FMO6 — Web: https://tinyurl.com/ycaloqk6 Polls: https://pollev.com/johnarmstron561
Lecture 4

Dr John Armstrong

King's College London

August 22, 2020

Feedback

- Generally very positive feedback. The most contentious issue is the pace.
- Two people said the room was unsatisfactory, but didn't say why.

Overview

What we've seen

- We have learned how to program Matlab
- We have learned the importance of testing
- We have learned how to use numerical integration to perform risk neutral pricing

What's to come

- Simulating prices for
  - Monte Carlo pricing (today)
  - Risk management
  - Simulation
- Pricing using PDE methods
- Optimization

Monte Carlo Pricing

Simulating stock prices

- We wish to simulate stock prices following the model
$$\mathrm{d}S_t = S_t(\mu\,\mathrm{d}t + \sigma\,\mathrm{d}W_t)$$
at the time T.
- "Simulating stock prices" means writing a function which generates stock prices that have exactly the same probability distribution as S_T.
- One way to simulate stock prices is first to simulate the log of the stock price z_T instead, then exponentiate.
- So we need to simulate z_T following
$$\mathrm{d}z_t = \left(\mu - \frac{1}{2}\sigma^2\right)\,\mathrm{d}t + \sigma\,\mathrm{d}W_t$$
- By definition of Brownian motion this means generating z_T which are normally distributed with mean $z_0 + (\mu - \frac{1}{2}\sigma^2)\,T$ and standard deviation $\sigma\sqrt{T}$.

Algorithm (Simulating Black–Scholes prices at time T)

- *Generate a normally distributed random number* ϵ *with mean* 0 *and standard deviation* 1*.*
- *Set*
$$\tilde{z}_T = z_0 + \left(\mu - \frac{1}{2}\sigma^2\right) T + \sigma\sqrt{T}\epsilon$$
- *Set*
$$\tilde{S}_T = \exp(\tilde{s}_T)$$
- *The probability density of*
$$\tilde{S}_T$$
is the same as that of
$$S_T$$
- *So* $\tilde{S}_T$ *is a simulated stock price.*

Mnemonic

We have the process

$$\mathrm{d}z_t = \left(\mu - \frac{1}{2}\sigma^2\right)\mathrm{d}t + \sigma\,\mathrm{d}W_t$$

Write:

$$\mathrm{d}z_t \mapsto \delta z_T = z_T - z_0$$

$$\mathrm{d}t \mapsto \delta T = T - 0$$

$$\mathrm{d}W_t \mapsto \sqrt{\delta T}\,\epsilon$$

To get:

$$z_T - z_0 = \left(\mu - \frac{1}{2}\sigma^2\right) T + \sigma\sqrt{T}\epsilon$$

Note on the Mnemonic

- The previous slide suggests approximating a continuous stochastic process with a discrete process with time intervals δT.
- This idea does work and is called the Euler method. It *approximately simulates* a continuous stochastic process. Such methods improve as δT tends to zero.
- Our algorithm for simulating stock prices *exactly simulates* Black Scholes stock prices no matter how large δT is.
- The point is that we can integrate Brownian motion exactly, but not more general processes.

Growth rate of noise

- Why do we say that
$$\mathrm{d}W_t \mapsto \sqrt{\delta T}\,\epsilon?$$
- The change in the stock price over time Nh is composed of N independent small changes over time periods of length h.
- By the central limit theorem (assuming the small changes have finite variance), the change in the stock price over time Nh has standard deviation proportional to $\sqrt{N}$.
- Therefore, the cumulative effect of the noise grows at a rate proportional to $\sqrt{\delta T}$. Sigma is defined to be the constant of proportionality.
- This is why sigma has units of years$^{-1/2}$.

Simulating at multiple time points

- Simulating the stock price at times $0 = t_0 < t_1 < t_2 < \ldots < t_n$ means generating values $\tilde{S}_{t_1}$, $\tilde{S}_{t_2}$, $\ldots \tilde{S}_{t_n}$ so that the joint distribution function of $S_{t_1}, \ldots S_{t_n}$ is the same as that of $\tilde{S}_{t_1}, \ldots \tilde{S}_{t_n}$.
- By the Markov property of the stock price, we simply need to simulate the stock to time t_1, then use this as the starting point of a simulation up to time t_2 and so on up to time t_n.

