E-kursuse "Simulation Methods in Financial Mathematics (MTMS.02.038)" materjalid

Aine maht 3 EAP

Raul Kangro, Ants Kaasik (Tartu Ülikool), 2012

Course introduction

Amount of credits: 3 EAP

Lecturer: Raul Kangro (Assoc. prof., Institute of Mathematical statistics, Tartu University)

Target Group: Master students of financial and actuarial mathematics program

Breaf description: This course gives an overview of and practical experiece in the different aspects of applying Monte-Carlo methods for pricing various derivative securities. Several methods of speeding up Monte-Carlo computations are studied

Goals of the course: The goals of the course are

1. To give an overview of the possibilities of using simulation methods (Monte-Carlo methods) for pricing various assets, especially finanancial options
2. To give practical skills in effective application of simulation methods for numerical solutions of various problems of mathematical finance.

Learning outcomes: After completing the course the students

1. Know the principles and practical error estimates of the Monte-Carlo method; are able to generate the trajectories of the solutions of stochastic differential equations (especially stock price trajectories) and know how to use the skill in constructing simulation methods for pricing financial options
2. Know several variance reductions methods for speeding up the Monte-Carlo simulation (antithetic variates, control variables, stratified sampling) and are able to use them in option pricing
3. Know the concept of the quasai-random variables and are able to use them in option pricing.

Topics of the course: Introduction to the R package. Generating random numbers. MC method for numerical computation of expected values of random variables. Numerical evaluation of integrals using MC methoc. Implementing Black-Scholes formulas in R. Simulating the trajectories of stock prices. Implemening Euler's method. The analysis of the convergence rate of the method. Euler's method in the case of a general market model. Milstein's method and a weakly second order method for generating stock prices. Variance reduction methods: antithetic variates and control variates, importance sampling and stratified sampling. Additional points about stratified sampling, using stratified sampling for generating the trajectories of the stock price. Pricing Asian options by MC. Using stratified sampling for pricing Asian options. Quasi-Monte-Carlo methods, Halton points, Sobol points. Computing the option price sensitivities with MC. Using MC for pricing American options

Independent works: There are 8 homeworks giving up to 5 points each. The homeworks are due one week after they were handed out. The late submissions are allowed but the maximal score for such submissions is reduced by 50%. The solutions of

the practical homeworks have to be submitted through the Moodle web page of the course as R files. The maximal total score for homeworks is 40 points.

Requirement to be met for final assessment: At least 20 points (50%) for homeworks is required for qualifying for the final examination.

Composition of the final grade: 60% of the total score is given by the final exam, 40% comes from the homework assignments. The final exam takes place in a computer lab and consists of 3 computational problems related to option pricing. The final grade is determined by the total score as follows: the score less than 50 gives F, from 50 to 59.9 gives E, from 60 to 69.9 gives D, from 70 to 79.9 gives C, from 80 to 89.9 gives B and a score of 90 or more gives A

E-learning activities: The course materials are divided between 16 study weeks and can be used for independent study or as supporting materials for the computer labs. The lab handouts contain all of the theoretical materials that are required for this course. The solutions of the homeworks have to be submitted through Moodle. The final examination has to be taken in person.

Additional information: Raul Kangro raul.kangro@ut.ee

Creation of the web page of the course was supported by European Union

Before you start

1) Remember that everything is CASE SENSITIVE in R. This means that caps make a difference.

2) Also, functions in R mostly work ELEMENTWISE by default! This means that a lot of operations can be accomplished with a brief line of code.

3) Several functions can be used in the same line of code (i.e. output of one function is used as (perhaps partial) input for another) and if needed lines of code can be "joined" by putting a semicolon between them.

4) It is not necessary to leave any spaces anywhere, this is just for display purposes.

5) Character # can be used for commenting the code (i.e. everything following this symbol on a line is ignored by R). Make sure that you do COMMENT YOUR CODE. It is invaluable when you are returning to use your code later on.

Getting Started

`variable <- value` setting a *value* for *variable*, often **=** is used instead

```
a <- 3
b = 9
```

`?function` documentation of a function

```
?sum
```

`apropos("string")` lists all the functions that contain *string* in its name

```
apropos("sum")
```

`install.packages("packagename")` installs the package named *packagename* into the users computer

```
install.packages("gsl")
```

`library("packagename")` loads that package

`library(help="packagename")` opens an overview of the package and the functions in it

Basic Operators

`+ - * / ** ^ %/% %%` binary operators addition, substraction, multiplication, division, raising to a power (two variants), integer division, remainder

```
a * b #27
b ** (1 / 2) #3
b ** 1 / 2 #4.5
a %% b #3
```

Vectors

`c(...)` combines arguments into a vector

```
c(1, 2) #1 2
c(2 * c(a, b), a) #6 18 3
```

`from : to` produces a vector of integers with increment (plus or minus) one

```
a : b  #3 4 5 6 7 8 9
a : -1  #3 2 1 0 -1
1 : 2 + 3  #4 5
```

`seq(from, to, by)` identical but *by* specifies the increment; instead of *by* argument *length.out* can be used to specify the desired length of the sequence

```
seq(a, -1, -2) #3 1 -1
seq(a, -1, -3) #3 0
seq(a, -1, length.out=5) #3  2  1  0 -1
```

`rep(x, times)` replicate argument *x times* times; *each* can be used to replicate each element of *x each* times

```
rep(c(2, 5), times=2) #2 5 2 5
rep(c(2, 5), each=4) #2 2 2 2 5 5 5 5
```

Math & Stat

`length(x)` number of elements in argument *x*

`abs(x)` absolute value of elements of argument *x*

`max(...)` maximal element of all elements in the arguments

```
max(a, 6) #6
max(c(a, b), 6) #9
```

`min(...)` minimal element of all elements in the arguments

`sum(...)` sum of all elements in the arguments

`prod(...)` product of all elements in the arguments

`log(x)` natural logarithm of elements of argument *x*; argument *base* can be used to set a different base

```
log(9, base=3) #2
```

`exp(x)` exponent of elements of argument *x*

`mean(x)` arithmetic mean of elements of argument *x*

```
mean(c(a, b)) #6
```

`sd(x)` standard deviation of elements of argument *x*

```
var(c(a, b)) #18
```

`var(x)` variance of argument *x*

`cor(x, y)` correlation between vectors *x* and *y*

`cov(x, y)` covariance between vectors *x* and *y*

`round(x, digits)` round the elements of *x* to the number of decimal places specified by *digits*

`pmax(...)` positionwise maxima

```
pmax(c(4, 5), c(a, b)) #4 9
```

`pmin(...)` positionwise minima

`cumsum(x)` cumulative sum of the elements of argument *x*

```
cumsum(c(a, b, a)) #4 13 17
```

`cumprod(x)` cumulative product of the elements of argument *x*

`cummax(x)` cumulative maximum of the elements of argument *x*

```
cummax(c(a, b, a)) #4 9 9
```

`cummin(x)` cumulative minimum of the elements of argument *x*

Matrices

`matrix(x, nrow, ncol)` creating a matrix with *nrow* rows and *ncol* columns from elements of *x* by first filling the first column (top to bottom), then the second, etc

```
matrix(c(2, 3), nrow=2, ncol=2)  #2 2
                                 #3 3
```

`diag(x)` forms a diagonal matrix with elements of *x* on the main diagonal; argument *nrow* can still be used

`%*%` binary operator for matrix multiplication

`t(x)` transposes argument *x*

`solve(x)` inverse of the square matrix *x*

`dim(x)` dimensions of argument *x* i.e. number of rows and columns for a matrix*;* returns *NULL* for vectors

```
dim(diag(2, nrow=2))  #2 2
```

`rowSums(x)` sum the elements of *x* by rows

`colSums(x)` sum the elements of *x* by columns

`cbind(...)` combine arguments side-by-side

```
cbind(c(2, 3), c(4, 5), c(a, b)) #2 4 3
                                 #3 5 9
```

`rbind(...)` combine arguments top to bottom

```
rbind(c(2, 3), c(4, 5))    #2 3
                           #4 5
```

`apply(X,MARGIN,FUN)` apply function *FUN* to the object *X* by rows if *MARGIN=1* or by columns if *MARGIN=2*

Logicals

== >= > < <= != %in% binary operators equal to, greater than or equal to, greater than, less than, less than or equal to, not equal to, and is contained in producing logical objects consisting of *TRUE* and/or *FALSE*

```
a > b  #FALSE
a + 6 == b #TRUE
a != b #TRUE
c(a, 2) %in% c(2, b, 1) #FALSE TRUE
```

! unary operator for negating the logical object

```
!c(FALSE, TRUE) #TRUE FALSE
```

& | binary operators AND and OR for combining logical objects

```
a == 3 | b == 3 #TRUE
```

Probability distributions

rnorm(n) random generation of *n* numbers from a standard normal (Gaussian) distribution; arguments *mean* and *sd* can be used to specify distribution parameters

```
set.seed(16)
rnorm(2) #1.147829 -0.468412
rnorm(2, mean=3) #4.096216 1.555771
```

dnorm(x) value of the probability density function (pdf) of a standard normal distribution at elements of *x*; arguments *mean* and *sd* can be used to specify distribution parameters

pnorm(q) value of the cumulative distribution function (cdf) of a standard normal distribution at elements of *q*; arguments *mean* and *sd* can be used to specify distribution parameters

```
pnorm(0) #0.5
```

qnorm(p) value of the quantile function i.e. inverse cdf of a standard normal distribution at elements of *q*; arguments *mean* and *sd* can be used to specify distribution parameters

```
qnorm(0.95) #1.644854
```

Similar functions exist for many other distributions e.g. **runif** generates uniformly distributed random numbers (from the unit interval) and **rexp** exponentially distributed random numbers; arguments for specifying the parameters have different names

Data extraction

```
x = 2 : 10
y = 9 : 1
z = c(5, 2, 3)
```

```
y[3] #7                          element at a specific position
z[-2] #5 3                       all elements except one at a specific position
y[1 : 3] #9 8 7                  elements at specific positions
y[-(1 : 7)] #2 1                 all elements except those at specific positions
x[c(2, 4, 6)] #3 5 7             elements at specific positions
z[c(TRUE, FALSE, TRUE)] #5 3     elements at positions TRUE
x[y > 4] #2 3 4 5 6              elements at positions TRUE
x[x > 3 & x < 5] #4              elements at positions TRUE
```

```
m = rbind(x, y)
```

```
m[2, 3] #7                       element at a specific position
m[1, ] #2 3 4 5 6 7 8 9 10       specified row
m[ , 2] #3 8                     specified column
m[-1, 1] #9                      specified sub-matrix
m[1, c(1, 3)] #2 4               specified sub-matrix
```

Plotting

`plot(x, y)` plot the points coordinates of which are defined by the vectors *x* and *y* elementwise; argument *type* can be used to change the plotting style e.g. *"p"* for points, *"l"* for lines, *"o"* for overplotted points and lines

```
plot(x, y, type="l")
```

`matplot(x, y)` plot the series of points coordinates of which are defined by the matrices *x* and *y* elementwise with series in columns; if only one matrix is given then this is assumed to be *y*; *type* can be used as before

```
matplot(m, type="l")
x1 = (1 : 200) / 100; y1 = matrix(0, length(x1), 2)
y1[ , 1] = cos(x); y1[ , 2] = sin(x); matplot(x1, y1, type="l")
```

Programming

`function(arglist){expressions}` function definition; functions are usually named and not used "on the spot"; giving default values to arguments allows execution of a function without specifying values for all the arguments; values can be arbitrary (e.g. function names); when the function body consists of a single expression then the curly braces can be omitted

```
mltpl_tbl = function(x=1:9, y=1:9){x %*% t(y)}
mltpl_tbl(1:2, 1:2)    #1 2
                       #2 4
univ_func = function(obj, f) f(obj)
```

`if(condition){expressions}` *expressions* are executed only if *condition* is *TRUE*

```
if(runif(1) > runif(1)) print("first was bigger")
```

`if(condition){expressions1}else{expressions2}` if *condition* is TRUE then *expressions1* are executed, if not then *expressions2*

```
if(runif(1) > runif(1)){
      print("first was bigger")
      }else{
      print("second was bigger")
}
```

`for(variable in sequence){expressions}` a cycle where a *variable* takes the value of the first element of the *sequence* and *expressions* are then executed, then the *variable* takes the value of the second element of the *sequence* and *expressions* are again executed; this continues until the *sequence* is exhausted or *expressions* cause the cycle to end prematurely

```
N = 1000
n = 500
meanv = rep(0, N)
for(i in 1 : N){
      meanv[i] = mean(runif(n))
}
var(meanv) #should be approximately 1/(12*n)
```

***`while(condition)`*`{expressions}`** a cycle where *expressions* are executed if the *condition* is *TRUE* initially; then the *condition* is re-checked and if it is still true then *expressions* are executed again; this continues until the *condition* is *FALSE* or *expressions* cause the cycle to end prematurely

```
init = 1:10
summand = init
while(max(summand) > 0.00001){
      summand = summand / 2
      init=init+summand
}
```

An introduction to R language

The book addresses some topics of R language that are relevant to the course

Table of Contents

How to use the book

When reading the book, it is good idea to have the R program also running on your computer. When an example is given in the text, try it immediately out by typing (or copying) it to the R console window. If a command defines a variable, then you can see the value of that variable by just typing the name of the variable in the command window and pressing the return (or enter) key.

After finishing reading a topic, please also try to find answers to the questions at the end of each chapter. Finally, look also at the answers given in the subchapter Sample answers. If your answer does not appear among the answers given, then it does not mean that your answer is wrong. As long as it produces the required result, it is correct.

Creating vectors

(Numeric) vectors are collections of numbers

There are many ways to create a vector in R:

1) just join several numbers or existing vectors together by **c()** function.
For example,
`x=c(3.4,5)`
creates a vector x with two elements.
`y=c(x,7)`
creates a vector y with elements 3.4,5,7. You can use as arguments for c other commands that produce vectors, like
`z=c(0,rnorm(5))`
which produces a vector starting with 0 and then containing 5 random numbers from the standard normal distribution.
Remark: actually it is possible to use c() to join together things of different types, like `z=c(1,"hallo",sin)`, where the last is a function name, but we are not going to use those more exotic possibilities in the course.

2) creating a vector of constant values (zeros, ones or some other values) with the function rep(value,n):
`x=rep(0,10)`
creates a vector containing 10 zeros

3) a vector of consecutive integers can be formed by the construction n1:n2, where n1 and n2 are integers. The result is a vector containing all having n1 as the first element, n2 as the last element and containing all integers between those values. For example
`x=5:10`
produces a vector x with elements 5,6,7,8,9,10 and the command
`y=3:0`
produces a vector y with elements 3,2,1,0.

4) There is a command **seq()** for producing linearly spaced (meaning with constant distance between consequtive elements) vectors. The most useful forms are `seq(from, to, by=stepsize)` which gives a seqence starting with the value from and increments each next element by the value stepsize, and `seq(from, to, length.out=n)`, which produces n equally spaced numbers starting from the value from and ending with value to (so the step size is (to-from)/(n-1)). Examples:
`x=seq(1.5,6,by=2)`
produces a vector with values 1.5,3.5,5.5 and
`y=seq(0,1,length.out=5)`
produces a vector with values 0,0.25,0.5,0.75,1

5) by using any function that returns a vector as a result, for example
`x=runif(10)`
produces a vector of 10 uniformly distributed random variables from the interval [0,1]

There is one type of vectors that we will be using apart from numeric ones (even if this use is perhaps implicit) -- logical vectors. These have TRUE and FALSE as its possible elements. These can be abbreviated as T and F. Creation of such vectors is still straightforward. For example
`x=c(T,F,F)`
creates one. We will see in the next chapter why logical vectors can be very useful.

Questions:

1. Write a command that creates a vector x with elements 10,0,1.5
2. Write a command that creates a vector x starting with 5 ones and then 16 twos without writing explicitly out all of the elements
3. Write a command that creates a vector y with values 10,9,8,...,0
4. Write a command that produces a vector x with values 1,1.1,1.2,...,3
5. Write a command that produces 101 equally spaced number between -1 and 1 (including -1 and 1).

Sample answers

1. x=c(10,0,1.5)
2. x=c(rep(1,5),rep(2,16))
3. y=10:0 or y=seq(10,0,by=-1) or y=seq(10,0,length.out=11)
4. y=seq(1,3,by=0.1) or y=seq(1,3,length.out=21). Later we see, that a possible command is also y=1+(0:20)*0.1
5. seq(-1,1,length.out=101) or seq(-1,1,by=0.02), it is also possible to write -1+(0:100)*0.2

Manipulating vectors

When we have created vectors our aim is typically to make use of them. Many functions can be used.

1) + - * / ** ^ %/% %% are examples of binary operators (performing addition, substraction, multiplication, division, raising to a power (two variants), integer division, integer division remainder, respectively). They all operate elementwise as do most of the functions in R.
For example after creating
`x=c(3.4,5)`
`y=c(1,7)`
we can write
`x+y`
to get a vector with elements 4.4,12 as a result. This suggests that the vectors that we are going to use should have the same length. This is not necessarily true as R recycles elements if the vectors are not of equal length. This is why
`2*x`
produces vector 6.8,10 as a result.
Remark: Recycling will take place whenever the lengths don't match but it makes sense only when longer vector length is a multiple of shorter vector length, otherwise a warning message is produced.

2)Examples of (elementwise) operators producing logical vectors are **== >= > < <= != %in%**(respectively equal to, greater than or equal to, greater than, less than, less than or equal to, not equal to, and is contained in). For example
x>y
returns T,F as the first element of x is greater than the first element of y but the second element of xis not greater than the second element of y. Command
c(x,2) %in% c(2,y,1)
returns F,F,T as neither of the elements in x is contained among the elements of the second vector but the third element (2) is. Logical vectors can further be combined with **&** and **|** standing for logical AND and OR (and operating elementwise). Thus
x>y|y>x
produces T,T as the elements in respective positions are different.
It is also interesting that logical vectors can be used in arithmetic operations. R simply translates T to 1 and F to 0.

3) Some functions use all the elements in the input vector to produce a single number (one-element vector) as a result. For example
`mean(x)`
produces 4.2.

4)We might need just some elements of a vector. This is accomplished with square brackets. Let us define
x=*2:10*
y=9:1
z=c(5,2,3)
and then we can

1. ask for the third element in vector y with `y[3]`
2. ask for all the elements in vector z apart from the second element with *z[-2]*
3. ask for the first three elements in y with *y[1:3]*
4. ask for the last four elements in y with *y[-(1:5)]*
5. ask for the second, fourth and sixth element from x with *x[c(2,4,6)]*
6. ask for specific elemnts from z by specifying the required elements by a logical vector like *z[c(TRUE,FALSE,TRUE)]*
7. ask for specific elemnts from x by specifying the required elements by a logical vector which is created using a condition like *x[y>4]* or *x[x>3 & x<5]*
8. write an index after any expression that produces a vector, for example `(x-y)[2]` and `sqrt(x)[3]` are valid usages of indeces.

Remark: positions in a vector can also have names and then these names can be used for extracting data from the vector but we will not be using this option in the course.

5) Replacing elements in a vector is possible by first selecting the elements that need replacing (as shown previously) and then writing the replacements on the other side of the equality sign. For example

```
y[1:5]=c(1,3,11,2,9)
```

replaces the first five elements in vector y by 1,3,11,2,9 respectively.

Questions:

1. Write a command that creates a vector x with elements 2,4,8,16,32,64
2. Write a command that reverses the order of that vector x so that it becomes 64,32,16,8,4,2
3. Write a command that selects elements from x that are multiples of 16.
4. Look at the the function reference sheets and find the function for rounding numbers. Write commands that generate two vectors x1 and x2, each with 10 random numbers from the standard normal distribution and then rounds the elements of these vectors to integers. Write a command that finds how many of the numbers in the first rounded vector are contained in the second rounded vector.
5. Generate 100 random numbers from the interval [0,1]. Find the element that causes the cumulative sum of these numbers to get larger than 20. Hint: use the function reference sheets to find a suitable function.

