
4 qreg q[1];
5
6 creg c[1];
7
8
9 h q[0];
10 measure q[0] -> c[0];
11
12
```

[Open in composer](#)

Finally, in Qiskit we find a listing of the same circuit into Python code, as seen below, including two modules: NumPy and Qiskit. Clicking on the blue button on the bottom right will open this code in Jupyter Lab.

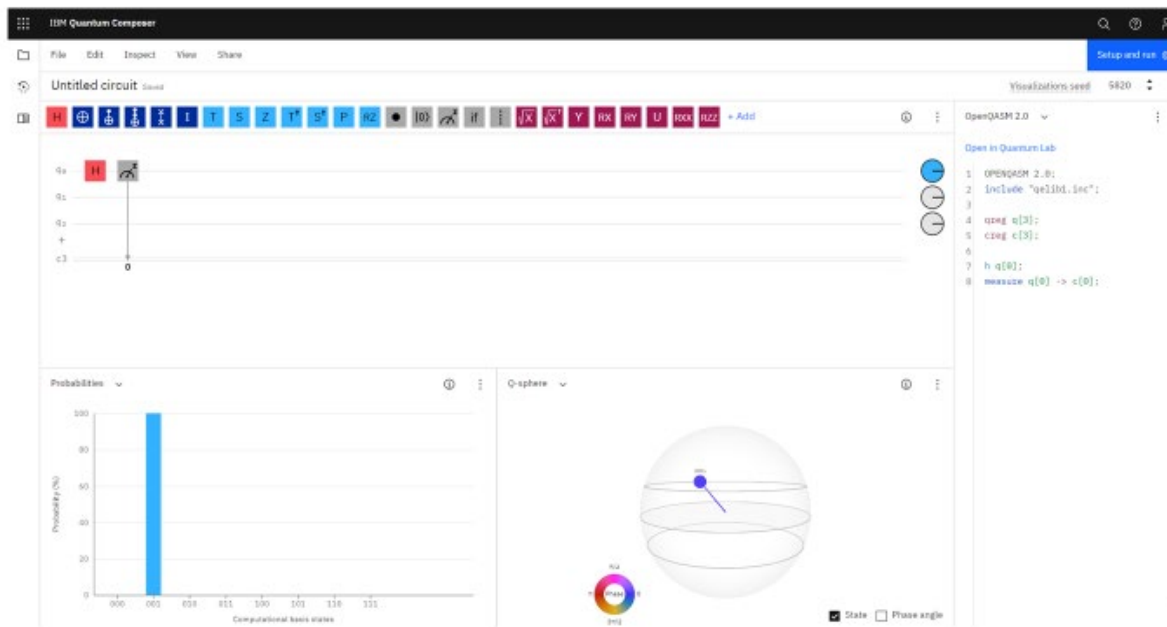


**STEP 8 : Clicking on QISKIT will show the Python code of the circuit.**

Thus we have generated a sequence of 1000 random numbers, random zeroes and ones, using the simulator. In the next exercise, we will do the same in a real quantum computer.

# 1-bit QRNG Using Lima 5 qubit in a Real Quantum Computer

We start as before by going to the page of IBM Quantum Experience. We construct our circuit with one single qubit, one Hadamard Gate and a measurement device after it.



**STEP 1: Circuit construction**

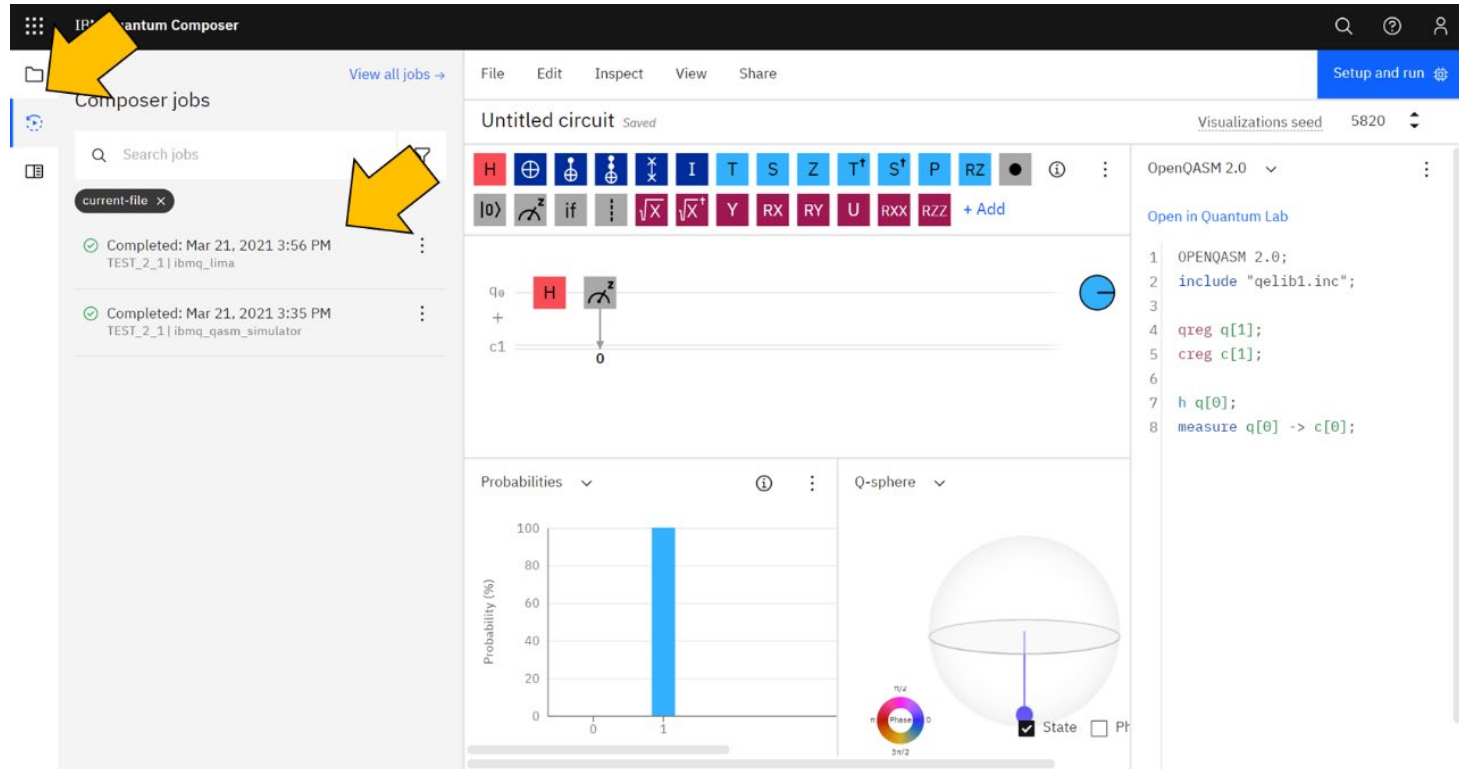
However, now after clicking on Job run settings we go to System and select `ibmq_lima`, which is a 5 qubit system offered by IBM. We select again 1,000 shots of the circuit and name it `TEST_2_1`. We now click on the blue button called `Run on ibmq_lima`, on the bottom right corner.

