Developping R packages with C++

A beginner guide

Hervé Perdry

2023-10-19

Table des matières

About this document

This document assumes that the reader is familiar with R; no previous knowledge of C++ is assumed.

The first chapter presents rapidly the main data structures used in R (vectors, matrix, factors, list, data frames), showing in particular how

The second chapter presents the very bases of C++ and RCpp. In this chapter you will use the function `Rcpp::sourceCpp` to compile C++ code from R. All the example code is available on github.

The third chapter shows how to create a R package. The resulting package can be installed from github.

After that, every chapter is associated to a package that you can install to test directly the code in it.

Les comparaisons de temps d'exécution qui apparaissent ici ont été obtenues avec une installation de R « standard » (pas de librairie comme openBlas ou autre), une compilation avec `clang++`, sur une machine linux disposant de 8 cœurs à 3.60 GHz, avec un cache de 8 MB. Des comparaisons avec d'autres compilateurs ou sur d'autres machines peuvent donner des résultats (très) différents, tant en valeur des temps d'exécution qu'en comparaison des performances.

L'idéal serait d'amener les lecteurs d'une part à une bonne connaissance des possibilités offertes par `Rcpp`, d'autre part au niveau nécessaire pour ouvrir `The C++ programming language` de Bjarne Stroustrup – on recommande, avant de se frotter à cet énorme et patibulaire ouvrage de référence (1300 pages), le plus court et plus amène `A tour of C++` du même auteur.

Chapitre 1

Objets de R

On supposera que les objets de R sont bien connus. Dans ce court chapitre nous allons simplement voir comment examiner leur structure.

1.1 R objects have types

L'instruction `typeof` permet de voir le type des objets. Considérons trois vecteurs, une matrice, une liste, un data frame, un facteur.

1.1.1 Numerical types

```
> typeof( c(1.234, 12.34, 123.4, 1234) )
[1] "double"
```

```
> typeof( runif(10) )
[1] "double"
```

```
> M <- matrix( rpois(12, 2), 4, 3)
> typeof(M)
[1] "integer"
```

```
> F <- factor( c("F", "M", "F", "F") )
> typeof(F)
[1] "integer"
```

Il y a deux types de variables numériques : `double` (nombres « à virgule », en format dit « flottant ») et `integer` (entiers). Les entiers s'obtiennent en tapant `0L`, `1L`, etc; certaines commandes renvoient des entiers:

```
> typeof(0)
[1] "double"
```

```
> typeof(0L)
[1] "integer"
```

```
> typeof(0:10)
[1] "integer"
```

```
> typeof( which(runif(5) > 0.5) )
[1] "integer"
```

```
> typeof( rpois(10, 1) )
[1] "integer"
```

On remarque que le facteur `F` a pour type `integer`. Ce petit mystère s'éclaircira bientôt.

1.1.2 Logical

We shall see later that, internally, the logical `TRUE` and `FALSE` are stored as integers `1` and `0`. They however have their proper type.

```
> typeof( c(TRUE, FALSE) )
[1] "logical"
```

1.1.3 Lists

Data frame are lists. This will be clarified soon.

```
> L <- list(a = runif(10), b = "dada")
> typeof(L)
[1] "list"
```

```
> D <- data.frame(x = 1:10, y = letters[1:10])
> typeof(D)
[1] "list"
```

1.1.4 A glimpse on the objects type

Pour examiner le contenu d'un objet avec une information sur son type, on peut utiliser `str`.

```
> str(M)
 int [1:4, 1:3] 2 0 2 1 4 2 0 5 5 2 ...
```

```
> str(F)
 Factor w/ 2 levels "F","M": 1 2 1 1
```

```
> str(L)
List of 2
 $ a: num [1:10] 0.4707 0.8502 0.4636 0.0762 0.1888 ...
 $ b: chr "dada"
```

```
> str(D)
'data.frame':   10 obs. of  2 variables:
 $ x: int  1 2 3 4 5 6 7 8 9 10
 $ y: chr  "a" "b" "c" "d" ...
```

1.2 R objects have attributes.

Les objets de R ont des « attributs ». Ainsi donner des noms aux éléments d'un vecteur revient à lui donner un attribut `names`.

```
> c <- runif(4)
> names(c) <- c("elt1", "elt2", "elt3", "elt4")
> c
     elt1      elt2      elt3      elt4
0.9969619 0.6725960 0.5119153 0.9567324
```

```
> attributes(c)
$names
[1] "elt1" "elt2" "elt3" "elt4"
```

Ce qui différentie une matrice d'un vecteur, c'est l'attribut `dim`:

```
> attributes(M)
$dim
[1] 4 3
```

Les data frames et les facteurs ont également des attributs :

```
> attributes(D)
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> attributes(F)
$levels
[1] "F" "M"

$class
[1] "factor"
```

Les attributs peuvent être modifiés avec la syntaxe `attributes(x) <- ...` ou un individuellement avec `attr(x, which)` :

```
> attr(M, "dim")
[1] 4 3
```

```
> attr(M, "dim") <- c(2L, 6L)
> M
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    2    2    4    0    5    2
[2,]    0    1    2    5    2    0
```

1.3 How to look further at the objects' structure

La fonction `dput` permet d'obtenir une forme qui peut être copiée dans une autre session R ; ceci permet parfois d'obtenir des informations plus précises sur la représentation interne d'un objet. Nous allons l'utiliser ici pour mieux comprendre la construction des matrices, des data frames, et des facteurs.

Il est nécessaire de jeter au préalable un œil à l'aide de `structure` pour mieux comprendre le résultat. On y précise notamment :

```
For historical reasons (these names are used when deparsing),
attributes '".Dim"', '".Dimnames"', '".Names"', '".Tsp"' and
'".Label"' are renamed to '"dim"', '"dimnames"', '"names"',
'"tsp"' and '"levels"'.
```

1.3.1 Matrices are vectors

```
> dput(M)
structure(c(2L, 0L, 2L, 1L, 4L, 2L, 0L, 5L, 5L, 2L, 2L, 0L), .Dim = c(2L,
6L))
```

Une matrice est un vecteur muni d'un attribut `dim` (qui apparaît comme `.Dim` dans le résultat de `dput`).

1.3.2 Data Frame are lists

```
> dput(D)
structure(list(x = 1:10, y = c("a", "b", "c", "d", "e", "f",
"g", "h", "i", "j")), class = "data.frame", row.names = c(NA,
-10L))
```

Un data frame est une liste munie d'un attribut `class = "data.frame"` et d'un attribut `row.names` (ici, la valeur de cet attribut est la convention pour « 4 lignes non nommées »).

1.3.3 Factors are integer vectors

```
> dput(F)
structure(c(1L, 2L, 1L, 1L), .Label = c("F", "M"), class = "factor")
```

```
> levels(F)
[1] "F" "M"
```

Un facteur est qu'un vecteur d'entiers muni d'attributs `class = "factor"`. et `levels` (les niveaux du facteur), qui apparaît dans `structure` sous le nom `.Label` ; cet attribut est également accessible via la fonction `levels`.

On peut par exemple fabriquer un facteur à partir d'un vecteur d'entiers, ainsi :

```
> G <- c(2L, 1L, 1L, 2L)
> attributes(G) <- list(levels = c("L1", "L2"), class = "factor")
> G
[1] L2 L1 L1 L2
Levels: L1 L2
```

1.4 Pour les apprentis sorciers

La fonction interne `inspect` permet de voir l'adresse où se trouve l'objet, son type (d'abord codé numériquement, par exemple 13 pour `integer` puis le nom conventionnel de ce type, `INTSXP`), et quelques autres informations ; les objets complexes (leurs attributs) sont déroulés.

```
> .Internal(inspect( 1:10 ))
@5581dc2e4d58 13 INTSXP g0c0 [REF(65535)]  1 : 10 (compact)
```

```
> .Internal(inspect( c(0.1, 0.2) ))
@5581dc294da8 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2
```

```
> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@5581dc357618 14 REALSXP g0c2 [REF(2)] (len=2, tl=0) 0.1,0.2
```

```
> names(a) <- c("A", "B")
> .Internal(inspect( M ))
@5581db577768 13 INTSXP g0c4 [REF(5),ATT] (len=12, tl=0) 2,0,2,1,4,...
ATTRIB:
  @5581db5ec5b8 02 LISTSXP g0c0 [REF(1)]
    TAG: @5581d7a2c390 01 SYMSXP g1c0 [MARK,REF(3708),LCK,gp=0x4000] "dim" (has value)
    @5581db5eae40 13 INTSXP g0c1 [REF(65535)] (len=2, tl=0) 2,6
```

```
> .Internal(inspect( L ))
@5581da582028 19 VECSXP g1c2 [MARK,REF(3),ATT] (len=2, tl=0)
  @5581d8c5bb68 14 REALSXP g1c5 [MARK,REF(2)] (len=10, tl=0) 0.470737,0.850207,0.463579,0.0762363,0.188
  @5581da326868 16 STRSXP g1c1 [MARK,REF(4)] (len=1, tl=0)
    @5581da326830 09 CHARSXP g1c1 [MARK,REF(1),gp=0x60] [ASCII] [cached] "dada"
ATTRIB:
  @5581d98a2f30 02 LISTSXP g1c0 [MARK,REF(1)]
    TAG: @5581d7a2bfa0 01 SYMSXP g1c0 [MARK,REF(65535),LCK,gp=0x4000] "names" (has value)
    @5581da581fe8 16 STRSXP g1c2 [MARK,REF(65535)] (len=2, tl=0)
      @5581d7d0f268 09 CHARSXP g1c1 [MARK,REF(36),gp=0x61] [ASCII] [cached] "a"
      @5581d8033408 09 CHARSXP g1c1 [MARK,REF(15),gp=0x61] [ASCII] [cached] "b"
```

Les plus braves pourront consulter le code de cette fonction, ainsi que tout le code de R, à cette adresse : [https://github.com/wch/r-source/tree/trunk/src] plus précisément pour `inspect`, dans `src/main/inspect.c`...

```
## Loading required package: readr
```

Chapitre 2

Introducing C++

Types (integer, floats, bool), arrays, and flow control statements.

2.1 Using `Rcpp::sourceCpp`

If you are using linux, install R, Rstudio and the Rcpp package (use `install.packages("Rcpp")`), as well as a C++ compiler such as g++.

If you are using Windows or macOS, install R, Rstudio, the Rcpp package, and Rtools. The simplest way to make sure everything works is to follow the following instructions:

1. Installer Rcpp via la commande install.packages("Rcpp")
2. Cliquer dans le menu `file > new > c++ file` (ou l'équivalent en français) pour créer un nouveau fichier C++ ; un fichier contenant quelques lignes d'exemples va être créé. Sauvez ce fichier puis cliquez sur Source. Rstudio doit vous proposer d'installer "Rtools" : acceptez.
3. Cliquer à nouveau sur Source. Tout doit fonctionner...! Vous êtes prêt à apprendre le C++.

Les exemples de code proposés utilisent souvent le standard C++11. Pour pouvoir compiler avec ce standard, n'omettez pas la ligne `// [[Rcpp::plugins(cpp11)]]` in the source files. Altenatively, using `Sys.setenv("PKG_CXXFLAGS" = "-std=c++11")` inside a R session will enable C++11 compilation once for all.