Sample answers

1. x=2**(1:6)
2. x=x[6:1] or x=rev(x)
3. x[x%%16==0]
4. x1=round(rnorm(10)); x2=round(rnorm(10)); sum(x1 %in% x2)
5. x=runif(100); x[cumsum(x)>20][1] or x[min(which(cumsum(x)>20))]

Functions

Every (valid) command that you use in R makes use of a function. Sometimes this might be implicit but it is actually always so. You have already seen many useful functions that are predefined and many others are available in different R packages. However, throughout the course it will be very useful to write several functions of your own.

1) How do we define a function?
Typically with a command similar to this
`function_name=function(arguments_list){expressions}`
So a function really is comprised of three parts: it's name, it's arguments and the function body (this is the main part that is inside the curly brackets).

Let us consider an example. Suppose we want to write a function that returns the n largest elements in decreasing order from a vector. But this n is not fixed in advance.

1. So we first decide on its name and think it should be topn.
2. Secondly we think about the arguments that the function needs. Obviously the vector from which the large elements are sought from. So suppose we call this vector x. And also the amount n.
3. Now the function body. In the function body we must use the input arguments with the names that we gave them in the previous part. We can find that R has a predefined function sort that we can use to make our work easier.

So our function can be defined as follows:
```
topn=function(x,n){
 sort(x,decreasing=T)[1:n]
}
```

2) How do we use this function?
We use it as we have seen previously. We just must remember what kind of arguments the function is expecting. The first one must be a vector and the second one a positive integer.
Remark: You may notice that we may provide arguments that do not make sense. For example n might be larger than the number of elements in x. Of course our function will not work properly with such arguments.

So suppose we first generate 100 random numbers from a standard normal distribution and store them.
`a=rnorm(100)`
Now we ask for the top ten by writing
`topn(a,10)`

3) How does our function work?
We can imagine what goes on. The function first sets
`x=a`
`n=10`
because these were the input values provided. Then it executes the expressions in the function body. There is just a single command there.
`sort(x,decreasing=T)[1:n]`
The result of this command is returned as output.

4) Why can't I just write `sort(rnorm(100),decreasing=T)[1:10]`?
You could, but imagine that you want to perform this operation several times (and the number of times is not known in advance) and each time the number of elements in the input vector is randomly generated and the size of the top (value of n) is also not fixed. Using the function topn meakes the code shorter and easier to uderstand.

5) Give me another example of defining and using a function!
Ok, lets take a more complicated example. Imagine now that we want to find the smallest and largest number in a vector that exceed some number which is not fixed in advance. So we define

```
sl=function(x,y=0){
 x1=x[x>y]
 large=max(x1)
 small=min(x1)
 return(c(small,large))
}
```

Note some differences:

1. Argument y has a default value
2. New objects are created inside the function body
3. Function body ends with expression making use of the return command

So?

1. Default values allow us to not provide input values when executing the function (so if we are happy with y=0 we must only provide a value for x)
2. These (temporary) variables (x1, large, small) are only accesible inside the function and when the function has terminated you will not be able to use them anymore (using those names x1, large, small).
3. When the function body contains several expressions then it is customary to write explicitly what the function should return as output. If we do not use return then the value of the last expression inside the function body is returned. This is why the function topn could be defined as it was (without return).

6) What can be used as arguments of a function?
Anything. You just have to keep in mind what kind of object is meant to be given for an argument when writing the body of the function and use the name of the argument accordingly. During this course it is often useful to write functions that take names of other functions as arguments. As an example, suppose we want to investigate the behavior of the top n values of random variables from a probability distributions for various distributions. Let the arguments of the function be N (the value of random values generated), n (the number of largest values to select) and gen (the name of a function that for a given value of n generates n random numbers). By using our earlier function topn we get write the following code:

```
topn_rand=function(N,n,gen){
 x=gen(N)#note that here gen is assumed to be the name of a function of one variable
 return(topn(x,n))
}
```

and use the function like this:

```
topn_rand(100,10,runif) #10 largest uniformly distributed random numbers out of 100
topn_rand(100,10,rnorm) #the same for normally distributed random numbers
```

Questions:

1. What would the function `sl` do when we would totally omit `return` from the the last line in its body (i.e. the last line would be `c(small,large)`)?
2. Write a function *meanse* that gets a vector as an input and returns the mean and standard error of the mean as output. For computing the standard error of the mean, assume that the vector consists of values of independent and identically distributed random variables.
3. Perform a test that makes use of this function to check whether the mean of 10 000 random numbers from a standard normal distribution is closer to zero than twice the value of the standard error of the mean. Provide the code for the test.
4. What should you change in the code of the function `topn` if you would like to select whether you want the largest or the smallest elements when using the function? Provide the new code.
5. Suppose we want to compare two vectors and out of those elements in the first vector that are larger than their respective counterparts in the second vector find the smallest and largest elements. Write the code of a function that would do this.

Sample answers

1. It would still do what it did previously as the output from the last line (of the function body) would be returned if there is no return statement in the function body.
2. meanse=function(x){c(mean(x),sd(x)/sqrt(length(x)))}
3. result=meanse(rnorm(10000)); abs(result[1])<2*result[2]
4. topbottomn=function(x,n,top=T){sort(x,decreasing=top)[1:n]} where we have set largest elements as the default value
5. If you carefully consider what the function sl does then you realize that it will do exactly that if you provide the two vectors as input.

Programming basics

Simulation is our main activity throughout this course and this usally means performing some particular step for a large number of times. Also, sometimes the exact value of this large number depends on the (stochastic) results that we have obtained previously. We thus need to know how to accomplish this with code. We look at 4 important constructs.

1) if(condition){expressions} means that expressions are executed only if condition is TRUE. So with

```
if(runif(1)>runif(1)) print("first was bigger")
```

first two random numbers from the unit interval are generated and then if the first one is larger than the second one then a message is printed. If this is not the case then the expressions are not evaluated at all.
Remark: as with functions, when there is a single expression, curly braces are not needed.

2) if(condition){expressions1}else{expressions2} is an extension of this, where depending on whether the condition is TRUE or FALSE different expressions get evaluated. So with

```
if(runif(1)>runif(1)){
   print("first was bigger")
   }else{
   print("second was bigger")
}
```

again two random numbers from the unit interval are generated and then if the first one is larger than the second one then a message is printed. However, if this is not the case then the other message is printed.
Remark: Negation of "first was bigger" in our case is of course "second was bigger or equal" not simply "second was bigger" but equality of two random number from a continuous distribution is out of the question.
Remark: It is still possible to omit curly braces with just a single expression but then make sure that the whole construct is on the same line. With the curly braces as above it is also important to not start a new line of code with else as otherwise R will think that there is no else and only evaluate the if part.
Remark: Of course the body of else (i.e. expressions2) may contain another if or if...else construct.

3) for(variable in sequence){expressions} is an example of a cycle. The basic idea is that variable takes the value of the first element of the sequence and expressions are then executed, then the variable takes the value of the second element of the sequence and expressions are again executed; this continues until the sequence is exhausted. Thus

```
N=1000
n=500
meanv=rep(0,N)
for(i in 1:N){
   meanv[i] = mean(runif(n))
}
```

first produces three objects and then begins the cycle. As a first step variable i gets value 1 (first value in the sequence). In the body of the cycle the first element of vector meanv is changed. Now variable i gets value 2 (second value in the sequence) and in the body of the cycle the second element of vector meanv is changed. This cycle continues till i=1000 (last value in the sequence) and in the last step the last element of meanv is changed. Then the cycle terminates as i has received its last value.
Remark: It is best not to use cycles in R unless absolutely necessary as it is usally possible to produce a code that executes much faster when no (explicit) cycles are present.

4) while(condition){expressions} is a different kind of cycle where expressions are executed if the condition is TRUE initially; then the condition is re-checked and if it is still true then expressions are executed again; this continues until the condition is FALSE. Basically this looks ver much like a for cycle presented previously but the difference is that the number of times the cycle body is run is not predetermined. So

```
init=1:10
summand=init
while(max(summand)>0.00001){
   summand=summand/2
```

```
   init=init+summand
}
```
after the two variables have been defined a cycle starts. As the maximum value of the summand is 10 the expressions are evaluated and all the elements of summand are divided by 2 (it is then 0.5,1,1.5,2,2.5,3,3.5,4,4.5,5). The value of init is increased by the new value of summand. Now the maximum value of the summand is 5 and the expressions are evaluated again -- the elements of summand are halved again and these new values are added to init. Eventually even the last (which is the largest) element of summand is not greater than 0.00001 and then the cycle is terminated (i.e. the condition is no longer TRUE and the expressions are no longer evaluated).

Remark: Also for for and while cycles curly braces are not obligatory when a single expression is in the body of the cycle.

Typically constructs presented are combined -- e.g. the body of a for cycle may contain an if construct.

Questions:

1. In the previous chapter you constructed a function `meanse` and performed a test with this function. Now write code that performs this test 100 times and stores the test results in a vector.
2. In the manipulating vectors chapter you generated 100 random numbers from the interval [0,1] and found the element that caused the cumulative sum of these numbers to get larger than 20. Now accomplish this with code that uses a for cycle and does not generate more random numbers than necessary.
3. And now write a similar code that uses a while cycle.
4. When you compare the execution time of the three possible codes then you probably won't see much of a difference because they all execute quite fast. However can you think of a situation where the first two solutions won't work properly.

Sample answers

1. reps=100
   test=rep(0,reps)
   for (i in 1:reps){result=meanse(rnorm(10000)); test[i]=(abs(result[1])<2*result[2])}
   test
2. rsum=0
   a=0
   for (i in 1:100){
     if (rsum<=20){
       a=runif(1)
       rsum=rsum+a
     }
   }
   a
   #remark: it is possible to use the break command to stop computations of a for cycle, so we could add `else{break}` for the if command to avoid unneccesary computations.
3. rsum=0
   a=0
   while (rsum<=20){
       a=runif(1)
       rsum=rsum+a
   }
   a
4. Even if highly improbable it is still possible that the sum of the first 100 numbers from the unit interval [0,1] is not greater than 20. Thus, if our aim is to find the first value that couses the cumulative sum to be larger than 20 without any bound for the number of random variables to be generated, the only absolutely correct solution is the one with while.

Matrices

We have already learned how to use vectors but sometimes it can be more convenient to arrange the data into matrices i.e. have several vectors next to each other.

1) To create a matrix function **matrix** can be used. We just need to provide the elements (in a vector) for the matrix and also inform R about the number of rows and columns the matrix has (i.e. set matrix dimensions). For example
`m1=matrix(c(2,3,2,1), nrow=2, ncol=2)`
produces a 2x2 square matrix where the first column is 2,3 and the second column 2,1.
Remark: Recycling takes place when the provided vector does not have enough elements (but is of a suitable length) so giving just a single element will work.

2) We can put vectors side-by-side or top to bottom to produce matrices. The respective functions are **cbind** and **rbind**. So
`cbind(c(2,3),c(2,1))`
and
`rbind(c(2,2),c(3,1))`
will also produce the matrix that was produced at the previous step. Input of those functions can also be a matrix.

Remark: Of course matrix can also be an output of some function so this is yet another example of producing a matrix.

3) What might we do with a matrix? We could add or subtract or multiply or divide or comapre them elementwise as we did with vectors using the same operators as long as the matrices have matching dimensions.
However more elaborate things are binary operator **%*%** for matrix multiplication, function **t** for transposing a matrix and function **solve** for inverting a matrix. So, for example,
`m1%*%m1`
produces a 2x2 matrix with first column 10,9 and second column 6,7 and
`solve(m1)`
produces a 2x2 matrix with first column -0.25,0.75 and second column 0.5,-0.5.
Remark: Transposing a vector produces a matrix with one row

4) Function **apply** is also noteworthy as it allows us to apply some function to the matrix by rows or by columns. First argument is the matrix, second is typically 1 or 2 specifying whether the function should be applied by columns or by rows and the thrid argument is the function to use. So
`apply(m1,2,prod)`
produces a vector 6,2 as the first element is the product of the elemnts in the first column and the second is the product of the elements in the second column.

5) The principle of extracting elements from a matrix is a generalisation of the concept of extracting elements from a vector and again makes use of the square brackets. The difference is that now there are two positions inside the brackets -- first for rows and the other for columns. Let us first define
`m2=cbind(m1,m1)`
and then we can

1. ask for the element in the 2nd row and 3rd column with `m2[2,3]`
2. ask for all the elements in the first row with `m2[1,]`
3. ask for all the elements in the 4th column with `m2[,4]`
4. ask for all the elements in the 1st colum except the one in the first row with `m2[-1, 1]`
5. ask for the 1st and 3rd elements in the 1st row with `m2[1, c(1, 3)]`
6. ask for the submatrix that consists of rows 1 and 2 and columns 2 and 3 with `m2[c(1,2),c(2,3)]`
7. ask for all the elements in the matrix that are greater than 1 with `m2[m2>1]`

Remark: As we see from example 7, it is possible to use a logical matrix to specify the positions from which we

want to extract the elements. However, as this does not follow the usual square brackets extraction style for matrices (there is no comma inside), both the logical matrix and the matrix that we are extracting the data from are first converted into a vector (columns are stacked under each other from left to right) and the output will also be a vector (even if it could theoretically have a matrix layout).

Questions:

1. Make a 5x3 matrix X with columns 1:5, 2:6 and 6:2. Now use X to make another matrix Y)with dimensions 5x2) that has the mean of the first two columns of X as its first column and the third column of X as its second column.
2. Now form a 3x2 matrix D with columns 0.5,0.5,0 and 0,0,1 and use X and D to produce Y.
3. Produce a sequence with 1:9 and then form a "multiplication table" based on this sequence. Hint: use matrix multiplication but make sure one of the arguments is a matrix as otherwise it returns the inner product.
4. Write the code that first generates 10000 random numbers from the unit interval [0,1], arranges them into a 100x100 matrix and then calculates the geometric means of each row and column (i.e. 200 geometric means in total) and finally the arithmetic mean of these 200 values.

Sample answers

1. X=matrix(c(1:5,2:6,6:2),nrow=5); Y=cbind((X[,1]+X[,2])/2,X[,3])
2. D=matrix(c(0.5,0.5,0,0,0,1),nrow=3); X%*%D
3. a=1:9; a%*%t(a)
4. mat=matrix(runif(10000),nrow=100); mean(c(apply(mat,1,prod)**(1/100),apply(mat,2,prod)**(1/100)))
   or
   mat=log(matrix(runif(10000),nrow=100)); mean(c(exp(rowMeans(mat)),exp(colMeans(mat))))

Efficient programming

Now that we have covered the basics (i.e the essential knowledge without which it is impossible to complete the course) we switch our attention to advanced subjects that make life easier. First, we cover some tips that help us to arrive at an answer more quickly.

1) Use **vectorized operations** whenever possible. A lot of functions can act on vectors (elementwise). Many tasks can be accomplished with a cycle but this is mostly inefficient. This typically remains true even if we might do some extra work.

Suppose we have to generate 10 000 numbers from a standard normal distribution and divide by 2 the positive numbers and multiply by 2 the negative numbers. This can of course be accomplished with a cycle that has an if else construct inside. However it is much more efficent to do it differently:

```
a=rnorm(10000)
a[a>0]=a[a>0]/2
a[a<0]=a[a<0]*2
```

or

```
a*((a<0)*2+(a>0)/2)
```

where the second option works even if we would have to change the sign or do something similar that could cause problems in the first one.
Remark: Only very rarely does it make sense to use a cycle in R when it is possible to vectorize. One of these cases is when we would otherwise run out of memory.
Remark: When the situation is tight with memory it might be worthwhile to erase/overwrite all the objects that are no longer used in the code that follows. Function *rm* can be used for deleting objects.

2) Typically **built-in R functions** operate much quicker than those that are manually added. However, the efficiency might be lost if the built-in function actually does what we need but also a lot more (i.e performs some tasks that are not needed by us). Sometimes functions operate differently depending on the type of the input.

Suppose we have to generate 1000 random numbers and calculate the standard deviation. Then we have to do it 999 more times. We can simply write.

```
sd(matrix(rnorm(1000000),nrow=1000))
```

3) Growing objects is very slow.

Suppose we have to generate random numbers from a normal distribution, standard deviation of which is random and determined by an exponential distribution. We stop when our sample includes an element that has an absolute value of at least 30 but have to return the full sample. Seems that we cannot avoid a cycle but at least we should not grow our sample vector too many times.

```
step=100000
samp=rnorm(step,sd=rexp(step))
smax=max(abs(samp))
while (smax<30){
new=rnorm(step,sd=rexp(step))
samp=c(samp,new)
smax=max(abs(new))
}
```

Remark: If no extra elements are allowed it is possible to fix where the maximum occurs in the while cycle and then cut off the tail of the sample after the cycle has finished.

4) When it seems that a cycle is needed there might still be a way to avoid it (at least explicitly). Sometimes the *apply* function can be an alternative.
Remark: If you really want to avoid cycles then functions *sapply*, *mapply* and *replicate* might help. If we still have to use a cycle constructs *next* and *break* might speed up the cycle.

Questions:

1. Write code that will generate a vector y based on vector x. When x[i]<1 then y[i] should be -1 and it should be 1 otherwise.
2. Simulate numbers from the standard normal distribution until three sample elements have exceeded 5. Present the sample elements that are less than -3.
3. Find the median for each of the 1000 samples, each containing 1000 random numbers from the standard exponential distribution. Plot the histogram of sample medians.

Sample answers

We first might get some estimate of how long this might take. 1-pnorm(5) shows that a sample element exceeds 5 about once in 4 million attempts. Thus 2 (or more) in 1 million is not typical and we are willing to throw some numbers away if it were to happen.

```
sample=NULL
check=0
large=5
small=(-3)
step=1000000
while (check<3){
samp=rnorm(step)
big=which(samp>large)
if (length(big)==0) {sample=c(sample,samp[samp
```

Simulation Methods in Financial Mathematics
Computer Lab 1

Goals of the lab:

- To familiarize yourself with R
- To learn to use the Monte-Carlo method and estimate its error
- To learn to calculate integrals using simulation methods

Monte-Carlo method or simulation method is a computational algo rithm for estimating the mean of a random variable. The method is based on performing indep endent trials and averaging these results. That is, if we are interested in $\mathbb{E}Y$ for some random variable Y, we generate n independent values $Y_1, Y_2, \ldots, Y_n$ from the distribution of Y and estimate

$$\mathbb{E}Y \approx H_n = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

It is important to understand, that for a given random variable Y the expected value $\mathbb{E}Y$ is a constant (non-random real number) but the the approximatio n H_n computed by MC method is a random number, so the error of the computed result is also random. So, when using MC method, we can never be completely sure that the error of the r esult we get is as small as we want but we can make the probability of getting a res ult with a large error very small.

The estimation of the error of H_n can be based on the Central Limit Theorem: for large enough n we have that the error of H_n (i.e. $|H_n - \mathbb{E}Y|$) is less than

$$\varepsilon = -\frac{\Phi^{-1}(\alpha/2)\sigma_Y}{\sqrt{n}}$$

with approximate probability $1-\alpha$. Here Φ denotes the cumulative distribution function of the standard normal distribution. The inverse of a cumulativ e distribution function is called the quantile function of that distribution, so actually t he quantile function of the standard normal distribution is used in the error estimate. The st andard deviation σ_Y of the random variable Y is also estimated by using $Y_1, \ldots, Y_n$.

In most cases relevant Y can be expressed as $Y = g(X)$, where X is a random variable (or random vector) with known distribution and g is some given function. In this case we generate values of Y by applying the function g to the generated values of X.

Tasks:

1. We start using the MC method. Let $Y = X^2$, where X has the standard uniform distribution. Using the sample of size $n = 1000$ find an estimate of $\mathbb{E}Y$, calculate the error estimate for $\alpha = 0.1$ and the actual error.

2. Let us write our first useful function for applying MC method in ma ny different situations. Namely, define the function `MC1` with four arguments: the name of a function g, the name of a function that for a given n generates n random variables X, the number n of random variables to be generated, and the value of α used in computing the error estimate. The function should return a vector of t wo numbers; the estimat of $\mathbb{E}[g(X)$ and the error estimate. Additionally, define the function $f(x) = x^2$ and compute the value $MC1(f, runif, 100, 0.1)$. Is the result correct?

