UNIVERSITE DE LISALA

CENTRE INTERUNIVERSITAIRE DE RECHERCHE PLURIDISCIPLINAIRE (CIREP)
STATUT : UNIVERSITE PUBLIQUE
Web : www.cirep.ac.cd
Email : info@cirep.ac.cd

NOTES DE COURS DE PROGRAMMATION

OBJECTIFS DU COURS

Objectif général

L'objectif général de ce cours est d'approfondir les connaissances des étudiants en matière de modélisation mathématique et de résolution de problèmes complexes à l'aide de techniques avancées de programmation.

Objectifs spécifiques :

- ✓ Comprendre les concepts et les méthodes de la programmation stochastique, qui implique la modélisation et l'optimisation de systèmes sous l'incertitude.
- ✓ Apprendre à formuler des problèmes d'optimisation stochastique et à utiliser des outils mathématiques avancés tels que la théorie des probabilités, les processus stochastiques et la programmation linéaire stochastique pour les résoudre.
- ✓ Explorer les applications de la programmation stochastique dans des domaines tels que la finance, la logistique, l'ingénierie et la gestion des opérations.
- ✓ Se familiariser avec les techniques de programmation mathématique avancée, telles que la programmation non linéaire, la programmation quadratique, la programmation convexe et la programmation semi-définie.
- ✓ Développer la capacité à analyser et interpréter les résultats des modèles mathématiques et à prendre des décisions éclairées basées sur ces analyses.

En combinant ces objectifs spécifiques, les étudiants seront en mesure d'approfondir leur compréhension des concepts avancés en programmation stochastique et mathématique, tout en acquérant des compétences pratiques pour résoudre efficacement des problèmes complexes dans divers domaines d'application.

Plan du cours

Remarques préliminaires

Ce cours est probablement un peu en avance sur un programme classique d'un étudiant en mathématiques appliquées. Ainsi nous ne verrons pas tous les classiques des méthodes de Monte–Carlo mais seulement ses éléments incontournables ! Ce cours commence par la génération de variables aléatoires puis poursuit par l'intégration par Monte–Carlo. Pour ce deuxième partie, j'ai décidé d'illustrer la thématique avec des applications en Mathématiques financières. Bien sûr c'est juste « une carotte » afin de rendre ce cours plus intéressant et il ne vous ait aucunement demandé de connaître les Mathématiques financières—ni de les apprécier d'ailleurs ;-)

L'évaluation du cours se fera par **deux** notes :

— une note sur un devoir sur table (avec peut être l'aide de l'ordinateur) ;

— une note sur un projet que vous ferez en binôme et pour lequel un rapport en LaTeXde 5 à 10 pages devra être rédigé.

L'évaluation portera tout autant sur la théorie que vous aurez appris tout au long de ce cours que sur sa mise en pratique avec R. Votre engagement personnel comptera également pour une grande partie de la note.

D'ailleurs en passant puisque vous allez utiliser le langage R (et que ce cours n'est pas un cours sur R), je vous invite très fortement à jeter un oeil à mon cours sur R (ou tout autre ressource qui vous plaira)

www.math.univ-montp2.fr/~ribatet/docs/LogicielsScientifiques/cours.pdf

Mais aussi d'avoir votre feuille de pompe toujours sur vous (et que je vous mets plus bas)

http://cran.r-project.org/doc/contrib/Short-refcard.pdf

Sachez que la programmation ne s'apprend pas réellement avec un prof, c'est donc à vous de vous y mettre. Il faut à tout prix fournir un travail régulier, n'hésitez donc pas si vous en avez la possibilité de venir en cours avec votre ordinateur portable.

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07

Getting help

Most R functions have online documentation.

help(topic) documentation on topic
?topic id.
help.search("topic") search the help system
apropos("topic") the names of all objects in the search list matching the regular expression "topic"
help.start() start the HTML version of help
str(a) display the internal *str*ucture of an R object
summary(a) gives a "summary" of a, usually a statistical summary but it is *generic* meaning it has different operations for different classes of a
ls() show objects in the search path; specify pat="pat" to search on a pattern
ls.str() str() for each variable in the search path
dir() show files in the current directory
methods(a) shows S3 methods of a
methods(class=class(a)) lists all the methods to handle objects of class a

Input and output

load() load the datasets written with save
data(x) loads specified data sets
library(x) load add-on packages
read.table(file) reads a file in table format and creates a data frame from it; the default separator sep="" is any whitespace; use header=TRUE to read the first line as a header of column names; use as.is=TRUE to prevent character vectors from being converted to factors; use comment.char="" to prevent "#" from being interpreted as a comment; use skip=n to skip n lines before reading data; see the help for options on row naming, NA treatment, and others
read.csv("filename",header=TRUE) id. but with defaults set for reading comma-delimited files
read.delim("filename",header=TRUE) id. but with defaults set for reading tab-delimited files
read.fwf(file,widths,header=FALSE,sep="",as.is=FALSE) read a table of *f*ixed *width f*ormatted data into a 'data.frame'; widths is an integer vector, giving the widths of the fixed-width fields
save(file,...) saves the specified objects (...) in the XDR platform-independent binary format
save.image(file) saves all objects
cat(..., file="", sep=" ") prints the arguments after coercing to character; sep is the character separator between arguments
print(a, ...) prints its arguments; generic, meaning it can have different methods for different objects
format(x,...) format an R object for pretty printing
character or factor columns are surrounded by quotes ("); sep is the field separator; eol is the end-of-line separator; na is the string for missing values; use col.names=NA to add a blank column header to get the column headers aligned correctly for spreadsheet input
sink(file) output to file, until sink()

Most of the I/O functions have a file argument. This can often be a character string naming a file or a connection. file="" means the standard input or output. Connections can include files, pipes, zipped files, and R variables.

On windows, the file connection can also be used with description = "clipboard". To read a table copied from Excel, use
x <- read.delim("clipboard")
To write a table to the clipboard for Excel, use
write.table(x,"clipboard",sep="\t",col.names=NA)
For database interaction, see packages RODBC, DBI, RMySQL, RPgSQL, and ROracle. See packages XML, hdf5, netCDF for reading other file formats.

Data creation

c(...) generic function to combine arguments with the default forming a vector; with recursive=TRUE descends through lists combining all elements into one vector
from:to generates a sequence; ":" has operator priority; 1:4 + 1 is "2,3,4,5"
seq(from,to) generates a sequence by= specifies increment; length= specifies desired length
seq(along=x) generates 1, 2, ..., length(along); useful for for loops
rep(x,times) replicate x times; use each= to repeat "each" element of x each times; rep(c(1,2,3),2) is 1 2 3 1 2 3; rep(c(1,2,3),each=2) is 1 1 2 2 3 3
data.frame(...) create a data frame of the named or unnamed arguments; data.frame(v=1:4,ch=c("a","B","c","d"),n=10); shorter vectors are recycled to the length of the longest
list(...) create a list of the named or unnamed arguments; list(a=c(1,2),b="hi",c=3i);
array(x,dim=) array with data x; specify dimensions like dim=c(3,4,2); elements of x recycle if x is not long enough
matrix(x,nrow=,ncol=) matrix; elements of x recycle
factor(x,levels=) encodes a vector x as a factor
gl(n,k,length=n*k,labels=1:n) generate levels (factors) by specifying the pattern of their levels; k is the number of levels, and n is the number of replications
expand.grid() a data frame from all combinations of the supplied vectors or factors
rbind(...) combine arguments by rows for matrices, data frames, and others
cbind(...) id. by columns

Slicing and extracting data

Indexing vectors

x[n]	n^{th} element
x[-n]	all *but* the n^{th} element
x[1:n]	first n elements
x[-(1:n)]	elements from n+1 to the end
x[c(1,4,2)]	specific elements
x["name"]	element named "name"
x[x > 3]	all elements greater than 3

Indexing lists

x[n]	list with elements n
x[[n]]	n^{th} element of the list
x[["name"]]	element of the list named "name"
x$name	id.

Indexing matrices

x[i,j]	element at row i, column j
x[i,]	row i
x[,j]	column j
x[,c(1,3)]	columns 1 and 3
x["name",]	row named "name"

Indexing data frames (matrix indexing plus the following)

x[["name"]]	column named "name"
x$name	id.

Variable conversion

as.array(x), as.data.frame(x), as.numeric(x), as.logical(x), as.complex(x), as.character(x), ... convert type; for a complete list, use methods(as)

Variable information

is.na(x), is.null(x), is.array(x), is.data.frame(x), is.numeric(x), is.complex(x), is.character(x), ... test for type; for a complete list, use methods(is)
length(x) number of elements in x
dim(x) Retrieve or set the dimension of an object; dim(x) <- c(3,2)
dimnames(x) Retrieve or set the dimension names of an object
nrow(x) number of rows; NROW(x) is the same but treats a vector as a one-row matrix
ncol(x) and **NCOL(x)** id. for columns
class(x) get or set the class of x; class(x) <- "myclass"
unclass(x) remove the class attribute of x
attr(x,which) get or set the attribute which of x
attributes(obj) get or set the list of attributes of obj

Data selection and manipulation

which.max(x) returns the index of the greatest element of **x**
which.min(x) returns the index of the smallest element of **x**
rev(x) reverses the elements of x
sort(x) sorts the elements of x in increasing order; to sort in decreasing order: rev(sort(x))
cut(x,breaks) divides x into intervals (factors); breaks is the number of cut intervals or a vector of cut points
match(x, y) returns a vector of the same length than x with the elements of x which are in y (NA otherwise)
which(x == a) returns a vector of the indices of x if the comparison operation is true (TRUE), in this example the values of i for which x[i] == a (the argument of this function must be a variable of mode logical)
choose(n, k) computes the combinations of k events among n repetitions $= n!/[(n-k)!k!]$
na.omit(x) suppresses the observations with missing data (NA) (suppresses the corresponding line if x is a matrix or a data frame)

unique(x) if x is a vector or a data frame, returns a similar object but with the duplicate elements suppressed

table(x) returns a table with the numbers of the differents values of x (typically for integers or factors)

subset(x, ...) returns a selection of x with respect to criteria (..., typically comparisons: x$V1 < 10); if x is a data frame, the option select gives the variables to be kept or dropped using a minus sign

sample(x, size) resample randomly and without replacement size elements in the vector x, the option replace = TRUE allows to resample with replacement

prop.table(x,margin=) table entries as fraction of marginal table

Math

sin,cos,tan,asin,acos,atan,atan2,log,log10,exp

max(x) maximum of the elements of x

min(x) minimum of the elements of x

range(x) id. then c(min(x), max(x))

sum(x) sum of the elements of x

diff(x) lagged and iterated differences of vector x

prod(x) product of the elements of x

mean(x) mean of the elements of x

median(x) median of the elements of x

quantile(x,probs=) sample quantiles corresponding to the given probabilities (defaults to 0,.25,.5,.75,1)

weighted.mean(x, w) mean of x with weights w

rank(x) ranks of the elements of x

var(x) or cov(x) variance of the elements of x (calculated on $n-1$); if x is a matrix or a data frame, the variance-covariance matrix is calculated

sd(x) standard deviation of x

cor(x) correlation matrix of x if it is a matrix or a data frame (1 if x is a vector)

var(x, y) or cov(x, y) covariance between x and y, or between the columns of x and those of y if they are matrices or data frames

cor(x, y) linear correlation between x and y, or correlation matrix if they are matrices or data frames

round(x, n) rounds the elements of x to n decimals

log(x, base) computes the logarithm of x with base base

scale(x) if x is a matrix, centers and reduces the data; to center only use the option center=FALSE, to reduce only scale=FALSE (by default center=TRUE, scale=TRUE)

pmin(x,y,...) a vector which *i*th element is the minimum of x[i], y[i],...