The screenshot shows the IBM Quantum Composer interface. On the left, a quantum circuit is visible with two qubits, `q0` and `c1`. `q0` has an `H` gate, and `c1` has a `Z` gate. Below the circuit, a probability plot shows a single bar at state `1` with 100% probability. The main panel on the right is titled 'Set up and run your circuit'. It is divided into two steps. Step 1, 'Choose a system or simulator', lists available systems. `ibmq_lima` is selected, showing it is online with 5 qubits and 8 pending jobs. Step 2, 'Choose your settings', shows the provider as `ibmq-open/main`, shots set to 1000, and the job name as `TEST_2_1`. A blue button at the bottom right is labeled 'Run on ibmq\_lima'. Yellow arrows indicate the workflow: one points from the circuit editor to the system selection, and another points from the settings to the run button.

**STEP 2: Selecting the settings and running the circuit**



Afterwards, in the Composer Jobs section in the right hand side, the job will appear as QUEUED.



The screenshot displays the IBM Quantum Composer interface. On the left, the 'Composer jobs' panel shows a list of completed jobs. Two yellow arrows highlight the interface: one points to the 'Composer jobs' header, and the other points to the job list. The job list contains two entries, both marked as 'Completed' with green checkmarks and timestamps from March 21, 2021. The first job is 'TEST\_2\_1 | ibmq\_lima' and the second is 'TEST\_2\_1 | ibmq\_qasm\_simulator'. The main workspace shows an 'Untitled circuit' with a quantum circuit diagram and a corresponding QASM code block on the right. The circuit diagram includes a Hadamard gate on qubit 0, followed by a measurement. The QASM code is as follows:

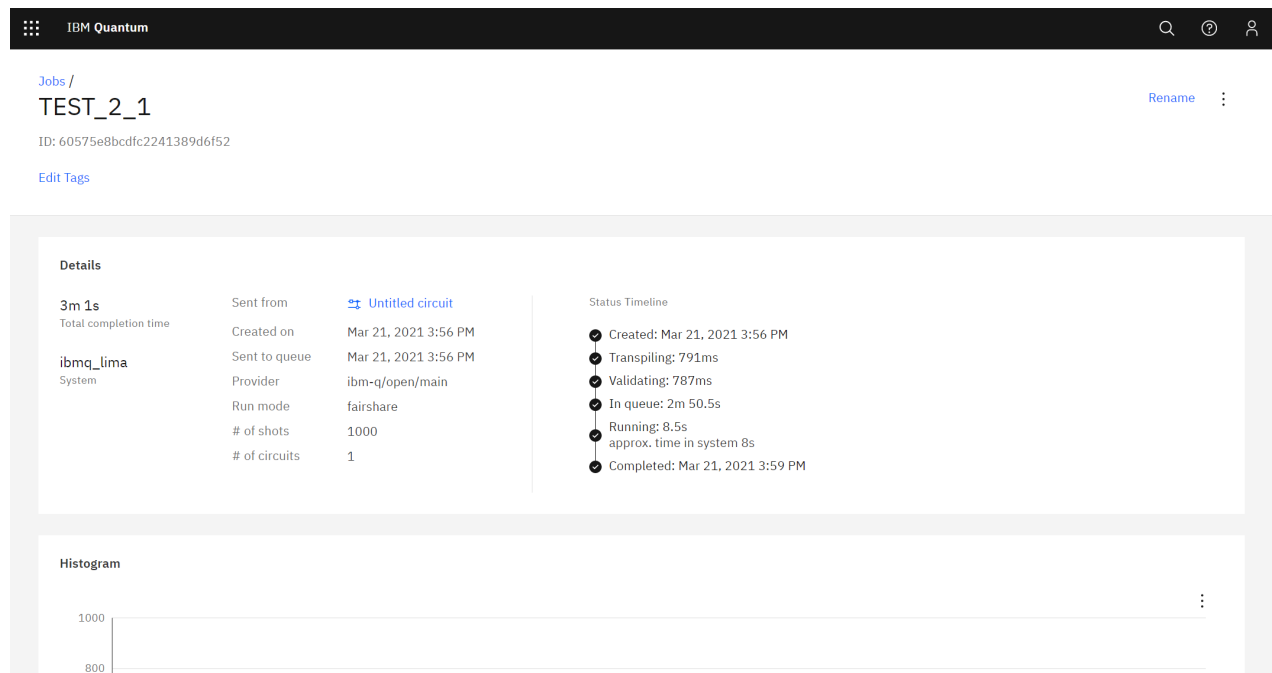
```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3
4 qreg q[1];
5 creg c[1];
6
7 h q[0];
8 measure q[0] -> c[0];
```

Below the circuit diagram, there is a 'Probabilities' section showing a bar chart with a single bar at 100% for state 1. To the right of the probabilities is a 'Q-sphere' visualization showing a sphere with a point on its surface representing the current state.