2.2 Hello world

```
// file: vec.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector vec0(int n) {
  Rcpp::NumericVector x(n);
  return x;
}

// accès aux éléments
// [[Rcpp::export]]
```

```
Rcpp::IntegerVector vec1(int n) {
  Rcpp::IntegerVector x(n);
  for(int i = 0; i < n; i++) {
    x[i] = i*i;
  }
  return x;
}
```

Il faut toujours commencer par saluer le monde. Créez un fichier `hello.cpp` contenant le code suivant :

```
// file: hello.cpp
#include <Rcpp.h>
#include <iostream>
//[[Rcpp::export]]
void hello() {
  Rcpp::Rcout << "Hello world!\n";
}
```

Compilez le depuis R (il faut avoir installé le package Rcpp) :

```
library(Rcpp)
sourceCpp("hello.cpp")
```

(ou, si vous utilisez R Studio, cliquez sur « source »…). Appelez ensuite la fonction en R :

```
> hello()
Hello world!
```

Dans le programme C++, les directives d'inclusion `#include` servent à inclure des librairies. La librairie `Rcpp.h` permet l'interaction avec les objets de R ; la définition de l'objet `Rcpp::Rcout`, un « flux de sortie » (output stream) qui permet l'écriture dans la console R y est incluse. La librairie `iostream` contient en particulier la définition de l'opérateur `<<`. Elle n'est en fait pas nécessaire ici car `Rcpp.h` contient une directive d'inclusion similaire.

2.3 Why types are necessary

C++ est un langage compilé et non interprété. Le compilateur est le programme qui lit le code C++ et produit un code assembleur puis du langage machine (possiblement en passant par un langage intermédiaire).

Les instructions de l'assembleur (et du langage machine qui est sa traduction numérique directe) manipulent directement les données sous formes de nombre codés en binaire, sur 8, 16, 32 ou 64 bits. La manipulation de données complexes (des vecteurs, des chaines de caractères) se fait bien sûr en manipulant une suite de tels nombres.

Pour que le compilateur puisse produire de l'assembleur, il faut qu'il sache la façon dont les données sont codées dans les variables. La conséquence est que toutes les variables doivent déclarées, et ne pourrons pas changer de type ; de même, le type des valeurs retournées par les fonctions doit être fixé, ainsi que celui de leurs paramètres.

Les fantaisies permises par R (voir ci-dessous) ne sont plus possibles (étaient-elles souhaitables ?).

```
fantaisies <- function(a) {
  if(a == 0) {
    return(a)
  } else {
    return("Non nul")
  }
}
```

```
> fantaisies(0)
[1] 0
```

```
> fantaisies(1)
[1] "Non nul"
```

```
> fantaisies("0")
[1] "0"
```

```
> fantaisies("00")
[1] "Non nul"
```

La librairie standard de C++ offre une collection de types de données très élaborés et de fonctions qui les manipulent. Nous commencerons par les types fondamentaux : entiers, flottants, booléens.

2.4 Integers

There are several types of integers in C++.

2.4.1 The four main types of integers.

Compilez ce programme qui affiche la taille (en octets) des quatre types d'entiers (signés) (le résultat peut théoriquement varier d'une architecture à l'autre, c'est-à-dire qu'il n'est pas fixé par la description officielle du C++).

```
// file: int_types.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void int_types() {
  char a;
  short b;
  int c;
  long int d;
  int64_t e;

  Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcout << "sizeof(c) = " << sizeof(c) << "\n";
  Rcout << "sizeof(d) = " << sizeof(d) << "\n";
  Rcout << "sizeof(e) = " << sizeof(e) << "\n";
}
```

Notez une nouveauté ci-dessus : la directive `using namespace Rcpp` qui permet de taper `Rcout` au lieu de `Rcpp::Rcout`. C'est commode mais à utiliser avec parsimonie (et à bon escient) : il n'est en effet pas rare que des fonctions appartenant à des namespace différents portent le même nom. La syntaxe `namespace::fonction` permet d'éviter toute ambiguïté.

```
> int_types()
sizeof(a) = 1
sizeof(b) = 2
sizeof(c) = 4
sizeof(d) = 8
sizeof(e) = 8
```

Une compilation sous Windows ne produit pas les mêmes résultats: les `long int` ne font que 32 bits. Pour une implémentation portable, la solution est d'utiliser des types où la taille est explicite, comme `int64_t`.

Les entiers de R correspondent au type `int`(sur 32 bits) mais cela ne vous empêche pas de manipuler dans vos fonctions C++ des entiers plus courts ou plus longs si vous en avez besoin.

2.4.2 Unsigned integers

Il existe aussi des types non signés, par exemple `unsigned int` ou `unsigned char` ; et des raccourcis variés, par exemple `size_t` pour `unsigned long int` ou `uint16_t` pour des entiers non signées de 8 bits.

```
// file: non_signes.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void non_signes() {
  int16_t x = 32766;
  uint16_t y = 32766;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
}
```

Sur 16 bits, les entiers non signés vont de -32768 à 32767, et les entiers signés de 0 à 65535:

```
> non_signes()
x = 32766, y = 32766
x = 32767, y = 32767
x = -32768, y = 32768
x = -32767, y = 32769
```

2.4.3 Numerical overflow

```cpp
// file: overflow.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void overflow() {
  unsigned short x(65530);
  Rcout << "x = " << x << "\n" ;
  x = x+5;
  Rcout << "x = " << x << "\n" ;
  x = x+5;
  Rcout << "x = " << x << "\n" ;
}
```

```
> overflow()
x = 65530
x = 65535
x = 4
```

2.4.4 Notre première fonction « non void »

Écrivons notre première fonction qui renvoie une valeur. Son type doit être déclaré comme ceci :

```cpp
// file: somme_entiers.cpp
//[[Rcpp::export]]
int somme_entiers(int a, int b) {
  return a+b;
}
```

Et testons la :

```
> somme_entiers(1L, 101L)
[1] 102
```

```
> somme_entiers(1.9,3.6)
[1] 4
```

Que se passe-t-il ? Utilisez la fonction suivante pour comprendre.

```cpp
// file: cast_to_int.cpp
//[[Rcpp::export]]
int cast_to_int(int x) {
  return x;
}
```

2.4.5 Initialisation des variables

Il est nécessaire d'initialiser les variables.

```cpp
// file: uninit.cpp
//[[Rcpp::export]]
int uninit() {
  int a; // a peut contenir n'importe quoi
  return a;
}
```

Testons :

```
> uninit()
[1] 0
```

```
> uninit()
[1] 0
```

On aura parfois 0, mais pas systématiquement (cela dépend de l'état de la mémoire). On peut initialiser `a` lors de la déclaration : `int a = 0;`

2.5 Les flottants

Il y a trois types de nombres en format flottant. Le type utilisé par R est le `double` de C++.

2.5.1 The free types of floating point numbers

```cpp
// file: float_types.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void float_types() {
  float a;
  double b;
  long double c;
  Rcpp::Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcpp::Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcpp::Rcout << "sizeof(c) = " << sizeof(c) << "\n";
}
```

2.5.2 Précision du calcul

Parenthèse de culture informatique générale. Voici ce que répond R au test `1.1 - 0.9 = 0.2`.

```
> 1.1 - 0.9 == 0.2
[1] FALSE
```

Pourquoi ? Est-ce que C++ fait mieux ? (Rappel : R utilise des `double`).

Sur les architectures courantes, les nombres au format `double` sont codés sur 64 bits (voir ci-dessus, taille 8 octets). C'est un format « à virgule flottante », c'est-à-dire qu'ils sont représentés sous la forme $a2^b$, ou a et b sont bien sûr codés en binaire (sur 53 bits – dont un bit 1 implicite – pour a, 11 pour b, et un bit de signe).

Cette précision finie implique des erreurs d'arrondi. Pour plus de détails, voir Wikipedia sur la norme IEEE 754 : https://fr.wikipedia.org/wiki/IEEE_754

Quelle est la différence entre les nombres ci-dessus ?

```
> (1.1 - 0.9) - 0.2
[1] 5.551115e-17
```

C'est-à-dire 2^{-54} (une erreur au 53e chiffre…). Affichons la représentation interne des nombres en question avec la fonction `bits` du package `pryr`.

```
> pryr::bits(1.1 - 0.9)
[1] "00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011100"
```

```
> pryr::bits(0.2)
[1] "00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011010"
```

2.5.3 Valeurs spéciales et extrêmes

Il y a des valeurs spéciales en C++ comme en R : une valeur infinie, et une valeur non-définie `NaN`, pour *not a number*.

```
// file: divise.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
double divise(double a, double b) {
  double r = a/b;
  Rcpp::Rcout << a << " / " << b << " = " << r << std::endl;
  return r;
}
```

```
> divise(1,2)
1 / 2 = 0.5
[1] 0.5
```

```
> divise(1,0)
1 / 0 = inf
[1] Inf
```

```
> divise(-1,0)
-1 / 0 = -inf
[1] -Inf
```

```
> divise(0,0)
0 / 0 = -nan
[1] NaN
```

En C++, la fonction `numeric_limits` permet d'obtenir les valeurs extrêmes que peuvent prendre les `double`.

```cpp
// file: numeric_limits.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void numeric_limits() {
  Rcpp::Rcout
    << "plus petite valeur positive 'normale' = "
    << std::numeric_limits<double>::min()         << "\n"
    << "plus petite valeur positive           = "
    << std::numeric_limits<double>::denorm_min()  << "\n"
    << "plus grande valeur                    = "
    << std::numeric_limits<double>::max() << "\n"
    << "epsilon                               = "
    << std::numeric_limits<double>::epsilon()     << "\n";
}
```

```
> numeric_limits()
plus petite valeur positive 'normale' = 2.22507e-308
plus petite valeur positive           = 4.94066e-324
plus grande valeur                    = 1.79769e+308
epsilon                               = 2.22045e-16
```

2.5.4 Constantes numériques

Attention, si le R considère que `0` ou `1` est un `double` (il faut taper `0L` ou `1L` pour avoir un `integer`), pour C++ ces valeurs sont des entiers. Pour initialiser proprement un `double` il faudrait normale taper `0.` ou `0.0`, etc; cependant le compilateur fera la conversion de type si nécessaire.

2.6 Opérateurs arithmétiques

Les opérateurs arithmétiques sont bien entendu `+`, `-`, `*` et `/`. Pour les entiers, le modulo est `%`.

```cpp
// file: division_entiere.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void division_entiere(int a, int b) {
  int q = a / b;
  int r = a % b;
  Rcpp::Rcout << a << " = " << b << " * " << q << " + " << r << std::endl;
}
```

```
> division_entiere(128, 7)
128 = 7 * 18 + 2
```

À ces opérateurs, il faut ajouter des opérateurs d'assignation composée `+=`, `-=`, `*=` et `/=` qui fonctionnent ainsi : `x += 4;` est équivalent à `x = x + 4`, et ainsi de suite. Il y a aussi les opérateurs d'incrémentation `++` et de décrémentation `--`.

```cpp
// file: operateurs_exotiques.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
```

```
void operateurs_exotiques(int a) {
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "a *= 2;" << std::endl;
  a *= 2;
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "int b = a++;" << std::endl;
  int b = a++; // post incrementation
  Rcpp::Rcout << "b = " << b << std::endl;
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "int c = ++a;" << std::endl;
  int c = ++a; // pre incrementation
  Rcpp::Rcout << "c = " << c << std::endl;
  Rcpp::Rcout << "a = " << a << std::endl;
}
```

```
> operateurs_exotiques(3)
a = 3
a *= 2;
a = 6
int b = a++;
b = 6
a = 7
int c = ++a;
c = 8
a = 8
```

2.7 Conversion de type: les “cast”

Le compilateur peut réaliser une conversion d’un type à l’autre: on parle de *cast*. Cette conversion peut être implicite, lors par exemple d’une copie d’un type `double` vers un type `int` ; elle peut être explicite, lors par exemple de la copie d’une valeur de type `double` vers un `int` ; elle peut être rendue explicite en mettant un nom de type entre parenthèses devant une variable : `(int) x` fera une conversion de `x` vers le type `int` (si le type de `x` rend ça possible).

```
// file: cast.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void cast(int x, int y) {
  double a = x; // cast implicite
  double b = (double) y; // cast explicite
  double q1 = x / y; // cast implicite (à quel moment a-t-il lieu ?)
  double q2 = (double) x / (double) y; // cast explicite
  Rcpp::Rcout << "q1 = " << q1 << "\n";
  Rcpp::Rcout << "q2 = " << q2 << "\n";
}
```

Cet exemle montre les écueils du cast implicite :

```
> cast(4,3)
q1 = 1
q2 = 1.33333
```

Lors du calcul de `q1`, le cast a été fait après la division entière… était-ce le comportement désiré ?