3. Use the function $MC1$ to repeat the first task 100 times and produce three vectors: average, error_estimate, actual_error. How many times did the actual error exceed the error estimate?

4. When using the MC method, we usually do not know beforehand how large a sample should be generated in order to get an answer that is accurate enough for our purposes. Thus simulation continues until the required precision is achieved (or in some cases until we cannot wait any longer). In order to do that we first set the number of random variables to be generated at one go and after generating this number of values we estimate the error. If the error is not small enough we repeat the generation process and estimate the error again by using all generated values. Since the number of generations needed for achieving the desired accuracy can be very large, we do not store the previously generated random variables (to avoid memory problems), and thus we cannot use the R functions to calculate the mean and standard deviation of the sample. Instead we store only the sum, the sum of squares and the total number of values of Y generated so far. The standard deviation can then be estimated as

$$\sigma_Y \approx \sqrt{\frac{|sum\_of\_squares\_of\_y - (sum\_of\_y)^2/n|}{n-1}}.$$

Write a function which takes as the input a function g, a function gen which generates values from the distribution of X, allowed error ε and α – the probability of exceeding the allowed error and would return the estimate (with given precision with probability $1-\alpha$) of the expected value of $Y = g(X)$.

5. **Homework** (Deadline 16.02.2012) Definite integrals

$$\int_a^b g(x)\,dx$$

can be viewed as expected values of a function by multiplying and dividing the integrand by a suitable probability density function:

$$\int_a^b g(x)\,dx = \int_a^b \frac{g(x)}{f_X(x)}\,f_X(x)\,ds = \mathbb{E}(\tilde{g}(X)),$$

where X is a random variable with the probability density function f_X such that $f_X(x) > 0,\ x \in [a,b]$ and

$$\tilde{g}(x) = \frac{g(x)}{f_X(x)} I_{[a,b]}(x).$$

Here $I_{[a,b]}(x)$ is the indicator function of the set $[a,b]$ having value 1 when x belongs to that interval and 0 otherwise and it is not needed if the density f_X or the function g is constantly zero outside of the interval $[a,b]$. So there are many ways to compute by MC the same definite integral (different choices of the random variable X give different methods with different convergence properties).

Use the Monte-Carlo method to calculate with precision $\varepsilon = 0.01$ (with probability $\alpha = 0.05$) the integrals

$$\int_{-1}^{2} \sqrt{1+x^2}\,dx$$

and

$$\int_3^{100} e^{-x^{1.5}+4}\,dx.$$

Hint: the indicator function $I_{[a,b]}(x)$ can be written in R as $(x >= a) * (x <= b)$.

Simulation Methods in Financial Mathematics
Computer Lab 2

Goals of the lab:

- To familiarize yourself with Brownian motion and the Black-Scholes model of the market
- To program the Black-Scholes call and put option pricing formulas
- To understand that the prices of options can be calculated as expected values

A standard (also called vanilla) option is a contract written by a seller that conveys to the buyer the right – but not the obligation – to buy (in the case of a call option) or to sell (in the case of a put option) in a future a particular asset, such as a piece of property, or shares of stock or some other underlying security, such as, among others, a futures contract, for the price specified in the contract. In return for granting the option, the seller collects a payment (the premium) from the buyer. We will deal with stock options.

For example, an European call option gives the buyer the right on a fixed future date and time (time T) to buy a share of the fixed company for a fixed price E; such a contract is equivalent to the right to receive the amount $p(S(T))$ at time T, where $p(s) = \max(s - E, 0)$. The function p is called the payoff function of the option. A similar put option gives the owner the right to sell a share at some fixed timepoint with some fixed price; the respective payoff function is $p(s) = \max(E - s, 0)$.

Actually, there are many different types of options and not all of them are related to buying or selling something, but all options can be viewed as contracts giving the owner the right to receive in the future a payment which value is determined by the future price (or prices) of the underlying asset (or assets). So it is important to be able to compute the prices for of options with arbitrary payoff functions.

To find the price of an option, we will first need to model the share price. Based on the model we can deduce the rule for calculating the price. One of the most commonly used models is the Black-Scholes model

$$dS(t) = S(t) \cdot (\mu \cdot dt + \sigma \cdot dB(t)),$$

where μ is trend, σ is the volatility of the stock price (quantifies the risk of the instrument) and B is the standard Brownian motion. In a general case μ and σ can depend on time, stock price and the Brownian motion. In the current lab, however, we deal with constant μ and σ.

Tasks:

1. The standard bownian motion is defined by the following properties

   - $B(0) = 0$;
   - increments $B(t_2) - B(t_1)$ are normally distributed $N(0, \sqrt{t_2 - t_1})$ and independent for disjoint intervals.

   To generate the paths of a Brownian motion we need to split the interval $[0, T]$ into m disjoint intervals and the values of the Brownian motion at the time instants

$t_i = i \cdot \frac{T}{m}$ can be generated as $B(t_{i+1}) = B(t_i) + X_i$, where X_i iid normally distributed $N(0, \sqrt{\frac{T}{m}})$ random variables.
Produce a graph consisting of 10 different paths of a Brownian motion in interval $[0, 0.5]$, by dividing the latter into $m = 100$ subintervals. It is recommended to store the paths in a matrix with dimensions 100×10 (one trajectory in each column).

2. When μ and σ are constant, then it is known that,the stock prices corresponding t o the Black-Scholes model are distributed as

$$S(t) = S(0)e^{(\mu - \sigma^2/2)t + \sigma B(t)}.$$

Thus we have a one-to-one correspondence between the paths of the Browni a n motion and the paths of a stock price.
Assume $S(0) = 100$, $\mu = 0.1$, $\sigma = 0.5$ and produce a graph of 10 stock price paths in interval $[0, 0.5]$, by dividing the latter into $m = 100$ subintervals.

3. For a constant risk-free interest rate r, stock dividend percent D and volatility σ, one can calculate the option prices for the Black-Scholes model exactly . Program the Black-Scholes formulas for European call and put options. The resp ective formulas are

$$C(S, E, T, r, \sigma, D, t) = Se^{-D(T-t)}\Phi(d_1) - Ee^{-r(T-t)}\Phi(d_2),$$
$$P(S, E, T, r, \sigma, D, t) = -Se^{-D(T-t)}\Phi(-d_1) + Ee^{-r(T-t)}\Phi(-d_2),$$

where

$$d_1 = \frac{\ln(\frac{S}{E}) + (r - D + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}},\ d_2 = d_1 - \sigma\sqrt{T-t},$$

S is the stock price at time t and Φ is the cdf of a standard normal distribution (function `pnorm` in R). Make a graph of call and put option prices at $t = 0$ and the respective payoff functions in the interval $0 \leq S \leq 200$ when $r = 0.03$, $\sigma = 0.4$, $E = 100$, $T = 1$, $D = 0$, by computing the option prices for integer values of S.

4. **Homework** (deadline 23.02.2012) Make an experiment to test the fact that for European calls and puts we can calculate the option price as an expected value

$$H(t, T, S(t)) = \mathbb{E}[e^{-r(T-t)}p(S(T))],$$

where $S(T)$ is a random variable defined as

$$S(T) = S(t)e^{(r - D - \frac{\sigma^2}{2})(T-t) + \sigma(B(T) - B(t))} \tag{1}$$

and p is the option payoff function. Use the Monte-Carlo method with error probability $\alpha = 0.05$ and allowed error 0.05 and the the parameters from the previous task to compute the prices of put and call options at $t = 0.2$ when $S(0.2) = 95$. Output the price obtained by MC method, the exact price according to Black-Sch oles formulas and the difference of the results for both put and call option. Compute also the price of the option with the pay-off function

$$p(s) = \min(\frac{s}{20}, 5)$$

withe the same accuracy. (Hint: use the solution of Task 4 from the previous lab. Thus one needs to define a generator which for a given n would output n values of $S(T)$ and a suitable function g, the expected value of which needs to be computed. For defining the generator of the stock prices, one can benefit from the fact that $B(T) - B(t)$ has distribution $N(0, \sqrt{T-t})$.)

Simulation Methods in Financial Mathematics
Computer Lab 3

Goals of the lab:

- To program Euler's method for providing a solution to the stochastic differential equation (SDE)
$$dS(t) = S(t) \cdot (\mu \cdot dt + \sigma \cdot dB(t))$$
at time T so that the solution satisfies the initial value $S(0) = S_0$.
- To study the weak and strong convergence rate of Euler's method experimentally.

As we saw in the first lab, the idea of Monte-Carlo (or simulation) method is very simple: we just have to generate certain random variables X, apply a function g to generated values and calculate the average the results together with an error estimate. It turns out that the simplicity is only apparent: it is not always easy to generate efficiently the values of the random variables and that it is not always obvious what random variables and function g to use for a given problem.

The prices of many financial instruments (including options) can be expressed as expected values of certain random variables and therefore it is possible to apply MC method for computing the prices. In Lab 2 and homework problem 2 we saw that when we assume the validity of the Black-Scholes market model with constant volatility, then generating the future values of stock prices is easy and hence it is quite straightforward to use Monte-Carlo method to compute option prices. When the market model does not have constant parameters (e.g. σ is not constant), the explicit distribution of the stock prices at the exercise time T is not available. Thus we have to calculate approximate stock prices using the respective trajectories of the Brownian motion. In the present lab we study the easiest method for generating the approximate stock prices – Euler's method. In order to analyze the error introduced by approximate stock price generation, we start with the market model with constant μ and σ, so that we know the exact results.

Fix the terminal value T. In order to generate the values of $S(T)$ using Euler's method, we fix m – the number of steps and denote $dt = \frac{T}{m}$. Now we calculate

$$S_i = S_{i-1}(1 + \mu\, dt + \sigma X_i), \quad i = 1, 2, \ldots, m,$$

where $X_i \sim N(0, \sqrt{dt})$ and independent.

When using numerical methods for solving SDEs, two types of errors are distinguished. Firstly, how much does the generated stock price differ from the exact stock price (both based on the same trajectory of the Brownian motion). The rate of convergence of this error (in m) is called the strong rate of convergence. Secondly, how rapidly does the difference between $E[g(\tilde{S}(T))]$ and $E[g(S(T))]$ go to zero as m increases, where $\tilde{S}(T)$ are the approximate stock prices (generated using e.g. by Euler's method) and g is some function of interest. This rate is called the weak rate of convergence. Both rates are measured in terms of the maximal power q for which the respective errors are less than or equal to $\frac{c}{m^q}$ for some constant q. It can be shown that for most reasonably "nice" functions g (for example, for functions with bounded first derivative) the weak convergence rate is at least as large as the strong convergence rate.

It is important to understand that when calculating the price of an option by replacing $S(T)$ with its approximation produced by the Euler's method and then using the Monte-Carlo method to estimate the expected value the error of the final result consists of two

components: the error of the Monte-Carlo method which we can reduce by increasing the number of simulations n and the error introduced because of the wrong distribution of the stock prices which we can reduce by decreasing step length m. If we let n increase without a bound all that is left is the second error.

Tasks:

1. Program a function for generating n stock prices based on the Euler's method `S_euler(n,S0,m,T,mu,sigma)`. Use this function to calculate the price of an European put option when $m = 40, S0 = 50, E = 50, T = 0.5, D = 0.1, r = 0.03, \sigma = 0.7$, using the Monte-Carlo method and allowing the MC error to exceed 0.01 with probability $\alpha = 0.05$ (for the stock price generation, take $\mu = r - D$).

2. Study the strong convergence rate of the Euler's method experimentally. First write a function `S_euler_error(n,S0,m,T,mu,sigma)`, which returns the difference between the $S(T)$ obtained by the Euler's method and the exact value based on the formula
$$S(T) = S0 \cdot e^{(\mu - \frac{\sigma^2}{2})T + \sigma B(T)}$$
Then use the Monte-Carlo method to estimate the expected value of the absolute difference. NB! One has to use the same trajectories on both instances i.e. when calculating the exact value of $S(T)$ one needs to set $B(T) = \sum_{i=1}^{m} X_i$, where X_i are the random variables used with Euler's method. Use the parameters from the first task and trend $\mu = r - D$. To analyze the dependence on m calculate the expected value for $m = 5, 10, 20, 40, 80$ and the use the R command
`nls(log(error)~log(c)-q*log(m),start=list(c=1,q=1))`to estimate the rate of convergence. In that command `error` has to be the vector of errors and `m` has to be a vector of steps used.

3. Determine experimentally the weak rate of convergence of the Euler's method for the put option of the previous task using $m = 3, 6, 12, 24$ steps. NB! The allowed error for the Monte-Carlo method has to be significantly smaller than the error that depends on m!

Simulation Methods in Financial Mathematics
Computer Lab 4

Goal of the lab:

- To learn to use Euler's method for generating stock prices (which enable us to price options) when the actual distribution of the stock prices is unknown. To learn to compute option prices with a given accuracy when using a numerical method for generating stock prices.

The Euler's method for solving a stochastic differential equation (SDE) of the form

$$dY(t) = \alpha(t, Y(t))\, dt + \beta(t, Y(t))\, dB(t),\ Y(0) = Y_0$$

can be presented as

$$Y_{k+1} = Y_k + \alpha(t_k, Y_k)(t_{k+1} - t_k) + \beta(t_k, Y_k)X_k,\ k = 0, 1, \ldots, m-1$$

where $X_k \sim N(0, \sqrt{t_{k+1} - t_k})$ are iid random variables and Y_k are the approximate values of $Y(t_k)$. Typically we take $t_k = k \cdot \frac{T}{m}$, which in turn means that $t_{k+1} - t_k = \Delta t = \frac{T}{m}$. More generally, the Euler's method for solving a system of N SDE's of the form

$$dY_i(t) = \alpha(t, Y_1(t), \ldots, Y_N(t))\, dt + \beta(t, Y_1(t), \ldots, Y_N(t))\, dB_i(t),\ Y_i(0) = Y_{i0}, i = 1, \ldots, N$$

is

$$Y_{i,k+1} = Y_{ik} + \alpha(t_k, Y_{1k}, \ldots, Y_{Nk})(t_{k+1} - t_k) + \beta(t_k, Y_{1k}, \ldots, Y_{Nk})X_{ik},\ i = 1, \ldots, N,\ k = 0, 1, \ldots, m-1,$$

where the vectors $(X_{1k}, \ldots, X_{Nk})$ are iid random variables with the same n-dimensional normal distribution as $(B_1(t_{k+1}) - B_1(t_k), \ldots, B_N(t_{k+1}) - B_N(t_k))$.

Euler's method for Black-Scholes market model

$$dS(t) = S(t)(\mu(t, S(t))\, dt + \sigma(t, S(t))\, dB(t)) \tag{1}$$

with constant step $\Delta t = \frac{T}{m}$ has the form

$$S_{i+1} = S_i \cdot (1 + \mu(t_i, S_i)\Delta t + \sigma(t_i, S_i)X_i), \tag{2}$$

where the random variables X_i are independent and with distribution $N(0, \sqrt{\Delta t})$.

Let V be the price of an European option with the expiration date T and pay-off function p, then

$$V = E(\exp(-rT)p(S(T))),$$

where $S(t),\ 0 \le t \le T$ follows certain stochastic differential equation (SDE). If the SDE can not be solved exactly, then instead of $S(T)$ we use S_m, thus we use Monte-Carlo method to compute an approximate value V_m of V, where

$$V_m = E[e^{-rT}p(S_m)].$$

It is known that if p is continuous and has bounded first derivative (ie it is Lipshitz continuous), then Euler's method is weakly convergent with rate 1, hence

$$|V - V_m| = \frac{C}{m} + o(\frac{1}{m}),$$

where C is a constant that does not depend on m and $m \cdot o(\frac{1}{m}) \to 0$ as $m \to \infty$. Actually, a more precise relation

$$V - V_m = \frac{C_1}{m} + o(\frac{1}{m}),$$

where $C = |C_1|$, holds and we use that later in estimating the coefficient C.

Thus, if we use S_m instead of $S(T)$ and use Monte-Carlo method with allowed error ε at a specified allowed error probability α, then the total error of the computed number $\hat{V}_{m,\varepsilon}$ is

$$|V - \hat{V}_{m,\epsilon}| \le |V - V_m| + |V_m - \hat{V}_{m,\epsilon}| \le \frac{C}{m} + o(\frac{1}{m}) + \epsilon.$$

The last term is the error of the Monte-Carlo method and can be chosen by us. So, in order to compute the option price V with a given error ε, we should choose large enough m (so that the term $\frac{C}{m}$ is small enough, for example less than $\frac{\varepsilon}{2}$) and then use MC method with allowed error $\epsilon = \frac{\varepsilon}{2}$). There is one trouble: we do not know C. One possibility to estimate C is as follow:

1. Choose some values for m_0, ϵ_0 for m and MC error ϵ. The value of m_0 should not be too small, but very large values take too much computation time; the value of the allowed error ϵ_0 should be sufficiently small (we discuss it in more detail in the next step). In practice we usually use $m0 = 5$ or $m0 = 10$.

2. Use MC method twice to compute $\hat{V}_{m_0,\epsilon_0}$ and $\hat{V}_{2m_0,\epsilon_0}$. The value ϵ_0 is small enough if the results differ significantly more than by ϵ_0. If we use too large value of ϵ_0, then we overestimate the value of C and hence the final value of m in the following steps and our final computations may take too much time.

3. Estimate the value of C. We use the inequality

$$|V_{m_0} - V_{2m_0}| \leq |V_{m_0} - \hat{V}_{m_0,\epsilon_0}| + |V_{2m_0} - \hat{V}_{2m_0,\epsilon_0}| + |\hat{V}_{m_0,\epsilon_0} - \hat{V}_{2m_0,\epsilon_0}|.$$

   If we use the more precise infrmation about V_m and V_{2m} and assume that the terms $o(\frac{1}{m_0})$ and $o(\frac{1}{2m_0})$ are practically zero, then it follows that

$$\begin{aligned} C &\leq 2m_0 \cdot (|\hat{V}_{m_0,\epsilon_0} - \hat{V}_{2m_0,\epsilon_0}| + |V_{m_0} - \hat{V}_{m_0,\epsilon_0}| + |V_{2m_0} - \hat{V}_{2m_0,\epsilon_0}|) \\ &\leq 2m_0 \cdot (|\hat{V}_{m_0,\epsilon_0} - \hat{V}_{2m_0,\epsilon_0}| + 2\epsilon_0) =: \bar{C}. \end{aligned}$$

4. Choose m_1 such that $\frac{\bar{C}}{m_1} \leq \frac{\varepsilon}{2}$ and compute $\bar{V}_{m_1,\frac{\varepsilon}{2}}$. The last result is an approximation of the true option price which satisfies the desired error estimate, if the starting value of m_0 was large enough so that the additional error terms of order $o(\frac{1}{m})$ are practically equal to zero. In this course we do not consider methods of determining if the starting value of m_0 was sufficiently large and take the result of the last computation to be the desired answer.

Tasks:

1. Find the value of an European call option with strike price $E = 98$ at time $t = 0$ with precision 0.1, when $\alpha = 0.05$, $r = 0.05$, $D = 0$, $T = 0.5$, $S(0) = 100$ and $\sigma(t, s) = 0.7 - 0.7\, e^{-0.01s}$. To solve the problem we have to choose an m so that the error due to m would be sufficiently small.

2. Future risk free interest rate is actually not a constant and can be considered to be a random variable. In the case Black-Scholes model with a random interest rate the prices of European options can be computed as

$$Price = E[\exp(-\int_0^T r(t)\, dt)p(S(T))],$$

   where

$$dS(t) = S(t)((r(t) - D)\, dt + \sigma\, dB_1(t))$$

   and $r(t)$ follows a suitable stochastic differential equation. We conside so calle Cox-Ingersoll-Ross model

$$dr(t) = a(b - r(t))\, dt + \sigma_2 \sqrt{r(t)}\, dB_2(t),$$

   where $B_1(t)$ andf $B_2(t)$ are independent Brownian motions. So we have a system of stochastic differential equations for S and r. Write a function that for a given values of parameters $D, \sigma, \sigma_2, T, a, b, m$ and n generates n values of the future stock prices $S(T)$ by using Euler's method with m time steps for solving the system of SDEs for S and r.