Simulating 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*
$$\tilde{z}_{t_i} = \tilde{z}_{t_{i-1}} + \left(\mu - \frac{1}{2}\sigma^2\right)\delta t_i + \sigma\sqrt{\delta t_i}\epsilon_i$$
- *Define* $\tilde{S}_{t_i} = \exp(z_{t_i})$.
- $\tilde{S}_{t_i}$ *simulate the stock price at the desired times.*
- *These formulae all match our mnemonic exactly.*

Comment

- The first algorithm is a special case of this.
- I could have just written this algorithm initially, but I want to emphasize that if you are only interested in stock prices at time T there is no need to simulate intermediate times.

Turning this into code

Example

A stock follows the Black–Scholes price process with drift $\mu = 0.05$ and $\sigma = 0.1$. The initial stock price is $S_0 = 100$.
Simulate the stock price every day for 1 year and plot the result.

We'll write a function that answers the question directly and then try to generalize our code.

```
function  plotBSPath()
T = 1.0;
nSteps = 365;
S0 = 100;
mu = 0.05;
sigma = 0.1;
```

The interesting bit

```
dt = T/nSteps;
logS0 = log( S0);
logS = zeros(nSteps,1);
for t=1:nSteps
    eps = randn();
    dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
    if (t==1)
        lastLogS = logS0;
    else
        lastLogS = logS(t-1);
    end
    logS(t) = lastLogS + dlogS;
end
```

Plotting the result

```
S = exp( logS );
times = (1:nSteps)*dt;
plot(times,S);
```

- Using the last three slides, write a MATLAB function to solve the problem.
- Experiment with some different values of μ and σ to see if you think the results look correct.

Generalizing the code

- Let us generalize the code by writing a function that takes the following parameters
  - `S0`, `mu` and `sigma` describing the model
  - Parameters `T` and `nSteps` describing the total time interval for the simulation and the number of equally sized pieces to divide the time interval into.
  - A parameter `nPaths` indicating the number of price paths to be simulated.
- We would like the function to return a matrix of simulated stock prices. Each row should correspond to a scenario and each column should correspond to a time step.
- The function should also return a vector of time points indicating the time corresponding to each column.

Version 1 - idea

- The most obvious idea is to take our existing code and repeat it `nPaths` times using a `for` loop
- This will give a solution containing two `for` loops, so I've called the function `generateBSPaths2Loops`
- In fact we will be able to eliminate these loops when we optimize our code.

Version 1 - code

```
function [ S, times ] = generateBSPaths2Loops( ...
    T, S0, mu, sigma, nPaths, nSteps  )

dt = T/nSteps;
logS0 = log( S0);
logS = zeros(nPaths,nSteps);
for p=1:nPaths
    for t=1:nSteps
        eps = randn();
        dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
        if (t==1)
            lastLogS = logS0;
        else
            lastLogS = logS(p,t-1);
        end
        logS(p,t) = lastLogS + dlogS;
    end
end
S = exp(logS);
times = dt:dt:T;
end
```

Version 2 - idea

- In MATLAB you can often “vectorize” your code to eliminate loops this:
  - Makes it easier to read
  - Often makes it considerably faster
- When you repeat the same operation across independent scenarios, you can always vectorize.
- We’ve already seen that A(1:end,j) means the j-th column of x. You can abbreviate this to $A(:,j)$.

Version 2 - code

```
function [ S, times ] = generateBSPaths1Loop( ...
    T, S0, mu, sigma, nPaths, nSteps )

dt = T/nSteps;
logS0 = log( S0);
eps = randn(nPaths, nSteps);
dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
logS = zeros(nPaths,nSteps);
for t=1:nSteps
    if (t==1)
        lastLogS = logS0;
    else
        lastLogS = logS(:,t-1);
    end
    logS(:,t) = lastLogS + dlogS(:,t);
end
S = exp(logS);
times = dt:dt:T;
end
```

Using cumsum

$$\texttt{cumsum}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_1 + x_2 \\ x_1 + x_2 + x_3 \\ \dots \\ x_1 + x_2 + x_3 + \dots + x_n \end{pmatrix}$$

- `cumsum(v)` computes the cumulative sums of a vector `v`.
- `cumsum(A)` computes the cumulative sums of the rows of a matrix `A`.
- `cumsum(A,1)` computes the cumulative sums of the rows of a matrix `A`. Rows are the first dimension.
- `cumsum(A,2)` computes the cumulative sums of the columns of a matrix `A`. Columns are the second dimension.

