15-110 Principles of Computing – S19

Lecture 24:
Simulation, Random numbers, Monte Carlo - 1

Teacher:
Gianni A. Di Caro

The data journey of a scientist / scientific practitioner

Natural and social sciences are based on observation data, and there are plenty of data around, from scientific experiments, statistical offices, social networks, own measurements, ….

- **Find / download data files** from data sources of interest:
  - Collection of sites with financial data: https://pydata.github.io/pandas-datareader/stable/remote_data.html
  - Collection of data repositories in multiple fields: https://www.nature.com/sdata/policies/repositories
  - Vast repository of research data: https://www.re3data.org/
  - U.S. government's open data repository: https://www.data.gov/
  - Selection of repositories: https://www.datasciencecentral.com/profiles/blogs/20-free-big-data-sources-everyone-should-check-out
- **Generate** your own data and **store/share** them on **files:**
  - ✓ Perform **real-world experiments** (in some cases it might be impractical, expensive, time-consuming)
  - ✓ Define **mathematical, computational, probabilistic models** of the phenomena under study
  - ✓ Use **Monte Carlo simulation** (*random numbers*) to generate data based on the models

The data journey of a scientist / scientific practitioner

- **Read** the data, according to their records/fields format, defined according to **standard or custom formats**
  - ✓ **CSV** data format
  - ✓ **JSON** data format
  - ✓ **…**
- **Describe** the available data:
  - ✓ **Compute measures** of interest to summarize/describe the data using **descriptive statistics**
  - ✓ **Visualize** the data and/or the results from the descriptive analysis using different visualization formats
- **Build on data** to find **correlations**, make **inference** and **predictions**, find **regularities** and **optima,** study **dynamics**
  - ✓ Use **inferential statistics**
  - ✓ Use **machine learning** and **AI techniques**
  - ✓ Use **operations research** techniques
  - ✓ Use **simulation** to simulate possible scenarios, system's evolution, possible futures, time-space interactions
    - ✓ **Deterministic simulation**
    - ✓ **Stochastic simulation**: **Monte Carlo** methods, using **random numbers** generation

Generate your own data: Monte Carlo simulation

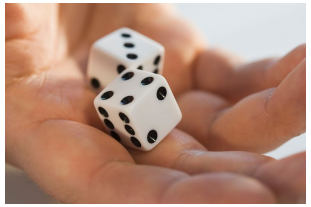
- **Generate** your own data (and **store/share** them on **files):**

  1. Define **mathematical, computational, probabilistic models** of the phenomena under study
  2. Use **Monte Carlo simulation** (*random numbers*) to generate data based on the models

"The first thoughts and attempts I made to practice [the **Monte Carlo method**] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later [in 1946] I]described the idea to John von Neumann and we began to plan actual calculations." (*Stanislav Ulam*, 1983)

A simple example: Pascal's problem

- **Blaise Pascal** (1623-1662) is the father of probability theory, that grew out of his interest for gambling games
- One of Pascal's problems was: given 24 rolls of a pair of fair dice, what is the probability of not rolling a double six?



Probability?

- On the first roll the probability of rolling a 6 on each die is $p_1 = \frac{1}{6}$

  → The probability of rolling a 6 with both dice is $p_1 \times p_1 = \frac{1}{36}$ (probability of the joint event)

  → The probability of **not rolling a double 6 on the first roll is:** $\bar{p}_1 = \left(1 - \frac{1}{36}\right) = \frac{35}{36}$

- Every roll of the dice is independent from the previous roll
- The **probability of *not* rolling a double six for n times** is the product of the probability of not rolling a double six each single time: $\bar{p}_1 \times \bar{p}_1 \times \cdots \times \bar{p}_1 = (\bar{p}_1)^n$

→ The probability of *losing the game*, by not rolling a double six 24 consecutive times is $\left(\frac{35}{36}\right)^{24} = 0.509$

At Pascal's time this was still a difficult question to answer mathematically ….

Pascal's problem: computational version (Monte Carlo algorithm)

Ingredients for MC simulation: a generative model of the phenomenon under consideration + random numbers

- ✓ Generative model: When dealing with a game, the generative model for MC simulation is "easy" to find, since it's naturally specified by the same rules of the game