pmax(x,y,...) id. for the maximum

cumsum(x) a vector which *i*th element is the sum from x[1] to x[i]

cumprod(x) id. for the product

cummin(x) id. for the minimum

cummax(x) id. for the maximum

union(x,y), **intersect(x,y)**, **setdiff(x,y)**, **setequal(x,y)**, **is.element(el,set)** "set" functions

Re(x) real part of a complex number

Im(x) imaginary part

Mod(x) modulus; abs(x) is the same

Arg(x) angle in radians of the complex number

Conj(x) complex conjugate

convolve(x,y) compute the several kinds of convolutions of two sequences

fft(x) Fast Fourier Transform of an array

mvfft(x) FFT of each column of a matrix

filter(x,filter) applies linear filtering to a univariate time series or to each series separately of a multivariate time series

Many math functions have a logical parameter na.rm=FALSE to specify missing data (NA) removal.

Matrices

t(x) transpose

diag(x) diagonal

%*% matrix multiplication

solve(a,b) solves a %*% x = b for x

solve(a) matrix inverse of a

rowsum(x) sum of rows for a matrix-like object; **rowSums(x)** is a faster version

colsum(x), **colSums(x)** id. for columns

rowMeans(x) fast version of row means

colMeans(x) id. for columns

Advanced data processing

apply(X,INDEX,FUN=) a vector or array or list of values obtained by applying a function FUN to margins (INDEX) of X

lapply(X,FUN) apply FUN to each element of the list X

tapply(X,INDEX,FUN=) apply FUN to each cell of a ragged array given by X with indexes INDEX

by(data,INDEX,FUN) apply FUN to data frame data subsetted by INDEX

merge(a,b) merge two data frames by common columns or row names

xtabs(a b,data=x) a contingency table from cross-classifying factors

aggregate(x,by,FUN) splits the data frame x into subsets, computes summary statistics for each, and returns the result in a convenient form; by is a list of grouping elements, each as long as the variables in x

stack(x, ...) transform data available as separate columns in a data frame or list into a single column

unstack(x, ...) inverse of stack()

reshape(x, ...) reshapes a data frame between 'wide' format with repeated measurements in separate columns of the same record and 'long' format with the repeated measurements in separate records; use (direction="wide") or (direction="long")

Strings

paste(...) concatenate vectors after converting to character; sep= is the string to separate terms (a single space is the default); collapse= is an optional string to separate "collapsed" results

substr(x,start,stop) substrings in a character vector; can also assign, as substr(x, start, stop) <- value

strsplit(x,split) split x according to the substring split

grep(pattern,x) searches for matches to pattern within x; see ?regex

gsub(pattern,replacement,x) replacement of matches determined by regular expression matching sub() is the same but only replaces the first occurrence.

tolower(x) convert to lowercase

toupper(x) convert to uppercase

match(x,table) a vector of the positions of first matches for the elements of x among table

x %in% table id. but returns a logical vector

pmatch(x,table) partial matches for the elements of x among table

nchar(x) number of characters

Dates and Times

The class Date has dates without times. POSIXct has dates and times, including time zones. Comparisons (e.g. >), seq(), and difftime() are useful. Date also allows + and -. ?DateTimeClasses gives more information. See also package chron.

as.Date(s) and **as.POSIXct(s)** convert to the respective class; format(dt) converts to a string representation. The default string format is "2001-02-21". These accept a second argument to specify a format for conversion. Some common formats are:

%a, %A Abbreviated and full weekday name.
%b, %B Abbreviated and full month name.
%d Day of the month (01–31).
%H Hours (00–23).
%I Hours (01–12).
%j Day of year (001–366).
%m Month (01–12).
%M Minute (00–59).
%p AM/PM indicator.
%S Second as decimal number (00–61).
%U Week (00–53); the first Sunday as day 1 of week 1.
%w Weekday (0–6, Sunday is 0).
%W Week (00–53); the first Monday as day 1 of week 1.
%y Year without century (00–99). Don't use.
%Y Year with century.
%z (output only.) Offset from Greenwich; -0800 is 8 hours west of.
%Z (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input. See ?strftime.

Plotting

plot(x) plot of the values of x (on the *y*-axis) ordered on the *x*-axis

plot(x, y) bivariate plot of x (on the *x*-axis) and y (on the *y*-axis)

hist(x) histogram of the frequencies of x

barplot(x) histogram of the values of x; use horiz=FALSE for horizontal bars

dotchart(x) if x is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)

pie(x) circular pie-chart

boxplot(x) "box-and-whiskers" plot

sunflowerplot(x, y) id. than plot() but the points with similar coordinates are drawn as flowers which petal number represents the number of points

stripplot(x) plot of the values of x on a line (an alternative to boxplot() for small sample sizes)

coplot(x~y | z) bivariate plot of x and y for each value or interval of values of z

interaction.plot (f1, f2, y) if f1 and f2 are factors, plots the means of y (on the *y*-axis) with respect to the values of f1 (on the *x*-axis) and of f2 (different curves); the option fun allows to choose the summary statistic of y (by default fun=mean)

matplot(x,y) bivariate plot of the first column of x *vs.* the first one of y, the second one of x *vs.* the second one of y, etc.

fourfoldplot(x) visualizes, with quarters of circles, the association between two dichotomous variables for different populations (x must be an array with dim=c(2, 2, k), or a matrix with dim=c(2, 2) if $k = 1$)

assocplot(x) Cohen–Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table

mosaicplot(x) 'mosaic' graph of the residuals from a log-linear regression of a contingency table

pairs(x) if x is a matrix or a data frame, draws all possible bivariate plots between the columns of x

plot.ts(x) if x is an object of class "ts", plot of x with respect to time, x may be multivariate but the series must have the same frequency and dates

ts.plot(x) id. but if x is multivariate the series may have different dates and must have the same frequency

qqnorm(x) quantiles of x with respect to the values expected under a normal law

qqplot(x, y) quantiles of y with respect to the quantiles of x

contour(x, y, z) contour plot (data are interpolated to draw the curves), x and y must be vectors and z must be a matrix so that dim(z)=c(length(x), length(y)) (x and y may be omitted)

filled.contour(x, y, z) id. but the areas between the contours are coloured, and a legend of the colours is drawn as well

image(x, y, z) id. but with colours (actual data are plotted)

persp(x, y, z) id. but in perspective (actual data are plotted)

stars(x) if x is a matrix or a data frame, draws a graph with segments or a star where each row of x is represented by a star and the columns are the lengths of the segments

symbols(x, y, ...) draws, at the coordinates given by x and y, symbols (circles, squares, rectangles, stars, thermometres or "boxplots") which sizes, colours ... are specified by supplementary arguments

termplot(mod.obj) plot of the (partial) effects of a regression model (mod.obj)

The following parameters are common to many plotting functions:

add=FALSE if TRUE superposes the plot on the previous one (if it exists)

axes=TRUE if FALSE does not draw the axes and the box

type="p" specifies the type of plot, "p": points, "l": lines, "b": points connected by lines, "o": id. but the lines are over the points, "h": vertical lines, "s": steps, the data are represented by the top of the vertical lines, "S": id. but the data are represented by the bottom of the vertical lines

xlim=, ylim= specifies the lower and upper limits of the axes, for example with xlim=c(1, 10) or xlim=range(x)

xlab=, ylab= annotates the axes, must be variables of mode character

main= main title, must be a variable of mode character

sub= sub-title (written in a smaller font)

Low-level plotting commands

points(x, y) adds points (the option type= can be used)

lines(x, y) id. but with lines

text(x, y, labels, ...) adds text given by labels at coordinates (x,y); a typical use is: plot(x, y, type="n"); text(x, y, names)

mtext(text, side=3, line=0, ...) adds text given by text in the margin specified by side (see axis() below); line specifies the line from the plotting area

segments(x0, y0, x1, y1) draws lines from points (x0,y0) to points (x1,y1)

arrows(x0, y0, x1, y1, angle= 30, code=2) id. with arrows at points (x0,y0) if code=2, at points (x1,y1) if code=1, or both if code=3; angle controls the angle from the shaft of the arrow to the edge of the arrow head

abline(a,b) draws a line of slope b and intercept a

abline(h=y) draws a horizontal line at ordinate y

abline(v=x) draws a vertical line at abcissa x

abline(lm.obj) draws the regression line given by lm.obj

rect(x1, y1, x2, y2) draws a rectangle which left, right, bottom, and top limits are x1, x2, y1, and y2, respectively

polygon(x, y) draws a polygon linking the points with coordinates given by x and y

legend(x, y, legend) adds the legend at the point (x,y) with the symbols given by legend

title() adds a title and optionally a sub-title

axis(side, vect) adds an axis at the bottom (side=1), on the left (2), at the top (3), or on the right (4); vect (optional) gives the abcissa (or ordinates) where tick-marks are drawn

rug(x) draws the data x on the *x*-axis as small vertical lines

locator(n, type="n", ...) returns the coordinates (*x, y*) after the user has clicked n times on the plot with the mouse; also draws symbols (type="p") or lines (type="l") with respect to optional graphic parameters (...); by default nothing is drawn (type="n")

Graphical parameters

These can be set globally with **par(...)**; many can be passed as parameters to plotting commands.