**STEP 3: In the Job Composer job appears in the list**

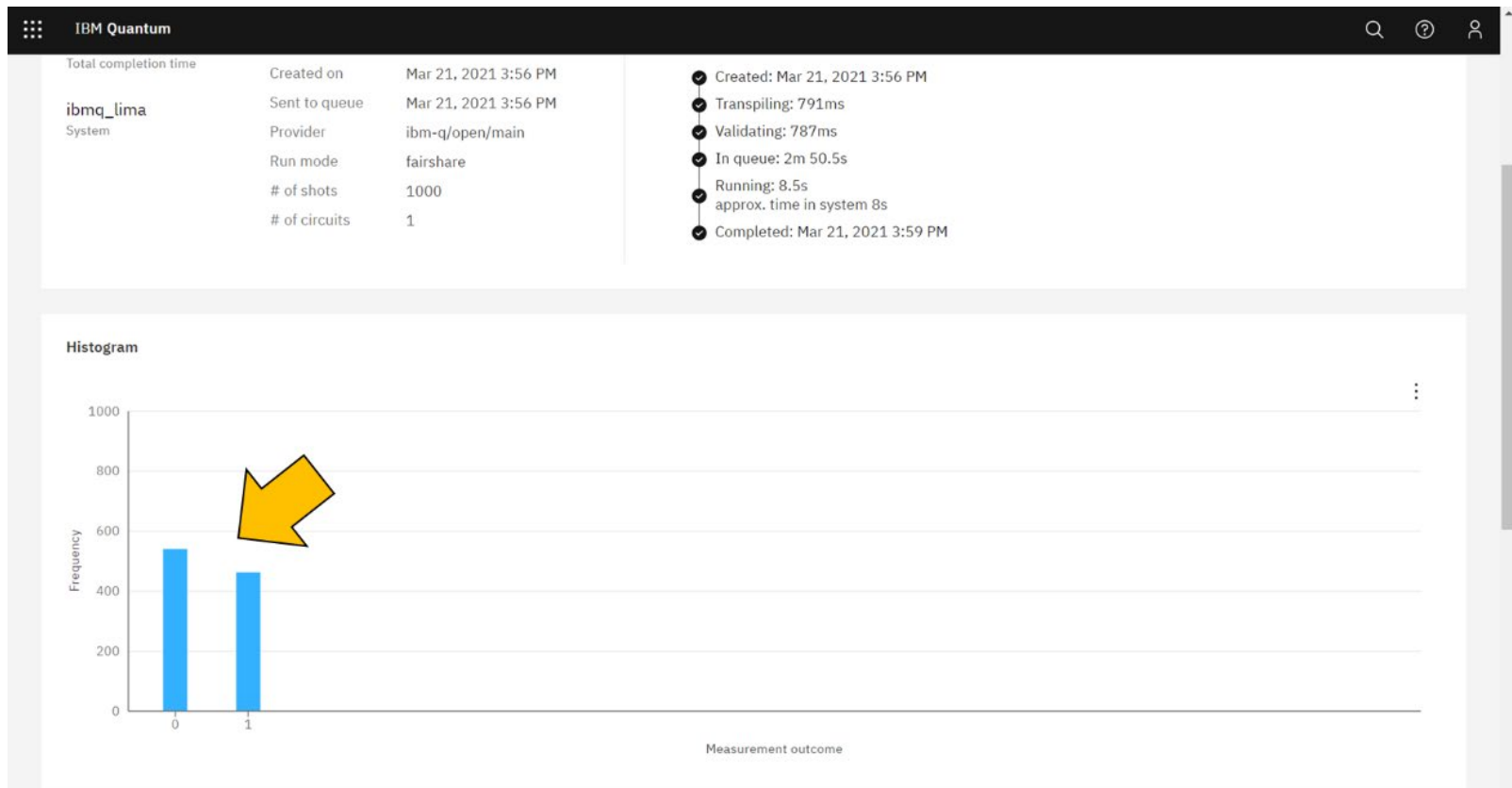
When the Job changes to COMPLETED, which might take several minutes due to the many requests present in the various systems, you can go to the left black border on the page and find the icon for Jobs and click it.

Clicking from the list on the specific job sent, we find as before the details of the run conducted. In this case, our request was on a queue for a few minutes and took 8.5 seconds to be executed.



**STEP 4:results from Job**

Scrolling down gives the summary results of the simulations in terms of a histogram. Indicating 539 zeros and 461 ones.

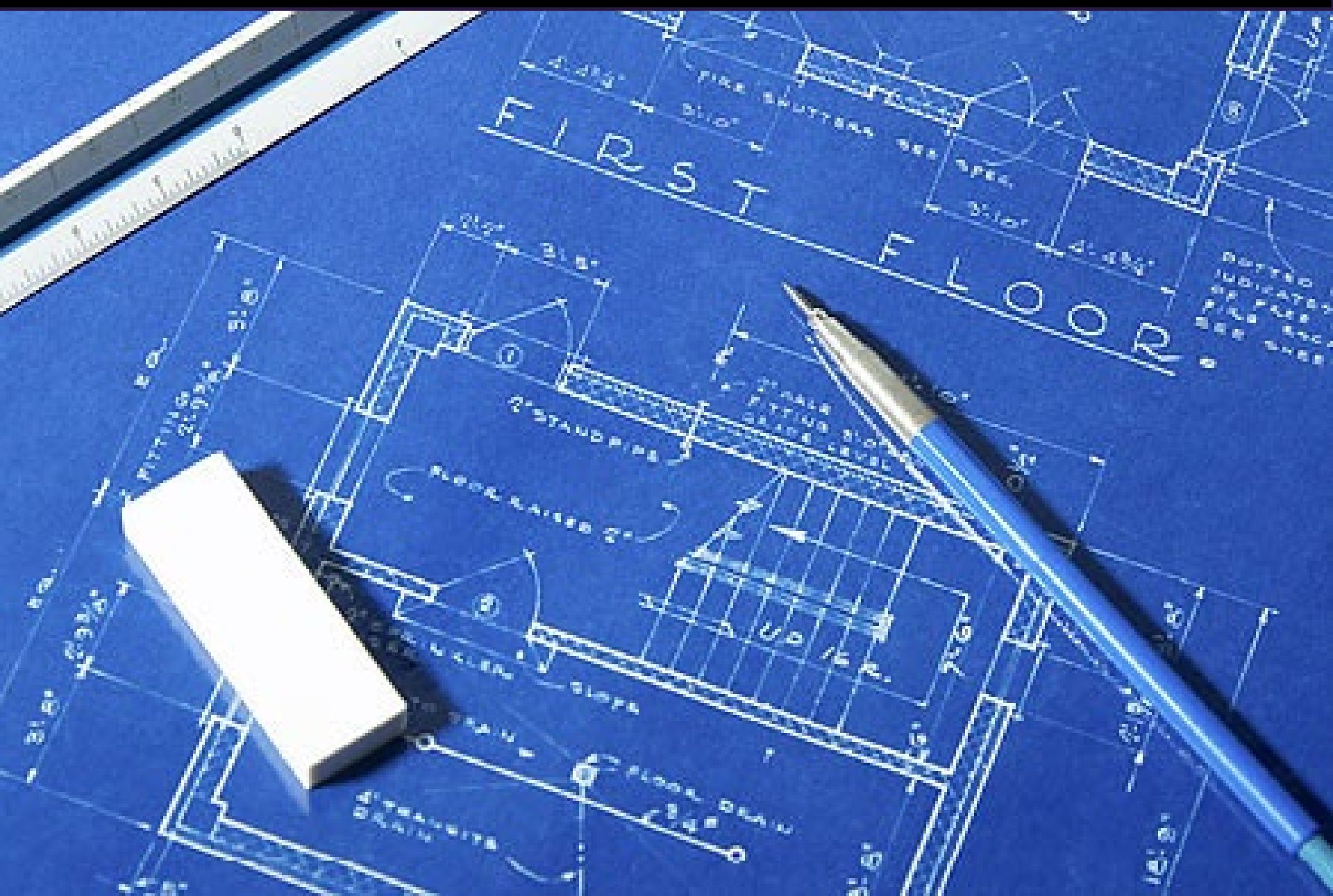


**STEP 5: histogram of the results**

As before, by scrolling further down, we have a visual representation of our circuit and its automatically-generated equivalent (or “transpiled”) circuit actually used in the Lima system to run our task.

We have been able to run again our simple, 1 qubit circuit multiple times (1,000 in fact) to obtain 1000 random samples of 0s and 1s. Now we will take these series of 0 and 1 as building blocks to construct first binary numbers from then, then integers, then decimal numbers in the range  $[0,1]$  and finally samples from a Standard Normal Distribution.

# LABORATORY 5



# LABORATORY 5: Quantum Binary Random Numbers in QISKIT

In this laboratory we will create our own Python programs in our computer based on the principles discussed before with the online service IBM Quantum Experience. We will do this in our own Python environment (I use Anaconda Individual and Jupyter Lab to run all the Labs). I run each code in Jupyter by simple copy and pasting it into a cell and pressing Shift+Enter. All the subsequent examples are based on the quantum random computer simulator offered by QISKIT, but the setting can be changed to use a real quantum computer as will be illustrated in the next chapter.

To start with, I this laboratory we will start by generating binary numbers, i.e. bits and bytes.

## Code 2.7 Quantum 1-bit generator

In this code we construct a simple 1 qubit circuit with a H-Gate and repeat its measurement 8 times (using 8 shots).

