Monte_Carlo

November 11, 2018

In [1]:
```
#include <algorithm>
#include <iostream>
#include <random>

using namespace std;
```

In [2]:
```
#include "xplot/xfigure.hpp"
#include "xplot/xmarks.hpp"
#include "xplot/xaxes.hpp"
```

In [3]:
```
const double pi = std::acos(-1);
const int num_points = 1000;
std::vector<double> cx;
std::vector<double> cy;

for (double a = 0.0; a <= pi / 2; a += pi / (2 * num_points)) {
    cx.push_back(std::sin(a));
    cy.push_back(std::cos(a));
}
```

In [4]:
```
xpl::figure fig;
fig.padding_x = 0.025;
fig.padding_y = 0.025;

xpl::linear_scale clx, cly;
xpl::lines circumference{ clx, cly };
circumference.colors = std::vector<std::string>({ "red" });
circumference.x = cx;
circumference.y = cy;
fig.add_mark(circumference);

xpl::axis cax{ clx }, cay{ cly };
cay.orientation = "vertical";
fig.add_axis(cax);
fig.add_axis(cay);
```

1 Introduction to Monte Carlo Methods

FE 522 C++ Programming in Finance

1.1 Monte Carlo Methods

- "Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results." (https://en.wikipedia.org/wiki/Monte_Carlo_method)
- There are several applications of Monte Carlo methods in different areas of knowledge.
- Random number generators based on different probability functions may be used when implementing a Monte Carlo method, depending on the application.

1.2 A classical example: estimating π with a simulation

In [5]:
```
fig
```

Out[5]:
```
A Jupyter widget
```

In [6]:
```
const double true_pi = std::acos(-1);
```

In [7]:
```
cout << "True Pi = " << true_pi << '\n';
```

```
True Pi = 3.14159
```

In [8]:
```
random_device rd; // Will be used to obtain a seed for the random number engine
mt19937 gen{ rd() }; // Standard mersenne_twister_engine seeded with rd()
uniform_real_distribution<> urd{ 0.0, 1.0 };
```

In [9]:
```
vector<double> px;
vector<double> py;

for (int i = 0; i < 1000; ++i) {
    px.push_back(urd(gen));
    py.push_back(urd(gen));
}
```

In [10]:
```
xpl::figure fig2;
fig.padding_x = 0.025;
fig.padding_y = 0.025;

xpl::linear_scale sx, sy;
xpl::scatter scatter{ sx, sy };
scatter.x = px;
scatter.y = py;
fig2.add_mark(scatter);

fig2.add_mark(circumference);
```

```
         xpl::axis ax{ sx }, ay{ sy };
         ay.orientation = "vertical";
         fig2.add_axis(ax);
         fig2.add_axis(ay);
```

In [11]:
```
fig2
```

Out[11]: A Jupyter widget

In [12]:
```
double simulate_pi(int num_simulations)
{
    double x = 0.0;
    double y = 0.0;
    int inside_circle = 0;

    for (int i = 0; i < num_simulations; ++i) {
        x = urd(gen);
        y = urd(gen);

        if (x * x + y * y < 1.0)
            ++inside_circle;
    }

    return 4.0 * inside_circle / num_simulations;
}
```

In [13]:
```
cout << "True Pi = " << true_pi << '\n';
cout << "Simulated Pi = " << simulate_pi(1000) << '\n';
```

```
True Pi = 3.14159
Simulated Pi = 3.136
```

In [14]:
```
cout << "True Pi = " << true_pi << '\n' << '\n';
cout << "# Sim." << '\t' << "Sim. Pi" << '\n';

for (int i = 1; i < 6; ++i) // [10; 100,000]
    cout << pow(10, i) << '\t' << simulate_pi(pow(10, i)) << '\n';
```

```
True Pi = 3.14159

# Sim.         Sim. Pi
10         2.8
100         3.32
1000         3.212
10000         3.1016
100000         3.134
```

1.3 Pricing Derivatives

- There are several complex derivatives that do not have an analytical solution for their price calculations.
- Monte Carlo methods provide simple and flexible approaches to price such financial instruments.
- Monte Carlo methods can handle several different random factors that affect the model, e.g. options with multiple underlying instruments, their volatility, and different interest rates.
- Monte Carlo methods also allow for the incorporation of more realistic pricing processes, e.g. models with price *jumps*.
- Monte Carlo methods may, however, be computationally inefficient in their most basic form.