adj controls text justification (0 left-justified, 0.5 centred, 1 right-justified)

bg specifies the colour of the background (ex. : bg="red", bg="blue", ... the list of the 657 available colours is displayed with colors())

bty controls the type of box drawn around the plot, allowed values are: "o", "l", "7", "c", "u" ou "]" (the box looks like the corresponding character); if bty="n" the box is not drawn

cex a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, cex.axis, the axis labels, cex.lab, the title, cex.main, and the sub-title, cex.sub

col controls the color of symbols and lines; use color names: "red", "blue" see colors() or as "#RRGGBB"; see rgb(), hsv(), gray(), and rainbow(); as for cex there are: col.axis, col.lab, col.main, col.sub

font an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for cex there are: font.axis, font.lab, font.main, font.sub

las an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)

lty controls the type of lines, can be an integer or string (1: "solid", 2: "dashed", 3: "dotted", 4: "dotdash", 5: "longdash", 6: "twodash", or a string of up to eight characters (between "0" and "9") which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example lty="44" will have the same effect than lty=2

lwd a numeric which controls the width of lines, default 1

mar a vector of 4 numeric values which control the space between the axes and the border of the graph of the form c(bottom, left, top, right), the default values are c(5.1, 4.1, 4.1, 2.1)

mfcol a vector of the form c(nr,nc) which partitions the graphic window as a matrix of nr lines and nc columns, the plots are then drawn in columns

mfrow id. but the plots are drawn by row

pch controls the type of symbol, either an integer between 1 and 25, or any single character within ""

1 ● 2 △ 3 + 4 × 5 ◇ 6 ▽ 7 ⊠ 8 ✳ 9 ⊕ 10 ● 11 ⧖ 12 ⊞ 13 ● 14 ⧄ 15 ■
16 ● 17 ▲ 18 ◆ 19 ● 20 • 21 ● 22 □ 23 ◇ 24 △ 25 ▽ * * . X X a a ? ?

ps an integer which controls the size in points of texts and symbols

pty a character which specifies the type of the plotting region, "s": square, "m": maximal

tck a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if tck=1 a grid is drawn

tcl a value which specifies the length of tick-marks on the axes as a fraction of the height of a line of text (by default tcl=-0.5)

xaxt if xaxt="n" the *x*-axis is set but not drawn (useful in conjonction with axis(side=1, ...))

yaxt if yaxt="n" the *y*-axis is set but not drawn (useful in conjonction with axis(side=2, ...))

Lattice (Trellis) graphics

xyplot(y~x) bivariate plots (with many functionalities)

barchart(y~x) histogram of the values of y with respect to those of x

dotplot(y~x) Cleveland dot plot (stacked plots line-by-line and column-by-column)

densityplot(~x) density functions plot

histogram(~x) histogram of the frequencies of x

bwplot(y~x) "box-and-whiskers" plot

qqmath(~x) quantiles of x with respect to the values expected under a theoretical distribution

stripplot(y~x) single dimension plot, x must be numeric, y may be a factor

qq(y~x) quantiles to compare two distributions, x must be numeric, y may be numeric, character, or factor but must have two 'levels'

splom(~x) matrix of bivariate plots

parallel(~x) parallel coordinates plot

levelplot(z~x*y|g1*g2) coloured plot of the values of z at the coordinates given by x and y (x, y and z are all of the same length)

wireframe(z~x*y|g1*g2) 3d surface plot

cloud(z~x*y|g1*g2) 3d scatter plot

In the normal Lattice formula, `y x|g1*g2` has combinations of optional conditioning variables `g1` and `g2` plotted on separate panels. Lattice functions take many of the same arguments as base graphics plus also `data=` the data frame for the formula variables and `subset=` for subsetting. Use `panel=` to define a custom panel function (see `apropos("panel")` and `?llines`). Lattice functions return an object of class trellis and have to be `print`-ed to produce the graph. Use `print(xyplot(...))` inside functions where automatic printing doesn't work. Use `lattice.theme` and `lset` to change Lattice defaults.

Optimization and model fitting

`optim(par, fn, method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN")` general-purpose optimization; `par` is initial values, `fn` is function to optimize (normally minimize)

`nlm(f,p)` minimize function `f` using a Newton-type algorithm with starting values `p`

`lm(formula)` fit linear models; `formula` is typically of the form `response termA + termB + ...`; use `I(x*y) + I(x^2)` for terms made of nonlinear components

`glm(formula,family=)` fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution; `family` is a description of the error distribution and link function to be used in the model; see `?family`

`nls(formula)` nonlinear least-squares estimates of the nonlinear model parameters

`approx(x,y=)` linearly interpolate given data points; `x` can be an xy plotting structure

`spline(x,y=)` cubic spline interpolation

`loess(formula)` fit a polynomial surface using local fitting

Many of the formula-based modeling functions have several common arguments: `data=` the data frame for the formula variables, `subset=` a subset of variables used in the fit, `na.action=` action for missing values: `"na.fail"`, `"na.omit"`, or a function. The following generics often apply to model fitting functions:

`predict(fit,...)` predictions from `fit` based on input data

`df.residual(fit)` returns the number of residual degrees of freedom

`coef(fit)` returns the estimated coefficients (sometimes with their standard-errors)

`residuals(fit)` returns the residuals

`deviance(fit)` returns the deviance

`fitted(fit)` returns the fitted values

`logLik(fit)` computes the logarithm of the likelihood and the number of parameters

`AIC(fit)` computes the Akaike information criterion or AIC

Statistics

`aov(formula)` analysis of variance model

`anova(fit,...)` analysis of variance (or deviance) tables for one or more fitted model objects

`density(x)` kernel density estimates of `x`

`binom.test()`, **`pairwise.t.test()`**, **`power.t.test()`**, **`prop.test()`**, **`t.test()`**, ... use `help.search("test")`

Distributions

`rnorm(n, mean=0, sd=1)` Gaussian (normal)

`rexp(n, rate=1)` exponential

`rgamma(n, shape, scale=1)` gamma

`rpois(n, lambda)` Poisson

`rweibull(n, shape, scale=1)` Weibull

`rcauchy(n, location=0, scale=1)` Cauchy

`rbeta(n, shape1, shape2)` beta

`rt(n, df)` 'Student' (*t*)

`rf(n, df1, df2)` Fisher–Snedecor (*F*) (χ^2)

`rchisq(n, df)` Pearson

`rbinom(n, size, prob)` binomial

`rgeom(n, prob)` geometric

`rhyper(nn, m, n, k)` hypergeometric

`rlogis(n, location=0, scale=1)` logistic

`rlnorm(n, meanlog=0, sdlog=1)` lognormal

`rnbinom(n, size, prob)` negative binomial

`runif(n, min=0, max=1)` uniform

`rwilcox(nn, m, n)`, `rsignrank(nn, n)` Wilcoxon's statistics

All these functions can be used by replacing the letter `r` with `d`, `p` or `q` to get, respectively, the probability density (`dfunc(x, ...)`), the cumulative probability density (`pfunc(x, ...)`), and the value of quantile (`qfunc(p, ...)`, with $0 < p < 1$).

Programming

`function( arglist ) expr` function definition

`return(value)`

`if(cond) expr`

`if(cond) cons.expr else alt.expr`

`for(var in seq) expr`

`while(cond) expr`

`repeat expr`

`break`

`next`

Use braces *{}* around statements

`ifelse(test, yes, no)` a value with the same shape as `test` filled with elements from either `yes` or `no`

`do.call(funname, args)` executes a function call from the name of the function and a list of arguments to be passed to it

Chapitre 1

Introduction

La modélisation stochastique est un terme générique dont le but principal est de représenter un comportement par des modèles probabilistes. La plupart du temps ce que nous cherchons à modéliser présente un aléas (telle que la précipitation par exemple) mais ce n'est pas toujours le cas.

En ce qui nous concerne pour ce cours nous allons nous concentrer sur une petite partie de la modélisation stochastique : la **simulation** et les **méthodes de Monte Carlo**.

Les méthodes de Monte Carlo visent à reproduire le comportement d'un processus (au sens physique du terme) aléatoire. Cela dit les données générées par ces méthodes n'ont (la plupart du temps) rien d'aléatoires mais son générées par des algorithmes **déterministes**. Les **résultats pourraient donc être en principe prévisibles**—si nous avions entièrement connaissance du procédé.

Avant 1920 Utilisation de simulation afin de récupérer ou suggérer la loi théorique (Student–t, aiguille de Buffon et π).

1940 Approximation d'intégrale pour la 1ère bombe atomique.

1950 Algorithme de Metropolis

1970 Développement d'algorithmes pour la simulation selon des lois standards

1980 Méthodes MCMC et bootstrap

1990 Méthodes MCMC largement répandues (BUGS, ...)

2000 ABC, SMC, ...

Petite citation de William Sealy Gosset, alias Student, en 1908 sur l'utilité de la simulation

> « *Before I had succeeded in solving my problem analytically, I had endeavoured to do so empirically. The material used was a correlation table containing the height and left middle finger measurements of 3000 criminals, from a paper by W. R. MacDonell (Biometrika, Vol. I., p. 219). The measurements were written out on 3000 pieces of cardboard, which were then very thoroughly shuffled and drawn at random. As each card was drawn its numbers were written down in a book, which thus contains the measurements of 3000 criminals in a random order. Finally, each consecutive set of 4 was taken as a sample—750 in all—and the mean, standard deviation, and correlation of each sample determined. The difference between the mean of each sample and the mean of the population was then divided by the standard deviation of the sample, giving us the z of Section III.* »

On se doute que William Gosset a dû se faire plaisir en faisant tous ces calculs certes simples mais vraiment pas intéressants. . . Aujourd'hui nous avons l'ordinateur et il se démotive bien moins vite que nous sur ce genre de tâches barbantes.

L'objectif de ce cours est donc de vous montrer comment ont peu utiliser l'ordinateur pour répondre à des problèmes bien précis. Cela dit l'ordinateur est un outil et il faut donc savoir s'en servir à bon escient. En gros, ce n'est pas parce que l'on utilise l'ordinateur que l'on n'aura pas besoin de maîtriser la théorie ! ! !

Remarque. Pour être parfaitement honnête, ce sera un premier pas et vous ne connaîtrez que les méthodes basiques. Cela dit en M1 ça deviendra bien plus sérieux et séduisant ! ! !

Chapitre 2

Simulation selon une loi donnée

Avant de commencer ce chapitre, je souhaite reprendre l'introduction faite par Bernard Bercu et Djalil Chafaï dans leur libre « Modélisation stochastique et simulation ».

Supposons que je joue à pile ou face avec vous mais qu'après 20 lancés, je n'ai fait que des faces. Vous seriez très sceptique quand à l'**honnêteté de ma pièce**. Pourtant je pourrais vous répondre que **toutes les suites de vingt lancers sont équiprobables et de probabilité** 2^{-20} **!!!**

Clairement ce qui vous gêne dans la suite

$$1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1,$$

c'est qu'elle semble être **moins aléatoire** que la suite

$$1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1.$$

Mais au fait c'est quoi une **suite aléatoire** ? Nous entrons clairement dans un débat philosophique ici et nous ne perdrons pas de temps avec. Nous reviendrons juste très rapidement sur les propriétés souhaitables de telles suites aléatoires lors de la Section 2.1.2.