Version 3 - Code

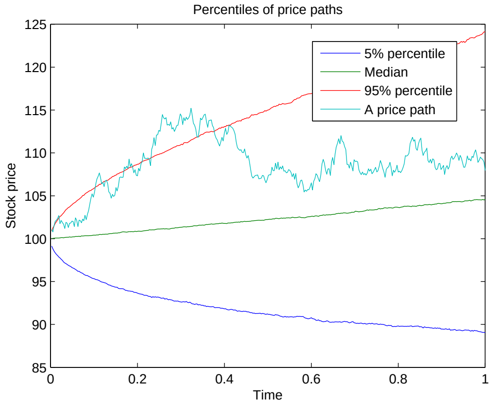
```
function [ S, times ] = generateBSPaths( ...
    T, S0, mu, sigma,nPaths, nSteps   )

dt = T/nSteps;
logS0 = log( S0);
eps = randn( nPaths, nSteps );
dlogS = (mu-0.5*sigma^2)*dt + sigma*sqrt(dt)*eps;
logS = logS0 + cumsum( dlogS, 2);
S = exp(logS);
times = dt:dt:T;

end
```

- You can think of `cumsum` as meaning “integrate” in the code above.
- Notice that this code is quite a bit simpler than the first version

Plot of percentiles



Using the `prctile` function, we can create a plot that shows a random price path for our problem and also the percentiles for the price at each time

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 $t_1, t_2, \ldots, t_n$

- *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*

Knockout options

Example

A discrete up and out call option with strike K and barrier B and maturity T is an option that pays 0 if the stock price is ever above B at the end of the business day. If it reaches time T without hitting the barrier, its payoff is given by $\max(S_T - K, 0)$

The payoff is determined entirely by the prices at the end of each business day.

Example

An Asian call option with maturity T has its payoff determined by the average price $\overline{S}$ at the close of the last n days of trading up to and including maturity. The payoff is given by $\max(\overline{S} - K, 0)$

Variations on a theme

- An option that becomes worthless if the price goes below a barrier is called a down and out option.
- Up-and-out and down-and-out options are termed "knockout options".
- A knock in option is one where the option is worthless unless the stock price crosses a barrier
- One has "up-and-in" and "down-and-in" options.
- You can create knockout puts, digital Asians etc..
- One can also mathematically model continuous barrier and Asian options. We can approximate these by using a large number of time points.

Trivial example

- European put and call options can be priced by this Monte Carlo method.
- This method is equivalent to one of the Monte Carlo integration techniques covered last week (exercise: convince yourself)
- Monte Carlo pricing is always a numerical integration technique (we are computing expectations and expectations are defined as integrals)
- If we use n points on the price path then we are computing an n-dimensional integral
- So Monte Carlo integration will be better than variants on the rectangle rule etc. if n is greater than roughly 3 or 4.

American options

- You *cannot* use our algorithm to price American options.
- This is because the payoff of an American option is not specified in the contract as a function of the price at fixed times.
- There is a more sophisticated technique called American Monte Carlo, but we will not discuss that in this course.

Estimating the error

- By the central limit theorem, we expect that the sample mean of the discounted payoff is approximately normally distributed with standard deviation
$$\frac{\tilde{\sigma}_P}{\sqrt{N}}$$
where $\tilde{\sigma}_P$ is the population standard deviation of the discounted payoff.
- The sample standard deviation $\tilde{\sigma}_S$ is an estimator for the population standard deviation (although it is slightly biased, the bias can be ignored for large N)
- So we estimate that the standard error of our price is
$$\frac{\tilde{\sigma}_S}{\sqrt{N}}$$
- We can then use this to construct approximate confidence intervals.

Pricing an up-and-out call option

Example

We wish to price a discrete up-and-out call option with barrier B, strike K and maturity T where one tests to see if the option has hit the barrier at *nSteps* evenly spaced times over the lifetime of the option.
The stock price follows the Black Scholes model with parameters S_0, μ, r, σ as usual.
We wish to write a function to price the option using Monte Carlo simulations with *nPaths* paths.
The function should also return an estimate of the error.

Strategy

- We already have a function that computes a matrix of prices with rows corresponding to scenarios and columns corresponding to times.
- We would like to write a function that computes the payoff of our option as a vector given the matrix of prices
- We will then be able to price the option by taking the mean of this vector and discounting.