- ✓ Random numbers: we will come back to this, for the time being, let's just use the generator of random numbers, from the `random` module

```
import random
for i in range(100):
    r = random.choice([1,2,3,4,5,6])
    print(r, '', end='')
```

```
2 3 6 6 5 4 4 3 4 2 5 1 3 3 5 6 6 3 4 4 1 2 2 1 3 3 5 4 5 1 6 4 4 2 5 6 1 3 5 2 5 6 6 2 2 2 4 6 6 6 2 6
4 6 3 4 2 6 2 2 5 1 4 4 5 3 2 4 3 3 5 1 1 4 6 1 6 4 4 6 5 3 2 6 1 1 2 4 1 6 5 6 6 2 4 2 6 3 3 5
```

```
1 6 5 3 1 1 2 3 2 3 6 1 3 2 4 4 5 6 1 4 3 2 1 2 4 2 4 3 5 4 4 1 1 6 3 6 5 5 5 5 3 4 5 6 2 2 1 5 5 5 3
6 4 2 3 6 2 1 6 4 6 4 1 1 4 6 2 6 2 3 3 5 5 3 2 6 5 4 1 4 5 4 2 2 3 1 3 4 1 3 6 4 5 5 3 1 2 5 4
```

```
5 6 1 5 2 5 5 5 5 3 2 2 1 6 2 1 4 2 5 3 6 1 1 3 5 1 6 5 3 1 1 3 3 2 2 2 2 5 2 6 4 4 4 2 5 3 2 6 4 1 3 1
6 1 1 3 1 1 1 4 3 1 1 5 1 4 5 3 6 5 4 6 2 1 1 2 1 1 5 5 6 1 3 2 2 6 1 3 6 3 1 3 6 6 6 3 4 3 2 1
```

Can we predict the next number?

Pascal's problem: computational version (Monte Carlo algorithm)

```
import random
def roll_die():
    return random.choice([1,2,3,4,5,6])

def pascal_problem(num_trials):
    both_six = 0
    for t in range(num_trials):
        for i in range(24):
            die_1 = roll_die()
            die_2 = roll_die()
            if die_1 == 6 and die_2 == 6:
                both_six += 1
                break
    return both_six / num_trials
```

Generation of random outcomes for rolling a single die

Generative model for simulating playing the game

```
prob = pascal_problem(1000000)
print('Combinatorial probability of losing the game: {:4.3f}'.format((35/36) ** 24))
print('Empirical probability of losing the game: {:4.3f}'.format(1 - prob))
```

Combinatorial probability of losing the game: 0.509
Empirical probability of losing the game: 0.509

With MC simulations we can easily try out “what if…”

- What if the dice are not fair? (e.g., more weight is put on the six face of both dice)

```
import random
def roll_die():
    return random.choice([1,2,3,4,5,6,6)

prob = pascal_problem(20000)
print('Combinatorial probability of losing the game (fair): {:4.3f}'.format((35/36) ** 24))
print('Empirical probability of losing the game (unfair): {:4.3f}'.format(1 - prob))
```

```
Combinatorial probability of losing the game (fair): 0.509
Empirical probability of losing the game (unfair): 0.129
```

With MC simulations we can easily try out “what if…”

- What if only one of the dice is not fair? (e.g., more weight is put on the six face of one die)

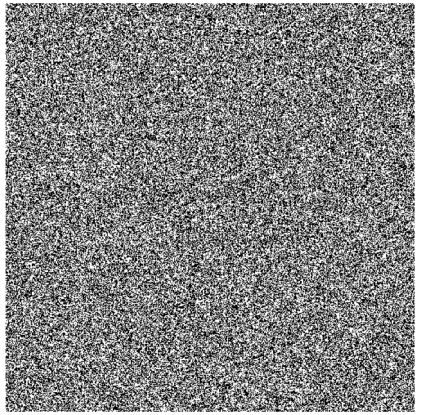
```
import random
def roll_die_unfair():
    return random.choice([1,2,3,4,5,6,6])

def roll_die():
    return random.choice([1,2,3,4,5,6])

def pascal_problem(num_trials, roll_1, roll_2):
    both_six = 0
    for t in range(num_trials):
        for i in range(24):
            die_1 = roll_1()
            die_2 = roll_2()
            if die_1 == 6 and die_2 == 6:
                both_six += 1
                break
    return both_six / num_trials

prob = pascal_problem(1000000, roll_1, roll_2)
print('Combinatorial probability of losing the game (fair): {:4.3f}'.format((35/36) ** 24))
print('Empirical probability of losing the game (unfair): {:4.3f}'.format(1 - prob))
```

```
Combinatorial probability of losing the game (fair): 0.509
Empirical probability of losing the game (unfair): 0.306
```