2.1 Le début du voyage : la $U(0, 1)$

Nous n'allons pas trop insister sur la génération de variable aléatoires uniformes sur $[0, 1]$ mais juste parler des notions / approches élémentaires qu'ils faut absolument connaître au moins pour votre culture générale.

2.1.1 Générateurs linéaires congruentiels

Avant de pouvoir simuler des nombres aléatoires selon une loi spécifique, il faut déjà pouvoir le faire sur la **loi de base** : la loi uniforme sur l'intervalle $[0, 1]$, notée $U(0, 1)$. C'est que nous allons voir dans cette section.

S'il existe différentes méthodes pour arriver à ce but nous allons nous concentrer sur les **générateurs linéaires congruentiels**, i.e., dont les **nombres pseudo aléatoires** sont donnés par la suite

$$X_{n+1} \equiv aX_n + b \mod m, \tag{2.1}$$

où les entiers a, b et m doivent être choisis minutieusement afin que la séquence générée ait de bonnes propriétés et la valeur initiale X_0 est appelée la **graine aléatoire**. Clairement les valeurs $X_1, X_2, \ldots \in \{0, 1, \ldots, m-1\}$ et l'on obtient donc comme désiré des valeurs dans $[0, 1]$ en considérant les valeurs X_i/m.

Voici quelques exemples de générateurs :

rand() (langage C ANSI) $m = 2^{31}$, $a = 1103515245$ et $b = 12345$;

drand48() (langage C ANSI) $m = 2^{48}$, $a = 25214903917$ et $b = 11$;

RANDU (qui était implémenté sur les ordinateurs IBM) $m = 2^{31}$, $a = 65539$ et $c = 0$;

Maple $m = 10^{12} - 11$, $a = 427419669081$ et $b = 0$.

Notons au passage que les deux derniers générateurs sont **multiplicatifs** (puisque $b = 0$).

Remarque. La logiciel R que nous allons utiliser utilise par défaut le générateur **Mersenne Twister**—bien que d'autres générateurs soient disponibles.

Clairement les générateurs de type (2.1) sont périodiques et de période $p \leq m$. Essayons de connaître un peu mieux les propriétés de ce type de générateurs mais afin de nous faciliter la tâche, on supposera pour la suite que $b = 0$.

Remarque. Dans le cas multiplicatif, on ne peut clairement pas poser $X_0 \equiv 0$ sinon $X_i \equiv 0$ pour tout $i \geq 0$. La période des générateurs congruentiels multiplicatifs est donc au plus $m - 1$.

Il est possible de construire des générateurs multiplicatifs de période maximale, i.e., égale à $m - 1$, comme le montre le théorème suivant (donné sans preuve).

Théorème 2.1.1. *Une condition nécessaire pour qu'un générateur multiplicatif soit de période $m-1$ est que m soit un nombre premier.*

Par le théorème de Lagrange sur les groupes, la période divise alors nécessairement $m - 1$. Cela dit elle est égale à $m - 1$ si et seulement si a est une racine primitive *de m, i.e., $a \neq 0$ et*

$$a^{(m-1)/p} \not\equiv 1 \mod m,$$

pour tout facteur premier p de $m - 1$.

Exemple 2.1. Considérons le cas où $m = 2^{31} - 1$, qui est un nombre premier—si si je vous le dis. Puisque $m - 1 = 2 \times 3^2 \times 7 \times 11 \times 31 \times 151 \times 331$, on peut en déduire que 7 est une racine primitive de m et qu'il en est de même pour $a = 7^5 = 16807$.

Remarque. Le générateur Mersenne Twister, qui n'est pas un générateur multiplicatif, a pour période $2^{19937-1}$! ! !

2.1.2 Validation des générateurs

Comme déjà évoqué plus haut, nous aimerions que le générateur possède de bonnes propriétés. S'il est clair que l'on souhaite avoir une grande période, ce n'est pas le seul critère à prendre en compte comme le montre le générateur débile suivant

$$X_{n+1} \equiv X_n + 1 \mod m, \qquad X_0 = 0,$$

et m arbitrairement grand.

Pour faire simple (car c'est une notion compliquée), il faut vérifier deux hypothèses :

— l'uniformité sur $[0, 1]$;

— l'indépendance.

Pour tester l'uniformité (du moins marginalement) on peut utiliser des notions de

statistiques que vous verrez en M1 (probablement) appelées *tests d'hypothèses* (test du χ^2, Kolmogorov–Smirnov, . . .) Tester l'indépendance est un peu plus délicat et typiquement les tests sont basés sur la manière dont l'hypercube unité $[0, 1]^d$ se remplit par le générateur—où les points remplissant l'hypercube sont de la forme $(U_l, \ldots, U_{l+d-1})$, $l = 1, \ldots, N - d + 1$.

Chapitre 2

2.2 La méthode d'inversion

La **méthode d'inversion** est une technique générique dont l'objectif est de générer un n–échantillon selon une loi donnée et ceci à partir de réalisations indépendantes d'une $U(0, 1)$. Cette méthode est de loin la **plus simple** et se base sur la **fonction de répartition inverse.**

Définition 2.2.1 (Fonction de répartition et son inverse généralisée)**.**
Soit X une variable aléatoire à valeur dans $\mathbb{R}$. On appelle **fonction de répartition** la fonction

$$F(x) = \Pr(X \leq x), \qquad x \in \mathbb{R}. \tag{2.2}$$

La fonction F est une fonction croissante et **cadlag**—cela dit pour nous elle sera souvent continue, i.e., sans aucun point de discontinuité.

Ceci nous permet de définir son **inverse généralisée**, F^{-1}, définie par

$$F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}, \qquad u \in [0, 1]. \tag{2.3}$$

La fonction F^{-1} est croissante puisque F l'est et est parfois appelée **fonction quantile**.

Lemme 2.2.1. *Pour tout $x \in \mathbb{R}$ et $u \in [0, 1]$, on a $u \leq F(x) \Leftrightarrow F^{-1}(u) \leq x$.*

Démonstration.

Remarque. Évidemment si F est bijective (et ce sera bien souvent le cas pour nous), alors F^{-1} correspond à la réciproque de F.

Proposition 2.2.2. *Soient F une fonction de répartition quelconque et $U \sim U(0, 1)$. Alors $F^{-1}(U)$ est une variable aléatoire de fonction de répartition F.*

Démonstration.

Exemple 2.2 (Loi exponentielle)**.**
Rappelons qu'une variable aléatoire exponentielle de paramètre $\lambda > 0$, utilisée notamment pour modéliser les durées de survie, a pour fonction de répartition

$$F(x) = 1 - \exp(-\lambda x), \qquad x \geq 0.$$

Calculons sa fonction réciproque (F étant clairement bijective).

$$F(x) = u \Leftrightarrow 1 - \exp(-\lambda x) = u \Leftrightarrow x = -\frac{\ln(1-u)}{\lambda}.$$

Ainsi pour générer un n–échantillon selon une Exp(λ) on pose

$$(X_1, \ldots, X_n) = \left(-\frac{\ln U_1}{\lambda}, \ldots, -\frac{\ln U_n}{\lambda} \right),$$

où on a utilisé au passage que $U \overset{d}{=} 1 - U$.

Voici maintenant une implémentation en R.

```
> myrexp <- function(n, lambda)
+     return(-log(runif(n)) / lambda)
>
> ## Simulation d'une Exp(5)
> lambda <- 5
> x <- myrexp(500, lambda)
>
> ## Comparaison histogramme / densite
> hist(x, freq = FALSE)##freq = FALSE pour aire = 1
> curve(dexp(x, lambda), xlim = c(0, max(x)), col = "red", add = TRUE)
```

Notons au passage que pour que cette méthode soit **efficace** (au sens informatique), il faut que F^{-1} soit rapidement calculable.

La méthode d'inversion prend une forme particulière lorsque le domaine de définition de variable aléatoire X à simuler est un ensemble fini $\{x_1, \ldots, x_k\}$, $k \in \mathbb{N}$. Sans perte de généralité, nous pouvons supposer que $x_1 < x_2 < \cdots < x_k$. Ainsi on a donc

$$X = F^{-1}(U) = \begin{cases} x_1, & \text{si } U \leq p_1 \\ \vdots & \\ x_j, & \text{si } \sum_{l=1}^{j-1} p_l < U \leq \sum_{l=1}^{j} p_l \\ \vdots & \\ x_k, & \text{si } \sum_{l=1}^{k-1} p_l < U \leq \sum_{l=1}^{k} p_l = 1, \end{cases}$$

avec $p_j = \Pr(X = x_j)$, $j = 1, \ldots, k$.

Exemple 2.3 (Loi de Bernoulli)**.**
Rappelons que la loi de Bernoulli de paramètre $p \in (0, 1)$ et notée Ber(p) est définie par

$$\Pr(X = j) = \begin{cases} p, & \text{si } j = 1 \\ 1 - p, & \text{si } j = 0. \end{cases}$$

Ainsi un n-échantillon distribué selon une Ber(p) est générée selon

$$(X_1, \ldots, X_n) = \left(1_{\{U_1 \leq p\}}, \ldots, 1_{\{U_n \leq p\}} \right), \qquad U_1, \ldots, U_n \overset{\text{iid}}{\sim} U(0, 1).$$

Cette méthode peut être lente car elle peut nécessiter beaucoup de test de la forme $\sum_{l=1}^{j-1} p_l < U \leq \sum_{l=1}^{j} p_l$ et il peut donc être rentable de réordonner les x_j par valeurs de probabilités décroissantes—on augmente ainsi la probabilité que $U \leq p_1$ par exemple.

2.3 La méthode du rejet

Cette méthode est très connue et elle doit donc faire partie intégrante de votre répertoire de statisticien. Elle est parfois également appelée **méthode d'acceptation–rejet**.

Pour cette méthode, nous nous plaçons dans le cadre où nous savons simuler des variables aléatoires de densité de probabilité g ainsi que des $U(0,1)$ mais que notre but est de générer des variables aléatoires de densité f dont le **support est contenu dans celui de** g.

Remarque. J'appellerai la densité g la « densité outil »—les anglais utilisent quand à eux l'appelation « instrumental density ».

Supposons qu'il existe une constante $M > 0$ telle que pour tout $x \in \mathbb{R}$ on ait

$$\frac{f(x)}{g(x)} \leq M < \infty.$$

La méthode du rejet procède de la manière suivante :

1. Générer $Y \sim g$ et $U \sim U(0,1)$ indépendemment ;
2. Poser $X = Y$ si $U \leq f(Y)/\{Mg(Y)\}$, sinon retourner en 1.
3. Renvoyer Y.