3. **Homework** (Deadline 08.03.2012). Find the price of a call option with maximum error of 0.1 when considering the market model with a stochastic interest described in task 2. Use the parameter valuesf (except r) from the first task and additionally let $a = 1$, $b = 0.06$, $\sigma_2 = 0.1$ and $r(0) = 0.03$. When computing the expected value, replace the integral with the product of T and the mean value of r for a given trajectory. For this, the generator should return a two-column matrix, where the first column contains the approximate values of $S(t)$ and the secon column contains the average interest rates for each generated trajectory. In order to use the general functions of applying MC methods we should rewrite the function g so that it works correctly when it's argument is the result of the generator, so a correct form is

```
g=function(x){return(exp(-T*x[,2])*p_call(x[,1]))}
```

Simulation Methods in Financial Mathematics
Computer Lab 5

Goal of the lab:

- To learn to use higher order discretization methods for stock price generation and study their error rates

There are many numerical schemes of different strong and weak convergence rates for solving stochastic differential equations. For example, a stochastic differential equation of the form

$$dS(t) = S(t)(\mu(t, S(t))\, dt + \sigma(t, S(t))\, dB(t)) \tag{1}$$

can be solved by using the following methods:

Euler's method

$$S_{i+1} = S_i \cdot (1 + \mu(t_i, S_i)\Delta t + \sigma(t_i, S_i)X_i), \tag{2}$$

Milstein's method

$$S_{i+1} = S_i \cdot \left(1 + \mu(t_i, S_i)\Delta t + \sigma(t_i, S_i)X_i + \frac{1}{2}(S_i\frac{\partial\sigma}{\partial s}(t_i, S_i) + \sigma(t_i, S_i))\sigma(t_i, S_i)(X_i^2 - \Delta t)\right) \tag{3}$$

and

A weakly second order method

$$\begin{aligned} S_{i+1} &= S_i \cdot (1 + \mu(t_i, S_i)\Delta t + \sigma(t_i, S_i)X_i) \\ &+ \frac{1}{2}\left(L_1\mu(t_i, S_i)\Delta t^2 + (L_2\mu(t_i, S_i) + L_1\sigma(t_i, S_i))\Delta t\, X_i + L_2\sigma(t_i, S_i)(X_i^2 - \Delta t)\right), \end{aligned} \tag{4}$$

where random variables X_i are independent and have distribution $N(0, \sqrt{\Delta t})$ and the operators L_1 and L_2 are defined by

$$L_1 f(t, s) = s\frac{\partial f}{\partial t}(t, s) + s \cdot \mu(t, s) \cdot (f(t, s) + s\frac{\partial f}{\partial s}(t, s)) + \frac{s^2\sigma(t, s)^2}{2}(2\frac{\partial f}{\partial s}(t, s) + s\frac{\partial^2 f}{\partial s^2}(t, s)),$$

$$L_2 f(t, s) = s\sigma(t, s)(f(t, s) + s\frac{\partial f}{\partial s}(t, s)).$$

For example, if $\mu(t, s) = \mu$ (a constant), then $L_1\mu(t, s) = s \cdot \mu^2$, $L_2\mu(t, s) = s\sigma(t, s) \cdot \mu$.

Tasks:

1. Program methods (3) and (4) for generating solutions of the SDE (1) at time T when the Black-Scholes market model with constant coefficients is assumed.
2. Experimentally find the strong convergence rates of (3) and (4) using procedure `nls` in the case when $S(0) = 100,\ T = 0.5, \mu = 0.1$ and $\sigma = 0.6$.
3. Experimentally find the weak convergence rates of (3) and (4) for an European put option when $E = 100$, $T = 1$, $r = 0.05$, $S(0) = 100$, $D = 0$, $\sigma = 0.5$.

Homework 4 (deadline 15.03.2012) Use the weakly second order method for compung the price of the put option with the total error less than 0.1 with probability 0.95 in the case $S(0) = E = 100,\ T = 1.0,\ r = 0.05,\ D = 0.1$ and $\sigma(s) = e^{-s/100} + 0.3$. Note that in the process of choosing an appropriate value of m one has to take into account the second order convergence rate, so the procedure of of the previous lab for computing the price of an option with a given accuracy has to be slightly modified.

Simulation Methods in Financial Mathematics
Computer Lab 6

Goal of the lab:

- To learn to use antithetic variates and control variates for variance reduction in the Monte Carlo method.

Antithetic variates. Suppose we know, how to generate Y and $\tilde{Y}$ that have the same distribution but are negatively correlated. Then it is reasonable to generate a random variable $Z = \frac{Y+\tilde{Y}}{2}$ as the variance of it is smaller than the variances of Y and $\tilde{Y}$. This means that the Monte Carlo method will work faster. In option pricing the easiest way to introduce antitethic variates is generating pairs of stock prices $(S(T), \tilde{S}(T))$, where the random variables used to generate the second are the ones used for the first but with changed sign. Assuming the payoff function is monotone, its values will be negatively correlated for those price pairs and we can calculate

$$Price = E[e^{-rT}\frac{p(S(T)) + p(\tilde{S}(T))}{2}].$$

Control variates. Control variates can be used when in addition to the random variable of interest, Y, we also know how to generate another random variable Z for which its expected value is known. This means that we are also able to generate $Y_1 = Y - a(Z - EZ)$ and if Y and Z are strongly correlated and a is chosen wisely, Y_1 can have a variance that is a magnitude smaller than the variance of Y. The constant a should approximate $\frac{\mathrm{cov}(Y,Z)}{DZ}$. In this lab the price of an European call option with strike price $E = 100$ at time $t = 0$ is examined when $r = 0.1$, $D = 0.05$, $T = 0.5$, $S(0) = 102$ and $\sigma(t, s) = 0.5 - 0.3\,e^{-0.01s}$.
Tasks:

1. Enhance the function that calculates expected values with Monte-Carlo method so that it would output the number of generations used (in thousands) to achieve a specific precision. Use the function for pricing the call option by generating stock prices with Euler's method with $m = 20$ and precision 0.05.

2. Enhance the generator of stock prices using Euler's method so that it would output a matrix with two colums consisting of pairs of antithetic stock prices. Function g also needs to be updated to be able to use the output. Compare the number of generations needed (when using pairs the number of generations is actually double the amount of the counter value)

3. Modify the generator that uses the Euler's method so that it would use the same Brownian motion to output the stock prices for a market model with constant volatility and a market model with a non-constant volatility. For the former use the exact stock price formula. Let S be the stock price corresponding to the non-constant volatility and $\bar{S}$ the stock price corresponding to the constant volatility. Use the Monte Carlo method (with the settings of this lab and constant volatility $\sigma_1 = 0.4$) to find the expected value of

$$e^{-rT}p(S) - a(e^{-rT}p(\bar{S}) - h).$$

   Here h is the exact price of the option for the market model with constant volatility and a is the estimate of

$$\frac{\mathrm{cov}(p(S), p(\bar{S}))}{\mathrm{var}(p(\bar{S}))}$$

   based on 1000 generations. Function p is the payoff function for an European call option. Compare the number of generations needed with the corresponding number of the previous task.

Simulation Methods in Financial Mathematics
Computer Lab 7

Goal of the lab:

- To learn to use importance sampling and stratified sampling for s peeding the numerical option pricing process.

The idea of **importance sampling** comes from the fact that we can compute instead of the expected value of $g(X)$ in the case of random variable X with probability density function f_X the expected value of $\frac{g(Y)f_X(Y)}{f_Y(Y)}$ using a random variable Y with the probability density function f_Y:

$$E(g(X)) = \int_{-\infty}^{\infty} g(x)f_X(x)\,dx = \int_{-\infty}^{\infty} \frac{g(y)f_X(y)}{f_Y(y)} f_Y(y)\,dy = E[\frac{g(Y)f_X(Y)}{f_Y(Y)}].$$

This idea is very useful when X is such that $g(X)$ has large values with low probability and Y increases the chances of seeing the large values of g with higher probability.
Using this idea it can be shown that we can also calculate the price of t he option as

$$Price = E[e^{-rT-\eta B(T)-\frac{\eta^2 T}{2}} p(S(T))],$$

where the stock price $S(T)$ corresponds to the market model

$$dS(t) = S(t)((r - D + \eta\sigma)\,dt + \sigma dB(t)).$$

This is especially useful for out-of-the-money options when there is relati vely low probability of the payoff function of becoming positive. To reduce the vari ance of the random variable expectation of which is to be found, we must choose η so that $S(0) \cdot e^{(r-D+\eta\sigma)T}$ would be in the region where the payoff function p is not equal to zero. This kind of methodology is known as importance sampling because we increase the probability of generating important values of the random variable. The described metho d is also applicable when the volatility σ is not constant and can also be used in the case of american options.
Stratified sampling is based on specifying (disjoint) events $(A_i)_{i=1}^k$ that partition the probability space and then invoking the formula

$$E(Y) = \sum_{i=1}^{k} E(Y|A_i)P(A_i).$$

There are several possibilities for the latter, including the followin g:

- one can replace the calculation of the expected value of Y by the calculation of the expected value of $\sum_{i=1}^{k} P(A_i)Y_i$, where Y_i are independent and are distributed as Y given the event A_i;
- Conditional expectations on the right-hand side can be calculated sep arately using the MC method and then a weighted sum of the results can be calculated. In order to get the final result with error ε, one should compute the the expected value at each stratum with the error $\frac{\varepsilon}{\sqrt{P(A_i)}}$.

Since these two approaches turn out to be equivalent, we use only the second one. In this lab we use the system of events

$$A_i = \{B(T) \in (\sqrt{T}\Phi^{-1}(\frac{i-1}{k}), \sqrt{T}\Phi^{-1}(\frac{i}{k})]\},\ i = 1, 2, \ldots, k,$$

thus $P(A_i) = \frac{1}{k},\ i = 1, 2, \ldots, k$.
In the following we will be interested in a call option with $S(0) = 50$, $r = 0.1$, $\sigma = 0.5$, $T = 1$, $t = D = 0$, $E = 100$.
Tasks:

1. Use importance sampling to find the price of an European call opt ion with precision 0.01 and error probability 0.05. Use the exact formula for stock price generation. Determine the best value of η (best being the one with which least number of generated random variables is needed) with precision 0.1.

2. Write a procedure, that uses the representation $S(T) = S(0)e^{(r-D-\frac{\sigma^2}{2})T+\sigma B(T)}$ and the number i of the strata to generate stock prices corresponding to the event A_i. Values of the Brownian motion from the conditional distribution can be generated as $\sqrt{T}\Phi^{-1}(X_i)$, where $X_i \sim U(\frac{i-1}{k}, \frac{i}{k})$. Use this procedure to find the price of the call option when using k with values $k = 5$, 10, 20, 100 with precision 0.01 (take the probability of an error as $\alpha = 0.05$). Compare the number of generations (that is the sum of the number of generations for all strata) with the ordinary MC method as well as with the importance sampling method.

Simulation Methods in Financial Mathematics
Computer Lab 8

Goal of the lab:

- To continue to study the use of stratified sampling for speeding up the numerical option pricing process.

Stratified sampling is based on specifying (disjoint) events $(A_i)_{i=1}^k$ that partition the probability space and then invoking the formula

$$E(Y) = \sum_{i=1}^{k} E(Y|A_i)p_i,$$

where $p_i = P(A_i)$. It is known that maximal variance reduction is achieved when in stratum i the number n_i of generated random variables is proportional to $\sigma_i p_i$ (where σ_i^2 is the conditional variance in stratum i); it is also known that when we generate n_i random variables in stratum i then the variance of

$$Z = \sum_{i=1}^{k} p_i \sum_{j=1}^{n_i} \frac{Y_{ij}}{n_i}$$

is equal to

$$DZ = \sum_{i=1}^{k} \frac{p_i^2 \sigma_i^2}{n_i}$$

and $EZ = EY$. Thus one possibility for generating the optimal number of Y_i is setting the proportionality constant C to some fixed value and in every stratum generating random variables until $n \geq Cp_i \cdot \sigma_i$ where the value of σ_i is estimated.

Proofs of the previous results can be found, for example, in P. Glasserma nn, "Monte Carlo Methods in Financial Engineering", Section 4.3.

We will continue to use te system of events

$$A_i = \{B(T) \in (\sqrt{T}\Phi^{-1}(\frac{i-1}{k}), \sqrt{T}\Phi^{-1}(\frac{i}{k})]\},\ i = 1, 2, \ldots, k,$$

and thus $p_i = P(A_i) = \frac{1}{k},\ i = 1, 2, \ldots, k$. We want to price a call option when $r = 0.1$, $\sigma = 0.4$, $T = 0.5$, $t = D = 0$, $E = 100$.

Tasks:

1. Write a function for calculating the expected value so that proport ionality constant C and stratum probability p can be given as input and the expected value is calculated as the average of n generated values, where n is the smallest number in whole hundreds such that $n_{\geq} C \cdot p \cdot \sigma_Y$ (where σ_Y is the estimate of the standard deviation of the random variable, whose expected value we are calculating). The o utput of the function should include the calculated expected value, estimated variance and the number of generations.

2. Program a procedure that would calculate the price of the option using stratified sampling, where the conditional expected value in every stratum is calcu lated using the previously completed function with some given C. Output the overall error estimate (calculated assuming that Z is normally distributed, so that with probability α we have $|Z - EY| \leq -\Phi^{-1}(\frac{\alpha}{2})\sqrt{DZ}$) and the number of total genrations in addition to the calculated expected value.

3. **Homework 5** (deadline 05.04.2012) Compare the number of random variable g enerations needed for this optimal method with those of the stratified sam pling method of the previous lab when $S0 = 75$, 100, 125, $k = 10\,20$, 40, 80 and the allowed error is 0.01. Thus a proportionality constant C must be picked so that the error estimate is approximately equal to the allowed error. (Hint: if we multip ly C by a certan number x, then the error of the final result is divided approximately by $\sqrt{x}$)

Simulation Methods in Financial Mathematics
Computer Lab 9

Goal of the lab:

- To learn to use stratified sampling for speeding the numerical option pricing process in situations where the price of the option is depenent on the path of the stock price.

We denote by A' the transpose of a matrix A and by writing $Y \sim N(0, s)$ we mean that Y is normally distributed with mean 0 ans standard deviation s.
When $\mathbf{v} = (v_1, \ldots, v_m)'$ is a non-zero vector (i.e. $\|\mathbf{v}\| = \sqrt{v_1^2 + \ldots + v_m^2} > 0$), $W \sim N(0, a\|\mathbf{v}\|)$ and $Z_i \sim N(0, a)$, $i = 1, \ldots, m$ are independent random variables and $a > 0$, then it is easy to check that by defining a vector X of random variables X_i, $i = 1, \ldots, m$ as

$$\mathbf{X} = \frac{W}{\|\mathbf{v}\|^2}\mathbf{v} + \mathbf{Z} - \frac{(\mathbf{v}'\mathbf{Z})}{\|\mathbf{v}\|^2}\mathbf{v}$$

we have that the components of X are independent and have distribution $N(0, a)$, and also $\mathbf{v}'\mathbf{X} = W$. Indeed, X is normally distributed since it is a linear combination of (jointly) normally distributed random variables and by a direct calculation we get that the covariance matrix $E(XX')$ is of the form aI_m, where I_m is the $m \times m$ identity matrix.

If we want to generate at once more than one vector, each corresponding to different value of W, then the former formula can be written as

$$\mathbf{X} = \frac{1}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{W} + \mathbf{Z} - \frac{(\mathbf{v}\mathbf{v}'\mathbf{Z})}{\|\mathbf{v}\|^2},$$

where $\mathbf{X}$ is now a $m \times n$ matrix with independent normally $N(0, a)$ distributed random variables, $\mathbf{W}$ is a $1 \times n$ matrix (a row vector) of independent $N(0, a\|\mathbf{v}\|)$ distributed random variables and $\mathbf{Z}$ is a $m \times n$ matrix with independent normally $N(0, a)$ distributed random variables. The matrix $\mathbf{X}$ has now the property $\mathbf{v}'\mathbf{X} = \mathbf{W}$ (i.e. each column sums with weights v_i to the value of W_i). Thus we can generate independent normally distributed random variables so that we first generate the value of a lienar combination of the variables and then determine the variables itself.

This result allows us to stratify the generation of normally distributed random variables X_i according to any given linear combination (e.g. according to the sum) of the random variables - we just have to generate the values W from a given stratum and to determine X_i by the above formula.

We use the previous result to generate the increments of a Brownian motion $B(t_1) - B(0), B(t_2) - B(t_1), \ldots, B(T) - B(t_{m-1})$ so that their sum $B(T)$ would be in a given stratum (i.e. $\mathbf{v} = (1, 1, \ldots, 1)'$). To accomplish this we first need to generate the value of W (from the desired stratum) according to the distribution $N(0, \sqrt{T})$ and calculate the vector of Brownian motion increments X using the formula presented (for intervals that have equal lengths, $a = \sqrt{\frac{T}{m}}$).

It is useful to know that in R the matrix multiplication is $\%*\%$ and transposed matrix can be obtined by the function $t()$.

Tasks:

1. Write a procedure that, given the inputs k, m, T, would draw k different Brownian motion paths such that every stratum (based on the value of $B(T)$ and defined as in the previous lab) would include the terminal value of exactly one path.

2. Enhance the stock price generation function that is based on Euler's method and non-constant volatility so that one could use the optimal stratified sampling. Using optimal stratified sampling find the price of an European call with precision 0.01 in the case when $r = 0.06$, $D = 0.03$, $\sigma(s) = \frac{95}{95+s}$, $T = 0.5$, $S(0) = 105$, $E = 100$, using $\alpha = 0.05$.

Simulation Methods in Financial Mathematics
Computer Lab 10

Goal of the lab:

- To learn to use Monte-Carlo method for pricing Asian options

An Asian option is an option which payoff depends on the average sto ck price. Let $A(T)$ be the average stock price for the period [0, T] i.e.

$$A(T) = \frac{1}{T}\int_0^T S(t)\,dt.$$

The most typical payoff functions for Asian options are $p(s,a) = \max(s-a,0)$ (i.e. at time T the value of the option is $p(S(T),A(T)) = \max(S(T)-A(T),0)$; it is known as the average strike call option), $p(s,a) = \max(a-s,0)$ (average strike put option), $p(s,a) = \max(a-E,0)$ ja $p(s,a) = \max(E-a,0)$ (average price call and put options with strike price E). When the stock price is governed by the Black-Scholes market model, then the price of all the named options at time $t = 0$ can be calculated as the expected value

$$Hind = E[e^{-rT}p(S(T),A(T))],$$

where $S(T)$ corresponds to the Black-Scholes market model with trend $\mu = r - D$. Thus to use the MC method we need to generate (in addition to the stock pri ces at time T) the average values of stock prices which depend also on the intermediat e values of the stock price paths. The simplest way of calculating the average is us ing the average of $S(i\frac{T}{m}),\ i = 0,1,\ldots,m-1$; a better approximation can be calculated as

$$A(T) \approx \frac{1}{m}\sum_{i=1}^{m} S_{i-1}\left(1 + (r-D)\frac{T}{2m} + \frac{\sigma}{2}(B(t_i) - B(t_{i-1}))\right).$$

The idea leading to the improved formula is to write

$$\frac{1}{T}\int_0^T S(t)\,dt = \frac{1}{T}\sum_{i=1}^{m}\int_{t_{i-1}}^{t_i} S(t)\,dt = \frac{1}{T}\sum_{i=1}^{m}\int_{t_{i-1}}^{t_i}(S(t_{i-1}) + \int_{t_{i-1}}^{t} dS(\tau))\,dt,$$

using the equation for $dS(\tau)$ and to approximate the resulting double integrals by replacing the integrands wiht their values at the beginning of the integratio n intervals.