2.8 Booléens

Le type `bool` peut prendre les valeurs vrai/faux. Il correspond au type `logical` de R.

```
// file: test_positif.cpp
// [[Rcpp::export]]
bool test_positif(double x) {
  return (x > 0);
}
```

Les opérateurs de test sont comme en R, >, >=, <, <=, == et !=. Les opérateurs logiques sont && (et), || (ou) et ! (non). **Attention !** Les opérateurs & et | existent également, ce sont des opérateurs logiques bit à bit qui opèrent sur les entiers.

```
// file: test_interval.cpp
// [[Rcpp::export]]
bool test_interval(double x, double min, double max) {
  return (min <= x && x <= max);
}
```

2.9 Tableaux de taille fixe

On peut définir des tableaux de taille fixe fixe (connue à la compilation) ainsi:

```
// file: petit_tableau.cpp
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void petit_tableau() {
  int a[4] = {0,2,7,11};
  SHOW(a)    // wut ?
  SHOW(a[0])
  SHOW(a[1])
  SHOW(a[2])
  SHOW(a[3])
}
```

L'occasion est saisie pour montrer l'utilisation d'une macro. La ligne `SHOW(a[0])` est remplacée par `Rcpp::Rcout << "a[0]" << " = " << (a[0]) << std::endl;` **avant** la compilation. Les macros peuvent rendre de grand services pour la clarté du code ou pour faciliter le débuggage « manuel ».

L'utilisation de parenthèse autour de `(x)` dans la définition de la macro est très conseillée : si on utilisait par exemple `SHOW(a == b)` il n'y a aucun problème avec la syntaxe `Rcout << (a == b) << std::endl;` mais `Rcout << a == b << std::endl;` pourrait poser des problèmes de priorité des opérateurs == et <<…

Le résultat de `SHOW(a)` sera expliqué plus tard (pointeurs).

2.10 Contrôle du flux d'exécution : les boucles

2.10.1 Boucles `for`

Plus de 90% des boucles `for` s'écrivent ainsi :

```
// file: ze_loop.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void ze_loop(int n) {
  for(int i = 0; i < n; i++) {
    Rcpp::Rcout << "i = " << i << std::endl;
  }
}
```

```
> ze_loop(4)
i = 0
i = 1
i = 2
i = 3
```

Le premier élément dans la parenthèse (ici, `int i = 0`) est l'initialisation ; il sera exécuté une seule fois, et c'est généralement une déclaration de variable (avec une valeur initiale). Le deuxième élément (`i < n`) est la condition à laquelle la boucle sera exécutée une nouvelle fois, c'est généralement une condition sur la valeur de la variable ; et le dernier élément (`i++`) est exécuté à la fin de chaque tour de boucle, c'est généralement une mise à jour de la valeur de cette variable.

Il est facile par exemple d'aller de 2 en 2 :

```
// file: bouclette.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void bouclette(int n) {
  for(int i = 0; i < n; i += 2) {
    Rcpp::Rcout << "i = " << i << std::endl;
  }
}
```

```
> bouclette(6)
i = 0
i = 2
i = 4
```

Pour revenir sur les types d'entiers : gare au dépassement arithmétique.

```
// file: arithmetic_overflow.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void arithmetic_overflow() {
  int x = 1;
  for(int i = 0; i < 33; i++) {
    Rcpp::Rcout << "2^" << i << " = " << (x) << "\n";
    x = 2*x;
```

```
  }
}
```

Essayer avec `unsigned int`, `long int`.

2.10.2 `continue` et `break`

Une instruction `continue` en cours de boucle fait passer au tour suivant :

```
// file: trois.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void trois(int n) {
  for(int i = 1; i <= n; i++) {
    Rcpp::Rcout << i << " " ;
    if(i%3 != 0)
      continue;
    Rcpp::Rcout << "\n";
  }
  Rcpp::Rcout << "\n";
}
```

```
> trois(9)
1 2 3
4 5 6
7 8 9
```

Quant à `break`, si s'agit bien sûr d'une interruption de la boucle.

```
// file: zz.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void zz(int n, int z) {
  for(int i = 0; i < n; i++) {
    Rcpp::Rcout << "A" ;
    if(i > z)
      break;
  }
  Rcpp::Rcout << std::endl;
}
```

```
> zz(14, 100)
AAAAAAAAAAAAAA
```

```
> zz(14, 5)
AAAAAAA
```

2.10.3 Boucles `while` et `do while`

Ces boucles ressemblent fort à ce qui existe en R. Dans un cas, le test est fait avant la boucle, dans l'autre il est fait après.

```cpp
// file: a_rebours_1.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void a_rebours_1(int n) {
  while(n-- > 0) {
    Rcpp::Rcout << n << " ";
  }
  Rcpp::Rcout << std::endl;
}

// [[Rcpp::export]]
void a_rebours_2(int n) {
  do {
    Rcpp::Rcout << n << " ";
  } while(n-- > 0);
  Rcpp::Rcout << std::endl;
}
```

```
> a_rebours_1(3)
2 1 0
```

```
> a_rebours_2(3)
3 2 1 0
```

On peut aussi utiliser `continue` et `break` dans ces boucles.

Considérons un exemple un peu moins artificiel : le calcul d'une racine carrée par l'algorithme de Newton. L'avantage de la syntaxe `do while` est apparent ici.

```cpp
// file: squareRoot.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
double squareRoot(double x, double eps = 1e-5) {
  double s = 1;
  do {
    s = 0.5*(s + x/s);
  } while( fabs(s*s - x) > eps);
  return s;
}
```

```
> squareRoot(2)
[1] 1.414216
```

```
> squareRoot(2, 1e-8)
[1] 1.414214
```

Cherchez sur le site `cpprefrence.com` la description des fonctions `abs` et `fabs`. Pourquoi ne pouvait-on pas utiliser `abs` ici ? Est-il raisonnable de proposer une valeur trop petite pour `eps` ? Proposer une modification de la fonction qui évite cet écueil.

2.11 Contrôle du flux d'exécution : les alternatives

2.11.1 `if` et `if else`

Cela fonctionne tout à fait comme en R ; la x

```
// file: mini.cpp
// [[Rcpp::export]]
double mini(double x, double y) {
  double re = 0;
  if(x > y) {
    re = y;
  } else {
    re = x;
  }
  return re;
}
```

```
> mini(22, 355)
[1] 22
```

2.11.2 `switch`

Un exemple simple devrait permettre de comprendre le fonctionnement de `switch`.

```
// file: combien.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void combien(int n) {
  switch(n) {
    case 0:
      Rcpp::Rcout << "aucun\n";
      break;
    case 1:
      Rcpp::Rcout << "un seul\n";
      break;
    case 2:
      Rcpp::Rcout << "deux\n";
      break;
    case 3:
    case 4:
    case 5:
      Rcpp::Rcout << "quelques uns\n";
      break;
    default:
      Rcpp::Rcout << "beaucoup\n";
  }
}
```

Chapitre 3

Création d'un package R

Pour diffuser votre travail, ou pour ne pas avoir besoin de recompiler vos fonctions à chaque fois, il faut créer un package R. `Rstudio` est d'une grande aide pour cela. Nous allons dans ce chapitre faire les premiers pas dans la création d'un package ; le tome du manuel de R intitulé *Writing R extensions* reste une référence indispensable.

Le package que nous allons créer s'appelle `introRcppPackages`. Vous pouvez le retrouver à l'adresse `https://github.com/introRcpp/introRcppPackages`.

3.1 Créer l'arborescence de fichiers

Pour commencer sélectionner le menu `File` l'option `New Project`, puis `New Directory`, puis `R package using Rcpp`. Vous allez pouvoir choisir à quel endroit le nouveau répertoire qui va contenir votre package (et portera son nom) sera installé. Nous choisissons d'appeler notre package `introRcppPackages`.

Le répertoire `introRcppPackages/` contient:

- les fichiers `DESCRIPTION` et `NAMESPACE`
- les répertoires `R/`, `src/` et `man/` dont nous allons parler plus bas
- des fichiers qui ne font pas partie du package :
  - un fichier `Read-and-delete-me` (obtempérez)
  - un fichier `introRcppPackages.Rproj` et un dossier (caché) `.Rproj.user/` qui sont utilisés par `Rstudio`
  - un fichier `.Rbuilbignore` qui contient des expressions régulières destinées à informer R de la présence de fichiers qui ne font pas partie du package…

Le contenu de `DESCRIPTION` est assez clair – vous pouvez et devez le modifier:

```
Package: introRcppPackages
Type: Package
Title: What the Package Does in One 'Title Case' Line
Version: 1.0
Date: 2020-03-24
Author: Your Name
Maintainer: Your Name <your@email.com>
Description: One paragraph description of what the package does as one or more full
          sentences.
License: GPL (>= 2)
Imports: Rcpp (>= 1.0.3)
LinkingTo: Rcpp
```

Il est possible d'inclure des informations supplémentaires, par exemple un champ `Encoding` pour spécifier la façon dont les éventuelles lettres accentuées sont encodées (`latin1` et `UTF-8` sont les solutions les plus fréquentes). J'insère pour ma part la ligne

```
Encoding: UTF-8
```

qui correspond à l'encodage par défaut sous Linux et Mac OS et me permet d'accentuer correctement mon prénom dans le champ `Author`. Les utilisateurs de Windows choisiront peut-être plus commodément l'encodage `latin1`, mais Rstudio peut gérer l'encodage de votre choix et vous demande de choisir lors de la première sauvegarde d'un fichier.

Le fichier `NAMESPACE` contient deux lignes importantes pour l'utilisation de fonctions écrites avec Rcpp:

```
useDynLib(introRcppPackages, .registration=TRUE)
importFrom(Rcpp, evalCpp)
```

La ligne

```
exportPattern("^[[:alpha:]]+")
```

dit à R que toutes les fonctions dont le nom commence par un caractère alphanumérique sont exportées du package. C'est très bien quand on ne développe que pour soi, pour un package destiné à la diffusion il est souvent nécessaire de modifier cela. Nous le ferons plus tard.

3.2 Inclure une fonction C++

Les fonctions C++ sont dans le répertoire `src/`. Il contient deux fichiers, `rcpp_hello_world.cpp` qui contient quelques exemples basiques ; et `RcppExports.cpp`, qui est généré par la fonction `Rcpp::compileAttributes()`. Dans un premier temps vous pouvez ignorer son contenu.