Clairement l'algorithme sera performant dès lors que l'événement

$$\left(U \leq \frac{f(Y)}{Mg(Y)} \right)$$

à une forte probabilité de se réaliser. Nous reviendrons sur ce point plus tard.

Proposition 2.3.1. *L'algorithme énoncé plus haut se termine en T itérations où T est une variable aléatoire de loi géométrique de paramètre $1/M$ et renvoie une variable aléatoire de densité f.*

Démonstration.

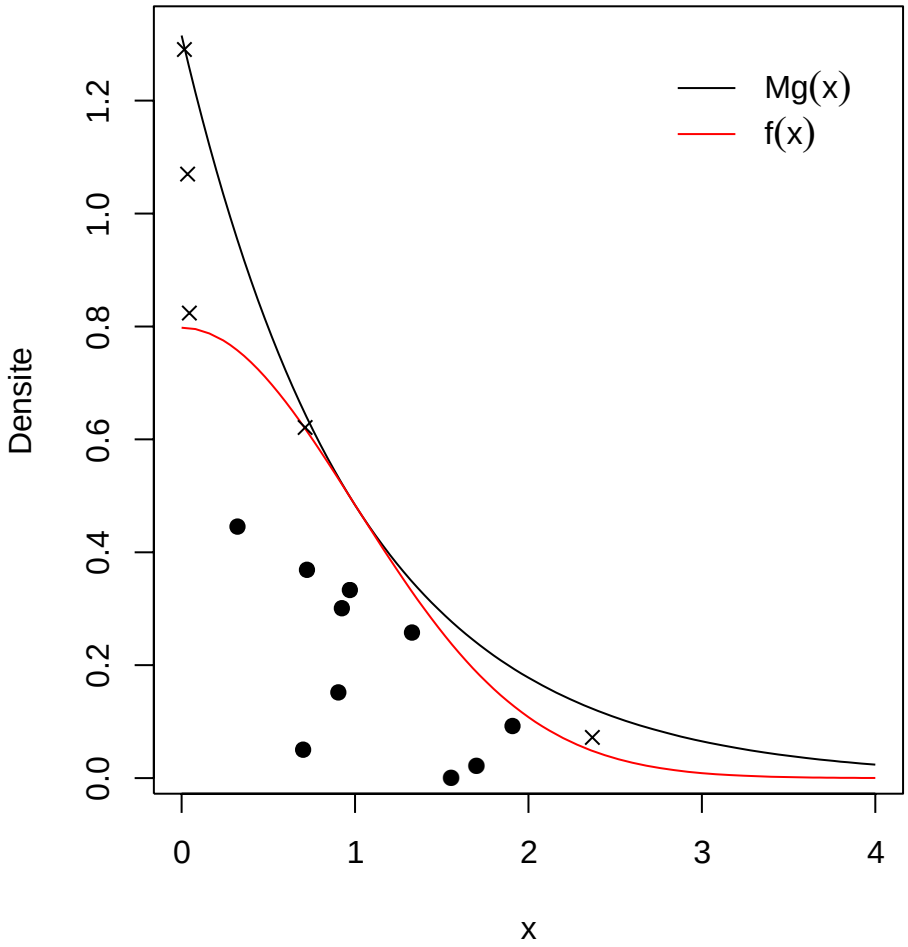

Figure 2.1 – *Illustration de la méthode du rejet via l'Exemple 2.4. Les points ont pour coordonnées* $\{Y_i, MU_i g(Y_i)\}$ *et les ronds sont acceptés, les croix rejetées.*

Exemple 2.4 (Loi demi–normale par une exponentielle)**.**
La loi demi–normale de paramètre $\sigma^2 = 1$ a pour densité

$$f(x) = \begin{cases} \sqrt{\frac{2}{\pi}} \exp\left(-\frac{x^2}{2}\right), & \text{si } y > 0 \\ 0, & \text{sinon.} \end{cases}$$

Elle porte donc bien son nom !

Pour tout $x > 0$, on a donc , en prenant pour g une Exp(1),

$$\frac{f(x)}{g(x)} = \frac{\sqrt{\frac{2}{\pi}}\exp\left(\frac{x^2}{2} - \right.}{\exp(-x)}$$

$$= \sqrt{\frac{2}{\pi}}\exp\left(-\frac{x^2 - 2x}{2}\right)$$

$$= \sqrt{\frac{2}{\pi}}\exp\left(-\frac{(x-1)^2 - 1}{2}\right)$$

$$\leq \sqrt{\frac{2}{\pi}}\exp(1/2) \approx 1.316.$$

La Figure 2.1 illustre cet exemple en simulant 10 réalisations indépendantes selon cette loi demi–normale. Voici le code R associé

```
> f <- function(x)
+     sqrt(2 / pi) * exp(-0.5 * x^2)
>
> g <- function(x)
+     exp(-x)
>
> M <- sqrt(2 / pi) * exp(0.5)
>
> ratio <- function(x)
+     M * g(x)
>
>
> n.sim <- 10 ## nb de v.a. a generer
> i <- 1 ##nb de v.a. generees
> ans <- rep(NA, n.sim) ## stocke les v.a. generees
> while (i <= n.sim){
+         U <- runif(1)
+         Y <- abs(rnorm(1))

+         if (U <= (f(Y) / (M * g(Y)))){
+             ans[i] <- Y
+             i  <- i + 1
+         }
+     }
>
> ans

 [1] 1.28825688 1.25588403 1.24018084 1.46666334 0.75421121 0.03724202
 [7] 0.39697197 0.46589588 1.11137043 0.18153538
```

Clairement d'après la Proposition 2.3.1 pour l'algorithme soit le plus efficace, **il faut que M soit le plus petit possible**—afin que l'on accepte plus facilement les v.a. générées. Souvent la borne M sera appelée **l'efficacité** de l'algorithme car elle correspond au nombre moyen de v.a. à générer pour en accepter une comme étant générée selon la loi d'intérêt—nous verrons cela en exercice.

2.4 Algorithme du ratio

Nous allons passer vite sur les détails théoriques de cette méthode car j'en ai fait un gros exercice, cf. Feuille 1 ! Les algorithmes de type ratio sont basés sur le théorème suivant.

Théorème 2.4.1. *Pour toute fonction positive h telle que $\int_{\mathbb{R}} h(x)dx < \infty$, posons*

$$C_h = \left\{(u, v) : 0 \le u \le \sqrt{h\left(\frac{v}{u}\right)}\right\}.$$

Alors C_h est d'aire finie et si (U, V) est uniformément distribué sur C_h alors la v.a. V/U a pour densité $h/\int h$.

L'idée concrète derrière ce théorème est de simuler des v.a. selon une densité f que l'on pourrait ne connaître qu'à une constante de proportionnalité près, i.e., $f \propto h$. Regardons comment cela fonctionne à travers un exemple.

Exemple 2.5. La loi Beta(a, b), $a, b > 0$, a pour densité

$$f(x) = \frac{x^{a-1}(1-x)^{b-1}}{\int_0^1 u^{a-1}(1-u)^{b-1}du} = \frac{x^{a-1}(1-x)^{b-1}}{B(a,b)}, \qquad x \in [0,1],$$

où $B(a, b)$ est la fonction (spéciale) Bêta.

Soit $h(x) = x^2(1-x)^2$, $0 \le x \le 1$, qui est bien sûr positive et d'aire finie, on a donc

$$C_h = \left\{(u, v) : 0 \le u \le \sqrt{h(v/u)}\right\} = \{(u, v) : 0 \le u \le (v/u)(1 - v/u)\}.$$

On générera donc des v.a. Beta(3, 3)

Un peu de calcul (exo à la maison) montre que $C_h \subset [0, 1/4] \times [0, 4/27]$, on peut donc simuler des $U(C_h)$ via la méthode du rejet en simulant d'abord des $U([0, 1/16] \times [0, 4/27])$. Voici donc le code associé.

```
> n.sim <- 10 ##nb de va a simuler
> i <- 1 ##nb de va deja simulees
> ans <- rep(NA, n.sim) ## stocke les v.a. simulees
>
> h <- function(x)
+     x^2 * (1 - x)^2 * (x >= 0) * (x <= 1)
>
> while (i <= n.sim){
+     ## Simulation U(C_h)
+     U <- runif(1, 0, 1/4)
+     V <- runif(1, 0, 4/27)

+     if (U <= sqrt(h(V/U))){
+         ans[i] <- V / U
+         i <- i + 1
+     }
+ }
```

2.5 Méthodes spécifiques

Les méthodes énoncées plus hauts sont génériques et peuvent donc s'appliquer à un grand nombre de situations. Ce n'est pas le cas pour les méthodes que nous allons présenter ici. Elles méritent cependant d'être connues au moins pour votre culture générale.

2.5.1 Méthode de Box–Muller

Cette méthode permet de générer des variables selon une loi normale standard, i.e., une $N(0, 1)$.

Proposition 2.5.1. *Soient* $R \sim Exp(1/2)$ *et* $\Theta \sim U(0, 2\pi)$ *mutuellement indépendantes. Alors*

$$X = \sqrt{R}\cos\Theta, \qquad Y = \sqrt{R}\sin\Theta,$$

sont deux $N(0, 1)$ *indépendantes.*

Démonstration.

L'algorithme de Box–Muller génère donc deux $N(0, 1)$ indépendantes. **Bien que séduisant sur le papier cet algorithme a le désavantage de reposer sur les fonctions trigonométriques** cos **et** sin **qui sont coûteuses en temps de calcul**. On peut toutefois contourner leur utilisation à l'**aide de la méthode du rejet** comme le montre l'algorithme suivant

1. Générer $U_1, U_2 \overset{iid}{\sim} U(0, 1)$ et poser $V_1 = 2U_1 - 1$ et $V_2 = 2U_2 - 1$;
2. Rejeter (V_1, V_2) si (V_1, V_2) est en dehors du disque unité, i.e., $S = V_1^2 + V_2^2 > 1$;
3. Poser

$$Z_1 = R\cos\Theta = V_1\sqrt{-2S^{-1}\ln S}$$
$$Z_2 = R\sin\Theta = V_2\sqrt{-2S^{-1}\ln S}$$

qui sont deux $N(0, 1)$ indépendantes.

Remarque. Faisons un petit aparté puisque nous parlons de la simulation de variables aléatoires gaussiennes. Clairement nous pouvons simuler selon une $N(\mu, \sigma^2)$ sans trop de mal à l'aide d'une $N(0, 1)$ puisque

$$Z \sim N(0, 1) \Rightarrow \mu + \sigma Z \sim N(\mu, \sigma^2).$$

Bien plus intéressant est la simulation de **vecteurs gaussiens**.

Définition 2.5.1 (Rappel)**.**
Un vecteur aléatoire X à valeurs dans $\mathbb{R}^d$, $d \geq 2$, est dit **gaussien** si toute combinaison linéaire de ses composantes est une variable aléatoire gaussienne.

