Monte Carlo Pricing

What we have learned:

- As exercises we have written a large amount of MATLAB style functionality in the file `matlib.cpp`. In particular we can:
  - Generate random numbers with `randUniform` and `randn`.
  - Generate plots with `plot` and `hist`.
  - Compute statistics of vectors with `min`, `mean`, `prctile` etc.
- We have learned how to write simple classes.

What we will do now:

- Add a function to `BlackScholesModel` to generate price paths.
- Test our price paths using `mean` etc. and plot them using `plot`.
- Write a class `MonteCarloPricer` that uses a `BlackScholesModel` to generate price paths and then uses risk-neutral pricing to price a `CallOption` by Monte Carlo.

Generate price path specification

We wish to write a function generatePricePath which takes a final date toDate and a number of steps nSteps and generates a random Black–Scholes Price path with the given number of steps.

```
class BlackScholesModel {
public:
    ... other members of BlackScholesModel ...

    std::vector<double> generatePricePath(
                                  double toDate,
                                  int nSteps) const;
};
```

Note that the class declaration effectively contains the specification. If you choose good function and variable names, you won't need too many comments.

Generate risk-neutral price path specification

We also want a function `generateRiskNeutralPricePath` which behaves the same, except it uses the $\mathbb{Q}$-measure to compute the path.

```
\begin{cpp}
class BlackScholesModel {
public:
    ... other members of BlackScholesModel ...

    std::vector<double> generateRiskNeutralPricePath(
                                  double toDate,
                                  int nSteps) const;
};
\end{cpp}
```

Private helper function

To implement these functions, we introduce a `private` function that allows you to choose the drift in the simulation of the price path.

```
class BlackScholesModel {
    ... other members of BlackScholesModel ...
private:
    std::vector<double> generateRiskNeutralPricePath(
                                double toDate,
                                int nSteps,
                                double drift) const;
};
```

This function is private because we've only created it to make the implementation easier. Users of the class don't need (or even want) to know about it.

Algorithm for Black–Scholes price paths

Algorithm

- *Define*
$$\delta t_i = t_i - t_{i-1}$$
- *Choose independent, normally distributed* ϵ_i*, with mean* 0 *and standard deviation* 1*.*
- *Define*
$$s_{t_i} = s_{t_{i-1}} + \left(\mu - \frac{1}{2}\sigma^2\right)\delta t_i + \sigma\sqrt{\delta t_i}\epsilon_i$$
- *Define* $S_{t_i} = \exp(s_{t_i})$*.*
- S_{t_i} *simulate the stock price at the desired times.*

Implement the helper function

```
vector<double> BlackScholesModel::generatePricePath(
        double toDate,
        int nSteps,
        double drift ) const {
    vector<double> path(nSteps,0.0);
    vector<double> epsilon = randn( nSteps );
    double dt = (toDate-date)/nSteps;
    double a = (drift - volatility*volatility*0.5)*dt;
    double b = volatility*sqrt(dt);
    double currentLogS = log( stockPrice );
    for (int i=0; i<nSteps; i++) {
        double dLogS = a + b*epsilon[i];
        double logS = currentLogS + dLogS;
        path[i] = exp( logS );
        currentLogS = logS;
    }
    return path;
}
```

Implement the public functions

```
vector<double> BlackScholesModel::generatePricePath(
        double toDate,
        int nSteps ) const {
    return generatePricePath( toDate, nSteps, drift );
}
```

```
vector<double> BlackScholesModel::
    generateRiskNeutralPricePath(
        double toDate,
        int nSteps ) const {
    return generatePricePath(
        toDate, nSteps, riskFreeRate );
}
```

Notice that with this design we've avoided writing the same complex code twice.

Implement a visual test

We'd like to see a price path, we can use the `LineChart` class.

```
void testVisually() {
    BlackScholesModel bsm;
    bsm.riskFreeRate = 0.05;
    bsm.volatility = 0.1;
    bsm.stockPrice = 100.0;
    bsm.date = 2.0;

    int nSteps = 1000;
    double maturity = 4.0;

    vector<double> path =
       bsm.generatePricePath( maturity, nSteps );
    double dt = (maturity-bsm.date)/nSteps;
    vector<double> times =
        linspace(dt,maturity,nSteps);
   LineChart lineChart;
   lineChart.setTitle("Stock price path");
   lineChart.setSeries(times, path);
   lineChart.writeAsHTML("examplePricePath.html");
}}
```

Extending `matlib`

- We've used the `linspace` function on the previous slide.
- This wasn't one of the homework exercises, but would have been easy enough.
- Adding new functions like this to `matlib` is so simple that we may do so from time to time without bothering to mention that we have done so.

```
void testRiskNeutralPricePath() {
    rng("default");

    BlackScholesModel bsm;
    bsm.riskFreeRate = 0.05;
    bsm.volatility = 0.1;
    bsm.stockPrice = 100.0;
    bsm.date = 2.0;

    int nPaths = 10000;
    int nsteps = 5;
    double maturity = 4.0;
    vector<double> finalPrices(nPaths,0.0);
    for (int i=0; i<nPaths; i++) {
        vector<double> path =
            bsm.generateRiskNeutralPricePath(
                 maturity, nsteps );
        finalPrices[i] = path.back();
    }
    ASSERT_APPROX_EQUAL( mean( finalPrices ),
        exp( bsm.riskFreeRate*2.0)*bsm.stockPrice,
             0.5);
}
```

Understanding the automated test

- If our risk-neutral pricing function is correct, then the discounted mean of the final stock price should equal the initial price.
- Since this test depends upon generating random numbers, we seed the random-number generator. I've written a function `rng` to do this. Just like `MATLAB` you should pass in the string `'default'`.