Créons un fichier `squareRoot.cpp`, contenant

```
#include <Rcpp.h>
#include <cmath>
// [[Rcpp::export]]
double squareRoot(double x) {
  if(x < 0)
    return NAN;
  double s = 1;
  // an eps that's scale with x
  double eps = x * std::numeric_limits<double>::epsilon() * 2;
  do {
    s = 0.5*(s + x/s);
  } while( fabs(s*s - x) > eps);
  return s;
}
```

Le nom du fichier n'a pas besoin de coincider avec celui de la fonction, c'est juste plus commode pour s'y retrouver. Vous pouvez maintenant cliquer sur `Install and Restart` (sous l'onglet `Build`). Rstudio compile le package et relance la session R, puis charge le package. Vous pouvez tester la fonction `squareRoot` !

```
library(introRcppPackages)
squareRoot(123)
## [1] 11.09054
```

3.3 Ce que Rstudio a fait avant la compilation

Rstudio a appelé la fonction `Rcpp::compileAttributes()` qui a modifié le fichier `RcppExports.cpp`. Elle a créé cette fonction

```
RcppExport SEXP _introRcppPackages_squareRoot(SEXP xSEXP) {
BEGIN_RCPP
    Rcpp::RObject rcpp_result_gen;
    Rcpp::RNGScope rcpp_rngScope_gen;
    Rcpp::traits::input_parameter< double >::type x(xSEXP);
    rcpp_result_gen = Rcpp::wrap(squareRoot(x));
    return rcpp_result_gen;
END_RCPP
}
```

qui est en fait celle qui est appelée par R. Comment cela ? La fonction R qui correspond est dans le fichier `R/RcppExports.R`:

```
squareRoot <- function(x) {
    .Call(`_introRcppPackages_squareRoot`, x)
}
```

3.4 Inclure une fonction R

La fonction `squareRoot` n'est pas totalement satisfaisante. Une bonne idée serait de vérifier – dans le code R – que l'utilisateur a bien passé un unique élément de type double. On peut créer dans le répertoire `R/` un fichier qu'on appelera par exemple `square.root.r` et qui contient

```
square.root <- function(x) {
  if( typeof(x) != "double" )
    stop("This function works on doubles")
  if( length(x) != 1 )
    stop("This function works on single numbers")
  squareRoot(x)
}
```

3.5 Contrôler quelles sont les fonctions exportées

Puisqu'on a créé `square.root`, on ne veut pas que l'utilisateur puisse utiliser `squareRoot`. On va donc modifier notre `NAMESPACE` pour qu'il contienne

```
useDynLib(introRcppPackages, .registration=TRUE)
importFrom(Rcpp,evalCpp)
export(square.root)
```

Ainsi la seule fonction exportée par notre package est `square.root`. On peut toujours, à nos risques et périls, utiliser la fonction non exportée avec la syntaxe `introRcppPackages:::squareRoot` (notez le triple deux-points).

3.6 Paramétrer la compilation

On peut inclure dans `src/` un fichier `Makevars` qui permet notamment de préciser quels flags doivent être inclus dans la commande de compilation. Si on utilise du C++11, il faut notamment inclure une ligne

```
PKG_CXXFLAGS = -std=c++11
```

sinon la compilation ne sera pas possible.

3.7 Documenter les fonctions avec `roxygen2`

Les fichiers de documentation sont inclus dans le répertoire `man/`. Ce sont des fichiers en `.Rd` qui peuvent être écrits à la main ; une solution qui s'avère à l'usage très commode (facilité d'écriture d'une part, de maintenance du package d'autre part) est de les faire générer par `roxygen2`.

Pour cela il faut tout d'abord installer ce package : `install.packages("roxygen2")`. Il est possible que l'installation soit pénible car ce package dépend de `xml2` qui nécessite d'installer d'autres composantes logicielles sur le système. Soyez attentifs aux messages d'erreur, ils sont informatifs. Une solution simple sous un linux de type Ubuntu est d'utiliser le gestionnaire de paquets pour installer `r-cran-xml2` ou `r-cran-roxygen2` !

Pour documenter la fonction `square.root`, placez le curseur dans cette fonction puis cliquez sur la baguette magique et choisissez `Insert Roxygen Skeleton`. Votre fichier ressemble maintenant à ceci:

```
#' Title
#'
#' @param x
#'
#' @return
#' @export
#'
#' @examples
square.root <- function(x) {
  if( typeof(x) != "double" )
    stop("This function works on doubles")
  if( length(x) != 1 )
    stop("This function works on single numbers")
  squareRoot(x)
}
```

On va compléter cette ébauche ainsi:

```
#' Computes a square root
#'
#' @param x a double vector of length 1
#'
#' @details This function is for pedagogical illustration only. Please use
#' `base::sqrt` in R or `sqrt` in C++.
#'
```

```
#' @return The square root of `x`
#' @export
#'
#' @examples
#' square.root(2)
#'

square.root <- function(x) {
  if( typeof(x) != "double" )
    stop("This function works on doubles")
  if( length(x) != 1 )
    stop("This function works on single numbers")
  squareRoot(x)
}
```

Maintenant, on va faire générer à Rstudio le fichier `man/square.root.Rd` qui correspond. Pour cela, il faut d'abord aller dans l'onglet `Build`, puis cliquer sur `More`, `Configure Build Tools`, cocher la case `Generate documentation with Roxygen` (cocher au minimum la case `Rd files`).

Ensuite, cliquez sur `Build > More > Document` pour faire générer ce fichier. Comme notre page d'aide contient des lettres accentuées, cela ne fonctionnera que si vous avez inséré dans `DESCRIPTION` la ligne `Encoding: UTF-8`. Le message d'erreur en l'absence de ce champ n'est pas très instructif, l'information se trouve dans un des warnings qui suit. En pratique, la documentation est généralement écrite en anglais, et les lettres accentuées y sont presque toujours absentes.

Une fois la documentation générée (regardez le contenu de `man/square.root.Rd`) vous pouvez réinstaller le package et admirer la page de documentation en tapant `?square.root` et `example(square.root)`.

3.8 Générer le fichier `NAMESPACE` avec `roxygen2`

Le tag `@export` de `roxygen2` signale que cette fonction est exportée. Cela permet à `roxygen2` de générer, en plus de l'aide, le fichier `NAMESPACE` et nous évite d'insérer à la main des lignes d'exportation comme `export(square.root)`. Cependant `roxygen2` refuse (sagement) d'effacer un `NAMESPACE` qu'il n'a pas généré lui-même.

Pour y remédier il semble qu'il n'y ait pas de meilleure solution que d'insérer au début du fichier `NAMESPACE` la ligne suivante, qui permet à `roxygen2` de considérer que le `NAMESPACE` peut être effacé:

```
# Generated by roxygen2
```

Il existe sûrement une solution plus propre. Un problème subsiste cependant: `roxygen2` ne génère pas les deux lignes indispensables au fonctionnement d'un package avec `Rcpp`, que nous avons mentionnées plus haut. La solution est d'insérer dans le répertoire `R/` un fichier à cette fin. Nous l'appellerons par exemple `zzz.r` et il contiendra les lignes suivantes:

```
#' @useDynLib introRcppPackages, .registration=TRUE
#' @importFrom Rcpp evalCpp
NULL
## NULL
```

Le `NULL` final peut être remplacé par un `0` ou ce que vous voulez (des appels aux fonctions `.onLoad` et `.onAttach` par exemple), mais il faut qu'il y ait un objet R à évaluer sinon le fichier n'est pas pris en compte par `roxygen2`.

square.root {introRcppPackages} R Documentation

Computes a square root

Description

Computes a square root

Usage

```
square.root(x)
```

Arguments

x	a double vector of length 1

Details

This function is for pedagogical illustration only. Please use 'base::sqrt' in R or 'sqrt' in C++.

Value

The square root of 'x'

Examples

```
square.root(2)
```

FIGURE 3.1 – Documentation de la fonction `square.root`

Chapitre 4

Manipuler les objets de R en C++

All the examples are in the R package... Install it with

```
> devtools::install_github("introRcpp/introRcppManipulation")
```

Load it with

```
library(introRcppManipulation)
##
## Attaching package: 'introRcppManipulation'
## The following objects are masked from 'package:introRcppBases':
##
##     vec0, vec1
```

4.1 Premiers objets Rcpp : les vecteurs

La librairie `Rcpp` définit des types `NumericVector`, `IntegerVector` et `LogicalVector` qui permettent de manipuler en C++ les vecteurs de R.

4.1.1 Créer des vecteurs, les manipuler

L'initialisation avec la syntaxe utilisée dans `vec0` remplit le vecteur de 0. Notez l'accès aux éléments d'un vecteur par l'opérateur `[]`; **contrairement à la convention utilisée par R, les vecteurs sont numérotés de 0 à n-1.**

```
// file: vec.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector vec0(int n) {
  Rcpp::NumericVector x(n);
  return x;
}

// accès aux éléments
// [[Rcpp::export]]
```

```
Rcpp::IntegerVector vec1(int n) {
  Rcpp::IntegerVector x(n);
  for(int i = 0; i < n; i++) {
    x[i] = i*i;
  }
  return x;
}
```

4.1.2 Exemple : compter les zéros

```
// file: countZeroes.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
int countZeroes(Rcpp::IntegerVector x) {
  int re = 0;
  // x.size() et x.length() renvoient la taille de x
  int n = x.size();
  for(int i = 0; i < n; i++) {
    if(x[i] == 0) ++re;
  }
  return re;
}
```

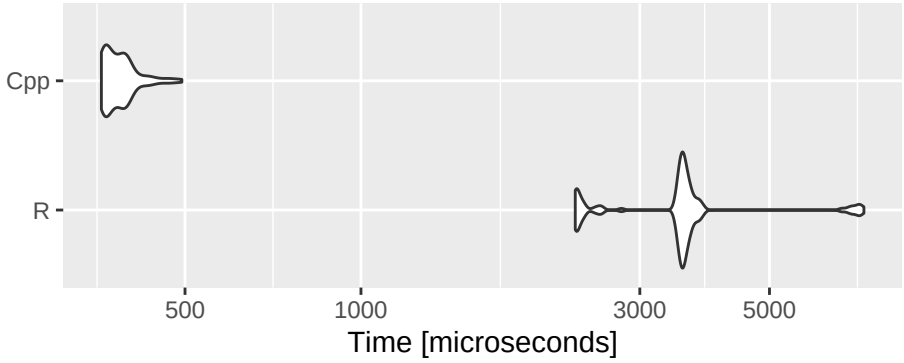
Comment les performances de cette fonction se comparent-elles avec le code R `sum(a == 0)` ?

```
> set.seed(1); a <- sample(0:99, 1e6, TRUE)
> countZeroes(a);
[1] 10017
```

```
> sum(a == 0)
[1] 10017
```

```
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp = countZeroes(a))
> mbm
Unit: microseconds
 expr      min        lq      mean   median        uq      max neval
    R 2326.257 3472.6710 3591.7389 3553.180 3636.5925 7254.144   100
  Cpp  359.655  365.2895  387.2598  380.917  397.4795  493.845   100
```

```
> ggplot2::autoplot(mbm)
```



La différence de vitesse d'exécution s'explique en partie par le fait que le code R commence par créer un vecteur de type `logical` (le résultat de `a == 0`), puis le parcourt pour faire la somme. Ceci implique beaucoup de lectures écritures en mémoire, ce qui ralentit l'exécution.

4.2 Vecteurs

4.2.1 Creating vectors

On a vu l'initialisation avec la syntaxe `NumericVector R(n)` qui crée un vecteur de longueur n, rempli de 0. On peut utiliser `NumericVector R(n, 1.0)` pour un vecteur rempli de 1 ; **attention à bien taper `1.0` pour avoir un `double` et non un `int`; dans le cas contraire, on a un message d'erreur difficilement compréhensible à la compilation**.