Si $X = (X_1, \ldots, X_d)^T$ est un vecteur gaussien, on définit son **espérance** par

$$\mathbb{E}(X) = \{\mathbb{E}(X_1), \ldots, \mathbb{E}(X_d)\}^T,$$

et sa matrice de **variance–(covariance)** par

$$\mathrm{Var}(X) = \mathbb{E}\left[\{X - \mathbb{E}(X)\}\{X - \mathbb{E}(X)\}^T\right]$$

Ainsi pour simuler un vecteur gaussien de moyenne μ et de matrice de covariance Σ—rappelons que Σ est alors nécessairement (semi) définie positive, il suffit d'utiliser l'algorithme suivant :

1. Générer le vecteur gaussien $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_d)^T$ où $\varepsilon_1, \ldots, \varepsilon_d \overset{\text{iid}}{\sim} N(0, 1)$;
2. Calculer la matrice C qui est la décomposition de Cholesky de Σ, i.e., $C^T C = \Sigma$;
3. Retourner $X = \mu + C^T \varepsilon$.

Démonstration.

2.5.2 Loi de Poisson

Rappelons qu'une variable aléatoire X de loi de Poisson de paramètre $\lambda > 0$ est définie par

$$\Pr(X = k) = \frac{\lambda^k}{k!} \exp(-\lambda), \qquad k = 0, 1, \ldots$$

A priori on pourrait être tenté d'utiliser l'algorithme donné lors de la Section 2.2 pour les variables aléatoires discrètes. C'est une mauvaise idée puisque X possède un nombre infini d'états et l'algorithme ne peut théoriquement pas s'appliquer—numériquement oui mais il sera très peu performant.

Une autre approche consiste à utiliser le résultat suivant.

Proposition 2.5.2. *Soient $E_1, E_2, \ldots$ des v.a.i.i.d. de loi Exp(λ), $\lambda > 0$. Alors, pour tout entier $n \geq 1$, on a*

$$\Pr(S_n \leq 1 \leq S_{n+1}) = \frac{\lambda^n}{n!} \exp(-\lambda), \qquad S_n = \sum_{l=1}^{n} E_l.$$

Démonstration.

□

On en déduit l'algorithme suivant pour simuler une Poisson(λ), $\lambda > 0$,

1. Posez $X \leftarrow 0$ et $S \sim \text{Exp}(\lambda)$;
2. Tant que $S < 1$ faire

   $S \leftarrow S + \text{Exp}(\lambda)$;

   $X \leftarrow X + 1$;
3. Retournez X.

Remarque. Mettre un exo où l'on initialise par $x \leftarrow 70$ par exemple avec une Poisson(100)– car la proba d'être inférieur à 70 est très faible...

2.5.3 Mélanges

En statistique on appelle un *mélange* une variable aléatoire qui typiquement fait intervenir un nombre fini de loi de probabilité. Plus précisément, la densité d'un mélange prend deux formes selon que le mélange soit *continu* ou *discret*

$$f(x) = \int g(x \mid y)p(y)\mathrm{d}y, \qquad f(x) = \sum_{i \in Y} p_i g_i(x),$$

où Y est le support de la loi « contrôlant » le mélange p.

L'algorithme permettant de générer des v.a. selon f est plus que naturel :

Mélange continu	Mélange discret
1. Générez $Y \sim p$	1. Générez $Y \sim p$;
2. Puis générez $X \sim g(\cdot \mid Y)$;	2. Puis générez $X \sim g_Y$;
3. Retournez X.	3. Retournez X.

Exemple 2.6. Loi de Student par mélange continu La loi de Student peut être vue comme un mélange continu. En effet il est connu (ou du moins à savoir pour vous !) que la variable aléatoire

$$T = Z\sqrt{\frac{\nu}{X}}, \qquad \nu > 0, \quad X \sim \chi^2_\nu, \quad Z \sim N(0,1),$$

suit une loi de Student à ν degrés de liberté.

Avec nos notations, ceci correspond à un mélange continu où p est la densité d'une χ^2_ν et $g(\cdot|\psi, Y)$ celle d'une loi Normale de moyenne nulle et de variance ν/Y. L'algorithme est donc

1. Générez $Y \sim \chi^2_\nu$;
2. Générez $T \sim N(0, \nu/Y)$;
3. Retournez T.

Remarque. mettre un exemple sur la loi négative binomiale $X \sim \text{Neg}(n, p)$, i.e., $X \mid Y \sim \text{Poisson}(Y)$ avec $Y \sim \text{Gamma}(n, p)$.

2.6 Avant de terminer ce chapitre...

... je tiens à préciser une chose **extrêmement importante**. Le langage R possède de base des fonctions pour générer des variables aléatoires selon de nombreuses loi de probabilités usuelles. Leurs implémentations ont été soigneusement construites par des experts reconnus mondialement et optimisées afin qu'elles soient en plus d'être fiable statistiquement, rapide d'exécution. Il est donc **parfaitement inutile** d'utiliser les codes que nous avons produits dans ce chapitre, ces derniers étaient là juste pour illustrer le cours.

Alors pourquoi avons nous vu tout cela me direz vous ? Tout simplement car R ne possède pas d'implémentation pour toutes les lois de probabilités et que vous aurez tôt ou tard besoin de le faire pour des distributions exotiques...

Chapitre 3

Intégration par Monte Carlo

Tout au long de ce chapitre nous allons nous intéresser à évaluer (je dis bien évaluer et non pas calculer) des intégrales de la forme

$$I = \int_{\mathcal{X}} h(x)\mathrm{d}x.$$

Ne vous fiez pas à l'apparence simpliste de ce problème car ce problème se rencontre extrêmement fréquemment en pratique. Alors bien sûr vous pensez tout de suite à l'intégration numérique déterministe (méthodes des trapèzes / de Simpson) et vous auriez bien raison. Sachez toutefois que ces méthodes bien qu'efficace en petites dimensions, ne le sont plus du tout pour des dimensions plus grandes. Ainsi une **intégration par Monte Carlo** pour le cas de la grande dimension sera bien plus profitable—car la vitesse de convergence est justement indépendante de cette dernière, contrairement aux méthodes déterministes ! ! !

3.1 Modèle de Black–Scholes

Afin de motiver ce chapitre nous allons travailler sur le modèle de Black–Scholes et de sa formule pour établir le prix juste d'une option d'achat européen. Comme ce n'est pas un cours de maths financières[1] nous n'irons pas trop dans les détails.

Le **modèle** de Black–Scholes modélise l'évolution temporelle du prix d'un actif, noté S_t, à l'aide d'une équation différentielle stochastique, i.e.,

$$\mathrm{d}S_t = rS_t\mathrm{d}t + \sigma S_t\mathrm{d}W_t,$$

où $\sigma > 0$ est la volatilité (écart type) et W_t un mouvement Brownien[2]

Il est facile (mais admis car bien trop tôt pour vous) de montrer à l'aide de la Formule d'Itô que la solution de cette équation différentielle stochastique est donnée par

$$S_t = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t\right).$$

Comme le mouvement Brownien vous est inconnu, on supposera juste qu'à temps $t > 0$ fixé, $W_t \sim N(0, t^2)$.

La **formule** de Black–Scholes établit la valeur théorique d'une option européenne.

1. et que je n'y connais pas grand chose…
2. Ce n'est pas grave si vous ne savez pas ce qu'est un mouvement Brownien…

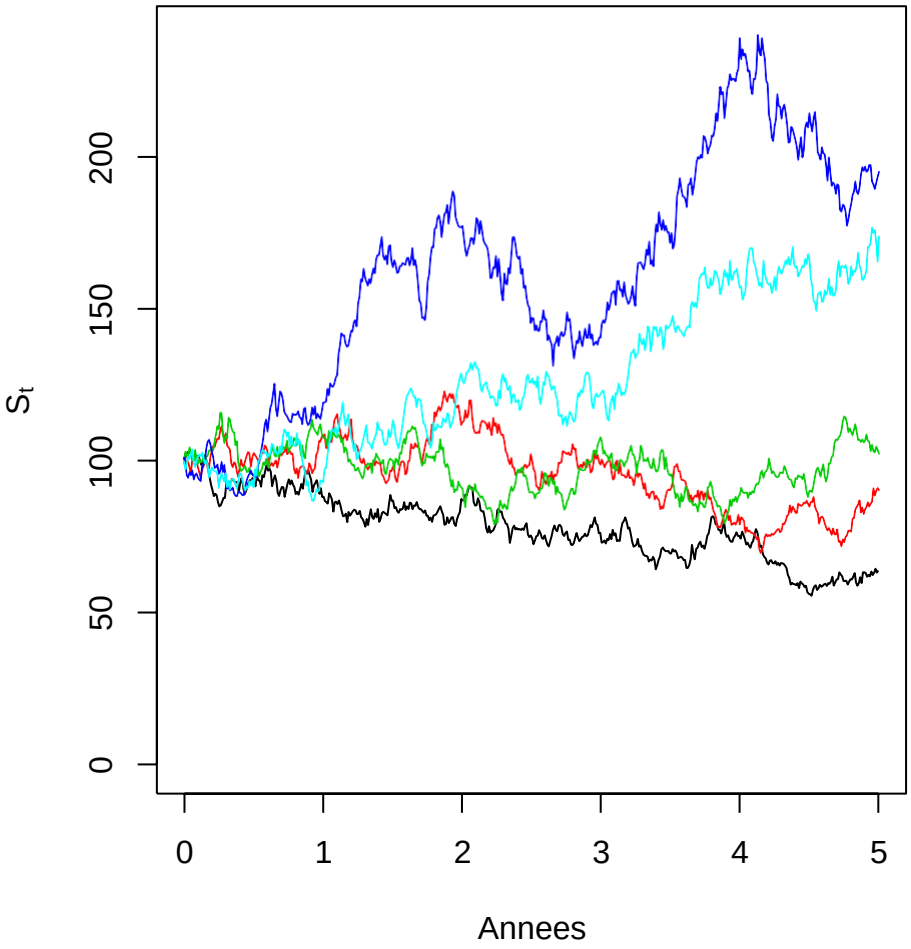

Figure 3.1 – *Quelques trajectoires du processus issu du modèle de Black–Scholes avec* $S_0 = 100$, $r = 0.05$, $\sigma = 0.15$.

Définition 3.1.1 (Option européenne)**.**
Une option est un contrat entre un acheteur et un vendeur. Cela donne a l'acheteur de l'option la possibilité, mais pas l'obligation, d'acheter (*call*) ou de vendre (*put*) un actif donné (action, devises. . .). Les prix fixés à l'avance ainsi que la durée de validité de l'option sont définis dans le contrat.