Step 1 - Implement the payoff function

```
function [ payoff ] = computeKnockoutPayoff2Loops(...
    strike, barrier, priceHistory )

nPaths = size( priceHistory, 1 );
nSteps = size( priceHistory, 2 );
payoff = zeros( nPaths, 1 );
for p=1:nPaths;
    knockedOut = 0;
    for t=1:nSteps
        if priceHistory(p,t)>barrier
            knockedOut = 1;
        end
    end
    if (~knockedOut)
        finalPrice = priceHistory( p, nSteps );
        if (finalPrice>strike)
            payoff(p)=finalPrice-strike;
        end
    end
end
end
```

Comments

- We don't need to pass the number of paths and the number of steps to this function, we've used the `size` function to deduce that from the `priceHistory`
- Since this code is completely repetitive across scenarios we know that we can vectorize it to improve efficiency. We'll do this shortly.
- Lets test our code first.

Step 2 - Test the payoff function

```
function testComputeKnockoutPayoff2Loops()

stockPrices = [100,101,102; 100,120,107; 100,103,108 ];
payoffs = computeKnockoutPayoff2Loops(105,110,stockPrices);
assertApproxEqual( payoffs(1), 0, 0.001);
assertApproxEqual( payoffs(2), 0, 0.001);
assertApproxEqual( payoffs(3), 3, 0.001);

end
```

Step 3 - Write the pricing function

```
function [price, errorEstimate]=priceKnockoutByMonteCarlo(...
    strike, barrier, T,...
    S0, r, sigma, ...
    nPaths, nSteps )

% Generate paths in risk neutral measure (mu=r)
priceHistory = generateBSPaths(T,S0,r,sigma,nPaths,nSteps);
payoffs = computeKnockoutPayoff(strike,barrier,priceHistory);
discountedPayoff = exp(-r*T)*payoffs;
price = mean( discountedPayoff );
errorEstimate = std( discountedPayoff )/sqrt(nPaths);

end
```

Step 4 - Test the pricing function

Exercise

Remarks

- The main pricing function is remarkably simple
- The main pricing function does little more than explain in English what the Monte Carlo pricing algorithm actually is.
- The real work is done in `generateBSPaths` and `computeKnockoutPayoff`.
- In the exam I may ask you to write pseudo-code for certain algorithms. I strongly recommend splitting your pseudo code into functions that explain the algortihm. Given how simple well written code is, you might well prefer to write actual code.
- We'll now see how to vectorize the `computeKnockoutPayoff` function.

Removing the loop across paths

```
function [ payoff ] = computeKnockoutPayoff1Loop( ...
    strike, barrier, priceHistory )

nPaths = size( priceHistory, 1 );
nSteps = size( priceHistory, 2 );

knockedOut = zeros( nPaths, 1);
for t=1:nSteps
    knockedOutThisTime = (priceHistory(:,t) > barrier);
    knockedOut = knockedOut | knockedOutThisTime;
end
finalPrice = priceHistory( :, nSteps );
inMoney = finalPrice>strike;
payoff=(~knockedOut).*inMoney.*(finalPrice-strike);
end
```

Understanding the vectorized code

- We have used the vectorization trick of using the fact that `knockedOut` and `inMoney` take the values `1` and `0` since `1` represents `true` and `0` represents false.
- This means

```
(~knockedOut).*inMoney.*(finalPrice-strike)
```

  Will only be non zero when we haven't knocked out and aren't in the money.
- We have used `|` which means element-by-element 'OR' just as `.*` means element-by-element multiplication.

Vectorizing again

- We are currently looping to find out if the price was ever above the barrier.
- Suppose we compute `priceHistory > barrier`. This will consist of 1's when the price is above the barrier and 0's when the price is below the barrier.
- This means that the maximum of `priceHistory>barrier` in each row will be 1 if the barrier knocked out and 0 otherwise.
- You can use `max(A,[],2)` to compute a vector of the maximum across the columns. `max(A,[],1)` computes the maximum across the rows.

No loops

```
function [ payoff ] = computeKnockoutPayoff( ...
    strike, barrier, priceHistory )

knockedOut = max( priceHistory>barrier, [], 2);
notKnockedOut = 1-knockedOut;
finalPrice = priceHistory(:,end);
inMoney = finalPrice>strike;
payoff = inMoney .* notKnockedOut .* (finalPrice-strike);

end
```

Which version to learn?