We will assume that the Black-Scholes market model with constant par ameters holds and fix $r = 0.1$, $D = 0$, $\sigma = 0.4$, $T = 0.5$, $S(0) = 100$. We will consider average strike calls and average price calls with strike price $E = 100$.

Tasks:

1. Write a generator which for a given value of n would generate n pairs of (terminal) stock price and average stock price. Calculate the stock prices using the exact formula (also use it when calculating the average stock prices); calcu late the average as the mean of $S(i\frac{T}{m}),\ i = 0,1,\ldots,m-1$. Find the weak convergence rate depending on m (by using the values $5, 10, 20, 40$ for m and a small enough MC error). For both options find m for which the error caused by the choice of m is less than 0.1 (using the result obtained for the weak rate of convergence).

2. Repeat the task when the average price is calculated according to the im proved formula. To study the rate of weak convergence use the values $2, 4, 6, 8$ for m and take 0.01 for the MC error.

Simulation Methods in Financial Mathematics
Computer Lab 11

Goal of the lab:

- To learn to use stratified sampling for speeding up the pricing of a sian options

We will consider the case where the average stock price is calculated usin g the formula

$$A(T) \approx \frac{1}{m}\sum_{i=1}^{m} S_{i-1}\left(1 + (r-D)\frac{T}{2m} + \frac{\sigma}{2}(B(t_i) - B(t_{i-1}))\right),$$

where $t_i = i\frac{T}{m}$. We already know, how to generate the increments of the Brownian motion $\Delta B = (B(t_1) - B(t_0), B(t_2) - B(t_1), \ldots, B(t_m) - B(t_{m-1}))'$ (here $\mathbf{a}'$ denotes the transpose of $\mathbf{a}$) so that given a vector $\mathbf{v} = (v_1, \ldots, v_m)'$ the stratifying would be based on the values of $v'\Delta B$:

- Generate the value of W from the desired stratum of $N(0, \|v\|\sqrt{\frac{T}{m}})$.
- Generate a random vector $\mathbf{Z} = (Z_1, \ldots, Z_m)'$, where $Z_i \sim N(0, \sqrt{\frac{T}{m}})$
- Calculate
$$\Delta B = \frac{1}{\|v\|^2} W\mathbf{v} + \mathbf{Z} - \frac{1}{\|v\|^2}\mathbf{v}(\mathbf{v}'\mathbf{Z}).$$

When we want to generate a matrix with dimensions $m \times n$ so that each column would represent the increments of the Brownian motion of a corresponding genera ted value of W (thus in total there are n values of W), then we need to modify the formula as follows: W must be generated as a row vector with n components, Z must be a $m \times n$ matrix and the increment matrix can then be calculated as

$$\Delta B = \frac{1}{\|v\|^2}\mathbf{v}\mathbf{W} + \mathbf{Z} - \frac{1}{\|v\|^2}(\mathbf{v}\mathbf{v}'\mathbf{Z}).$$

Using this formula there are several possibilities for stratificati on:

- When we want the the strata based on the values of $B(T)$, we take $\mathbf{v} = (1, 1, \ldots, 1)'$. This stratification should be appropriate for European options.
- When we want the the strata based on the average values of the Browni an motion $\frac{1}{T}\int_0^T B(t)\,dt$, we may approximate the integral by
$$\frac{1}{T}\int_0^T B(t)\,dt \approx \sum_{i=0}^{m-1}\frac{B(t_i)}{m} = \sum_{i=1}^{m-1}\sum_{j=1}^{i}\Delta B_j = \sum_{j=1}^{m-1}\frac{m-j}{m}\Delta B_j$$
and hence should use stratification with $\mathbf{v} = (\frac{m-1}{m}, \frac{m-2}{m}, \ldots, \frac{m-m}{m})'$. This stratification should be appropriate for Asian options when the payoff does not depend on the stock price at time T.
- When we want the the strata based on differences of $B(T)$ and the average values of the Brownian motion, we take $\mathbf{v} = (\frac{1}{m}, \frac{2}{m}, \ldots, \frac{m}{m})'$. This stratification should be appropriate for average strike options.

Tasks:

1. Let $m = 20$. Consider the pricing of Asian options with payoff functions $p(s, a) = \max(50 - a, 0)$ and $p(s, a) = \max(s - a, 0)$ using the MC method with error 0.01 in the case when $S(0) = 49$, $r = 0.05$, $D = 0.02$, $T = 0.5$, $\sigma = 0.5$. Compare the number of generations required when not using any variance reduction met hods and when using the optimal stratified sampling with the number of strat a $k = 40$ for all the abovementioned stratifications (3 different values of v).

2. **Homework 6** (deadline 26.04.2012). When volatility is not constant, then we have to use a numerical method for generating the stock prices and an approx imation for the average stock price. Implement MC for pricing Asian options by us ing Euler's method for generating the option prices and the simple formula for com puting the values of the average stock price. Use this method for computing the p rice of the average strike put option in the case $S0 = 90$, $r = 0.05$, $D = 0.02$, $T = 1$, $\sigma(s, t) = 0.4 + 0.5 \cdot e^{-\frac{s+5t}{100}}$, with the total error that is less than 0.01 with the probability 0 .95. Use an appropriate variance reduction method if necessary.

Simulation Methods in Financial Mathematics
Computer Lab 12

Goals of the lab:

- To learn the generation of Halton sequences for quasi-Monte Carlo simulation
- To learn to compute the price of an European option by using quasi-Monte Carlo method

The speed of convergence of Monte-Carlo methods for computing expected values is always $\frac{c}{\sqrt{n}}$, where n is the number of random variables generated. It is possible to reduce the constant c, but for Monte-Carlo methods the convergence speed with respect to n is always the same. It turns out that if we replace the random numbers in a Monte-Carlo method by specially constructed (non-random) values, it is possible to improve the convergence rate to $\frac{C}{n}$. The basis for this improvement are so called low discrepancy sequences which are sequences in the m-dimensional unit cube $[0,1]^m$ which for every value of n cover the cube in some sense as well as possible. One (the simplest possible) class of such a sequences is so called Halton sequences.

To generate Halton sequences we need to know how to represent a number, say k, in a number system that has base b where $b \geq 2$. To find this representation we can use the fact that if

$$k = \sum_{i=0}^{\infty} \alpha_i(k) b^i, \ \alpha_i(k) \in \{0, 1, \ldots, b-1\},$$

then the multipliers $\alpha_i(k)$ can be determined as

$$\alpha_i(k) = (k\%/\%b^i)\%\%b,$$

where $\%/\%$ symbolizes the process of finding the division quotient (an integer) and $\%\%$ symbolizes the process of finding the remainder. Denote

$$\psi_b(k) = \sum_{i=0}^{\infty} \frac{\alpha_i(k)}{b^{i+1}}.$$

When generating an m-dimensional Halton sequence, m numbers $b_1, b_2, \ldots, b_m$ that do not have a common factor (usually m first prime numbers are chosen) and point $\mathbf{x}_k$ is defined as a point with coordinates

$$x_{kj} = \psi_{b_j}(k), \ j = 1, \ldots, m.$$

When the expected value must be calculated from a function which uses m independent values from the uniform distribution $U(0,1)$ then we can replace these values with the coordinates of a point from the quasi-random sequence. Other distributions can also be replaced by applying the inverse of the cumulative distribution function to the coordinates.

Tasks:

Task 1 Write a function `Corput` with input $n1$, $n2$ and b, that would output the values of function $\psi_b(k)$ for $k = n1, n1+1, \ldots, n2$ (to deal with all the values of k simultaneously R function `outer` can be used). Produce the graphs showing the first 200 points of the Halton sequence using the function `Corput` in the case when $m = 2$ with a) $b_1 = 2, b_2 = 3$ and b) $b_1 = 17, b_2 = 19$.

Task 2 Consider pricing an European put option with $E = 100$ at time $t = 0$ using the Euler's method when $r = 0.1$, $D = 0$, $T = 0.5$, $S(0) = 105$ and $\sigma(t, s) = 0.6 - 0.5\, e^{-0.01s}$. We want to study how much the use of Halton sequence speeds up the convergence of the MC method in the cases when $m = 5$ and $m = 20$. To do this, replace the increments of the Brownian motion with values that are generated using a Halton sequence (coordinates of which are based on the first m prime numbers) and find the errors for cases $n = 10000, 20000, \ldots, 100000$. For comparison, find the errors for regular MC method for the same values of n. Use the knowledge that the expected value is $7,731$ when $m = 5$ and it is $7,577$ when $m = 20$.

Simulation Methods in Financial Mathematics
Computer Lab 13

Goal of the lab:

- To learn to use Quasi Monte Carlo methods for pricing financial options.

The procedure of implementing a Quasi Monte Carlo methods is as follows:

1. Implement a Monte-Carlo method for for computing EY, where Y is the random variable we are interested in. It can be a simple MC or a method using various variance reduction techniques.

2. Rewrite your MC method so that it is based on generating independent uniformly distributed random variables. This can be achieved, for example, by applying the knowledge that if F_X is the cumulative distribution function of a continuous random variable X, then the random variable $F_X^{-1}(U)$, where $U \sim U(0,1)$ is uniformly distributed in the interval $(0,1)$, has the same distribution as X. So, if our MC method uses random variables from the distribution $N(0,a)$, then we can generate them by
$$X = a\Phi^{-1}(U),$$
where $U \sim U(0,1)$ and Φ is the cumulative distribution function of the standard normal distribution.

3. If for generating one value of Y we use m uniformly distributed random variables, then replace them by coordinates of a point of a m-dimensional low discrepancy sequence.

4. Compute the final answer using as the simple average of the largest number of values of Y that can be generated in reasonable time.

The final answer is usually much more accurate than the answer obtained with a simple MC using the same number of generated values of Y. Unfortunately there are no efficient procedures for estimating the error of the answer obtained by this procedure that is called Quasi Monte Carlo method.

In this lab we use Sobol sequences. The Sobol sequences are based on the van der Corput sequences (like Halton sequences) but for every coordinate base 2 is used. Different coordinates are obtained by applying specifically constructed matrices that change the order of the sequence. The generation of a Sobol sequence is not too hard to implement, however we will use pre-defined functions in this lab.

There are several add-on packages of R that implement quasi random number generators. One such library is `randtoolbox` and we are going to use the generators from this library. The function for generating Sobol sequences is `sobol()` and the function for Halton sequences is `halton()`

Tasks:

1. Install and load package `randtoolbox`, if needed. Installation can usually be done by the command `install.packages("randtoolbox")` and loading by the command `library("randtoolbox")`. If you do not have administrative rights and R is not configured well, you still can install packages for your own use. For this make a catalog where you are going to install your own R packages. Then you can use commands

```
.libPaths("your catalog paths")
install.packages("randtoolbox",lib="your catalog path")
```

to install the package. Generate first 200 Sobol points for dimension 10 and visualize the placement by scatterplots that can be created with R function `pairs()`. Compare the placement with Halton points.

2. Consider pricing an European put option with $E = 100$ at time $t = 0$ using the Euler's method when $r = 0.1$, $D = 0$, $T = 0.5$, $S(0) = 105$ and $\sigma(t, s) = 0.6 - 0.5\, e^{-0.01s}$. We want to study how much the use of Sobol sequence speeds up the convergence of the MC method in the cases when $m = 5$ and $m = 20$. To do this, replace the increments of the Brownian motion with values that are generated using a Sobol sequence and find the errors for cases $n = 10000, 20000, \ldots, 100000$. For comparison, find the errors for regular MC method for the same values of n. Use the knowledge that the expected value is $7,731$ when $m = 5$ and it is $7,577$ when $m = 20$.

3. **Homework 7** (Deadline 10.05.2012) Assume that the Black-Scholes market model with constant parameters $r = 0.05$, $D = 0$, $\sigma = 0.5$, $T = 0.5$, $S(0) = 90$ holds. Consider pricing the average price put option (see Lab 10). Start from the Monte-Carlo method that uses the exact formula for computing the stock prices and the improved formula for computing the average stock price values. Modify this method for using Sobol points. Find the prices and error estimates for $m = 4$ and $m = 10$ corresponding to $n = 10000, 20000, \ldots, 100000$ and for $n = 1000000$ by the standard Monte-Carlo method and also prices by QMC using Sobol points for the same number of generations. Does QMC give more accurate answers for the same number of generations?

Simulation Methods in Financial Mathematics
Computer Lab 14

Goal of the lab:

- To learn a way to compute derivatives of the option price with respect to market parameters

Often the behavior of the random variable X depends on some parameter θ and therefore the expected value $E(g(X(\theta))$ depends also on that parameter, so we have

$$V(\theta) = E[g(X(\theta))]$$

and often one is interested in the value of the derivative of the expected value with respect to that parameter. For example, stock price (and hence the price of an option) depends on the initial stock price $S(0)$, the volatility σ, risk free interest rate r etc and we may want to know how the price changes if some of the parameters changes. A general way to compute the derivatives is to use finite difference approximations, for example

$$V'(\theta) = \frac{V(\theta+h) - V(\theta-h)}{2h} + O(h^2).$$

When using Monte-Carlo method for computing the values of V it is important not to compute $V(\theta+h)$ and $V(\theta-h)$ independently, but to write

$$\frac{V(\theta+h) - V(\theta-h)}{2h} = E[\frac{g(X(\theta+h)) - g(X(\theta-h))}{2h}]$$

and to compute this expected value by MC by using the same random variables for generating both the values of $X(\theta+h)$ and $X(\theta-h)$ at the same time.

Let us consider an European call option with $E = 100$, $T = 0.5$, assume $r = 0.05$, $D = 0$, $S(0) = 100$ and that the Black-Scholes market model holds.

Tasks:

1. Assume that the volatility is constant: $\sigma = 0.5$. Implement Monte-Carlo method for computing the derivative of the price of the option with respect to $S(0)$, using the exact formula for generating the stock prices. Compute the derivative at $S(0) = 100$ using $h = 20$ and allowed MC error 0.001 (corresponding to $\alpha = 0.05$).

2. If we want to compute the derivative with a given total error then we have to estimate the error coming from the choice of h. For this the Runge's method can be used. We assume that error coming from the choice of h is $C \cdot h^2$ with an unkonwn h. In order to esitmate C we do two computations, one for $h = 2h_0$ and the other on with $h = h_0$ (where h_0 is some value we choose) and using the difference of the answers it is possible to estimate C. After that we can choose h so that $C \cdot h^2 \leq \frac{\epsilon}{2}$, where ϵ is the allowe total error and do the final computation with this value of h and allowed MC error $\frac{\epsilon}{2}$. Find the derivative of the option price considered in the previous problem with total error less than 0.0005 with probability 0.95.

3. Consider the case of non-constant volatility $\sigma(s) = 1 - 0.5 \cdot e^{-(s-100)^2/200}$. Implement the procedure for computing the value of the derivative of the option price with respect to the initial stock price with a given accuracy. Use variance reduction techniques for speeding up the computation and find the value as exactly as you can so that the total computation time does not exceed 3 minutes; give an estimate of the total error of your answer.

Simulation Methods in Financial Mathematics
Computer Lab 15

The aim of this lab is to learn to use two additional methods for computing sensitivities of option prices.

Let us consider European options. For simplicity we also assume that the stock price behaves according to Black-Scholes market model with constant volatility. Then the price of an option with payoff function p is given by

$$H = E[e^{-rT}p(S(T))],$$

where T is the exercise time and the final stock price $S(T)$ is given by

$$S(T) = S_0 e^{(r-\sigma^2/2)T+\sigma B(T)}.$$

In addition to the difference quotients method considered in the previous lab, we consider the following two methods.

1. **Pathwise derivative method.** Let θ denote a parameter in the stock price formula. Assume that the function p is continuous with piecewise continuous and bounded derivative, then it can be shown that we can change the order of taking expected value and differentiation when computing the derivative of the price with respect to θ:
$$\frac{\partial H}{\partial \theta} = E[\frac{\partial(e^{-rT}p(S(T)))}{\partial \theta}].$$
For example, if $\theta = S_0$ we have
$$\frac{\partial H}{\partial S_0} = E[e^{-rT}p'(S(T))\frac{\partial S(T)}{\partial S0}] = E[e^{-rT}p'(S(T))\frac{S(T)}{S_0}].$$

2. **Likelihood ratio method.** Suppose we know the probability density function f_S of the final stock price. If θ is a market parameter, then the density function depends on θ (is a function of s and θ). Using the density function, we can write
$$H = \int_{-\infty}^{\infty} e^{-rT}p(s)f_S(s,\theta)\,ds.$$
Assuming again, that we can change the order of integration and differentiation, we get now
$$\begin{aligned}\frac{\partial H}{\partial \theta} &= \int_{-\infty}^{\infty} p(s)\left(\frac{\partial}{\partial \theta}(e^{-rT}f_S(s,\theta))\right)\frac{f_S(s,\theta)}{f_S(s,\theta)}\,ds \\ &= E[p(S(T))R(S(T),\theta)],\end{aligned}$$
where
$$R(s,\theta) = \frac{\frac{\partial}{\partial \theta}(e^{-rT}f_S(s,\theta))}{f_S(s,\theta)}.$$
If θ is not r of T, we can further write
$$R(s,\theta) = e^{-rT}\frac{\partial}{\partial \theta}\ln f_S(s,\theta).$$

Since according to our assumptions $S(T)$ is log-normally distributed, we have

$$f_S(s) = \frac{1}{s\sigma\sqrt{2\pi T}} \exp\left(-\frac{(\ln \frac{s}{S_0} - (r - \frac{\sigma^2}{2})T)^2}{2\sigma^2 T} \right).$$

Thus for $\theta = S_0$ the likelihood ratio method leads to the formula

$$\frac{\partial H}{\partial S_0} = E[e^{-rT} p(S(T)) \frac{\ln(S(T)/S_0) - (r - \sigma^2/2)T}{S_0 \sigma^2 T}].$$

Consider two European options: the usual call option and so called b inary call option with the payoff function

$$p(s) = \begin{cases} 1, & s \geq E, \\ 0, & s < E. \end{cases}$$

Let $T = 0.5$, $S_0 = 100$, $E = 100$, $\sigma = 0.5$, $r = 0.05$, $D = 0$.

- Task1: Compute the derivative of the usual call option with respect t o S_0 with both methods with accuracy 1%. Compare the number of generated stock prices.
- Task2: Compute the derivative of the binary call option with respect to S_0 with accuracy 1% by using a suitable method described in the lab.

Homework 8 (Deadline May 25, 2012) Find the derivatives of both us ual and binary call options with respect to σ with accuracy 1% with all suitable methods (including the method of Lab 14). Compare the numbers of generated stock prices. Which meth od is the best for each option type?

Simulation Methods in Financial Mathematics
Computer Lab 16

The aim of this lab is to practice using MC for pricing american options.

In this lab we consider pricing an American put option in the case of Heston market model

$$dS(t) = S(t)((r - D)\,dt + \sqrt{V(t)}\,dB_1(t)),$$
$$dV(t) = \kappa \cdot (\theta - V(t))\,dt + \xi\sqrt{V(t)}\,dB_2(t),$$

where B_1 and B_2 are uncorrelated Brownian motions. Recall that American options give the holder the right to exercise the option at any time before the maturity date T. It can be shown that the price of option can be computed as

$$price = \max_{\tau} \mathbb{E}[e^{-r\tau)}p(S(\tau))],$$

where τ is so called stopping time (exercise strategy, that uses information only from the past).

We consider the case $S(0) = 100$, $r = 0.05$, $D = 0$, $T = 0.5$, $E = 100$, $\kappa = 1$ $\xi = 0.1$, $\theta = 0.36$, $V(0) = 0.4$. For pricing options we discretize the model by choosing integer m, defining $t_i = \frac{iT}{m}$ and writing

$$S_{i+1} = S_i(1 + (r - D)\,\frac{T}{m} + \sqrt{V_i}\,X_{1,i}),$$
$$V_{i+1} = V_i + \kappa \cdot (\theta - V_i)\frac{T}{m} + \xi\sqrt{V_i}\,X_{2,i},$$

where $X_{1,i}$ and $X_{2,i}$ are independent, normally distributed, with variance $\frac{T}{m}$.