On peut utiliser `NumericVector R = no_init(n);` pour un vecteur non initialisé (ce qui fait gagner du temps d'exécution).

```
// file: zeros.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector zeros(int n) {
  IntegerVector R(n);
  return R;
}
```

```
// file: whatever.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector whatever(int n, int a) {
  IntegerVector R(n, a);
  return R;
}
```

```
// file: uninitialized.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector uninitialized(int n) {
```

```
  IntegerVector R = no_init(n);
  return R;
}
```

```
// file: favourites.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector favourites() {
  IntegerVector R = IntegerVector::create(1, 4, 8);
  return R;
}
```

```
> zeros(5)
[1] 0 0 0 0 0
```

```
> whatever(5, 2L)
[1] 2 2 2 2 2
```

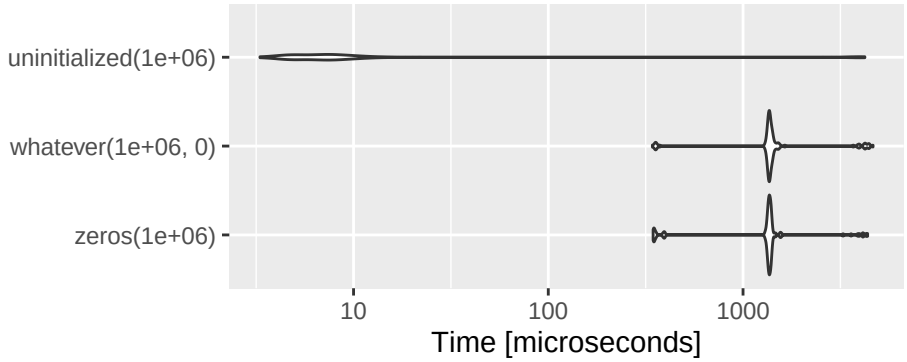
```
> uninitialized(5) # sometime 0s, not always
[1] -621187928      21889 -659868480      21889 -677197840
```

```
> favourites()
[1] 1 4 8
```

Comparons les performances des trois premières fonctions (comme à chaque fois, les résultats peuvent varier d'une architecture à l'autre).

```
> mbm <- microbenchmark::microbenchmark(zeros(1e6), whatever(1e6, 0), uninitialized(1e6))
> mbm
Unit: microseconds
                 expr     min        lq     mean    median       uq      max neval
        zeros(1e+06) 345.337 1338.1780 1449.917 1365.7535 1392.537 4363.645   100
   whatever(1e+06, 0) 342.420 1348.8335 1649.334 1371.1670 1417.071 4653.625   100
 uninitialized(1e+06)   3.307    4.8605  285.606    7.0805    8.801 4223.919   100
```

```
> ggplot2::autoplot(mbm)
```



4.2.2 Accessing elements

Using `x.size()` or `x.length()`. Beware O-based indices.

BLA BLA

4.2.3 Missing data

Cette fonction utilise `IntegerVector::is_na` qui est la bonne manière de tester si un membre d'un vecteur entier est `NA`.

```
// file: countNAs.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int countNAs(NumericVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += NumericVector::is_na(x[i]);
  }
  return(re);
}
```

Une nouveauté : le fichier `countNAS.h` ...

```
// file: countNAs.h
#include <Rcpp.h>
int countNAs(Rcpp::NumericVector x);
```

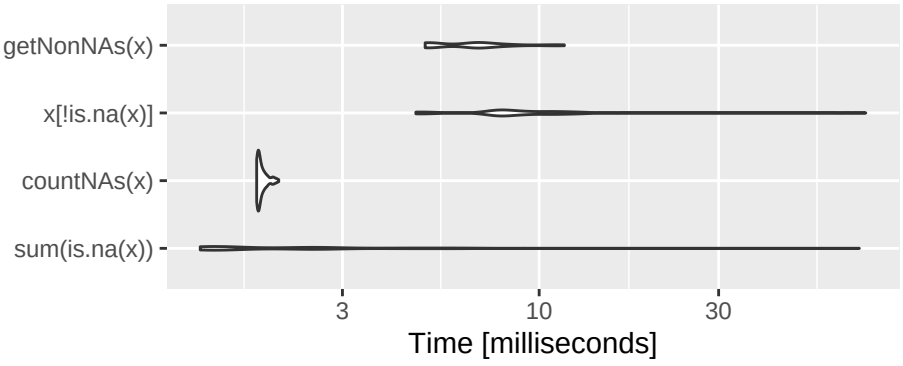
```
// file: getNonNAs.cpp
#include <Rcpp.h>
#include "countNAs.h"
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector getNonNAs(NumericVector x) {
  int nbNAs = countNAs(x);
  int n = x.size();
  NumericVector R(n - nbNAs);
  int j = 0;
  for(int i = 0; i < n; i++) {
    if(!NumericVector::is_na(x[i])) {
      R[j++] = x[i];
    }
  }
  return R;
}
```

Comparons ces deux fonctions avec leurs analogues R, `sum(is.na(x))` et `x[!is.na(x)]`.

```
> x <- sample( c(NA, rnorm(10)), 1e6, TRUE)
> mbm <- microbenchmark::microbenchmark( sum(is.na(x)), countNAs(x), x[!is.na(x)], getNonNAs(x) )
```

```
> mbm
Unit: milliseconds
          expr      min       lq     mean   median        uq       max neval
 sum(is.na(x)) 1.256205 1.354164 2.969596 1.525771  2.532421 70.990360   100
   countNAs(x) 1.774865 1.787200 1.831224 1.804517  1.853992  2.031615   100
  x[!is.na(x)] 4.701734 7.722991 9.303408 8.111391 10.354427 73.832494   100
  getNonNAs(x) 4.976842 5.101481 6.716029 6.788512  7.042927 11.670396   100
```

```
> ggplot2::autoplot(mbm)
```



4.3 Vecteurs nommés

Ça n'est pas passionnant en soi (on ne manipule pas si souvent des vecteurs nommés), mais ce qu'on voit là sera utile pour les listes et les data frames.

4.3.1 Créer des vecteurs nommés

Voici d'abord comment créer un vecteur nommé.

```
// file: createVec1.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector createVec1() {
  NumericVector x = NumericVector::create(Named("un") = 10, Named("deux") = 20);
  return x;
}
```

Application :

```
> a <- createVec1()
> a
  un deux
  10   20
```

Une syntaxe plus dense est possible :

```
// file: createVec2.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector createVec2() {
  NumericVector x = NumericVector::create(_["un"] = 10, _["deux"] = 20);
 return x;
}
```

Cela produit le même résultat.

```
> createVec2()
  un deux
  10   20
```

4.3.2 Accéder aux éléments par leurs noms

On utilise toujours la syntaxe `x[]` :

```
// file: getOne.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
double getOne(NumericVector x) {
  if(x.containsElementNamed("one"))
    return x["one"];
  else
    stop("No element 'one'");
}
```

Notez la fonction `Rcpp::stop` qui correspond à la fonction R du même nom.

```
> getOne(a)
Error in eval(expr, envir, enclos): No element 'one'
```

```
> getOne(b)
Error in getOne(b): object 'b' not found
```

4.3.3 Obtenir les noms d'un vecteur

Et voici comment obtenir les noms d'un vecteur.

```
// file: names1.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector names1(NumericVector x) {
  CharacterVector R = x.names(); // ou R = x.attr("names");
  return R;
}
```

Utilisons cette fonction:

```
> names1(a)
[1] "un"   "deux"
```

Cette fonction semble se comporter correctement, elle a cependant un gros défaut. Nous y reviendrons dans la section suivante.

4.4 Objets génériques : SEXP

Les objets R les plus génériques sont les `SEXP`, « S expression ». Les principaux types de `SEXP` sont illustrés par la fonction suivante.

```
// file: RType.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
std::string RType(SEXP x) {
  switch( TYPEOF(x) ) {
    case INTSXP:
      return "integer";
    case REALSXP:
      return "double";
    case LGLSXP:
      return "logical";
    case STRSXP:
      return "character";
    case VECSXP:
      return "list";
    case NILSXP:
      return "NULL";
    default:
      return "autre";
  }
}
```

Utiliser les types définis par Rcpp est généralement plus facile et plus sûr. Cependant à l'intérieur des fonctions Rcpp ils peuvent être utiles, par exemple dans le cas où une fonction peut renvoyer des objets de types différents, par exemple soit un `NILSXP`, soit un objet d'un autre type.

4.4.1 Exemple : vecteurs nommés (ou pas)

Testons à nouveau la fonction `names1`, en lui passant un vecteur non nommé.

```
> b <- seq(0,1,length=6)
> names1(b)
Error in eval(expr, envir, enclos): Not compatible with STRSXP: [type=NULL].
```

Bien sûr, le vecteur `b` n'a pas de noms ; la fonction `x.names()` a renvoyé l'objet `NULL`, de type `NILSXP`, qui ne peut être utilisé pour initialiser le vecteur `R` de type `STRSXP`. Une solution est d'attraper le résultat de `x.names()` dans un `SEXP`, et de tester son type avec `TYPEOF`.

```cpp
// file: names2.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector names2(NumericVector x) {
  SEXP R = x.names();
  if( TYPEOF(R) == STRSXP )
    return R;
  else
    return CharacterVector(0);
}
```

```
> names2(a)
[1] "un"   "deux"
```

```
> names2(b)
character(0)
```

4.4.2 Exemple : énumerer les noms et le contenu

On va utiliser l'opérateur `CHAR` qui, appliqué à un élément d'un `CharacterVector`, renvoie une valeur de type `const char *` c'est-à-dire un pointeur vers une chaîne de caractère (constante, ie non modifiable) « à la C » (voir chapitre dédié).

```cpp
// file: enumerate.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void enumerate(NumericVector x) {
  SEXP r0 = x.names();
  if(TYPEOF(r0) != STRSXP) {
    Rcout << "No names\n";
    return;
  }
  CharacterVector R(r0);
  for(int i = 0; i < R.size(); i++) {
    double a = x[ CHAR(R[i]) ];
    Rcout << CHAR(R[i]) << " : " << a << "\n";
  }
}
```

```
> enumerate(a)
un : 10
deux : 20
```

4.5 Facteurs

```cpp
// file: getLevels.cpp
#include <Rcpp.h>
```

```
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector getLevels(IntegerVector x) {
  SEXP R = x.attr("levels");
  switch(TYPEOF(R)) {
  case STRSXP:
    return R; // Rcpp prend soin que ce SEXP soit converti en CharacterVector
  case NILSXP:
    stop("No 'levels' attribute");
  default:
    stop("'levels' attribute of unexpected type");
  }
}
```

```
> x <- factor( sample(c("M","F"), 10, TRUE) )
> getLevels(x)
[1] "F" "M"
```

```
> x <- sample(1:2, 10, TRUE)
> # getLevels(x)
> attr(x, "levels") <- c(0.1, 0.4)
> # getLevels(x)
```

```
// file: someFactor.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector someFactor() {
  IntegerVector x = IntegerVector::create(1,1,2,1);
  x.attr("levels") = CharacterVector::create("F", "M");
  x.attr("class") = CharacterVector::create("factor");
  return x;
}
```

```
> someFactor()
[1] F F M F
Levels: F M
```

4.6 Listes et Data frames

Nous avons déjà vu les fonctions utiles dans le cas des vecteurs nommés, en particulier `containsElementNamed`.