1.4 European Option Pricing with Monte Carlo

- The value of an option, when considering a risk-neutral pricing model, is the value of its expected *payoff* discounted by the risk-free interest rate.
- Assuming a constant risk-free interest rate r, this would be the price of an European Option call:

$$C_t = e^{-r(T-t)} \mathbb{E}_t^*[\max(S_T - K, 0)].$$

- It is possible to obtain an estimate of the expected *payoff* by computing the average of a large number of simulated *payoffs*.
- For a given European Option which pays C_T at maturity date T, we first simulate the risk-neutral process for each of its variables, from their initial value until T. Then, we calculate the payoff $C_{T,j}$ for each simulation j.
- The value $C_{T,j}$ is then discounted using the simulated interest rates:

$$C_{0,j} = \exp\left(-\int_0^T r_u du\right) C_{T,j}.$$

- In the case of a constant interest rate:

$$C_{0,j} = \exp(-rT) C_{T,j}.$$

- If we repeat the simulations M times, it's possible to obtain an estimate of the real option price C_0 by computing the average of the values obtained in each simulation:

$$\hat{C}_0 = \frac{1}{M} \sum_{j=1}^{M} C_{0,j}.$$

- We are going to assume the premisses of the Black & Scholes model, with a constant interest rate.
- In order to implement the Monte Carlo method, we need to simulate the geometric brownian motion (GBM) of the underlying asset:

$$dS_t = rS_t dt + \sigma S_t dz_t.$$

- The logarithm of a GBM variable follows an arithmetic brownian motion, and is normally distributed. Because of that, its simulation is more computationally efficient. Given:

$$x_t = ln(S_t).$$

- Therefore:

$$dx_t = \nu dt + \sigma dz_t,$$

$$\nu = r - 0.5\sigma^2.$$

- To discretize the differential equation of x_t, we substitute the infinitesimals dx, dt and dz with very small Δx, Δt and Δz steps:

$$\Delta x = \nu \Delta t + \sigma \Delta z.$$

- We can write:

$$x_{t+\Delta t} = x_t + \nu \Delta t + \sigma(z_{t+\Delta t} - z_t).$$

- Thus:

$$S_{t+\Delta t} = S_t exp(\nu \Delta t + \sigma(z_{t+\Delta t} - z_t)).$$

- The random increment:

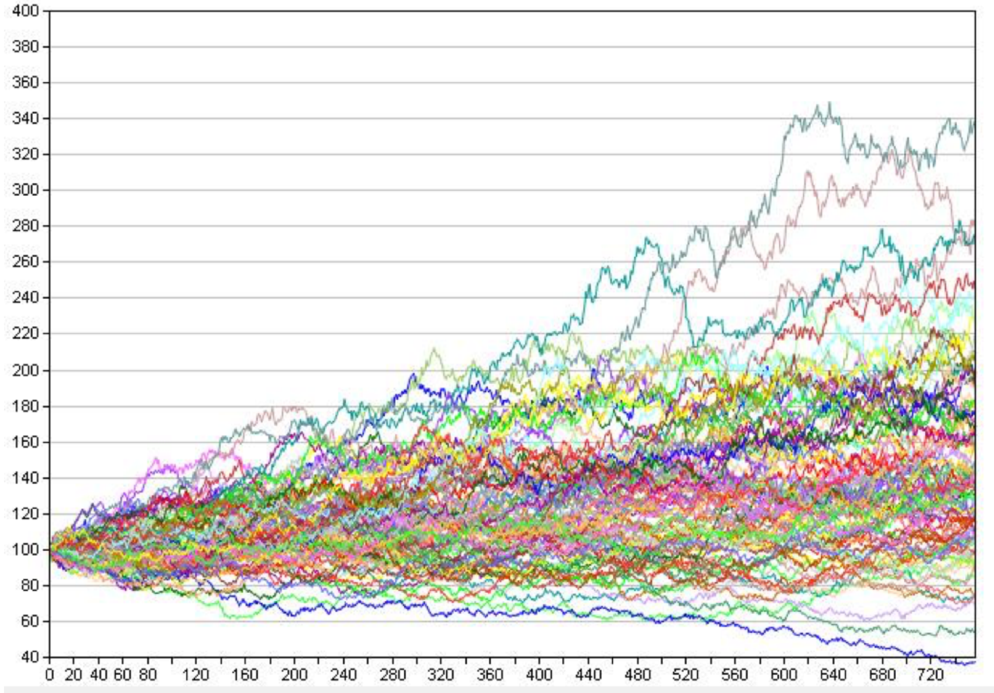
$$\Delta z = z_{t+\Delta t} - z_t,$$

has mean zero and variance Δt, so it can be simulated with random sampling of

$$\epsilon_t \sqrt{\Delta t},$$

where ϵ_t is a sample of a standard normal distribution ($N(0,1)$).