Il existe plusieurs type d'options (européenne, asiatique, américaine, . . .). L'option européenne signifie que le fait de bénéficier de son droit de vente/d'achat doit se faire à un instant T fixé par l'option. Ce n'est par exemple pas le cas d'une option américaine qui autorise cette vente/achat à n'importe quel moment d'une période fixée par l'option, i.e., sur $[0, T]$.

La formule dépend de cinq variables :

S_0 la valeur actuelle de l'actif sous-jacent ;

T la **date de maturité** ou **échéance**, i.e., le temps qui reste à l'option avant son échéance ;

K le **prix d'exercice** ou **strike**, i.e., le prix d'achat/vente de l'actif fixé par l'option ;

r le taux d'intérêt sans risque ;

σ la volatilité du prix de l'actif, i.e., son écart type annuel.

Dans le cas d'un call européen, le bénéfice (payoff in English) que je pourrais faire si je revendais immédiatement mon actif en $t = T$ serait bien entendu

$$\max\{S_t - K, 0\}.$$

Ainsi le « prix juste » d'un *call européen*, en finance on parle alors de prime de l'option, est alors défini par

$$C = \exp(-rT)\mathsf{E}\{\max(S_T - K, 0)\}. \tag{3.1}$$

Remarque. Le terme $\exp(-rT)$ est une correction due aux *intérêts composés*. Supposons que je place de M_0 euros aujourd'hui à un taux annuel de r et que les intérêts soient versés toutes les $1/n$ années, e.g., $n = 12$, alors au bout de T années j'aurais

$$M_T = M_0\left(1 + \frac{r}{n}\right)^{nT} \longrightarrow M_0 \exp(rT), \qquad n \to \infty.$$

Le fait de faire tendre $n \to \infty$ correspond aux *intérêts composés continûment* et convient parfaitement aux marchés financiers—hautes fréquences.

Ainsi le fait de multiplier par $\exp(rT)$ permet donc de « remonter le temps » afin de déterminer le prix juste de l'option au temps $t = 0$.

On peut montrer (nous ne le ferons pas) que le prix juste d'un call européen est

$$C = S_0\Phi(d_+) - \exp(-rT)K\Phi(d_-), \qquad d_\pm = \frac{1}{\sigma\sqrt{T}}\ln\frac{S_0}{K} + r \pm \frac{1}{2}\sigma^2\sqrt{T},$$

où Φ est la fonction de répartition d'une $N(0, 1)$.

Remarque. Pour un put européen, le prix juste de l'option est alors

$$P = \exp(-rT)\mathsf{E}\{\max(K - S_T, 0)\},$$

de sorte que l'on a

$$\begin{aligned}
C - P &= \exp(-rT)\mathsf{E}\{\max(S_T - K, 0) - \max(K - S_T, 0)\} \\
&= \exp(-rT)\mathsf{E}\{\max(S_T - K, 0) + \min(S_T - K, 0)\} \\
&= \exp(-rT)\mathsf{E}(S_T - K) \\
&= \exp(-rT)S_0\mathsf{E}\left[\exp\left(\left(r - \frac{\sigma^2}{2}\right)T + \sigma W_t\right)\right] - K\exp(-rT) \\
&= \exp(-rT)S_0\exp\left(\left(r - \frac{\sigma^2}{2}\right)T\right)\mathsf{E}\{\exp(\sigma W_t)\} - K\exp(-rT) \\
&\overset{\text{Laplace}}{=} \exp(-rT)S_0\exp\left(\left(r - \frac{\sigma^2}{2}\right)T\right)\exp\left(\frac{\sigma^2 T}{2}\right) - K\exp(-rT) \\
&= S_0 - K\exp(-rT).
\end{aligned}$$

*Culture générale : Cette égalité est connue sous le nom d'**arbitrage call–put**.*

3.2 Approche basique

L'intégration par Monte Carlo repose sur la loi (faible) des grands nombres.

Théorème 3.2.1 (Loi faible des grands nombres (rappel))**.**
Soient $X_1, X_2, \ldots$ sont des copies indépendantes d'une v.a. X d'espérance et de variance finies. Alors

$$\frac{1}{n}\sum_{i=1}^{n} X_i \xrightarrow{\text{proba}} \mathsf{E}(X) = \int_{\mathbb{R}} xf(x)dx, \qquad n \to \infty,$$

Démonstration.

Remarque. La version forte de la loi des grands nombres est bien plus puissante mais sa démonstration bien plus compliquée également. De plus elle ne nous sera pas d'une grande aide dans ce cours...

En effet bien souvent l'intégrale I peut s'écrire comme $I = \mathsf{E}(X)$ pour une v.a. X de loi convenable et l'on peut donc, via la loi des grands nombres, évaluer [3] (et donc pas calculer !) l'intégrale par

$$\hat{I} = \frac{1}{n}\sum_{i=1}^{n} f(X_i), \qquad X_1, \ldots, X_n \overset{\text{iid}}{\sim} X.$$

Bien entendu plus n sera grand plus notre évaluation sera précise... Le théorème central limite permet de connaître l'ordre de grandeur des erreurs commises.

Théorème 3.2.2 (Théorème central limite (rappel))**.**
Soient $X_1, X_2, \ldots$ sont des copies indépendantes d'une v.a. X d'espérance et de variance finies. Alors

$$\sqrt{n}\,\frac{\frac{1}{n}\sum_{i=1}^{n} X_i - \mathsf{E}(X)}{\sqrt{Var(X)}} \xrightarrow{\text{loi}} \mathcal{N}(0,1), \qquad n \to \infty.$$

Démonstration.

3. En statistiques on préfère alors dire *estimer*

Ce résultat montre que la vitesse de convergence vers I est en $\sqrt{n}$ mais dépend également de la variance de X. Clairement plus cette variance sera faible plus la précision sera grande...

Remarque. Pour bien enfoncer le clou, remarquez que la vitesse de convergence ne fait aucunement apparaître la dimension de l'intégrale... (sauf peut–être au travers de Var(X)).

Définition 3.2.1. On appellera précision de l'estimateur $\hat{I}$ la quantité $1.96\sqrt{\text{Var}(X)/n}$ que l'on estimera par

$$1.96\sqrt{\frac{(n-1)^2\sum_{i=1}^n (X_i-\bar{X})^2}{n}}, \qquad \bar{X}=\frac{1}{n}\sum_{i=1}^n X_i.$$

Note : vous verrez plus tard dans un autre cours que cela correspond à demi longueur d'un intervalle de confiance à 95%...

Revenons donc à nos moutons avec la formule de Black–Scholes (3.1) pour la prime d'un call européen. Avec tout ce que je viens de dire, on peut l'estimer facilement par

$$\frac{\exp(-rT)}{n}\sum_{i=1}^n \max\left(S_0\exp\left(r-\frac{\sigma^2}{2}\right)T+\sigma\sqrt{T}Z_i-K,0\right), \qquad Z_1,\ldots,Z_n \overset{\text{iid}}{\sim} N(0,1).$$

Exemple 3.1 (Black–Scholes simple)**.**
Application numérique avec $S_0 = 100$, $K = 100$, $\sigma = 0.15$, $r = 0.05$, $T = 1$.

```
> ## Fixons la graine pour pouvoir reproduire
> ## les memes resultats plus tard
> set.seed(123)
> S0 <- 100
> K <- 100
> sigma <- 0.15
> r <- 0.05
> t <- 1
```

```
>
> ## Calcul du prix theorique
> dplus <- (log(S0 / K) + (r + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> dminus <- (log(S0 / K) + (r - 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> call.theo <- S0 * pnorm(dplus) - exp(-r * t) * K * pnorm(dminus)
>
> n.sim <- 10^4
> Z <- rnorm(n.sim, 0, 1)
> St <- S0 * exp((r - 0.5 * sigma^2) * t + sigma * sqrt(t) * Z)
>
> call <- exp(-r * t) * pmax(St - K, 0)
>
> precision <- 1.96 * sqrt(var(call) / n.sim)
> c(call.theo, mean(call), precision)

[1] 8.5916583 8.5496363 0.2181641
```

3.3 Variables antithétiques

Supposons que nous ayons deux estimations[4] $\hat{I}_2$ et $\hat{I}_2$ de notre fameuse intégrale I. Ne serait il pas pertinent d'estimer alors I par $\tilde{I} = (\hat{I}_1 + \hat{I}_2)/2$? C'est l'idée de base des **variables antithétiques**.

Si de plus, comme se sera souvent le cas, $\mathrm{Var}(\hat{I}_1) = \mathrm{Var}(\hat{I}_2) = \mathrm{Var}(\hat{I})$, alors on a

$$\mathrm{Var}(\tilde{I}) = \frac{1}{2}\mathrm{Var}(\hat{I})(1 + \mathrm{Corr}(\hat{I}_1, \hat{I}_2),$$

de sorte que la $\mathrm{Var}(\tilde{I}) \leq \mathrm{Var}(\hat{I})$ dès lors que $\mathrm{Corr}(\hat{I}_1, \hat{I}_2) \leq 0$.

Lemme 3.3.1. *Soient $U \sim U(0,1)$ et g une fonction monotone définie sur $[0,1]$. Alors*

$$Corr\{g(U), g(1-U)\} < 0.$$

Démonstration.

4. Je devrais dire *estimateurs* mais vous ne l'avez pas encore vu alors…

En pratique on se sert du Lemme précédent avec $F^{-1}(U)$ et $F^{-1}(1-U)$ qui sont alors des v.a. négativement corrélées et de fonction de répartition F.

Remarque. Plus g est linéaire plus $g(U)$ et $g(1-U)$ sont négativement corrélées puisque évidemment $\mathrm{Corr}(U, 1-U) = -1$.

Exemple 3.2 (Black–Scholes antithétique)**.**
Du fait de la symétrie en 0 de la loi $N(0,1)$ on prendra comme variables antithétiques Z et $-Z$. Ainsi on a l'estimation suivante

$$\frac{1}{2n}\sum_{i=1}^{n} \hat{I}_1 + \hat{I}_2 \ ,$$

où $\hat{I}_1$ et $\hat{I}_2$ sont comme l'estimateur défini dans la Section 3 avec les Z_i et $-Z_i$ respectivement. Ceci s'écrit en R de la manière suivante

```
> ## Fixons la graine pour reproduire les memes resultats qu'avant
> set.seed(123)
> S0 <- 100
> K <- 100
> sigma <- 0.15
> r <- 0.05
> t <- 1
>
> ## Calcul du prix theorique
> dplus <- (log(S0 / K) + (r + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> dminus <- (log(S0 / K) + (r - 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> call.theo <- S0 * pnorm(dplus) - exp(-r * t) * K * pnorm(dminus)
>
> n.sim <- 10^4
> Z <- rnorm(n.sim, 0, 1)
> St <- S0 * exp((r - 0.5 * sigma^2) * t + sigma * sqrt(t) * Z)
> St.anti <- S0 * exp((r - 0.5 * sigma^2) * t - sigma * sqrt(t) * Z)
>
> call <- exp(-r * t) * pmax(St - K, 0)
> call.anti <- exp(-r * t) * pmax(St.anti - K, 0)
>
> cor(call, call.anti)

[1] -0.5781445

> precision.anti <- 1.96 * sqrt(0.5 * var(call) / n.sim *
+                                 (1 + cor(call, call.anti)))
>
> c(call.theo, mean(0.5 * (call + call.anti)), precision.anti)

[1] 8.5916583 8.5763419 0.1001959

> precision.anti / precision * 100

[1] 45.92687
```

On a donc augmenté la précision de notre estimation de près de 50% juste à l'aide d'un copié collé ! ! ! ! Des fois la fainéantise a du bon non ?