MonteCarloPricer specification

We want to write a class called `MonteCarloPricer` that:

- Is configured with `nScenarios`, the number of scenarios to generate. This should default to 10000.
- Has a function `price` which takes a `CallOption` and a `BlackScholesModel`, and computes (by Monte Carlo) the price of the `CallOption`.

We'll see that the declaration for `MonteCarloPricer` is pretty much the same thing as this specification.

MonteCarloPricer declaration

```
#pragma once

#include "stdafx.h"
#include "CallOption.h"
#include "BlackScholesModel.h"

class MonteCarloPricer {
public:
    /*  Constructor */
    MonteCarloPricer();
    /*  Number of scenarios */
    int nScenarios;
    /*  Price a call option */
    double price( const CallOption& option,
                  const BlackScholesModel& model );
};

void testMonteCarloPricer();
```

Revision

- The header file is called ...
- The header file always begins with ...
- We always `#include` ...
- A constructor looks like a function declaration except ...
- We pass the `option` and the `model` by ...
- Whenever we write code we ... it.

MonteCarloPricer.cpp

```cpp
#include "MonteCarloPricer.h"

#include "matlib.h"

using namespace std;

MonteCarloPricer::MonteCarloPricer() :
    nScenarios(10000) {
}
```

Revision

- The cpp file is called ...
- We always start a cpp file with ...
- The code beginning `MonteCarloPricer::MonteCarloPricer` is ...

Monte Carlo Pricing

Algorithm (Monte Carlo Pricing)

To compute the Black–Scholes price of an option whose payoff is given in terms of the prices at times t_1, t_2, ..., t_n:

- *Simulate stock price paths* in the risk-neutral measure. *i.e. use the algorithm above with* $\mu = r$.
- *Compute the payoff for each price path.*
- *Compute the discounted mean value.*
- *This gives an unbiased estimate of the true risk-neutral price.*

The implementation of price

```
double MonteCarloPricer::price(
        const CallOption& callOption,
        const BlackScholesModel& model ) {
    double total = 0.0;
    for (int i=0; i<nScenarios; i++) {
        vector<double> path= model.
                generateRiskNeutralPricePath(
                    callOption.maturity,
                    1 );
        double stockPrice = path.back();
        double payoff = callOption.payoff( stockPrice );
        total+= payoff;
    }
    double mean = total/nScenarios;
    double r = model.riskFreeRate;
    double T = callOption.maturity - model.date;
    return exp(-r*T)*mean;
}
```

Remarks

- We only need the final payoff to price a call option, so we only request one step in the price path.

We need a test

```
static void testPriceCallOption() {
    rng("default");

    CallOption c;
    c.strike = 110;
    c.maturity = 2;

    BlackScholesModel m;
    m.volatility = 0.1;
    m.riskFreeRate = 0.05;
    m.stockPrice = 100.0;
    m.drift = 0.1;
    m.date = 1;

    MonteCarloPricer pricer;
    double price = pricer.price( c, m );
    double expected = c.price( m );
    ASSERT_APPROX_EQUAL( price, expected, 0.1 );
}
```

Random Numbers

- The C++ random-number generator `rand` isn't very good. In particular it will give biased answers for large Monte Carlo simulations.
- A standard random-number generator for Monte Carlo simulations is called the Mersenne Twister algorithm.
- Implemented as `mt19937` in `<random>`
- `randUniform` and `randn` have been modified to use this class.

Generating random numbers

```
vector<double> randuniform( int n ) {
    vector<double> ret(n, 0.0);
    for (int i=0; i<n; i++) {
        ret[i] = (mersenneTwister()+0.5)/
                 (mersenneTwister.max()+1.0);
    }
    return ret;
}
```

Note the use of *operator overloading*.

Reseeding the random-number generator

```
void rng( const string& description ) {
    ASSERT( description=="default" );
    mersenneTwister.seed(mt19937::default_seed);
}
```

We are using a *static variable* here. This is a global variable associated with a class.

Summary

Key functionality for the course:	
`matlib`	Functionality similar to MATLAB
`BlackScholesModel`	Represents the Black–Scholes Model
`CallOption`	Represents a call option contract
`PutOption`	Represents a put option contract
`MonteCarloPricer`	Prices options by Monte Carlo
Non financial functionality:	
`LineChart`	Plots line charts
`Histogram`	Plots histograms
`PieChart`	Plots pie charts
`geometry`	Some elementary mathematical examples
Code you can use, but don't fully understand:	
`testing`	Macros to make testing less boring