La fonction suivante prend une liste `L` qui a un élément `L$alpha` de type `NumericVector` et renvoie celui-ci à l'utilisateur. En cas de problème un message d'erreur informatif est émis.

```
// file: getAlpha.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector getAlpha(List x) {
  if( x.containsElementNamed("alpha") ) {
```

```
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("alpha is not of type 'NumericVector'");
    return R;
  } else
    stop("No element named alpha");
}
```

Pour renvoyer des valeurs hétéroclites dans une liste c'est très facile:

```
// file: createList.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
List createList() {
  List L;
  L["a"] = NumericVector::create(1.0, 2.0, 4.0);
  L["b"] = 12;
  L["c"] = rnorm(4, 0.0, 1.0);
  return L;
}
```

```
> createList()
$a
[1] 1 2 4

$b
[1] 12

$c
[1] -1.2275839 -0.7487483  1.4054582 -0.1656251
```

Les data frames, ont l'a vu, sont des listes avec quelques attributs supplémentaires. En Rcpp cela fonctionne de la même façon, avec la classe `DataFrame`. Ils ont une certaine tendance à se transformer en liste quand on leur ajoute des éléments.

Here is a useful trick.

```
// file: createDF.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
DataFrame createDF() {
  List L;
  L["a"] = NumericVector::create(1.0, 2.0, 4.0, 8.0);
  L["b"] = CharacterVector::create("alpha", "beta", "gamma", "delta");
  L["c"] = rnorm(4, 0.0, 1.0);

  L.attr("class") = "data.frame";
  L.attr("row.names") = IntegerVector::create(NA_INTEGER, -4); // 4 unnamed rows
  return L;
}
```

```
> createDF()
  a     b          c
1 1 alpha -0.2627742
2 2  beta  0.3077454
3 4 gamma  1.1554909
4 8 delta  1.7582734
```

Chapitre 5

Sucre syntaxique

All the examples are in the R package… Install it with

```
> devtools::install_github("introRcpp/introRcppSugar")
```

Load it with

```
library(introRcppSugar)
##
## Attaching package: 'introRcppSugar'
## The following object is masked from 'package:introRcppManipulation':
##
##     countZeroes
```

La fonction R suivante fait un usage abondant de la vectorisation.

```
bonne.vectorisation <- function(x, y) {
  z <- 3*x + y
  if(any(x > 1))
    z <- z*2;
  sum( ifelse(z > 0, z, y) )
}
```

```
> set.seed(1); x <- rnorm(10); y <- rnorm(10)
> bonne.vectorisation(x,y)
[1] 26.99719
```

La transcription en C++ devrait faire intevenir trois boucles ; c'est un peu fastidieux. Le sucre syntaxique ajouté par les fonctions dites *Rcpp sugar* permet d'éviter de les écrire.

```
// file: fonction_sucree.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double fonction_sucree(NumericVector x, NumericVector y) {
  NumericVector z = 3*x + y;
  if( is_true(any(x > 1)) )
```

```
    z = z*2;
  return sum(ifelse(z > 0, z, y));
}
```

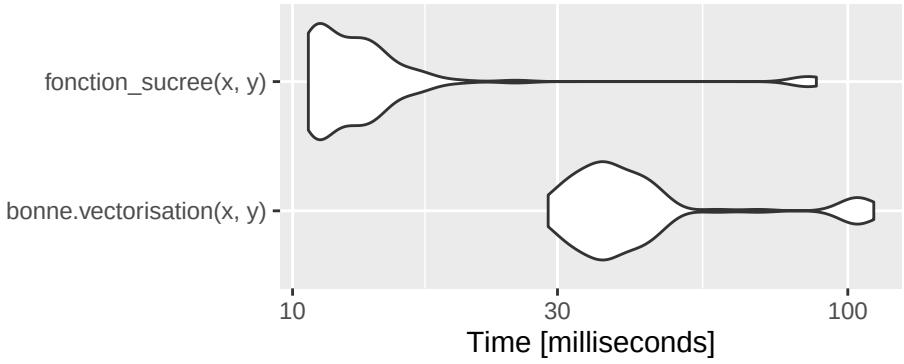
```
> fonction_sucree(x,y)
[1] 26.99719
```

5.1 Efficacité

Grâce à une implémentation soignée, les fonctions Rcpp sugar sont redoutablement efficaces.

```
> x <- rnorm(1e6)
> y <- rnorm(1e6)
> mbm <- microbenchmark::microbenchmark( bonne.vectorisation(x,y), fonction_sucree(x,y) )
> mbm
Unit: milliseconds
                      expr      min       lq     mean   median       uq       max neval
 bonne.vectorisation(x, y) 28.86106 34.07250 45.06973 37.28758 43.40480 111.38734   100
     fonction_sucree(x, y) 10.67524 11.03996 15.75968 12.62982 14.14164  87.79044   100
```

```
> ggplot2::autoplot(mbm)
```



Revenons cependant à notre exemple de comptage de 0 dans un vecteur.

```
// file: countZeroesSugar.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int countZeroesSugar(IntegerVector x) {
  return sum(x == 0);
}
```

Comparons cette solution à celle proposée au chapitre précédent.

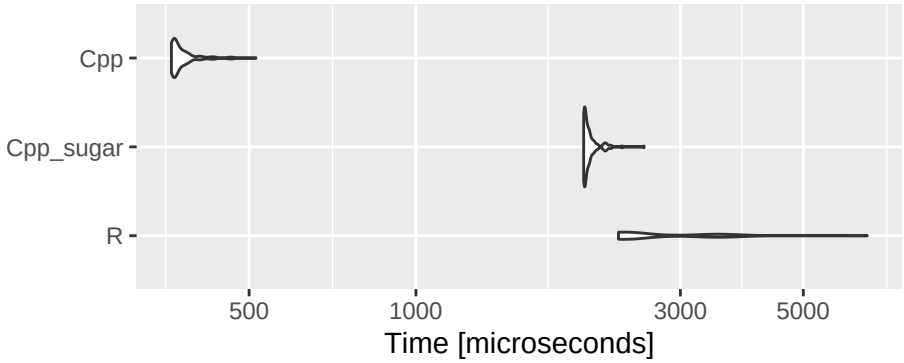
```
> set.seed(1); a <- sample(0:99, 1e6, TRUE)
> sum(a == 0);
[1] 10017
```

```
> countZeroesSugar(a);
[1] 10017
```

```
> countZeroes(a);
[1] 10017
```

```
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp_sugar = countZeroesSugar(a), Cpp = countZ
> mbm
Unit: microseconds
      expr      min        lq      mean    median        uq      max neval
         R 2319.400 2360.1975 2890.1546 2397.3765 3462.1045 6521.738   100
 Cpp_sugar 2007.557 2012.9200 2059.9149 2026.8840 2060.5935 2580.442   100
       Cpp  362.044  365.3935  381.2169  369.7425  384.2095  514.262   100
```

```
> ggplot2::autoplot(mbm)
```



On le voit, la fonction qui utilise le sucre syntaxique, bien que toujours très efficace, n'atteint pas toujours la performance d'une fonction plus rustique.

5.2 Un exemple de la Rcpp Gallery

Un exemple tiré de la Rcpp Gallery http://gallery.rcpp.org/articles/simulating-pi/ (estimation de π par la méthode de Monte-Carlo) et la variante avec une boucle.

```
// file: pi_sugar.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double pi_sugar(const int N) {
    NumericVector x = runif(N);
    NumericVector y = runif(N);
    // NumericVector d = sqrt(x*x + y*y);
    NumericVector d = x*x + y*y;
    return 4.0 * sum(d < 1.0) / N;
}

// [[Rcpp::export]]
double pi_boucle(const int N) {
```

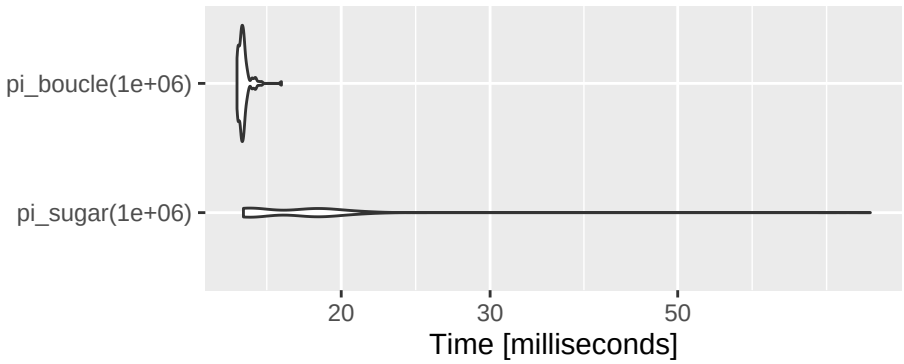
```
  int S = 0;
  for(int i = 0; i < N; i++) {
    double x = R::runif(0, 1);
    double y = R::runif(0, 1);
    S += (x*x + y*y) < 1.0; // cast implicite bool vers int
  }
  return (4.0 * (double) S / (double) N);
}
```

```
> pi_sugar(1e6)
[1] 3.14262
```

```
> pi_boucle(1e6)
[1] 3.141956
```

```
> mbm <- microbenchmark::microbenchmark(pi_sugar(1e6), pi_boucle(1e6))
> mbm
Unit: milliseconds
             expr      min       lq     mean   median       uq      max neval
  pi_sugar(1e+06) 15.32390 15.53474 18.14639 18.27965 18.99974 84.49878   100
 pi_boucle(1e+06) 15.05848 15.18475 15.32830 15.27909 15.37383 16.99576   100
```

```
> ggplot2::autoplot(mbm)
```



Chapitre 6

Exemple : Metropolis-Hastings

Install with

```
> devtools::install_github("introRcpp/introRcppMetropolis")
```

Load with

```
library(introRcppMetropolis)
```

6.1 L'algorithme

L'algorithme de Metropolis-Hastings permet de faire des tirages aléatoires dans une loi de densité proportionnelle à une fonction $\pi(x)$ positive – il n'y a pas besoin que $\int \pi(x)dx = 1$, autrement dit, on n'a pas besoin de connaître la constante de normalisation.

Nous présentons tout d'abord la notion de marche aléatoire, puis l'algorithme de Metropolis-Hastings.

6.1.1 Marche aléatoire

Une suite de valeurs aléatoires $x_1, x_2, \cdots \in \mathbb{R}^d$ est une marche aléatoire [1] si chaque point x_{t+1} est tiré dans une loi dont la densité ne dépend que x_t. On pourra noter $q(x|x_t)$ cette densité.

Un exemple simple est la marche aléatoire gaussienne : la densité $q(x|x_t)$ est la densité d'une la loi normale de variance $\sigma^2 I_d$ et d'espérance x_t. Cela revient à dire que

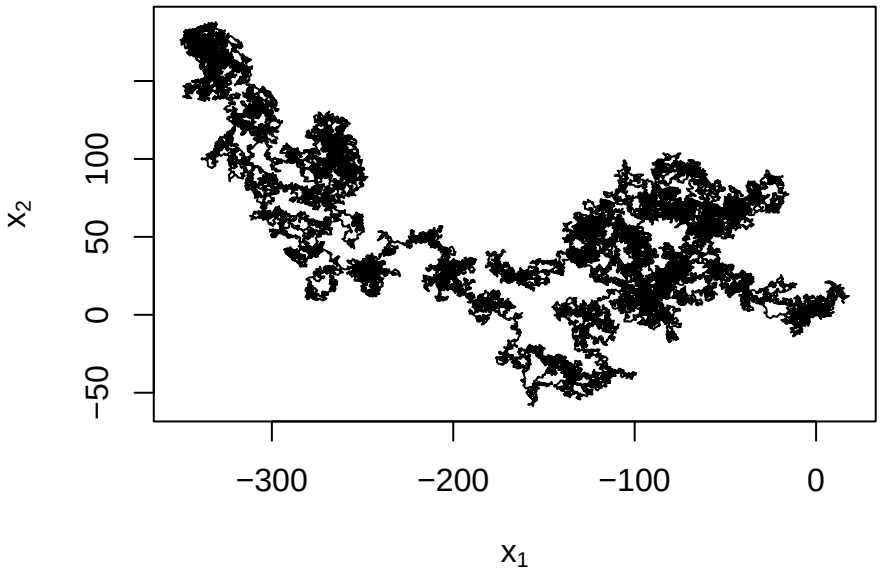
$$x_{t+1} = x_t + z$$

avec z tiré dans une loi normale centrée de variance $\sigma^2 I$.

