Imperial College
London

Python for Simulations: Random Numbers and Monte Carlo

Session 2

Imperial College
London

Key Ideas for Today

- **Random number generation:** Using Python to create randomness
- **Random walks:** Simulating price movements step by step
- **Loops:** Repeating operations to build simulations
- **Monte Carlo method:** Estimating expected values through simulation
- **Law of Large Numbers:** Why averaging many simulations works

Imperial College
London

Python's Random Tools

random module (built-in):

```
random.random() # Random float between 0 and 1
random.choice([options]) # Randomly select from list
random.randint(a, b) # Random integer between a and b
```

numpy.random (for numerical work):

```
np.random.normal(mean, std) # Normal random numbers
np.random.randn() # Standard normal (mean=0, std=1)
# Works with arrays for efficiency
```

Random seeds: Use `random.seed(42)` or `np.random.seed(42)` to make results reproducible.

Imperial College
London

Simulating a Coin Toss

Example setup: Simulate flipping a fair coin where Heads = +1 and Tails = -1.

Single coin toss:

```
import random
# Single coin toss
coin_flip = random.choice([-1, +1])
print(coin_flip) # Either -1 or +1
```

Multiple tosses:

```
# Simulate 10 coin tosses
tosses = [random.choice([-1, +1]) for i in range(10)]
print(tosses) # List of 10 random +1/-1 values
```

Key takeaway: `random.choice()` makes it easy to simulate discrete random events like coin flips or price movements.

Imperial College
London

The Discrete Random Walk

Toy model: At each step the price goes:

$$S_{n+1} = S_n + \varepsilon_n, \qquad \varepsilon_n \in \{-1, +1\}$$

Intuition:

- Price moves up/down randomly.
- Each step independent (news arrives unpredictably).

Properties:

- Easy to simulate.
- Captures unpredictability.
- But unrealistic: price can be negative.

Imperial College
London

Building a Random Walk in Python

Example setup:

Start with price $S_0 = 100$ • Simulate 100 steps • Each step: randomly add +1 or -1

```python
import random
# Initial setup
S = 100
prices = [S] # Store all prices
# Simulate 100 steps
for step in range(100):
change = random.choice([-1, +1])
S = S + change
prices.append(S)
print(prices[:10]) # First 10 prices
print(f"Final price: {prices[-1]}")
```

Key pattern: Initialize → Loop → Update → Store

Imperial College
London

Visualizing Random Walks with Matplotlib

Why visualize: Seeing the price path helps understand randomness and unpredictability.

```
import random
import matplotlib.pyplot as plt
# Generate random walk
S = 100
prices = [S]
for step in range(100):
S = S + random.choice([-1, +1])
prices.append(S)
# Plot the path
plt.plot(prices, 'b-', linewidth=2)
plt.xlabel('Time Step')
plt.ylabel('Price')
plt.title('Random Walk Simulation')
plt.grid(True)
plt.axhline(y=100, color='r', linestyle='--') # Starting price
plt.show()
```

Result: A jagged line showing unpredictable price movement.

Imperial College
London

For Loops — Repeating Operations

Why loops: Simulations require repeating calculations many times.

For loop syntax:

```
for variable in sequence:
# Code to repeat
```

Example 1 — Iterate over a range:

```
for i in range(5):
print(i) # Prints 0, 1, 2, 3, 4
```

Example 2 — Calculate multiple returns:

```
prices = [100, 105, 103, 108]
for i in range(len(prices) - 1):
ret = (prices[i+1] - prices[i]) / prices[i]
print(f"Return {i}: {ret:.2%}")
```

- `range(n)` generates numbers from 0 to n-1
- Loop variable (i) changes each iteration
- Indented code runs repeatedly

Imperial College
London

While Loops — Conditional Repetition

Syntax:

```
while condition:
# Code to repeat
```

When to use:

- For loops: known number of iterations
- While loops: repeat until condition becomes false

Example — Simulate until price doubles:

```
import random
S = 100
steps = 0
while S < 200:
S = S + random.choice([-1, +1])
steps += 1
print(f"Price doubled after {steps} steps")
```

Warning: Make sure the condition eventually becomes false, or the loop runs forever!

Imperial College
London

Normal Random Numbers with NumPy

Why normal distribution: Returns in finance are approximately normally distributed.

Single random number from N(0, 1):

```
import numpy as np
z = np.random.randn()
```

Multiple random numbers:

```
z_array = np.random.randn(1000) # 1000 standard normal values
```

Custom mean and standard deviation:

```
# N(mean=0.05, std=0.2)
returns = np.random.normal(0.05, 0.2, 1000)
```

mean (μ): center of distribution

std (σ): spread/volatility

size: how many numbers to generate

Use case: Simulating stock returns with realistic randomness.

Imperial College
London

Geometric Brownian Motion in Python

Discretized GBM formula:

$$S_{t+\Delta t} = S_t \times \exp[(\mu - \tfrac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\, Z] \quad \text{where } Z \sim N(0,1)$$

Python implementation:

```python
import numpy as np
# Parameters
S0 = 100 # Initial price
mu = 0.10 # Drift (10% annual return)
sigma = 0.20 # Volatility (20%)
T = 1.0 # Time horizon (1 year)
dt = 1/252 # Daily time step
steps = 252 # Trading days in a year
# Simulate price path
S = S0
prices = [S]
for t in range(steps):
Z = np.random.randn()
S = S * np.exp((mu - 0.5*sigma**2)*dt + sigma*np.sqrt(dt)*Z)
prices.append(S)
print(f"Final price: {prices[-1]:.2f}")
```

Key insight: GBM ensures prices stay positive (unlike simple random walk).