- I've shown how to price the code using three short functions `computeKnockoutPayoff`, `priceKnockoutCallOption` and `generateBSPaths`.
- We have three equivalent versions of `generateBSPaths` and `computeKnockoutPayoff`
- I would focus on understanding the final versions of the code. I introduced the other versions to help you understand the final version.

Should you vectorize your own code?

Good things about vectorization

- Vectorization makes the code more readable *once you know the standard tricks*.
- Vectorization may make the code faster

Bad things about vectorization

- Vectorization makes the code less readable if you don't know the standard tricks.
- Vectorization may make the code take longer for you to write (until you've mastered it)

I recommend writing what you find comes most naturally and optimizing only if there is a problem (or the question tells you to)

Unit testing and random numbers

- Algorithms that use random numbers will sometimes give poor answers by chance.
- If you're not careful this means that your tests will sometimes fail by chance.
- To make your tests reliable you should *seed the random number generator*.
    - Pseudo random number generators have a state which determines what they will do next.
    - Each time you generate a number they change state. This ensures they will generate a different number the next time.
    - Manually fixing the state is called *seeding* the random number generator.

Seeding the random number generator

- In MATLAB `rng('default')` sets the random number generator back to its default state.
- Therefore you should start unit tests of functions that use the random generator with a call of `rng('default')`.

Computing Greeks by Monte Carlo

- A trader will not thank you if you can compute the price but not the Greeks. Why is this?

Computing Greeks by Monte Carlo

- A trader is likely to be using some variant of the delta hedging strategy so they will want to make sure their portfolio is delta neutral
- If they can't ensure their portfolio is delta neutral, they won't trade because they can't hedge the risk of their position.
- In general, a price is useless without a trading strategy to achieve the price.

Numerical differentiation

Let f be a smooth function.

- We can approximate the derivative at x as
$$f'(x) = \frac{f(x+h)-f(x)}{h} \quad \textit{or} \quad f'(x) = \frac{f(x)-f(x-h)}{h}$$
for small h. These are called the *forward* and *backward* estimates. Error $O(h)$ by Taylor's theorem.
- A better approximation is
$$f'(x) = \frac{f(x+h)-f(x-h)}{2h}.$$
This is called the *central estimate*. Error $O(h^2)$.
- In the latter case, the first order error terms cancel and the error is bounded by
$$\sup_{c \in [x-h, x+h]} \frac{|f^{(3)}(c)|}{6} h^2$$

What h should you choose?

- We want to choose h that is small enough to give a reasonably accurate value value.
- We want to choose h so it isn't so small that rounding errors dominate the calculation.
- Choosing $h > \sqrt{\epsilon} x$ where ϵ denotes the accuracy of the computer will ensure that h isn't too small. For our purposes $\epsilon = 2.2 \times 10^{-16}$
- For ultimate accuracy one worries about whether $h + x$ can be represented accurately on the computer and tweaks the value accordingly.
- Since we will apply this to a Monte Carlo method, we needn't worry about choosing the optimal h.

How not to compute Monte Carlo delta

Do not:

- Compute the Monte Carlo price with an initial stock price of S_0
- Compute the Monte Carlo price with an initial stock price of $S_0 + h$
- Take the difference and divide by h

This is because the Monte Carlo prices are randomly generated. The random error will overwhelm the systematic difference we are trying to measure.

How to compute Monte Carlo delta

Algorithm (Monte Carlo Delta)

- *Compute the Monte Carlo price with an initial stock price of* $S_0 - h$
- *Compute the Monte Carlo price with an initial stock price of* $S_0 + h$ using exactly the same random numbers in the simulation
- *Take the difference and divide by* $2h$

Why this works

- Monte Carlo pricing is really just numerical integration
- Under reasonable conditions, the partial derivative of an integral is the integral of the partial derivative.
- Our Monte Carlo Delta algorithm is equivalent to computing the derivative of the pricing kernel numerically and then integrating by Monte Carlo integration.