La fonction suivante permet d'illustrer ceci avec $d = 2$. Elle réalise `B` étapes d'une marche aléatoire dont le point de départ est $x_1 = (0,0)$. Le résultat est présenté sous la forme d'une matrice à `B` lignes et deux colonnes. Le paramètre `sd` permet de spécifier la valeur de σ.

1. On pourra trouver d'autres définitions pour « marche aléatoire ». La définition donnée ici est celle d'une chaîne de Markov dite « homogène »ou parfois « stationnaire ».

```
Marche <- function(B, sd) {
  R <- matrix(0.0, nrow = B, ncol = 2)
  x <- R[1,]
  for(b in 2:B) {
    x <- x + rnorm(2, 0, sd = sd)
    R[b, ] <- x
  }
  return(R);
}
X <- Marche(5e4, sd = 0.8)
plot(X[,1], X[,2], xlab = expression(x[1]), ylab = expression(x[2]), type = "l")
```



6.1.2 L'algorithme

Voici l'algorithme pour faire des tirages dans une loi de densité proportionnelle à une fonction positive $\pi(x)$, définie sur $\mathbb{R}^d$. On part d'un point x_1 arbitraire, ou bien tiré au hasard dans une loi bien choisie. Supposons qu'on a $x_t \in \mathbb{R}^d$. On va tirer x_{t+1} en s'aidant d'une marche aléatoire de la façon suivante :

1. On génère une valeur y en faisant "un pas de marche aléatoire depuis x_t", autrement dit en tirant y dans la loi de densité $q(x|x_t)$.
2. On calcule
$$\rho = \frac{\pi(y)q(x_t|y)}{\pi(x_t)q(y|x_t)}.$$
3. Si $\rho \geqslant 1$, on pose $x_{t+1} = y$; sinon, $x_{t+1} = y$ avec probabilité ρ et $x_{t+1} = x_t$ avec probabilité $1-\rho$.

La valeur y s'appelle « valeur proposée »; l'étape 3 consiste à décider si on accepte ou non la proposition. En pratique, on peut la réaliser ainsi

— On tire u dans la loi uniforme $U(0,1)$
— Si $u < \rho$ on pose $x_{t+1} = y$ (on accepte y), et sinon $x_{t+1} = x_t$.

6.1.2.1 Cas particulier d'une marche symétrique

si pour tous x et y on a $q(x|y) = q(y|x)$ (la probabilité de faire un pas de y à x est la même que celle de faire un pas de x à y ; c'est le cas de la marche gaussienne donnée en exemple), alors on a simplement

$$\rho = \frac{\pi(y)}{\pi(x_t)}.$$

Dans ce cas, la valeur proposée y est toujours acceptée quand $\pi(y) > \pi(x_t)$, c'est-à-dire quand la marche aléatoire propose un point où la densité est plus grande qu'au point actuel.

6.1.2.2 Les propriétés du résultat

Si t est assez grand, alors x_t est approximativement de loi de densité $\pi(x)$ (ou proportionnelle à $\pi(x)$).

On pourrait donc utiliser cette méthode avec $t = 4000$ (par exemple) pour générer une valeur dans la loi voulue, puis recommencer, etc. C'est très coûteux en temps de calcul ; en fait pour la plupart des applications on peut garder toutes les valeurs au-delà d'une certaine valeur de t. Elles ne sont pas indépendantes mais cela n'est pas très gênant.

L'opération, souvent nécessaire, qui consiste à supprimer les premières valeurs (par exemple les 4000 premières) s'appelle le *burn-in*. Si il est important que les valeurs échantillonnées soient indépendantes, on peut s'en approcher en ne gardant, par exemple, qu'une valeur toutes les 100 itérations. Cette opération s'appelle le *thinning*.

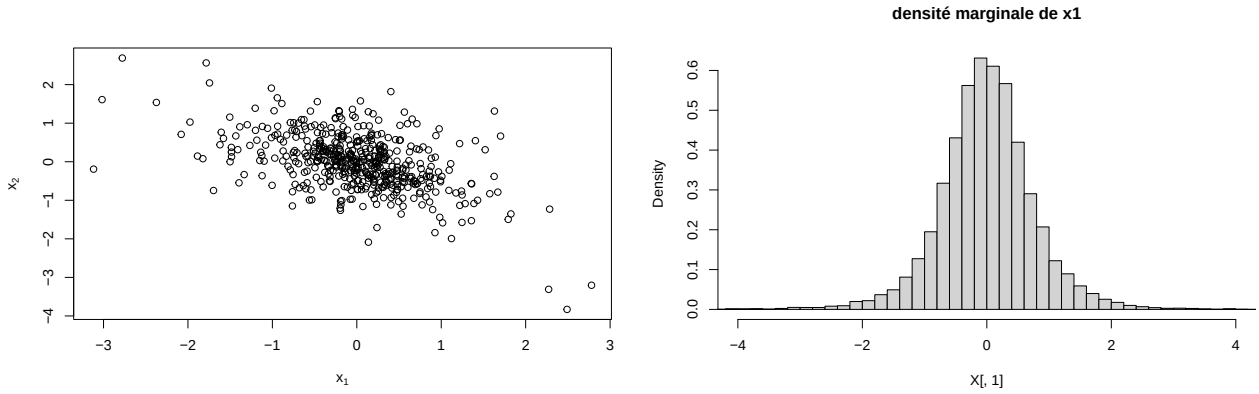
6.1.3 Application

On prend pour $x = (x_1, x_2)$, $\pi(x) = \left(1 + x_1^2 + x_1 x_2 + x_2^2\right)^{-3}$, et on utilise une marche aléatoire gaussienne comme celle présentée plus haut, qui est une marche symétrique : la formule simplifiée ci-dessus peut être utilisée.

L'implémenation en R n'est pas difficile:

```
PI <- function(x) (1 + x[1]**2 + x[1]*x[2] + x[2]**2)^(-3)
MH <- function(B, sd) {
  R <- matrix(0.0, nrow = B, ncol = 2)
  x <- R[1,]
  for(b in 2:B) {
    y <- x + rnorm(2, 0, sd = sd)
    rho <- PI(y) / PI(x)
    u <- runif(1)
    if(u < rho)
      x <- y
    R[b, ] <- x
  }
  return(R);
}
```

Voici un exemple de mise en œuvre :



6.2 Première version en C++

Une version obtenue en « traduisant » l'implémentation en R.

```
// file: Pi.cpp
#include <cmath>
double Pi(double x1, double x2) {
  return pow((1 + x1*x1 + x1*x2 + x2*x2),-3.0);
}
```

```
// file: MHcpp1.cpp
#include <Rcpp.h>
#include "Pi.h"
using namespace Rcpp;
//[[Rcpp::export]]
NumericMatrix MHcpp1(int B, double sd) {
  NumericMatrix R(B, 2);
  double x1 = 0.0, x2 = 0.0;
  for(int b = 1; b < B; b++) {
    double y1 = x1 + R::rnorm(0,sd);
    double y2 = x2 + R::rnorm(0,sd);
    double rho = Pi(y1, y2) / Pi(x1, x2);
    double u = R::unif_rand();
    if(u < rho) {
      x1 = y1;
      x2 = y2;
    }
    R(b,0) = x1;
    R(b,1) = x2;
  }
  return R;
}
```

6.3 Une version améliorée

Il est souhaitable d'éviter de recalculer `Pi(x1, x2)` à chaque tour de boucle, alors que cette valeur est déjà connue.

```cpp
// file: MHcpp2.cpp
#include <Rcpp.h>
#include "Pi.h"
using namespace Rcpp;

//[[Rcpp::export]]
NumericMatrix MHcpp2(int B, double sd, int burn = 0, int thin = 1) {
  NumericMatrix R(B, 2);
  thin = (thin < 1)?1:thin;
  double x1 = 0.0, x2 = 0.0;
  int b = 1;
  double pi_x = Pi(x1, x2);
  for(int k = 0; b < B ; k++) { // !! boucle exotique !!
    double y1 = x1 + R::rnorm(0,sd);
    double y2 = x2 + R::rnorm(0,sd);
    double pi_y = Pi(y1, y2) / Pi(x1, x2);
    double u = R::unif_rand();
    if(u * pi_x < pi_y) {
      x1 = y1;
      x2 = y2;
      pi_x = pi_y;
    }
    if(k > burn && (k % thin) == 0) {
      R(b,0) = x1;
      R(b,1) = x2;
      b++;
    }
    if((k % 1000) == 0) // toutes les mille itérations
      checkUserInterrupt();
  }
  return R;
}
```

Le paramètre `burn` permet de ne pas retenir les premières itérations ; le paramètre `thin` permet de ne retenir qu'une itération sur `thin` (pour réduire la dépendance entre les tirages successifs).

À noter : la fonction `checkUserInterrupt()` qui permet d'interrompre le programme en cas d'appui sur ctrl + C, ou sur le petit panneau STOP de R studio. On n'appelle pas cette fonction à chaque tour de boucle car elle est longue à exécuter !

À noter également : une boucle exotique, puisque la condition d'arrêt n'est pas sur le compteur de boucle `k`, mais sur `b`, qui est régulièrement incrémenté dans la boucle (mais, en général, pas à chaque tour).

Chapitre 7

Un peu plus de C++

Install the package with

```
> devtools::install_github("introRcpp/introRcppMore")
```

Load it with

```
library(introRcppMore)
##
## Attaching package: 'introRcppMore'
## The following object is masked from 'package:introRcppSugar':
##
##     countZeroes
## The following object is masked from 'package:introRcppManipulation':
##
##     countZeroes
```

7.1 Pointeurs

Les pointeurs sont un héritage de C. Il s'agit de « pointer » vers l'adresse d'un objet ou une variable. Un pointeur vers un `int` est un `int *`. Si p est un pointeur, `*p` est l'objet à l'adresse pointée par p.

Pour obtenir le pointeur vers x, on utilise `&x`.

```
// file: pointers.cpp
#include <Rcpp.h>
#include "show.h"
// [[Rcpp::export]]
void pointers() {
  int x = 12;   // entier
  int * p;      // pointeur vers un entier (non initialisé !!)
  p = &x;       // p pointe vers x
  SHOW(x);
  SHOW(p);
  SHOW(*p);
  Rcpp::Rcout << "On ajoute 1 à x\n";
  x += 1;
```

```
  SHOW(x);
  SHOW(p);
  Rcpp::Rcout << "On ajoute 1 à *p\n";
  *p += 1;
  SHOW(x);
  SHOW(p);
}
```

```
> pointers()
x = 12
p = 0x7ffe12a51244
*p = 12
On ajoute 1 à x
x = 13
p = 0x7ffe12a51244
On ajoute 1 à *p
x = 14
p = 0x7ffe12a51244
```

Nous avions déjà rencontré des pointeurs sans le savoir : les tableaux sont des pointeurs ! Plus précisément, si on a déclaré `int a[4]`, `a` pointe vers le premier élément du tableau, `a+1` vers le second, etc.

```
// file: arrays.cpp
#include <Rcpp.h>
#include "show.h"
// [[Rcpp::export]]
void arrays() {
  int a[4] = {10,20,30,40};
  SHOW(a);
  SHOW(a[0]);
  SHOW(*a);
  SHOW(a[2]);
  SHOW(a+2);
  SHOW(*a+2);
  SHOW(*(a+2));
}
```

```
> arrays()
a = 0x7ffe12a51230
a[0] = 10
*a = 10
a[2] = 30
a+2 = 0x7ffe12a51238
*a+2 = 12
*(a+2) = 30
```

Notez la différence entre `a` et `a+2` : la taille des `int` est prise en compte dans le calcul.