For pricing an American option we replace it with so-called Bermudan option, for which it is possible to sell the option only at time moments $t_i = i \cdot \frac{T}{m}$, $i = 1, 2, \ldots, m$.

Denote by W_i the value of option at $t = t_i$, then $W_m = p(S(T))$. Let C_i denote the continuation value of option at $t = t_i$ (the value after we decide not to exercise), then

$$C_i = \mathbb{E}(e^{-r\frac{T}{m}}W_{i+1} \mid \mathcal{F}_i),$$

where $\mathcal{F}_i$ is the information available at time t_i. If we know the continuation values of an Bermudan option at any time moment t_i, then it defines the optimal exercise strategy: exercise the option at the first time t_i for which the payoff value $p(S(t_i))$ is larger than the continuation value.

A numerical method using the observations above is Longstaff-Schwartz method:

1. Simulate n trajectories for stock prices and variances for time moments t_i, $i = 0, \ldots, m$.

2. For each time moment assume a form for the continuation value

$$C_i(\mathbf{x}) = \sum_{q=0}^{k} c_q\phi_q(\mathbf{x}),$$

where ϕ_q are basis functions (polynomials, for example) and $\mathbf{x}$ corresponds to the state of the market model (pair of S and V values in the case of Heston model).

3. Define $W_{m,j} = p(S_{m,j})$ for j-th trajectory

4. at each time moment t_i, $i = 1, 2, \ldots, m$ estimate the parameters of C_i by linear regression for $Y = e^{-r\frac{T}{m}} W_{i+1,j}$, using only the trajectories that satisfy $p(S_{i,j}) > 0$.

5. Define
$$W_{i,j} = \begin{cases} p(S_{i,j}), & \text{if } p(S_{i,j}) > C_i(\mathbf{x}_{i,j}), \\ e^{-r\frac{T}{m}} W_{i+1,j}, & \text{otherwise.} \end{cases}$$
Here $\mathbf{x}_{i,j}$ denotes the state of the market model at time $t = t_i$ for j-th trajectory.

6. compute the approximate price as the average of $e^{-r\frac{T}{m}} W_{1,j}$; also estimate the MC error using the variance of those numbers.

Task1 Write a procedure, that for a given n generates n trajectories of the stock prices and squared volatilities for Heston market model.

Task2 Assume
$$C_i(s, v) = c_0 + c_1(s - E/2) + c_2(v - \theta) + c_3(s - E/2)^2 + c_4(s - E/2)(v - \theta) + c_5(v - \theta)^2.$$
Generate $n = 10000$ trajectories and find (using the command `lm()` the coefficients of the continuation value function for $t = t_{m-1}$. Plot the graph of the continuation value function.

Task3 Implement Longstaff-Schwartz algorithm for pricing the option. Compute also the error estimate.

Task4 Use Longstaff-Schwartz algorithm for pricing an American straddle option with payoff
$$p(s) = |s - E|.$$
Assume the Black-Scholes market model with parameters $S(0) = 52$, $r = 0.05$, $D = 0$, $\sigma = 0.5$, $E = 50$, $T = 0.5$. Implement the method so that it is possible to give as the arguments the number of basis functions, the number of timesteps m and the number of trajectories n and to get the approximate option price with MC error estimate as the answer (there still remains the error that depends on m, it is not necessary to estimate that).

A method for deriving numerical methods for ordinary and stochastic differential equations

There are many ideas that can be used for deriving numerical methods for differential equations. We'll discuss one of them, namely integral expansion method.

Let us start from an ordinary differential equation

$$y'(t) = f(y(t), t).$$

We can rewrite it in a differential form:

$$dy(t) = f(y(t), t)dt.$$

For deriving a numerical method, we consider a small interval $[t_i, t_{i+1}]$ and use the differential equation to approximate the value $y(t_{i+1})$ in terms of the known value $y_i = y(t_i)$. Integrating the differential form ower the small interval, we get

$$y(t_{i+1}) - y_i = \int_{t_i}^{t_{i+1}} f(y(t), t)\, dt. \tag{1}$$

As any differentiable function $z(t)$ can be written as

$$z(t) = z(t_i) + \int_{t_i}^{t} dz(\tau)$$

we can write

$$f(y(t), t) = f(y_i, t_i) + \int_{t_i}^{t} d[f(y(\tau, \tau)]$$

and hence, after substituting this into (1) we have

$$y(t_{i+1}) = y_i + f(y_i, t_i)(t_{i+1} - t_i) + \int_{t_i}^{t_{i+1}} \left(\int_{t_i}^{t} d[f(y(\tau, \tau)] \right) dt. \tag{2}$$

If we stop at this point and throw away the double integral term, we get Euler's method

$$y_{i+1} = y_i + f(y_i, t_i)(t_{i+1} - t_i).$$

If we want a more accurate method, we should continue working with the double integral. Since

$$\begin{aligned} d[f(y(\tau, \tau)] &= \frac{\partial f}{\partial y}(y(\tau), \tau) dy(\tau) + \frac{\partial f}{\partial t}(y(\tau), \tau) d\tau \\ &= \left(\frac{\partial f}{\partial y}(y(\tau), \tau) f(y(\tau), \tau) + \frac{\partial f}{\partial t}(y(\tau), \tau) \right) d\tau \end{aligned}$$

we can denote

$$g(y, t) = \frac{\partial f}{\partial y}(y, t) f(y, t) + \frac{\partial f}{\partial t}(y, t)$$

and write

$$\begin{aligned} \int_{t_i}^{t} d[f(y(\tau, \tau)] &= \int_{t_i}^{t} g(y(\tau), \tau)\, d\tau \\ &= \int_{t_i}^{t} \left(g(y_i, t_i) + \int_{t_i}^{\tau} d[g(y(w), w)] \right) d\tau \\ &= g(y_i, t_i)(t - t_i) + \int_{t_i}^{t} \left(\int_{t_i}^{\tau} d[g(y(w), w)] \right) d\tau. \end{aligned}$$

Substituting it into (2) we get

$$y(t_{i+1}) = y_i + f(y_i,t_i)(t_{i+1}-t_i) + g(y_i,t_i)\frac{(t_{i+1}-t_i)^2}{2} + \int_{t_i}^{t_{i+1}} \left(\int_{t_i}^{t} \left(\int_{t_i}^{\tau} d[g(y(w),w)] \right) d\tau \right) dt. \quad (3)$$

If we throw away the triple integral, we get a second order method

$$y_{i+1} = y_i + f(y_i,t_i)(t_{i+1}-t_i) + \left(\frac{\partial f}{\partial y}(y_i,t_i) f(y_i,t_i) + \frac{\partial f}{\partial t}(y_i,t_i) \right) \frac{(t_{i+1}-t_i)^2}{2}.$$

In principle, we can continue expanding the triple integral to get a third order method and so on.

Similar ideas can be used for constructing numerical methods for stochastic differential equations but we have to use Itô's formula for computing differentials of stochastic processes.

Let us consider a stochastic differential equation of the form

$$d(Y(t)) = \alpha(t,Y(t))\,dt + \beta(t,Y(t))\,dB(t), \quad (4)$$

where $B(t)$ is the standard Brownian motion. By integrating over the interval $[t_i, t_{i+1}]$ we get

$$Y(t_{i+1}) - Y(t_i) = \int_{t_i}^{t_{i+1}} \alpha(t,Y(t))\,dt + \int_{t_i}^{t_{i+1}} \beta(t,Y(t))\,dB(t). \quad (5)$$

Now we have two integral terms and have to use expansions in both of them. In order to make expressions shorter, let us define operators (functions that take a function of two arguments as input and return another function of two arguments for the result) L_1 and L_2 by

$$(L_1 f)(t,y) = \frac{\partial f}{\partial t}(t,y) + \alpha(t,y)\frac{\partial f}{\partial y}(t,y) + \frac{\beta(t,y)^2}{2}\frac{\partial^2 f}{\partial y^2}(t,y)$$

and

$$(L_2 f)(t,y) = \beta(t,y)\frac{\partial f}{\partial y}(t,y),$$

where f is an arbitrary twice differentiable function of two variables t and y. Then Itô's formula for the process $f(t,Y(t))$ can be written as

$$df(t,Y(t)) = (L_1 f)(t,Y(t))\,dt + (L_2 f)(t,Y(t))\,dB(t)$$

wich, after integrating from t_i to t gives

$$f(t,Y(t)) = f(t_i,Y(t_i)) + \int_{t_i}^{t} (L_1 f)(z,Y(z))\,dz + \int_{t_i}^{t} (L_2 f)(z,Y(z))\,dB(z). \quad (6)$$

By using (6) in the cases $f = \alpha$ and $f = \beta$, we can write (5) as

$$Y(t_{i+1}) - Y(t_i) = \alpha(t_i,Y(t_i))\,\Delta t + \beta(t_i,Y(t_i))(B(t_{i+1}) - B(t_i)) + R, \quad (7)$$

where $\Delta t = t_{i+1} - t_i$ and

$$R = \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} (L_1\alpha)(z,Y(z))\,dz\,dt + \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} (L_2\alpha)(z,Y(z))\,dB(z)\,dt + \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} (L_1\beta)(z,Y(z))\,dz\,dB(t) + \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} (L_2\beta)(z,Y(z))\,dB(z)\,dB(t). \quad (8)$$

By throwing away terms with double integrals (the remainder R), we get Euler's method for stochastic differential equations.

By analysing the double integral terms one can show that the largest error is caused by trowing away the last term, where both integrals are with respect to Brownian motions. We can approximate this term better by expanding it. Applying (6 to $L_2\beta$, we get

$$\int_{t_i}^{t_{i+1}} \int_{t_i}^{t} (L_2\beta)(z, Y(z))\, dB(z)\, dB(t) = (L_2\beta)(t_i, Y(t_i)) \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} dB(z)\, dB(t) + \tilde{R},$$

where $\tilde{R}$ has triple integral terms. Since according to Itô's formula we have

$$\frac{1}{2} d\left((B(t) - B(t_i))^2 - t\right) = (B(t) - B(t_i))\, dB(t),$$

we get

$$(L_2\beta)(t_i, Y(t_i)) \int_{t_i}^{t_{i+1}} \int_{t_i}^{t} dB(z)\, dB(t) = \frac{1}{2}(L_2\beta)(t_i, Y(t_i))((B(t_{i+1}) - B(t_i))^2 - \Delta t).$$

If we add this term to Euler's method, we get Milstein's method for solving stochastic differential equations:

$$Y_{i+1} = Y_i + \alpha(t_i, Y_i)\, \Delta t + \beta(t_i, Y_i) X_{i+1} + \frac{1}{2}(L_2\beta)(t_i, Y_i)(X_{i+1}^2 - \Delta t). \tag{9}$$

If we use the formula (6 for expanding all terms of R and throw away triple integral terms, we get a method

$$\begin{aligned} Y_{i+1} &= Y_i + \alpha(t_i, Y_i)\, \Delta t + \beta(t_i, Y_i) X_{i+1} \\ &+ \frac{1}{2}\left[(L_1\alpha)(t_i, Y_i)\Delta t^2 + (L_2\alpha + L_1\beta)(t_i, Y_i)\Delta t\, X_{i+1} + (L_2\beta)(t_i, Y_i)(X_{i+1}^2 - \Delta t)\right], \end{aligned} \tag{10}$$