MATLAB implementation

```
function [ delta ] = computeDeltaByMonteCarlo( ...
    strike, barrier, T,...
    S0, r, sigma, ...
    nPaths, nSteps )

h = 10^(-6)*S0; % Won't cause rounding problems
                % but a minute change financially

rng('default');
p1 = priceKnockoutByMonteCarlo(strike,barrier,T,...
    S0-h, r, sigma, ...
    nPaths, nSteps );
rng('default');
p2 = priceKnockoutByMonteCarlo(strike,barrier,T,...
    S0+h, r, sigma, ...
    nPaths, nSteps );
delta = (p2-p1)/(2*h);

end
```

Explanation

- We have seeded the random number generator to ensure that the same random numbers are used at $S_0 - h$ and at $S_0 + h$.
- It would be better in practice to avoid generating the same random numbers twice as this inevitably be faster and would get rid of the bias of always using the same random numbers in calculations.

Second derivatives

$$
\begin{aligned}
f'(x-h) &\approx \frac{f(x)-f(x-h)}{h} \\
f'(x) &\approx \frac{f(x+h)-f(x)}{h} \\
f''(x) \approx f''(x-h) &\approx \frac{f'(x)-f'(x-h)}{h} \\
&\approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}
\end{aligned}
$$

Somewhat more rigorously, one can remark that this formula for f'' is accurate for quadratics and so, using Taylor's theorem one can give bounds on the error of the approximation.

Antithetic sampling

Let P_1 and P_2 be random variables with $\mathrm{E}(P_1) = \mathrm{E}(P_2)$. We wish to calculate this expectation.
Define

$$Z = \frac{P_1 + P_2}{2}.$$

Then

$$\begin{aligned}
\mathrm{E}(Z) &= \mathrm{E}(P_1) = \mathrm{E}(P_2) \\
\mathrm{Var}(Z) &= \frac{\mathrm{Var}(P_1) + \mathrm{Var}(P_2) + 2\,\mathrm{Cov}(P_1, P_2)}{4}
\end{aligned}$$

So, all other things being equal, if P_1 and P_2 are negatively correlated, the estimate $\mathrm{E}(Z)$ will be improved over the case when P_1 and P_2 are independent.

Monte Carlo with antithetic sampling

- Suppose that in our Monte Carlo pricing algorithm, we generate a random normally distributed vector ϵ and use this to compute a payoff $P_1 = P(S(\epsilon))$. We can use the vector $-\epsilon$ to compute another payoff $P_2 = P(S(-\epsilon))$. Here S is the stock price given ϵ, P then computes the payoff.
- $\text{Cov}(P_1, P_2)$ will be negative for many payoff functions (if ϵ leads to a good payoff, $-\epsilon$ will often be bad)
- So the estimate $\text{E}(Z)$ may be a better estimate than we would have obtained by taking independent samples P_1.
- So by computing the Monte Carlo price using a sample of random numbers ϵ_i and the antithetic sample $-\epsilon_i$ we will get a better estimate than we would obtain by simply doubling the sample size.

Exercises (p1)

- ★ What tests could you write for the `generateBSPaths` function? Write them and check that they pass. Make sure that the tests always pass whatever numbers are generated.

- ★ Use the `generateBSPaths` function to create a plot of the percentiles of the stock price over time as shown in the slides.

- ★ Sketch a graph of the how the price of a knockout option varies as the barrier changes. What is the minimum value and when is it obtained? What is the maximum value and when is it obtained? Explain your answer. Check that your sketch matches the results of the Monte Carlo simulation.

Exercises cont...

★ Price a knock in option and an Asian option. How can you use function passing to improve your code?

★ For our Monte Carlo pricing algorithm we need to generate normally distributed random numbers. Suppose that we do this by generating a uniformly distributed random number and then applying N^{-1}. Show that the Monte Carlo pricing algorithm then becomes equivalent to one of the Monte Carlo integration techniques from the last lecture.

★ Prove the convergence rates for the forward, backward and central estimates for the derivative using Taylor's theorem assuming that f is continuously differentiable to all orders.

★ Refactor the code so that the random numbers used to

Exercises cont...

★ We have stated an approximation formula for the second derivative in terms of the value of f at the points $\{x-h, x, x+h\}$. Assuming f is smooth, compute a bound on the error of this formula.

★ Use the approximation formula for the second derivative to write a function that computes the gamma of a European option using a Monte Carlo method. Check your answer.

★ Numerically compute the derivative of $\sin(x)$ in two different ways and plot a log-log plot of the error in the estimates.

★ Implement Monte Carlo pricing with antithetic sampling for a knockout option.