3.4 Variables de contrôle

Le principe des **variables de contrôle** repose sur l'égalité toute bête suivante

$$I = \mathbb{E}\{f(X)\} = \mathbb{E}\{f(X) - h(X)\} + \mathbb{E}\{h(X)\},$$

où, pour que la méthode ait un sens, $\mathbb{E}\{h(X)\} = \mu$ soit calculable explicitement mais aussi (et surtout !) que

$$\mathrm{Var}\{f(X) - h(X)\} \leq \mathrm{Var}\{f(X)\}.$$

Définition 3.4.1. La variable $h(X)$ est alors appelée **variable de contrôle**.

Ainsi on estimera I par une approche Monte–Carlo basique mais faîte sur $\mathbb{E}\{f(X) - h(X)\}$, i.e.,

$$\hat{I}_{\text{controle}} = \mu + \frac{1}{n}\sum_{i=1}^{n}\{f(X_i) - h(X_i)\}, \qquad X^1, \ldots, X_n \overset{\text{iid}}{\sim} X.$$

Clairement la variance de cet estimateur est alors

$$\mathrm{Var}(\hat{I}_{\text{controle}}) = n^{-1}\mathrm{Var}\{f(X) - h(X)\}.$$

Exemple 3.3 (Black–Scholes contrôle)**.**
Nous allons utiliser l'équation d'arbitrage put–call de la Remarque 3.1 pour définir une variable de contrôle.

En utilisant cette équation le prix juste d'un call européen peut se réécrire sous la forme

$$C = S_0 - K\exp(-rT) + \exp(-rT)\mathbb{E}\{\max(K - S_T, 0)\}.$$

On estime alors l'espérance par Monte Carlo classiquement, ce qui en R donne

```
> ## Fixons la graine pour reproduire les memes resultats qu'avant
> set.seed(123)
> S0 <- 100
> K <- 100
> sigma <- 0.15
> r <- 0.05
> t <- 1
>
> ## Calcul du prix theorique
> dplus <- (log(S0 / K) + (r + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> dminus <- (log(S0 / K) + (r - 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> call.theo <- S0 * pnorm(dplus) - exp(-r * t) * K * pnorm(dminus)
>
> n.sim <- 10^4
> Z <- rnorm(n.sim, 0, 1)
> St <- S0 * exp((r - 0.5 * sigma^2) * t + sigma * sqrt(t) * Z)
>
>
> put <- exp(-r * t) * pmax(K - St, 0)
> call <- put + S0 - K * exp(-r * t)
>
> precision.control <- 1.96 * sqrt(var(put) / n.sim)
>
> c(call.theo, mean(call), precision.control)
```

```
[1] 8.5916583 8.5878936 0.1235383

> precision.control / precision * 100

[1] 56.62631
```

Ce qui nous fait une précision accrue d'environ 40% par rapport à la première approche mais cependant une performance moindre par rapport à l'approche antithétique.

Cela dit on peut tout à fait combiner les deux ! ! ! Mais le calcul des erreurs sera alors plus complexe (et non mis)

```
> St.anti <- S0 * exp((r - 0.5 * sigma^2) * t - sigma * sqrt(t) * Z)
> put.anti <- exp(-r * t) * pmax(K - St.anti, 0)
> call.anti <- put.anti + S0 - K * exp(-r * t)
>
> cor(put, put.anti)

[1] -0.3475422

> precision.final <- 1.96 * sqrt(0.5 * var(put) / n.sim *
+                                  (1 + cor(put, put.anti)))
>
> c(call.theo, mean(0.5 * (call + call.anti)), precision.final)

[1] 8.59165831 8.64387624 0.07200217

> precision.final / precision * 100

[1] 32.80068
```

3.5 Échantillonnage préférentiel

L'échantillonnage préférentiel (*importance sampling* in English) est une technique qu'il faut à tout prix avoir dans sa trousse à outils. Bien que l'idée soit très simple, elle peut être d'une efficacité redoutable.

Nous allons motiver son utilisation par un exemple simpliste mais instructif. Supposons que nous soyons intéressés en l'évaluation de la fonction de répartition d'une $N(0, 1)$ en un point quelconque $x \in \mathbb{R}$, i.e.,

$$I(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{(2\pi)}} \exp\left(\frac{-y^2}{2}\right) dy.$$

Clairement on pourrait l'estimer par $\hat{I}(x) = n^{-1} \sum_{i=1}^{n} 1_{\{X_i \leq x\}}$ avec $X_1, \ldots, X_n \overset{iid}{\sim} N(0, 1)$ et sa variance vérifie, comme somme de Bernoulli,

$$\mathrm{Var}\{\hat{I}(x)\} = \frac{I(x)\{1 - I(x)\}}{n} \leq \frac{1}{4n}.$$

Nous en concluons que pour avoir une précision au moins égale à 10^{-4} il faut (approximativement)

$$2\sqrt{\frac{1}{4n}} \leq 10^{-4} \Leftrightarrow n \geq 10^8,$$

impliquant donc que cette approche est numériquement infaisable dès lors que l'on cherche à estimer de très faibles probabilités [5].

Essayons donc de faire mieux... Ainsi plutôt que de faire du Monte Carlo basique, i.e.,

$$I = \int h(x)\mathrm{d}x = \int m(x)f(x)\mathrm{d}x, \qquad f \text{ densité},$$

nous écrivons

$$I = \int m(x)\frac{f(x)}{g(x)}\mathrm{d}x,$$

où g est une nouvelle densité dont le support contient celui de $m \times f$.

Remarque. Parfois le rapport $w(x) = f(x)/g(x)$ est appelé **poids d'importance** ou **poids préférentiels**.

Ainsi en utilisant la loi des grands nombre comme précédemment, nous pouvons évaluer l'intégrale d'une nouvelle manière

$$\hat{I}_{\text{pref}} \frac{1}{n}\sum_{i=1}^{n} m(X_i)w(X_i), \qquad X_1, \ldots, X_n \overset{\text{iid}}{\sim} g.$$

Proposition 3.5.1. *Sous les conditions décrites juste avant, on a*

$$\mathsf{E}(\hat{I}_{\text{pref}}) = I, \qquad Var(\hat{I}_{\text{pref}}) = n^{-1}\left(\int m(x)^2\frac{f(x)}{g(x)}g(x)dx - I^2\right).$$

Démonstration.

□

On peut montrer (cf. Feuille 2) que la variance de l'estimateur par échantillonnage préférentiel est **nulle** lorsque $g(x) = m(x)f(x)/I$ ce qui est bien entendu d'une inutilité la plus totale puisque nous cherchons justement à estimer I ! ! ! Cela dit ce résultat nous suggère qu'un bon choix pour g devrait vérifier $g(x) \propto m(x)f(x)$.

5. Ce n'est pas juste un exemple théorique, pour l'étape de pré dimensionnement des barrages par exemple, la loi impose que l'ouvrage résiste à des événements de probabilités inférieures à 10^{-3} !

Exemple 3.4 (Black–Scholes avec une option peu « jouable »)**.**
Rappelons qu'un call européen donne la possibilité droit de vendre un actif au temps T aux prix S_T alors que nous l'avons acheté au prix K. Clairement le call européen est rentable dès lors que $S_T > K$ et le détenteur de l'option se fait un bénéfice de $S_T - K$—je néglige ici le prix de l'option.

Ainsi l'option a peu de chance d'être utilisée si $\Pr(S_T > K)$ est petite et nous nous ramenons donc à la situation expliquée en début de cette Section. C'est notamment le cas lorsque $S_0 \ll K$.

Plutôt que d'échantillonner selon une $N(0,1)$, prenons pour loi d'importance une $N(m,1)$ pour un $m \in \mathbb{R}$ bien choisi. On a donc

$$\hat{C}_{\text{pref}} = \frac{\exp(-rT)}{n} \sum_{i=1}^{n} \max\left\{ S_0 \exp\left(\left(r - \frac{\sigma^2}{2} \right) T + \sigma \sqrt{T} X_i \right) - K, 0 \right\} \exp\left(-\frac{X_i^2 - (X_i - m)^2}{2} \right)$$

$$= \frac{\exp(-rT)}{n} \sum_{i=1}^{n} \max\left\{ S_0 \exp\left(\left(r - \frac{\sigma^2}{2} \right) T + \sigma \sqrt{T} X_i \right) - K, 0 \right\} \exp\left(-\frac{2mX_i - m^2}{2} \right),$$

où $X_1, \ldots, X_n \overset{\text{iid}}{\sim} N(m, 1)$. Ce qui en R donne

```
> ## Fixons la graine pour reproduire les memes resultats qu'avant
> set.seed(123)
> S0 <- 50
> K <- 100
> sigma <- 0.15
> r <- 0.05
> t <- 1
>
> ## Calcul du prix theorique
> dplus <- (log(S0 / K) + (r + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> dminus <- (log(S0 / K) + (r - 0.5 * sigma^2) * t) / (sigma * sqrt(t))
> call.theo <- S0 * pnorm(dplus) - exp(-r * t) * K * pnorm(dminus)
>
> n.sim <- 10^4
> Z <- rnorm(n.sim, 0, 1)
>
> ## Monte-Carlo basique
> St <- S0 * exp((r - 0.5 * sigma^2) * t + sigma * sqrt(t) * Z)
> call <- exp(-r * t) * pmax(St - K, 0)
>
> ## Echantillonnage preferentiel
> m <- log(K / S0) / sigma
> X <- m + Z
>
> St.pref <- S0 * exp((r - 0.5 * sigma^2) * t + sigma * sqrt(t) * X)
> call.pref <- exp(-r * t) * pmax(St.pref - K, 0)
> weights <- exp(- m * X + 0.5 * m^2)
>
> c(call.theo, mean(call),

mean(call.pref * weights))[1]
```

```
1.983208e-05 0.000000e+00 2.005198e-05
```