Lab 2: Randomness

In this lab 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

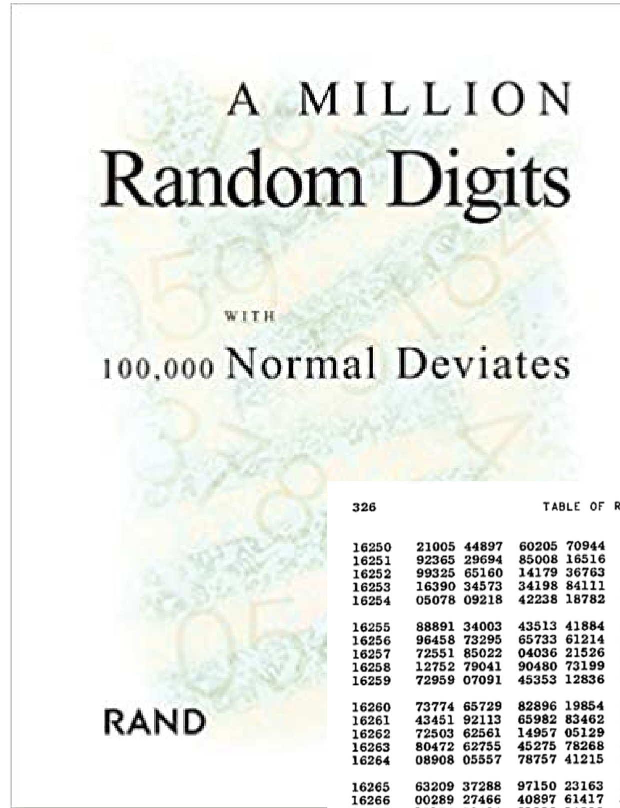
The Problem: Tossing a Coin

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?



326 TABLE OF RANDOM DIGITS

16250	21005	44897	60205	70944	51207	29121	54411	54853	96629	11803
16251	92365	29694	85008	16516	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	61536	73838	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	95593	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	53583	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.

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> and |1> 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> and |1> .

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 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).

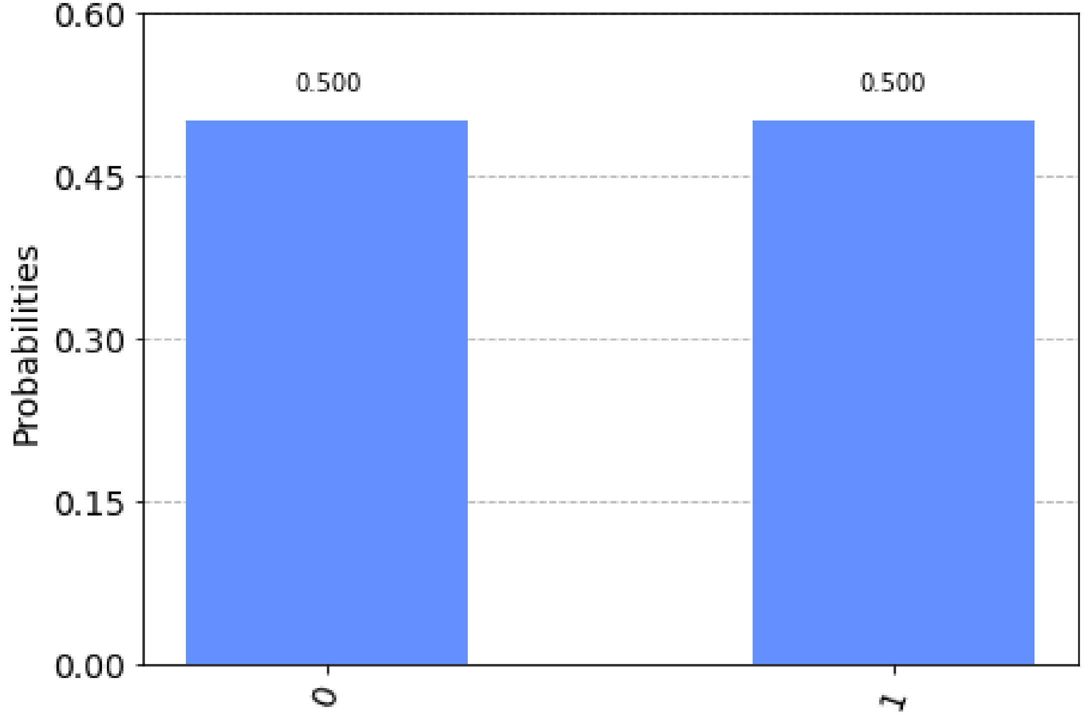
```
# CODE_2_7_QISKIT_1_qubit_8_shots

# 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

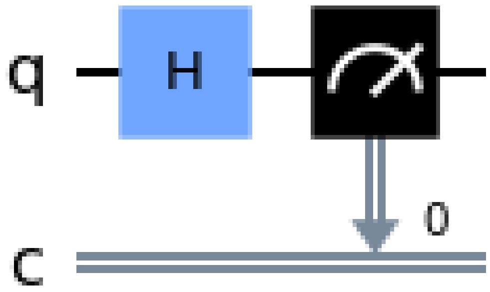
These values can be further summarized in terms of an histogram:



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.