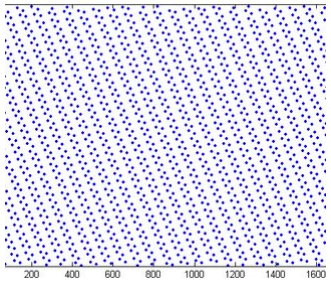
Random numbers and `random` module

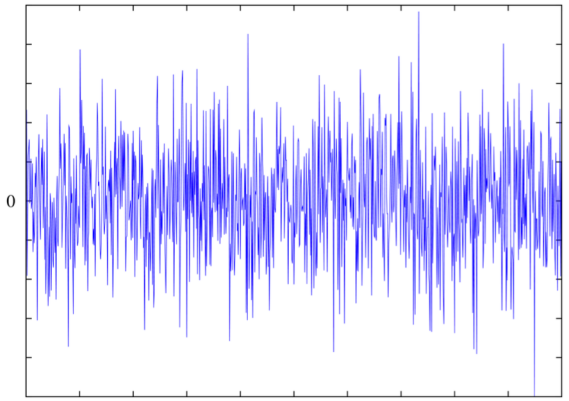
- The `random` module provides ***pseudo-random*** **number generators for various distributions**

✓ A **random phenomenon** is typically a phenomenon that we don't fully understand, such as it *deceives us* and we are not able to make reliable predictions about the phenomenon by looking at its behavior

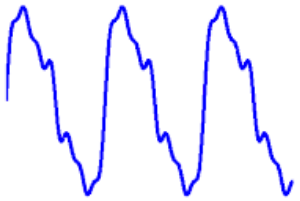


Where the next dot will appear?





What is the next value?



Head or tail?

```
def LCG(state, a=1664525, c=1013904223, m=2**32-1):
    next_state = (a * state + c) % m
    return (next_state)

state = 1
for i in range(100):
    state = LCG(state)
    print(state % 2, '', end='')
```

Random numbers and `random` module

- `randint(from, to):` method that returns a random integer n such that $from \leq n \leq to$, where both $from$ and to must be integer numbers. The interval is sampled **uniformly**

```
for i in range(100):
    print(random.randint(0,1), '', end='')
```

```
1 0 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 0 1 1 1 1 1 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 0 0 1 0
1 1 0 1 1 1 0 1 0 1 1 0 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0
```

```
1 0 1 0 1 0 1 1 1 1 0 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0
0 1 0 0 1 0 1 0 0 1 1 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 1 0 0 0 1 1 1 1 1 1 1 0 0 1
```

- ✓ The above sequences *look* random, it's almost impossible to reliably and systematically predict next outcomes
- ✓ However, they are generated by a rather simple algorithm: they are *not truly random*, IF we would know the algorithm it'd be very easy to exactly predict next outcomes → These are **pseudo-random sequences**
- ➢ An algorithm used by a *computer* for the generation of *randomly-looking sequences* is termed a **pseudo-random number generator (PRNG)**
- ➢ A PRNG produces sequences of numbers whose mathematical properties approximate those of sequences of *truly random numbers*

Seeding the generator

- A PRNG has a ***state*****:** each generated value uniquely depends on and is determined by the previously generated numbers since the first call of the generator
- It starts with a given initial **seed value** (state) and generates a new random number following each call of the generator, and the generated sequence depends on the initial seed
  - Invoking the generator with the same seed will produce the same sequence of values, while sequences started with different seeds are expected to be significantly different

- `seed(val=None)`: **initializes the random number generator** with the integer value `val`, if `val` is omitted the current internal time is used instead
- If not explicitly invoked, `seed()` is internally invoked by python: an *initialization* is always performed

Seeding the generator

```
seed = 12356
print('First call of PRNG, seed = ', seed)
random.seed(seed)
for i in range(100):
    print(random.randint(0,1), '', end='')
print('\n’)

random.seed(seed)
print('Second call of PRNG, seed = ', seed)
for i in range(100):
    print(random.randint(0,1), '', end='')
print('\n')

seed = 12357
random.seed(seed)
print('Third call of PRNG, seed = ', seed)
for i in range(100):
    print(random.randint(0,1), '', end='')
```

```
First call of PRNG, seed =  12356
0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 1 1 1
1 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

Second call of PRNG, seed =  12356
0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 1 1 1
1 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

Third call of PRNG, seed =  12357
0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0
0 1 1 1 0 1 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0
```

Why pseudo-random numbers?

- Why not to use, for instance, some randomly sampled value from the computer itself (e.g., current time in nanoseconds, or voltage levels, or cpu temperature) and use it to generate “true” random sequences?
  - Because it might be not obvious to asses the true randomness of the sample values
  - Because we would lose *repeatability*: every sequence would be different, every experiment using random numbers would produce different results
- Properties of true random sequences: …

Some graphical visualization and other useful methods

- A graphical representation of the generated values can help to assess (at least visually) the randomness of the sequences

- Let’s check this out in the notebook, together with additional useful methods from the random module …