Imperial College
London

Monte Carlo: Estimating by Averaging

Problem:

We want to compute expectations (e.g., future price, option payoff).

Monte Carlo method:

1. Simulate many random price paths
2. Compute the quantity of interest for each path
3. Take the average

$$\mathbb{E}[f(S_T)] \approx {}^{1}/_{N} \sum^{N}_{i=1} f(S_T^{(i)})$$

Intuition: Like polling — many random samples give the true average (by Law of Large Numbers).

Use cases:

- Option pricing
- Risk metrics (VaR)
- Portfolio optimization

Imperial College
London

Law of Large Numbers (LLN)

Intuition:

- If you repeat a random experiment many times,
- the average outcome gets closer to the true expected value.

Example: Toss a fair coin.

$$E[\text{Heads}] = 0.5$$

Python demonstration:

```python
import random
def simulate_tosses(n):
heads = sum([random.choice([0, 1]) for _ in range(n)])
return heads / n
print(f"10 tosses: {simulate_tosses(10)}") # May be noisy
print(f"1,000 tosses: {simulate_tosses(1000)}") # Closer to 0.5
print(f"100,000 tosses: {simulate_tosses(100000)}") # Very close to 0.5
```

Why we care in finance:

- Monte Carlo simulation relies on LLN
- More simulated paths → more accurate estimate

Imperial College
London

Monte Carlo for Option Pricing

Goal:

Estimate expected call option payoff.

Steps:

1. Simulate many stock price paths to maturity
2. Calculate payoff for each path: max(S_T - K, 0)
3. Average all payoffs

Python approach:

```
import numpy as np
def call_option_payoff(S_T, K):
return max(S_T - K, 0)
# Parameters
S0 = 100
K = 100
mu = 0.10
sigma = 0.20
T = 1.0
N_sims = 10000
```

Imperial College
London

Monte Carlo for Option Pricing (continued)

Monte Carlo simulation loop:

```
# Monte Carlo simulation
payoffs = []
for i in range(N_sims):
# Simulate final price
Z = np.random.randn()
S_T = S0 * np.exp((mu - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
# Calculate call option payoff
payoff = call_option_payoff(S_T, K)
payoffs.append(payoff)
# Estimate expected payoff
expected_payoff = sum(payoffs) / len(payoffs)
print(f"Expected payoff: £{expected_payoff:.2f}")
```

Imperial College
London

Complete Monte Carlo Example

Complete working example:

```python
import numpy as np
# Option parameters
S0 = 100 # Current stock price
K = 105 # Strike price
T = 0.5 # 6 months to expiry
mu = 0.12 # Expected return
sigma = 0.25 # Volatility
N = 50000 # Number of simulations
# Set seed for reproducibility
np.random.seed(42)
# Run Monte Carlo
payoffs = []
for i in range(N):
# Simulate stock price at maturity
Z = np.random.randn()
S_T = S0 * np.exp((mu - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
# Calculate call option payoff
payoff = max(S_T - K, 0)
payoffs.append(payoff)
# Results
avg_payoff = np.mean(payoffs)
std_payoff = np.std(payoffs)
print(f"Average payoff: £{avg_payoff:.2f}")
print(f"Std deviation: £{std_payoff:.2f}")
print(f"Based on {N:,} simulations")
```

Key pattern: This is the standard Monte Carlo template you'll use repeatedly.

Imperial College
London

Vectorization: Making Code Faster

Problem: Loops in Python can be slow for large simulations.

Loop version (slow):

```
payoffs = []
for i in range(10000):
Z = np.random.randn()
S_T = S0 * np.exp((mu - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
payoff = max(S_T - K, 0)
payoffs.append(payoff)
avg = np.mean(payoffs)
```

Vectorized version (fast):

```
Z = np.random.randn(10000) # All random numbers at once
S_T = S0 * np.exp((mu - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
payoffs = np.maximum(S_T - K, 0) # Vectorized max
avg = np.mean(payoffs)
```

Speed improvement: 10-100x faster for large simulations!

Key insight: NumPy operates on entire arrays, avoiding slow Python loops.

Imperial College
London

Summary of Session 2

What we learned:

> **Random number generation:** `random.choice()`, `np.random.randn()`
> **Random walks:** Simulating price movements step by step
> **For loops:** Repeating operations with `for i in range(n)`
> **While loops:** Conditional repetition
> **Normal random numbers:** `np.random.normal(mean, std)`
> **GBM simulation:** Realistic stock price paths
> **Monte Carlo method:** Simulate → Compute → Average
> **Law of Large Numbers:** More simulations = more accuracy
> **Vectorization:** Using NumPy for speed

Practice recommendations:

> Simulate your own random walks with different parameters
> Experiment with different numbers of Monte Carlo simulations
> Try pricing put options using Monte Carlo
> Visualize multiple price paths on the same plot

Next steps:

> Learn about conditional statements (if/else)
> Explore more advanced NumPy operations
> Study option Greeks through simulation
> Build portfolio simulations with multiple assets

Key takeaway:

You now have the tools to simulate financial scenarios and estimate expected values — the foundation of quantitative finance!