where $X_i \sim N(0, \sqrt{\Delta t})$ are independent random variables.

```
#exercise 1
n=100
X=runif(n)
Y=X**2 #or Y=X^2
EY=mean(Y)
#estimate the error
alpha=0.1 #the prob. of actual error being larger than the estimate
estimate=-qnorm(alpha/2)*sd(Y)/sqrt(n)
#exercise 2
MC1=function(g,Xgen,n,alpha){
    X=Xgen(n)
    Y=g(X)
    EY=mean(Y)
    estimate=-qnorm(alpha/2)*sd(Y)/sqrt(n)
    return(c(EY,estimate))
}
f=function(x){return(x^2)}
MC1(f,runif,100,0.1) #Yes, it is correct, the exact anser is 1/3 and the actual error is
smaller than estimate.
#Exercise 3
n=1000
alpha=0.1
exact=1/3
N=100 #we repeat the computation N times
average=rep(0,N)
error_estimate=rep(0,N)
actual_error=rep(0,N)
for(i in 1:N){
    result=MC1(f,runif,n,alpha)
    average[i]=result[1]
    error_estimate[i]=result[2]
    actual_error[i]=abs(result[1]-exact) #error is abs value of difference
}
#exercise 4
MC2=function(g,Xgen,error,alpha){
    N=10000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}
#try out
MC2(f,runif,0.001,0.05)
#can use other functions and generators
```

```
MC2(sin,rexp,0.01,0.05) #E(sin(X)), X has standard exponential distribution (lambda=1)
#if we want to use nonstandard generator, then we have to define corresponding function of
one variable
#runif can take more arguments, see ?runif
?runif
gen=function(n){return(runif(n,min=1,max=100))} #generates uniformly distributed random
numbers between 1 and 100
g=function(x){return(x*sin(x))}
MC2(g,gen,0.1,0.05)
```

```r
#Exercise 1
#first try: generating one trajectory of the Brownian motion
T=0.5
m=100 #the number of time steps
B=rep(0,m+1) #make a place for values corresponding to t=0, T/m,2*T/m,...,T
dt=T/m
for(i in 2:(m+1)){ #here we can decide what the variable of the cycle denotes. In the code here it denotes the index of the component of the vector of B values we are going compute
    Xi=rnorm(1,sd=sqrt(dt)) #or Xi=sqrt(dt)*rnorm(1)
    B[i]=B[i-1]+Xi #the current value is the previous one plus the random increment
}
#an alternate form, where i denotes the value that is known and in the cycle we compute the next one:
#for(i in 1:m){
#    Xi=sqrt(dt)*rnorm(1)
#    B[i+1]=B[i]+Xi
#}
t=seq(0,T,length.out=m+1) #t=(0:m)/m*T
plot(t,B,type="l")
#modification to get M trajectories together
M=10
B=matrix(0,nrow=m+1,ncol=M) #a pace for the values of the brownian motion. In each column of the matrix there is going to be one trajectory
for(i in 2:(m+1)){
    Xi=rnorm(M,sd=sqrt(dt)) # one value for each trajectory
    B[i,]=B[i-1,]+Xi #computes new values of all trajectories by using previous values (in the previous row) and adding a different random increment to each one
}
matplot(t,B,type="l")

#exercise 2
#let us use previous brownian motion values, so we have to compute just the corresponding stock prices
S0=100 #the starting price
S=matrix(S0,nrow=m+1,ncol=M) # a place of the values of the stock prices. For each time moment is different row, columns correspond to the price trajectories
mu=0.1
sigma=0.5
T=0.5
for(i in 2:(m+1)){
    time=(i-1)*dt # the first row corresponds to t=0, second row to t=dt, third one to t=2*dt etc
    S[i,]=S0*exp((mu-sigma**2/2)*time+sigma*B[i,]) # The i-th row is computed using the i-th row of the Brownian motion
}
matplot(t,S,type="l")


#Exercise 3

Put=function(S,E,T,r,sigma,D,t=0){
     tau=T-t
     d1=(log(S/E)+(r-D+sigma**2/2)*tau)/(sigma*sqrt(tau))
     d2=d1-sigma*sqrt(tau)
     value=-S*exp(-D*tau)*pnorm(-d1)+E*exp(-r*tau)*pnorm(-d2)
     return(value)
}
```

```
Call=function(S,E,T,r,sigma,D,t=0){
     tau=T-t
     d1=(log(S/E)+(r-D+sigma**2/2)*tau)/(sigma*sqrt(tau))
     d2=d1-sigma*sqrt(tau)
     value=S*exp(-D*tau)*pnorm(d1)-E*exp(-r*tau)*pnorm(d2)
     return(value)
}
r=0.03
sigma=0.4
E=100
T=1
D=0
Call(100,E,T,r,sigma,D) #compute just one value to see if the function is working correctly
Put(100,E,T,r,sigma,D) #the same for the put option

n=200 #how many points use for drawing the graph
S=seq(0,200,length.out=n) #make a vector of the stock prices
plot(S,Call(S,E,T,r,sigma,D),type="l") #here Call computes option prices for all stock prices

#defining payoff functions
p_put=function(S,E){
    pmax(E-S,0) #note that the function is pmax (from parallel maximum), not just max
}
matplot(S,cbind(Put(S,E,T,r,sigma,D),p_put(S,E)),type="l") #this is one way to put the
graphs of many functions to one picture
```

```
#exercise 1
S_euler=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        S[i+1,]=S[i,]*(1+mu*dt+sigma*Xi)
    }
    return(S[m+1,])
}

#define parameters
m=40
S0=50
E=50
T=0.5
D=0.1
r=0.03
sigma=0.7
mu=r-D
MCerror=0.01
alpha=0.05
#the function for computing expected values with MC method from the first lab
MC2=function(g,Xgen,error,alpha){
    N=10000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}

#for pricing an European option, we define the payoff function
p_put=function(S,E){
    pmax(E-S,0) #note that the function is pmax (from parallel maximum), not just max
}
#option price is the expected value of the discounted payoff, so define the corresponding
function
g=function(x){
        return(exp(-r*T)*p_put(x,E))#the values of the parameters r,T,E have to be defined
        outside of any function
}
#we also need to define a generator that for a given n computes n random variables (final
stock prices S(T) for European options)
generator=function(n){
```

```
    return(S_euler(n,S0,m,T,mu,sigma))
}
#now we can use the function MC2 to get an approximate option price
answer=MC2(g,generator,MCerror,alpha)


########################################Exercise 2
#When the market parameters are constant, there is an exact formula for S(T) in therms of
the final value B(T) of the brownian motion
#we want to see how different is the value computed by Euler's method from the exact value
S_euler_error=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    B=matrix(0,nrow=m+1,ncol=n) #to keep track of the values of Brownian motion
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt)) #the increment of the brownian motion
        B[i+1,]=B[i,]+Xi
        S[i+1,]=S[i,]*(1+mu*dt+sigma*Xi) #Euler's method uses the increments of the path of
        the Brownian motion
    }
    BT=B[m+1,] #for comparison we have to compute the exact value for the same brownian
    motion, so we need the final value
    S_exact=S0*exp((mu-sigma**2/2)*T+sigma*BT)# the exact value of S(T)
    return(S[m+1,]-S_exact)
}
#check if the code works
S_euler_error(n=10,S0,m=5,T,mu,sigma)
#we are actually interested in the mean absolute error
m=5
generator=function(n){
    return(S_euler_error(n,S0,m,T,mu,sigma))
}
error_5=MC2(abs,generator,error=0.01,alpha)
error_5
#if we use just 5 time steps to approximate the final value of S, the obtained result is not
very accurate
#let us see, how the difference between the exact and approximate stock prices change, when
we change m
m=10
error_10=MC2(abs,generator,error=0.01,alpha)
#OK, it got smaller. But how the error depends on m?
#for this we fit the curve c/m**q for the error. The parameter q is called the rate of
convergence
#large rate means that we usually get good results for smaller values of m
#To find the rate, we do many computations
m_values=c(5,10,20,40,80)
errors=rep(0,length(m_values))
for(i in 1:length(m_values)){
    m=m_values[i]
    errors[i]=MC2(abs,generator,error=0.01,alpha)[1]
    print(m)
}
#it turns out that instead of fitting c/m**q to errors we get a more accurate value if we
take logarithm before fitting
nls(log(errors)~log(c)-q*log(m_values),start=list(c=1,q=1))
#so the rate is approximately 0.5. This means, that if we want to reduce the error in S
values 2 times, we have to multiply the current value of m by four
```

```
#this is quite slow convergence, usually a large value of m is needed for the error to be
small
#the rate of convergence of the S values is called the strong convergence rate

##########################Exercise 3
#If we use Euler's method for generating values of S(T) in option pricing, we introduce an
additional error that depends on the value of m (the number of time steps)
#So, if we fix m and use MC2 to compute an option price, we usually do not get the exact
value of the price even when we let the error given to MC2 to go to 0
#We are interested, how the part of the error that comes from fixing a value of m behaves
#If we assume a BS market model with constant coefficients, we know the exact option price
#So we can study how the difference between the computed price and the exact price goes to 0
#by fitting the curve c/m**q to this difference gives us the weak convergence rate
#it is called weak rate because option prices can converge well even if the stock prices do
not converge very fast
#weak convergence rate is at least as large as the strong convergence rate, but can be larger

#the function for put:
Put=function(S,E,T,r,sigma,D,t=0){
     tau=T-t
     d1=(log(S/E)+(r-D+sigma**2/2)*tau)/(sigma*sqrt(tau))
     d2=d1-sigma*sqrt(tau)
     value=-S*exp(-D*tau)*pnorm(-d1)+E*exp(-r*tau)*pnorm(-d2)
     return(value)
}

#compute the exact value
exact=Put(S0,E,T,r,sigma,D,t=0) #the exact value of the put option
m_values=c(3,6,12,24)
errors=rep(0,length(m_values))
#we need the generator for stock prices
generator=function(n){
    return(S_euler(n,S0,m,T,mu,sigma))
}
#the function g is the same as in the exercise 1
for(i in 1:length(m_values)){
    m=m_values[i]
    errors[i]=MC2(g,generator,MCerror,alpha)[1]-exact
    print(m) #to see how many computations are finished. In R console you should choose menu
    Misc und uncheck Buffered Output
}
#check if the errors are clearly larger than MCerror
errors
#yes, MC error is 0.01, but the computed errors are above 0.05. So we can assume that the
numbers we see describe accurately the part of the error that comes from m
#What is the convergence rate?
nls(log(errors)~log(c)-q*log(m_values),start=list(c=1,q=1))
#it is higher than the weak convergence rate. I got 0.86. It can be proved that actually it
is 1. To see that, we should do computations with larger value of m and smaller value of MC
error, but it
#takes too much time.
```

```
m=ceiling(2*Cbar/total_error)
```

```
MC2=function(g,Xgen,error,alpha){
    N=10000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}


#exercise 1

#generator for constant volatility
#Euler's method (actually not needed for this lab)
S_euler=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        S[i+1,]=S[i,]*(1+mu*dt+sigma*Xi)
    }
    return(S[m+1,])
}
#Milstein's method, constant volatility
S_Milstein=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        S[i+1,]=S[i,]*(1+mu*dt+sigma*Xi+1/2*sigma**2*(Xi**2-dt))
    }
    return(S[m+1,])
}
#weakly second order method, constant mu and sigma
#define first functions needed in the method
L1mu=function(t,s){
   return(s*mu**2)
}
L2mu=function(t,s){
   return(s*sigma*mu)
}
L1sigma=function(t,s){
   return(s*mu*sigma)
```

```
}
L2sigma=function(t,s){
   return(s*sigma**2)
}
#define the method. Since functions work only if sigma and mu are correctly defined outside
of the funciton
#write this function also so that it takes sigma and mu from outside (do not use
corresponding arguments)
S_secondorder=function(n,S0,m,T){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        term1=S[i,]*(1+mu*dt+sigma*Xi)
        term2=1/2*(L1mu(t[i],S[i,])*dt**2)
        term2=term2+1/2*(L2mu(t[i],S[i,])+L1sigma(t[i],S[i,]))*dt*Xi
        term2=term2+1/2*L2sigma(t[i],S[i,])*(Xi**2-dt)
        S[i+1,]=term1+term2 #can also be written as one long expression
    }
    return(S[m+1,])
}



#-------------------------------------------------------------------------------------
#exercise 2 : finding strong convergence rates

#define functions that compute the differences between approximate final stock prices and
#exact stock prices for the same Brownian motion
#similar to what we did in Lab 3
S_Milstein_error=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    B=matrix(0,nrow=m+1,ncol=n)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        S[i+1,]=S[i,]*(1+mu*dt+sigma*Xi+1/2*sigma**2*(Xi**2-dt))
        B[i+1,]=B[i,]+Xi
    }
    BT=B[m+1,] #for comparison we have to compute the exact value for the same brownian
    motion, so we need the final value
    S_exact=S0*exp((mu-sigma**2/2)*T+sigma*BT)# the exact value of S(T)
    return(S[m+1,]-S_exact)
}
#the same for the weakly second order method
S_secondorder_error=function(n,S0,m,T,mu,sigma){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    B=matrix(0,nrow=m+1,ncol=n)
    for(i in 1:m){
        Xi=rnorm(n,sd=sqrt(dt))
        term1=S[i,]*(1+mu*dt+sigma*Xi)
        term2=1/2*(L1mu(t[i],S[i,])*dt**2)
        term2=term2+1/2*(L2mu(t[i],S[i,])+L1sigma(t[i],S[i,]))*dt*Xi
```

```
        term2=term2+1/2*L2sigma(t[i],S[i,])*(Xi**2-dt)
        S[i+1,]=term1+term2
        B[i+1,]=B[i,]+Xi
    }
    BT=B[m+1,] #for comparison we have to compute the exact value for the same brownian
    motion, so we need the final value
    S_exact=S0*exp((mu-sigma**2/2)*T+sigma*BT)# the exact value of S(T)
    return(S[m+1,]-S_exact)
}
#define parameters
S0=100
T=0.5
mu=0.1
sigma=0.6
m_values=c(5,10,20,40,80)
errors=rep(0,length(m_values))
#Strong convergence rate for Milsteins method
#define generator of differences between exact and approximate prices
generator=function(n){
    return(S_Milstein_error(n,S0,m,T,mu,sigma))
}
#find mean absolute errors for different m
for(i in 1:length(m_values)){
    m=m_values[i]
    errors[i]=MC2(abs,generator,error=0.01,alpha=0.05)[1]
    print(m)
}
#find the convergence rate
nls(log(errors)~log(c)-q*log(m_values),start=list(c=1,q=1))
#observed rate is close to 1 (theoretically is equal to 1)

#the same for the second order method
#define generator of differences between exact and approximate prices
generator=function(n){
    return(S_secondorder_error(n,S0,m,T,mu,sigma))
}
m_values=c(5,10,20,40,80)
errors=rep(0,length(m_values))
for(i in 1:length(m_values)){
    m=m_values[i]
    errors[i]=MC2(abs,generator,error=0.01,alpha=0.05)[1]
    print(m)
}
nls(log(errors)~log(c)-q*log(m_values),start=list(c=1,q=1))
#rate close to 1, strongly first order (like Milsteins method)
#conclusions: both Milstein's and the weakly second order method
#approximate final stock prices better than Euler's method
#the weakly second order method does not approximate final stock prices significantly better
than Milstein's method

#-----------------------------------------------------------------------------
#exercise 3: weak convergence rates
#define data
E=100
T=1
r=0.05
S0=100
```

```
D=0
mu=r-D #when pricing options, we take the trend parameter to be r-D
sigma=0.5
#function for the put option price for constant sigma
Put=function(S,E,T,r,sigma,D,t=0){
     tau=T-t
     d1=(log(S/E)+(r-D+sigma**2/2)*tau)/(sigma*sqrt(tau))
     d2=d1-sigma*sqrt(tau)
     value=-S*exp(-D*tau)*pnorm(-d1)+E*exp(-r*tau)*pnorm(-d2)
     return(value)
}

#compute the exact value
exact=Put(S0,E,T,r,sigma,D,t=0)
m_values=c(3,6,12,24)
errors=rep(0,length(m_values))

#weak convergence of the Milstein's method
#we need the generator for stock prices
generator=function(n){
    return(S_Milstein(n,S0,m,T,mu=r-D,sigma))
}
p_put=function(S,E){
    pmax(E-S,0) #note that the function is pmax (from parallel maximum), not just max
}
#option price is the expected value of the discounted payoff, so define the corresponding
function
g=function(x){
        return(exp(-r*T)*p_put(x,E))#the values of the parameters r,T,E have to be defined
        outside of any function
}
for(i in 1:length(m_values)){
    m=m_values[i]
    errors[i]=MC2(g,generator,0.01,0.05)[1]-exact
    print(m) #to see how many computations are finished. In R console you should choose menu
    Misc und uncheck Buffered Output
}
#check if the errors are clearly larger than MCerror
errors
#yes, MC error is 0.01, but the computed errors are above 0.03. So we can assume that the
numbers we see describe accurately the part of the error that comes from m
#What is the convergence rate?
nls(log(abs(errors))~log(c)-q*log(m_values),start=list(c=1,q=1))
#close to 1, theoretically 1
#so Milsteins's method is not better than Euler's method when computing option prices. Weak
convergence rates are the same
#repeat for the second order method
#converges very fast, so can not use many values of m so that MC error is smaller than
observed error
m_values=c(1,2,4,8) #the theoretical convergence rate is valid for large enough m, but we
can not use large values of m because computations become too slow
errors=rep(0,length(m_values))

generator=function(n){
    return(S_secondorder(n,S0,m,T))
}
for(i in 1:length(m_values)){
```

```
    m=m_values[i]
    errors[i]=MC2(g,generator,0.01,0.05)[1]-exact
    print(m) #to see how many computations are finished. In R console you should choose menu
    Misc und uncheck Buffered Output
}
#check if the errors are clearly larger than MCerror
errors
#the last error is not more than 2 times larger than the MC error, so we should repeat the
last comptation with smaller MCerror
#since the computations were quite slow, we just drop the last value before estimating the
convergence rate ...
#of cause the estimate is not very accurate, if a small number of computations is used
errors=errors[-4] #everything except the fourth value
m_values=m_values[-4]
nls(log(abs(errors))~log(c)-q*log(m_values),start=list(c=1,q=1))
#the observed convergence rate was close to 2. The theoretical weak convergence rate is 2
#this method is better than Euler's and Milstein's methods in terms of the weak convergence
rates
#usually a relatively small m is needed to obtain a very high accuracy. Each time we double
the value of m, the error is reduced approximately 4 times

#hint for homework: define functions sigma, dsigma (derivative of sigma) and d2sigma (the
second derivative) as functions of s
#then define L1mu, L2mu, L1sigma,L2sigma using those functions
#also modify the definition of the method S_secondorder so that it works when sigma is a
function of s
```

```r
MC2=function(g,Xgen,error,alpha){
    N=1000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}
```

```
MC2=function(g,Xgen,error,alpha){
    N=1000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}


#exercise 1

#generator for constant volatility
#Euler's method (actually not needed for this lab)
S_importance=function(n,S0,T,mu,sigma,eta=0){
    BT=rnorm(n,sd=sqrt(T))
    S=S0*exp((mu+eta*sigma-sigma**2/2)*T+sigma*BT)
    return(cbind(S,BT))
}
g=function(X){
    return(exp(-r*T-eta*X[,2]-eta**2*T/2)*p_call(X[,1],E))
    }
S0=50
E=100
r=0.1
sigma=0.5
D=0.0
T=1

MCerror=0.01
p_call=function(S,E){
    return(pmax(S-E,0))
}
generator=function(n){return(S_importance(n,S0,T,mu,sigma,eta))}
print("search for the best eta value")
for(eta in seq(1,3,by=0.1)){
answer1=MC2(g,generator,MCerror,0.05)
print(c(eta,answer1[1],answer1[2]))
}

#exercise 2
S0=50
E=100
r=0.1
sigma=0.5
```

```
D=0.0
T=1
S_strat=function(n,S0,r,D,sigma,T,i,k){
    U=runif(n,min=(i-1)/k,max=i/k)
    BT=sqrt(T)*qnorm(U) #normally distributed under the condition, that the value is in Ai
    S=S0*exp((r-D-sigma**2/2)*T+sigma*BT)
    return(S)
}
p_call=function(S,E){
    return(pmax(S-E,0))
}
g=function(x){return(exp(-r*T)*p_call(x,E))}
MCerror=0.01
#use stratified sampling, simple approach
for(k in c(5,10,20,40)){
price=0#here we add the contribution of each stratum
n=0#the total number of generated values
generator=function(n){
   return(S_strat(n,S0,r,D,sigma,T,i,k))
   }
for(i in 1:k){
   answer=MC2(g,generator,sqrt(k)*MCerror,0.05) #the error in the stratum should be the
   desired error divided by the probability of the stratum.
   price=price+1/k*answer[1]
   n=n+answer[2]
}
print(c(price,k,n))
}
#as k increases, the number of generated variables gets smaller
#for this problem (out of the money option) the importance sampling with the best eta is
better than the simple stratified sampling
#but it is possible to use a better version of the stratified sampling or to combine the
methods ...
```

```
#exercise 1
MC_stratified=function(g,Xgen,C,p_i){#p_i is the probability of the stratum, C is a positive
constant
    N=100#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    sdY=1 #any opositive number to make sure that the while cycle starts
    n=0 #the number of values generated so far
    while(n<C*p_i*sdY){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
    }
    return(c(sum_y/n,n,sdY**2)) #the estimate of EY is sum_y/n, instead of the estimated
    variance we could return for example p_i times the standard deviation
}
#exercise 2
r=0.1
sigma=0.4
T=0.5
D=0
E=100
S0=100
###############
C=1000
alpha=0.05
k=10
S_strat=function(n,S0,r,D,sigma,T,i,k){
    U=runif(n,min=(i-1)/k,max=i/k)
    BT=sqrt(T)*qnorm(U) #normally distributed under the condition, that value is in Ai
    S=S0*exp((r-D-sigma**2/2)*T+sigma*BT)
    return(S)
}
price_C=function(C,g,k){