- We now know how to simulate values of S_t.
- We divide the simulation period (from t to T) in N steps so that $\Delta t = (T-t)/N$.



MonteCarlo.png

- To estimate the price of an European Option call, all we have to do is to compute:

$$\hat{C}_0 = \exp(-rT)\frac{1}{M}\sum_{j=1}^{M}\max(S_{T,j} - K, 0).$$

In [15]:
```
enum OptionType {
    CALL,
    PUT
};
```

In [16]:
```
class EuropeanOption {
private:
    OptionType m_optionType; // option type
    double m_s;              // spot price
    double m_k;              // strike price
    double m_r;              // interest rate
    double m_t;              // time to maturity
    double m_vol;            // volatility

public:
    EuropeanOption(OptionType type, double s, double k, double r, double t, double vol
        : m_optionType{ type }
        , m_s{ s }
        , m_k{ k }
```

```
                , m_r{ r }
                , m_t{ t }
                , m_vol{ vol }
            {
            }

            double getPrice();
            double getMonteCarloPrice(int num_simulations, int num_steps);

        private:
            double N(double x);
        };
```

In [17]:
```
double EuropeanOption::getPrice()
{
    double price = 0.0;

    double d1 = (log(m_s / m_k) + (m_r + m_vol * m_vol / 2) * m_t) / (m_vol * sqrt(m_t
    double d2 = d1 - m_vol * sqrt(m_t);

    if (m_optionType == CALL)
        price = N(d1) * m_s - N(d2) * m_k * exp(-m_r * m_t);
    else
        price = N(-d2) * m_k * exp(-m_r * m_t) - N(-d1) * m_s;

    return price;
}
```

In [18]:
```
// Normal CDF
double EuropeanOption::N(double value)
{
    return 0.5 * erfc(-value * sqrt(0.5));
}
```

In [19]:
```
double EuropeanOption::getMonteCarloPrice(int num_simulations, int num_steps)
{
    double price = 0.0;

    double dt = m_t / num_steps;
    double nudt = (m_r - 0.5 * m_vol * m_vol) * dt;
    double volsqrtdt = m_vol * sqrt(dt);

    random_device rd;
    mt19937 gen{ rd() };
    std::normal_distribution<> dis{ 0, 1 };

    double x0 = log(m_s), x = 0.0, sum = 0.0;
```

```
            for (int i = 0; i < num_simulations; ++i) {
                x = x0;

                for (int j = 0; j < num_steps; ++j)
                    x += nudt + volsqrtdt * dis(gen);

                sum += (m_optionType == CALL) ? max(exp(x) - m_k, 0.0) : max(m_k - exp(x), 0.(
            }

            price = sum * exp(-m_r * m_t) / num_simulations;

            return price;
        }
```

```
In [20]: EuropeanOption eo{ CALL, 100.0, 100.0, 0.12, 3.0, 0.20 };
```

```
In [21]: cout << "Analytical Price: " << eo.getPrice() << '\n';
         cout << "Monte Carlo Price: " << eo.getMonteCarloPrice(1000, 1000) << '\n';
```

```
Analytical Price: 32.4468
Monte Carlo Price: 32.2859
```

```
In [22]: cout << "Analytical Price: " << eo.getPrice() << '\n' << '\n';
         cout << "# Sim." << '\t' << "Sim. Price" << '\n';

         for (int i = 1; i < 6; ++i) // [10; 100,000]
             cout << pow(10, i) << '\t' << eo.getMonteCarloPrice(pow(10, i), 1000) << '\n';
```

```
Analytical Price: 32.4468

# Sim.         Sim. Price
10        30.145
100        33.7273
1000        31.3861
10000        32.5799
100000        32.4253
```