```
# CODE_2_7_QISKIT_1_qubit_8_shots

# QISKIT generate 1-bit binary (0,1) with 1 qubit 8 shots


from qiskit import QuantumCircuit, execute, Aer, IBMQ

from qiskit.visualization import *

from qiskit.tools.jupyter import *


# Create a quantum circuit with 1 qubits and 1 classic bits
qcircuit = QuantumCircuit(1,1)


# Add an Hadamard gate to the qubit
qcircuit.h(0)


# Measure and link qubit into classical bit
qcircuit.measure([0],[0])


# Execute the circuit
backend = Aer.get_backend('qasm_simulator')

result = execute(qcircuit, backend, shots=8, memory = True).result()

counts = result.get_counts(qcircuit)


# Get individual shot results
shotlist = result.get_memory()


# Output

print(counts)

print(shotlist)

plot_histogram(counts)
```

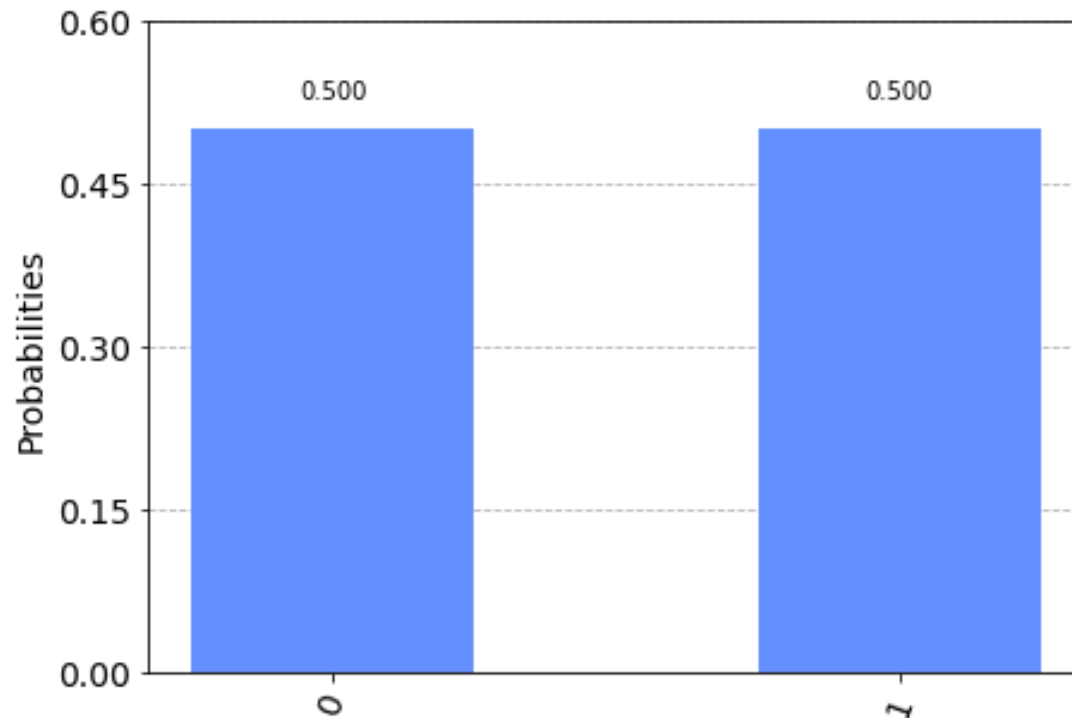
Running this circuit in Jupyter results in the following numerical results. Of the eight runs (shots), 4 times the qubit measured gave a zero (i.e. pointing up in the Bloch Sphere) and 4 times the qubit measured gave a one (i.e. pointing down in the Bloch Sphere). The results from the individual runs are given in the second row.

```
{'0': 4, '1': 4}  
['1', '0', '0', '0', '0', '1', '1', '1']
```

**OUTPUT Code 2.7: numerical results**



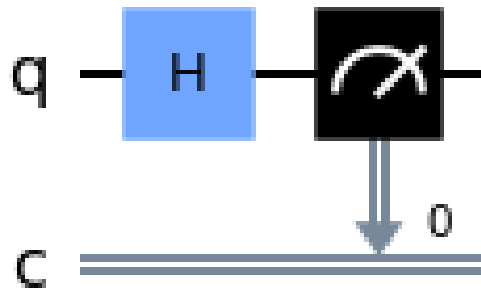
These values can be further summarized in terms of an histogram:



**OUTPUT Code 2.7: histogram**

Finally, if in Jupyter Lab we write in the next cell and execute (Shift+Enter) we can obtain a graph of our circuit:

```
# Draw the circuit  
qcircuit.draw()
```



**OUTPUT Code 2.7: circuit**

# References

James E. Gent. Random Number Generation and Monte Carlo Methods. Springer, 2013.

Paul Glasserman. Monte Carlo Methods in Financial Engineering. Springer, 2010.

Matsumoto, M.; Nishimura, T. (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". ACM Transactions on Modeling and Computer Simulation. 8 (1): 3–30.

Christian Kollmitzer, Stefan Schauer, Stefan Rass. Quantum Random Number Generation: Theory and Practice. Springer, 2020.