    generator=function(n){ #Since we want to change i inside the function, we have to define
    the generator here
        return(S_strat(n,S0,r,D,sigma,T,i,k))
    }
    price=0 #here we add the contribution of each stratum
    n=0#the total number of generated values
    DZ=0 #for error estimate
    for(i in 1:k){
        p_i=1/k #the probability of i-th stratum
        answer=MC_stratified(g,generator,C,p_i)
        price=price+1/k*answer[1]
        n=n+answer[2]
        DZ=DZ+p_i**2*answer[3]/answer[2] #if the MC_stratified returns
        p_i*standard_deviation, then the last term is answer[3]**2/answer[2]
    }
    error_estimate=-qnorm(alpha/2)*sqrt(DZ)
    return(c(price,error_estimate))
}
```

```
price_C(1000,g,10)
```

```
#task 1
Brown_stratified=function(k,m,T){#k-the number of stratums, m- the number of time steps,
T-the final time
    v=rep(1,m) #for the final value, we just sum the increments
    B=matrix(0,nrow=m+1,ncol=k) #trajectories are in columns
    a=sqrt(T/m) #the standard deviation of increments
    v_norm=sqrt(sum(v**2))
    for(i in 1:k){
        W=a*v_norm*qnorm(runif(1,min=(i-1)/k,max=i/k)) #the value for B(T) in the stratum
        Z=rnorm(m,sd=a)
        X=W/v_norm**2*v+Z-(t(v)%*%Z)/v_norm**2*v #increments of Brownian motion
        B[2:(m+1),i]=cumsum(X)
    }
    t=seq(0,T,length.out=m+1)
    matplot(t,B,type="l")
}
#task 2
#v has to be a vector of lenth m (the number of time steps)
S_euler_stratified=function(n,S0,m,T,mu,sigma,i,k,v){
    dt=T/m
    S=matrix(S0,nrow=m+1,ncol=n)
    t=seq(0,T,length.out=m+1)
    #compute the vector of increments
    a=sqrt(T/m) #the standard deviation of increments
    v_norm=sqrt(sum(v**2))
    W=a*matrix(v_norm*qnorm(runif(n,min=(i-1)/k,max=i/k)),nrow=1)
    #final values in the stratum
    Z=matrix(rnorm(m*n,sd=a),nrow=m)
    X=1/v_norm**2*v%*%W+Z-(v%*%t(v)%*%Z)/v_norm**2
    for(i in 1:m){
        S[i+1,]=S[i,]*(1+mu*dt+sigma(t=t[i],s=S[i,])*X[i,])
    }
    return(S[m+1,])
}
#now can use the method from the previous lab to price options
MC_stratified=function(g,Xgen,C,p_i){#p_i is the probability of the stratum, C is a positive
constant
    N=100#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    sdY=1 #any opositive number to make sure that the while cycle starts
    n=0 #the number of values generated so far
    while(n<C*p_i*sdY){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
    }
    return(c(sum_y/n,n,sdY**2)) #the estimate of EY is sum_y/n, instead of the estimated
    variance we could return for example p_i times the standard deviation
}
#data
r=0.06
D=0.03
```

```
T=0.5
S0=105
E=100
alpha=0.05
mu=r-D
sigma=function(t,s){
    return(95/(95+s))
}
#for call option
p_call=function(S,E){
    return(pmax(S-E,0))
}
g=function(x){return(exp(-r*T)*p_call(x,E))}

#the function for computing the price and error estimate for a given C by the optimal
stratified sampling
price_C=function(C,g,k){
    generator=function(n){ #Since we want to change i inside the function, we have to define
    the generator here
        return(S_euler_stratified(n,S0,m,T,mu,sigma,i,k,v))
    }
    price=0 #here we add the contribution of each stratum
    n=0#the total number of generated values
    DZ=0 #for error estimate
    for(i in 1:k){
        p_i=1/k #the probability of i-th stratum
        answer=MC_stratified(g,generator,C,p_i)
        price=price+1/k*answer[1]
        n=n+answer[2]
        DZ=DZ+p_i**2*answer[3]/answer[2] #if the MC_stratified returns
        p_i*standard_deviation, then the last term is answer[3]**2/answer[2]
    }
    error_estimate=-qnorm(alpha/2)*sqrt(DZ)
    return(c(price,error_estimate))
}
#compute price for a given m
m=10
v=rep(1,m) #v has to be of length m
C=1000
price_C(C,g,100)
#Computing with a given accuracy
total_error=0.01
m0=5
#First computation with given C
C=1000
m=m0
v=rep(1,m) #we have to change v whenever we change m in order to use the current generator
answer1=price_C(C,g,100)
V1=answer1[1]
m=2*m0
v=rep(1,m)
answer2=price_C(C,g,100)
V2=answer2[1]
#For optimal MC, the answers are computed with the same MC error eps0 as before
#This is not a problem, we just have to use the sum of estimated errors when we before used
2*eps0
#are the computations accurate enough to get a reasonably good estimate of the convergence
```

```
reate parameter C?
abs(V2-V1)>2*(answer1[2]+answer2[2])
#the result was FALSE, so we may get a quite bad estimate of m. So it may be a good idea to
make C larger (giving better estimates). Fut first check how large is the "bad" estimate of
m we get
#if it turns out to be relatively small, then it does not make sense to spend time for more
accurate estimates
#estimate C (valid for first order methods with weak convergence rate 1)
Cbar=2*m0*(abs(V1-V2)+answer1[2]+answer2[2])
m=ceiling(2*Cbar/total_error)
m
#I got 182, so it is quite large. Let us try to get a better estimate for m. The
computations were very farst, so we may multiply C by a relatively last number
#If we multiply C by x, the computation time increases approximately x times. For very farst
computations, we may try to multiply by relatively large x, say 100
C=100*1000
m=m0
v=rep(1,m) #we have to change v whenever we change m in order to use the current generator
answer1=price_C(C,g,100)
V1=answer1[1]
m=2*m0
v=rep(1,m)
answer2=price_C(C,g,100)
V2=answer2[1]
#For optimal MC, the answers are computed with the same MC error eps0 as before
#This is not a problem, we just have to use the sum of estimated errors when we before used
2*eps0
#are the computations accurate enough to get a reasonably good estimate of the convergence
reate parameter C?
abs(V2-V1)>2*(answer1[2]+answer2[2])
#the result was still FALSE, so we may still get a quite bad estimate of m. So it may be a
good idea to make C larger again (giving better estimates). Fut first check how large is the
"bad" estimate of m we get
#if it turns out to be relatively small, then it does not make sense to spend time for more
accurate estimates
#estimate C (valid for first order methods with weak convergence rate 1)
Cbar=2*m0*(abs(V1-V2)+answer1[2]+answer2[2])
m=ceiling(2*Cbar/total_error)
m
#now the estimate is not very large (I got 57). So it is perfectly OK to use that m for
final computation. But let us see, if we can get a better estimate with reasonable time
#Previous computations were not very fast, so let us multiply C by 4 only
C=4*100*1000
m=m0
v=rep(1,m) #we have to change v whenever we change m in order to use the current generator
answer1=price_C(C,g,100)
V1=answer1[1]
m=2*m0
v=rep(1,m)
answer2=price_C(C,g,100)
V2=answer2[1]
#For optimal MC, the answers are computed with the same MC error eps0 as before
#This is not a problem, we just have to use the sum of estimated errors when we before used
2*eps0
#are the computations accurate enough to get a reasonably good estimate of the convergence
reate parameter C?
abs(V2-V1)>2*(answer1[2]+answer2[2])
```

```
#the result is no TRUE
#let us estimate m
Cbar=2*m0*(abs(V1-V2)+answer1[2]+answer2[2])
m=ceiling(2*Cbar/total_error)
m
#It turned out for my current computations, that the more accurate estimate happened to be
the same as before (57), so we indeed did not gain much by the last repetition
#Let us compute the price with given accuracy. For this we have to use the m we found and
pick C so that MC error is less than total_error/2
v=rep(1,m) #need to redefine v. Let us keep C the same as in the last computation (we could
make it smaller if the last computations were very slow and the error part of the
#answer2 is smaller than total_error/2)
answer=price_C(C,g,100)
#the result was: price is 16.957282632, MC part of the error is 0.002405796
#the the answer is accurate enough (actually, the total error of the answer is smaller than
required, since MC part is less than total_error/2.
#if the answer was not accurate enough, we should redefine C bu multiplying it by
(answer[2]/(total_error/2))**2 or a slightly large number, to be on the safe side. I would
recommend to use an additional
#factor 0f 1.1
```

```
MC2=function(error,Xgenerator,g,alpha=0.05){
    n0=100000#the number of values to generate in one go
    n=n0
    X=Xgenerator(n0)
    Y=g(X)
    sum_y=sum(Y)
    sum_y2=sum(Y**2)
    sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
    error_estimate=-qnorm(alpha/2)*sd_estimate/sqrt(n)
   while(error_estimate>error){
        X=Xgenerator(n0)
        Y=g(X)
        sum_y=sum_y+sum(Y)
        sum_y2=sum_y2+sum(Y**2)
        n=n+n0
       sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
        error_estimate=-qnorm(alpha/2)*sd_estimate/sqrt(n)
    }
    return(c(sum_y/n,error_estimate,n))
}

gen_Asia1=function(S0,mu,sigma,T,m,n){
     dt=T/m
     S=S0
     A=0
     B=0
     for(i in 1:m){
        A=A+S/m
        dB=rnorm(n,sd=sqrt(dt))
        B=B+dB
        t=i*dt
        S=S0*exp((mu-sigma**2/2)*t+sigma*B)
     }
     return(cbind(S,A))
}
mvalues=c(5,10,20,40)
m=mvalues[1]
r=0.1
D=0
mu=r-D
sigma=0.4
T=0.5
S0=100
E=100
#exercise 1
#average strike
g=function(X){
    return(exp(-r*T)*pmax(X[,1]-X[,2],0))
}
gen=function(n){
    return(gen_Asia1(S0,mu,sigma,T,m,n))
}
price1=MC2(0.1,gen,g,0.05)
prices=rep(NA,length(mvalues))
MCerror=0.05
for(i in 1:length(mvalues)){
   m=mvalues[i]
```

```
   prices[i]=MC2(MCerror,gen,g,0.05)[1]
}
errors=prices[-length(mvalues)]-prices[-1]
errors
#find the convergence rate
nls(errors~C/mvalues[-length(mvalues)]**q*(1-1/2**q),start=list(C=1,q=1))
nls(log(errors)~log(C)-q*log(mvalues[-length(mvalues)])+log(1-1/2**q),start=list(C=1,q=1))
#weak convergence rate is 1

#computing option price with given accuracy
error=0.1
m=5
price1=MC2(MCerror,gen,g,0.05)[1]
m=10
price2=MC2(MCerror,gen,g,0.05)[1]
abs(price2-price1)>2*MCerror
#if TRUE, then OK, otherwise it may be a good idea to repeat
#the computations with smaller MCerror (therwise the final estimate of m may be too large)
Cbar=2*5*(abs(price2-price1)+2*MCerror)
m1=ceiling(Cbar/(error/2))
m=m1
final_price=MC2(error/2,gen,g,0.05)
```

```
MC2=function(error,Xgenerator,g,alpha=0.05){
    n0=100000#the number of values to generate in one go
    n=n0
    X=Xgenerator(n0)
    Y=g(X)
    sum_y=sum(Y)
    sum_y2=sum(Y**2)
    sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
    error_estimate=-qnorm(alpha/2)*sd_estimate/sqrt(n)
   while(error_estimate>error){
        X=Xgenerator(n0)
        Y=g(X)
        sum_y=sum_y+sum(Y)
        sum_y2=sum_y2+sum(Y**2)
        n=n+n0
       sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
        error_estimate=-qnorm(alpha/2)*sd_estimate/sqrt(n)
    }
    return(c(sum_y/n,error_estimate,n))
}

gen_Asia1=function(S0,mu,sigma,T,m,n){
     dt=T/m
     S=S0
     A=0
     B=0
     for(i in 1:m){
        A=A+S/m
        dB=rnorm(n,sd=sqrt(dt))
        B=B+dB
        t=i*dt
        S=S0*exp((mu-sigma**2/2)*t+sigma*B)
     }
     return(cbind(S,A))
}
gen_Asia2=function(S0,mu,sigma,T,m,n){
     dt=T/m
     S=S0
     A=0
     B=0
     for(i in 1:m){
        dB=rnorm(n,sd=sqrt(dt))
        A=A+S/m*(1+mu*T/(2*m)+sigma/2*dB)
        B=B+dB
        t=i*dt
        S=S0*exp((mu-sigma**2/2)*t+sigma*B)
     }
     return(cbind(S,A))
}
#exercise 1
m=20
r=0.05
D=0.02
mu=r-D
sigma=0.5
T=0.5
S0=49
```

```
E=50

gen_Asia2_strat=function(S0,mu,sigma,T,m,n,i,k,v){
     dt=T/m
     S=S0
     a=sqrt(dt)
     v=matrix(v,m,1)
     vnorm=sqrt(sum(v**2))
     W=matrix(a*vnorm*qnorm(runif(n,min=(i-1)/k,max=i/k)),nrow=1,ncol=n)#rnorm(1,sd=a*vnorm)
     Z=matrix(rnorm(m*n,sd=a),nrow=m,ncol=n)
     dB=1/vnorm**2*v%*%W+Z-v%*%t(v)%*%Z/vnorm**2
     A=0
     B=0
     for(j in 1:m){
        #dB=rnorm(n,sd=sqrt(dt))
        A=A+S/m*(1+mu*T/(2*m)+sigma/2*dB[j,])
        B=B+dB[j,]
        t=j*dt
        S=S0*exp((mu-sigma**2/2)*t+sigma*B)
     }
     return(cbind(S,A))
}


MC_strat_opt=function(Xgenerator_strat,g,pi,C){
    n0=100#the number of values to generate in one go
    n=n0
    X=Xgenerator_strat(n0)
    Y=g(X)
    sum_y=sum(Y)
    sum_y2=sum(Y**2)
    sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
    #error_estimate=-qnorm(alpha/2)*sd_estimate/sqrt(n)
   while(C*pi*sd_estimate>n){
        X=Xgenerator_strat(n0)
        Y=g(X)
        sum_y=sum_y+sum(Y)
        sum_y2=sum_y2+sum(Y**2)
        n=n+n0
       sd_estimate=sqrt(abs(sum_y2-sum_y**2/n)/(n-1))
    }
    return(c(sum_y/n,sd_estimate,n))
}
price1=function(C,k,g,v){
   Z=0
   DZ=0
   n=0
   gen=function(n){
   return(gen_Asia2_strat(S0,mu,sigma,T,m,n,i,k,v))}
   for(i in 1:k){
      pi=1/k
      tmp=MC_strat_opt(gen,g,pi,C)
      Z=Z+pi*tmp[1]
      n=n+tmp[3]
      DZ=DZ+pi**2*tmp[2]**2/tmp[3]
   }
   return(c(Z,-qnorm(alpha/2)*sqrt(DZ),n))
```

```
}




alpha=0.05

#average strike call
g=function(X){
    return(exp(-r*T)*pmax(X[,1]-X[,2],0))
}
gen=function(n){
    return(gen_Asia2(S0,mu,sigma,T,m,n))
}
price_mc=MC2(0.01,gen,g,0.05)
C=10000
v=rep(1,m)
price_opt=price1(C,40,g,v)
price_opt=price1(C*(price_opt[2]/0.01)^2,40,g,v)
v=1-1:m/m
price_opt2=price1(C,40,g,v)
price_opt2=price1(C*(price_opt2[2]/0.01)^2,40,g,v)
v=1:m/m
price_opt3=price1(C,40,g,v)
price_opt3=price1(C*(price_opt3[2]/0.01)^2,40,g,v)

#average price put
g=function(X){
    return(exp(-r*T)*pmax(E-X[,2],0))
}
gen=function(n){
    return(gen_Asia2(S0,mu,sigma,T,m,n))
}
price1_mc=MC2(0.01,gen,g,0.05)
C=10000
v=rep(1,m)
price_opt=price1(C,40,g,v)
price_opt=price1(C*(price_opt[2]/0.01)^2,40,g,v)
v=1-1:m/m
price_opt2=price1(C,40,g,v)
price_opt2=price1(C*(price_opt2[2]/0.01)^2,40,g,v)
v=1:m/m
price_opt3=price1(C,40,g,v)
price_opt3=price1(C*(price_opt3[2]/0.01)^2,40,g,v)
price1_mc
price_opt
price_opt2
price_opt3
```

```
#
b=2
k=11
n=32
digits=k%/%b^(0:(n-1))%%b
sum(digits/b**(1:n))
###many numbers in one go

k=1:11
digits=outer(k,b^(0:(n-1)),"%/%")%%b
digits%*%(1/b**(1:n))

Corput=function(n1,n2,b){
  n=32
  k=n1:n2
  digits=outer(k,b^(0:(n-1)),"%/%")%%b
  return(digits%*%(1/b**(1:n)))
}
 plot(Corput(1,200,2),Corput(1,200,3))
 plot(Corput(1,200,17),Corput(1,200,19))
#if b values get large, the unit square is not covered very well
#for relatively small number of points

#task 2
S_euler=function(n,S0,m,T,mu,sigma){
    S=S0
    dt=T/m
    for(i in 1:m){
             t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*sqrt(dt)*qnorm(runif(n)))#rnorm(n))
    }
    return(S)
}
sigma=function(s,t){
   return(0.6-0.5*exp(-0.01*s))
}
m=5
r=0.1
D=0
mu=r-D
T=0.5
S0=105

gen=function(n){
    S_euler(n,S0,m,T,mu,sigma)
}
g=function(S){
    return(exp(-r*T)*pmax(E-S,0))
}
E=100
N=100000
S=gen(N)
mean(g(S))
S_euler_qmc=function(n,S0,m,T,mu,sigma,b){
    S=S0
    dt=T/m
    for(i in 1:m){
```

```
            t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*sqrt(dt)*qnorm(Corput(1,n,b[i])))#rnorm(n))
    }
    return(S)
}
b=c(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53, 59, 61, 67, 71)
gen_qmc=function(n){
    S_euler_qmc(n,S0,m,T,mu,sigma,b)
}
S=gen_qmc(N)
mean(g(S))
```

```
#task 1
#.libPaths("h:/rlibs")
#install.packages("rngWELL",lib="h:/rlibs")
#install.packages("randtoolbox",lib="h:/rlibs")
#library(randtoolbox)
sobolpoints=sobol(200,10)
pairs(sobolpoints)
haltonpoints=halton(200,10)
pairs(haltonpoints)
#task 2
m=5
r=0.1
D=0
mu=r-D
T=0.5
S0=105
E=100
sigma=function(s,t){
   return(0.6-0.5*exp(-0.01*s))
}
#put option
g=function(S){
    return(exp(-r*T)*pmax(E-S,0))
}
#Euler's method for MC
S_euler=function(n,S0,m,T,mu,sigma){
    S=S0
    dt=T/m
    for(i in 1:m){
             t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*sqrt(dt)*qnorm(runif(n)))#rnorm(n))
    }
    return(S)
}
S_euler_qmc=function(n,S0,m,T,mu,sigma,qrn_gen){
    S=S0
    dt=T/m
    dB=sqrt(dt)*qnorm(qrn_gen(n,m)) #matrix of increments, trajectories in rows
    for(i in 1:m){
             t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*dB[,i])
    }
    return(S)
}
m=5
mcError=rep(NA,10)
qmcError=rep(NA,10)
exact=7.731
for(i in 1:10){
    n=i*10000
    mcError[i]=abs(mean(g(S_euler(n,S0,m,T,mu,sigma))-exact))
    qmcError[i]=abs(mean(g(S_euler_qmc(n,S0,m,T,mu,sigma,halton))-exact))
}

m=20
mcError=rep(NA,10)
qmcError=rep(NA,10)
```

```
exact=7.577
for(i in 1:10){
    n=i*10000
    mcError[i]=abs(mean(g(S_euler(n,S0,m,T,mu,sigma))-exact))
    qmcError[i]=abs(mean(g(S_euler_qmc(n,S0,m,T,mu,sigma,sobol))-exact))
}




#
b=2
k=11
n=32
digits=k%/%b^(0:(n-1))%%b
sum(digits/b**(1:n))
###many numbers in one go

k=1:11
digits=outer(k,b^(0:(n-1)),"%/%")%%b
digits%*%(1/b**(1:n))

Corput=function(n1,n2,b){
  n=32
  k=n1:n2
  digits=outer(k,b^(0:(n-1)),"%/%")%%b
  return(digits%*%(1/b**(1:n)))
}
 plot(Corput(1,200,2),Corput(1,200,3))
 plot(Corput(1,200,17),Corput(1,200,19))
#if b values get large, the unit square is not covered very well
#for relatively small number of points

#task 2
S_euler=function(n,S0,m,T,mu,sigma){
    S=S0
    dt=T/m
    for(i in 1:m){
             t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*sqrt(dt)*qnorm(runif(n)))#rnorm(n))
    }
    return(S)
}
m=5
r=0.1
D=0
mu=r-D
T=0.5
S0=105

gen=function(n){
    S_euler(n,S0,m,T,mu,sigma)
}
g=function(S){
    return(exp(-r*T)*pmax(E-S,0))
}
E=100
N=100000
```

```
S=gen(N)
mean(g(S))
S_euler_qmc=function(n,S0,m,T,mu,sigma,b){
    S=S0
    dt=T/m
    for(i in 1:m){
            t=(i-1)*dt
        S=S*(1+mu*dt+sigma(S,t)*sqrt(dt)*qnorm(Corput(1,n,b[i])))#rnorm(n))
    }
    return(S)
}
b=c(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53, 59, 61, 67, 71)
gen_qmc=function(n){
    S_euler_qmc(n,S0,m,T,mu,sigma,b)
}
S=gen_qmc(N)
mean(g(S))
```

```
MC2=function(g,Xgen,error,alpha){
    N=1000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    while(error_estimate>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
    }
    return(c(sum_y/n,n)) #the estimate of EY is sum_y/n
}
```

```
S0=100
E=100
T=0.5
D=0
r=0.05
sigma=0.5

#Task 1
#modification of MC so that answer is computed with a given relative error
MC2_relative=function(g,Xgen,error,alpha){
    N=1000#the number of variables to be generated in one go
    sum_y2=0 #the sum of the squares of all generated Y values
    sum_y=0 # the sum of the Y values generated so far
    error_estimate=error+1# Make the estimate to be large than the given error to start the
    while cycle
    n=0 #the number of values generated so far
    EY=1 #to make sure the while cycle starts
    while(error_estimate/abs(EY)>error){
        X=Xgen(N) #new set of X values
        Y=g(X) #corresponding Y values
        n=n+N # the total number of generated values is increased by N
        sum_y=sum_y+sum(Y) #total sum of Y values
        sum_y2=sum_y2+sum(Y**2) #the sum of squares of Y
        sdY=sqrt(abs(sum_y2-sum_y**2/n)/(n-1)) #estimate of the standard deviation of Y
        based on all generated values
        error_estimate=-qnorm(alpha/2)*sdY/sqrt(n) #the estimate of the error of the mean
        value of all generated Y values
        EY=sum_y/n
    }
    return(c(EY,n)) #the estimate of EY is sum_y/n
}

gener=function(n,S0,sigma){#generator of stock prices according to the exact formula
    B=sqrt(T)*rnorm(n)
    S=S0*exp((r-D-(sigma^2)/2)*T+sigma*B)
    return(S)
}

gen=function(n){return(gener(n,S0,sigma))}
g=function(S){return(exp(-r*T)*(S>=E)*(S/S0))} #for pathwise derivative

MC2_relative(g,gen, 0.01, 0.05)

#for likelyhood ratio method
g=function(S){exp(-r*T)*pmax(S-E,0)*(log(S/S0)-(r-0.5*sigma^2)*T)/(S0*(sigma^2)*T)}
MC2_relative(g,gen, 0.01, 0.05)


#task 2
# for binary option only the likelyhood ratio method from the current lab is suitable, since
the payoff function is not differentiable
g=function(S){exp(-r*T)*(S-E>=0)*(log(S/S0)-(r-0.5*sigma^2)*T)/(S0*(sigma^2)*T)}
MC2_relative(g,gen, 0.01, 0.05)
```

```
#lab 15
#pricing American options
#task 1
trajectories=function(n){
    dt=T/m
    S=matrix(S0,nrow=n,ncol=m+1)
    V=matrix(V0,nrow=n,ncol=m+1)
    for(i in 1:m){
       S[,i+1]=S[,i]*(1+(r-D)*dt+sqrt(V[,i])*sqrt(dt)*rnorm(n))
       V[,i+1]=V[,i]+kappa*(theta-V[,i])*dt+xi*sqrt(V[,i])*sqrt(dt)*rnorm(n)
    }
    return(list(S=S,V=V))
}
S0=100
r=0.05
D=0
T=0.5
E=100
kappa=1
xi=0.1
theta=0.36
V0=0.4
m=20
X=trajectories(5)
#task 2
phi1=function(s,v){
   return(s-E/2)
}
phi2=function(s,v){
   return(v-theta)
}
phi3=function(s,v){
   return((s-E/2)^2)
}
phi4=function(s,v){
   return((s-E/2)*(v-theta))
}
phi5=function(s,v){
   return((v-theta)^2)
}
n=10000
X=trajectories(n)
p=function(s,E){
  return(pmax(E-s,0))
}
W=p(X$S[,m+1],E)
i=m #time t=t[m-1]
Si=X$S[,i]
Vi=X$V[,i]
Y=exp(-r*T/m)*W
in_money=p(Si,E)>0
Ci=lm(Y~phi1(Si,Vi)+phi2(Si,Vi)+phi3(Si,Vi)+phi4(Si,Vi)+phi5(Si,Vi),subset=in_money)

#task 3
m=40
X=trajectories(n)
W=p(X$S[,m+1],E)
```

```
for(i in m:2){
   Si=X$S[,i]
   Vi=X$V[,i]
   Y=exp(-r*T/m)*W
   in_money=p(Si,E)>0
   Ci=lm(Y~phi1(Si,Vi)+phi2(Si,Vi)+phi3(Si,Vi)+phi4(Si,Vi)+phi5(Si,Vi),subset=in_money)
   W=Y
   Cvalues=predict(Ci,newdata=data.frame(Si=Si,Vi=Vi))
   p_values=p(Si,E)
   W[p_values>Cvalues]=p_values[p_values>Cvalues]
}
W=exp(-r*T/m)*W #at t=0, the value of a trajectory is discounted value at t=t[1]
price=mean(W)
alpha=0.05
MCerror=-qnorm(alpha/2)*sd(W)/sqrt(n)
```