En passant à une fonction un pointeur vers une variable, d'une part on permet la modification de cette variable, d'autre part on évite la copie de cette variable (intéressant quand on passe des objets de grande taille).

```cpp
// file: swap.cpp
#include <Rcpp.h>
#include "show.h"
// les arguments sont des pointeurs vers des entiers
inline void swap(int * x, int * y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

// [[Rcpp::export]]
void demonstrateSwap(int a, int b) {
  if(a > b) swap(&a,&b); // &a = pointeur vers a ...
  SHOW(a);
  SHOW(b);
}
```

Attention, la fonction `swap` ne peut pas être exportée vers R : les objets de R ne peuvent être transformés en `int *`, en pointeurs vers des entiers. Elle est destinée à n'être utilisée que dans notre code C++.

Le mot clef `inline` ci-dessus indique au compilateur qu'on désire que la fonction soit insérée dans le code aux endroit où elle est utilisée. Cela économise un appel de fonction…

```
> demonstrateSwap(12, 14)
a = 12
b = 14
```

```
> demonstrateSwap(14, 12)
a = 12
b = 14
```

7.2 Références

Le mécanisme derrière les références est similaire à celui des pointeurs ; la syntaxe est plus simple et permet d'éviter de promener des étoiles dans tout le code. Une référence à un entier est de type `int &`, et peut être manipulée directement comme un `int`. On l'initialise à l'aide d'une autre variable de type `int`.

```cpp
// file: references.cpp
#include <Rcpp.h>
#include "show.h"
// [[Rcpp::export]]
void references() {
  int x = 12;   // entier
  int & y = x;  // référence à x
  SHOW(x);
  SHOW(y);
  SHOW(&x);
  SHOW(&y);
  Rcpp::Rcout << "On ajoute 1 à x\n";
  x += 1;
  SHOW(x);
  SHOW(y);
```

```
  Rcpp::Rcout << "On ajoute 1 à y\n";
  y += 1;
  SHOW(x);
  SHOW(y);
}
```

```
> references()
x = 12
y = 12
&x = 0x7ffe12a51244
&y = 0x7ffe12a51244
On ajoute 1 à x
x = 13
y = 13
On ajoute 1 à y
x = 14
y = 14
```

C'est surtout utile pour spécifier les arguments d'une fonction. Voici ce que devient notre fonction `swap`:

```
// file: swap2.cpp
#include <Rcpp.h>
#include "show.h"

// les arguments sont des références à des entiers
inline void swap2(int & x, int & y) {
  int tmp = x;
  x = y;
  y = tmp;
}

// [[Rcpp::export]]
void demonstrateSwap2(int a, int b) {
  if(a > b) swap2(a,b); // on passe directement x et y
  SHOW(a);
  SHOW(b);
}
```

Cette fonction `swap2` ne peut pas plus être exportée que la fonction `swap` définie plus haut.

```
> demonstrateSwap2(12, 14)
a = 12
b = 14
```

```
> demonstrateSwap2(14, 12)
a = 12
b = 14
```

7.2.1 Autres exemples

Un appel `divideByTwo(x)` est équivalent à `x /= 2`. L'utilisation de cette fonction ne simplifie pas vraiment le code, mais ce mécanisme pourrait être utile pour des opérations plus complexes:

```
// file: divideByTwo.cpp
void divideByTwo(int & x) {
  x /= 2;
}

// [[Rcpp::export]]
int intLog2(int a) {
  int k = 0;
  while(a != 0) {
    divideByTwo(a);
    k++;
  }
  return k-1;
}
```

```
> intLog2(17)
[1] 4
```

Une autre utilité est de renvoyer plusieurs valeurs. Bien sûr quand on utilise Rcpp on peut manipuler des vecteurs ou des listes, mais utiliser des variables passées par référence pour récupérer les résultats d'un calcul est souvent très commode.

```
// file: quadratic.cpp
#include <Rcpp.h>
#include <cmath>
#include "show.h"

bool quadratic(double a, double b, double c, double & x1, double & x2) {
  double delta = b*b - 4*a*c;
  if(delta < 0)
    return false;
  double sqrt_delta = std::sqrt(delta);
  x1 = (-b - sqrt_delta)/(2*a);
  x2 = (-b + sqrt_delta)/(2*a);
  return true;
}

//[[Rcpp::export]]
void demoQuadratic(double a, double b, double c) {
  double x1, x2;
  bool solvable = quadratic(a, b, c, x1, x2);
  if(solvable) {
    SHOW(x1);
    SHOW(x2);
  } else {
    Rcpp::Rcout << "Pas de solution\n";
  }
}
```

```
> demoQuadratic(1,-5,6)
x1 = 2
x2 = 3
```

```
> demoQuadratic(1,0,1)
Pas de solution
```

7.3 Les objets de Rcpp sont passés par référence

La fonction suivante le démontre : les vecteurs de Rcpp sont passés par référence (ou sont des pointeurs, comme vous préférez). Ceci permet un grand gain de temps, en évitant la copie des données, mais a aussi des effets potentiellement indésirables.

```
// file: twice.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector twice(Rcpp::NumericVector x) {
  int n = x.size();
  for(int i = 0; i < n; i++) {
    x[i] *= 2;
  }
  return x;
}
```

```
> x <- seq(0,1,0.1);
> x
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
> twice(x)
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x # oups
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

De plus, quand on copie un objet en R, comme ci-dessous avec `y <- x`, la recopie des données n'a pas lieu dans un premier temps. Au début, `y` pointe vers le même objet que `x` ; la recopie n'aura lieu que si on modifie `y`. Quand on utilise `Rcpp` ce mécanisme n'est plus actif:

```
> x <- seq(0,1,0.1);
> y <- x
> twice(y)
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> y
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

La prudence est de mise! Notez qu'on a aussi le comportement suivant:

```
> x <- 1:5;
> x
[1] 1 2 3 4 5
```

```
> twice(x)
[1]  2  4  6  8 10
```

```
> x
[1] 1 2 3 4 5
```

Utiliser `typeof(x)` pour résoudre cette énigme.

À supposer que ce comportement soit indésirable, comment y remédier ?

```
// file: twice2.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector twice2(Rcpp::NumericVector x) {
  int n = x.size();
  // contrairement à ce qu'on pense, ceci ne copie pas x
  Rcpp::NumericVector y = x;
  for(int i = 0; i < n; i++) y[i] *= 2;
  return y;
}

// [[Rcpp::export]]
Rcpp::NumericVector twice3(Rcpp::NumericVector x) {
  int n = x.size();
  // il faut utiliser clone
  Rcpp::NumericVector y = Rcpp::clone(x);
  for(int i = 0; i < n; i++) y[i] *= 2;
  return y;
}
```

```
> x <- seq(0,1,0.1); invisible(twice2(x)); x
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x <- seq(0,1,0.1); invisible(twice3(x)); x
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

On peut également modifier « en place » les éléments d'une liste:

```
// file: incrementAlpha.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void incrementAlpha(List x) {
  if( x.containsElementNamed("alpha") ) {
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("elt alpha n'est pas un 'NumericVector'");
```

```
    NumericVector Alpha(R);
    Alpha = Alpha+1;  // sugar
  } else
    stop("Pas d'elt alpha");
}
```

Exemple:

```
> x <- list( alpha = c(0.1,7), beta = 1:4)
> x
$alpha
[1] 0.1 7.0

$beta
[1] 1 2 3 4
```

```
> incrementAlpha(x)
> x
$alpha
[1] 1.1 8.0

$beta
[1] 1 2 3 4
```

7.4 Surcharge

En C++ deux fonctions peuvent avoir le même nom, à condition que le type des arguments soit différent, ce qui permet au compilateur de les différentier. À noter : le type retourné par la fonction n'est pas pris en compte dans ce mécanisme.

Ceci s'appelle la *surcharge* des fonctions. On a également une surcharge des opérateurs (comme +, <<, etc).

Attention ! On ne peut pas exporter des fonctions surchargées avec Rcpp. Il faut donc maintenir ce mécanisme dans la partie strictement C++ de votre code.

Voici un exemple de fonction surchargée.

```
// file: d2.cpp
#include <Rcpp.h>
using namespace Rcpp;

double d2(double a, double b) {
  return (a*a + b*b);
}

int d2(int a, int b) {
  return (a*a + b*b);
}

//[[Rcpp::export]]
void exampleD2() {
  double x = 1.0;
```

```
  double y = 2.4;
  double z = d2(x, y);
  Rcout << "d2(x,y) = " << z << std::endl;
  int u = 1;
  int v = 4;
  int w = d2(u, v);
  Rcout << "d2(u,v) = " << w << std::endl;
}
```

```
> exampleD2()
d2(x,y) = 6.76
d2(u,v) = 17
```

On peut également avoir des fonctions qui ont le même nom, et un nombre différent d'arguments.

7.5 Templates

Les templates facilitent la réutilisation de code avec des types différents. Prendre le temps d'utiliser des méthodes propres pour éviter les copier-coller dans le code est de toute première importance quand il s'agit de « maintenir » le code.

7.5.1 Fonctions polymorphes

L'exemple ci-dessous définit un template `d2` ; lors de la compilation de la fonctions `addition_int` et `addition_double` ce template sera instancié avec `TYPE = int` puis avec `TYPE = double`.

```
// file: d2.h
#ifndef _D2_
#define _D2_
template<typename TYPE>
TYPE d2(TYPE a, TYPE b) {
  return a*a + b*b;
}
#endif
```

```
// file: exampleTemplate.cpp
#include <Rcpp.h>
#include "d2.h"
//[[Rcpp::export]]
void exampleTemplate() {
  double x = 1.0;
  double y = 2.4;
  double z = d2(x, y);
  Rcpp::Rcout << "d2(x,y) = " << z << std::endl;
  int u = 1;
  int v = 4;
  int w = d2(u, v);
  Rcpp::Rcout << "d2(u,v) = " << w << std::endl;
}
```

Exemple:

```
> exampleTemplate()
d2(x,y) = 6.76
d2(u,v) = 17
```

Si le compilateur peinait à trouver par quoi il faut remplacer `TYPE`, il serait possible de préciser en tapant par exemple `d2<float>`. Ce template peut fonctionner dès que les opérateurs `+` et `*` sont définis.

```
// file: exempleTemplate2.cpp
#include <Rcpp.h>
#include "d2.h"
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector exempleTemplate2(NumericVector x, NumericVector y) {
  return d2(x,y);
}
```

```
> exempleTemplate2( c(1,2), c(3,1) )
[1] 10  5
```

On peut donner un autre exemple en faisant un template pour la fonction `swap` que nous avions écrite il y a quelques chapitres.

```
// file: demonstrateSwap3.cpp
#include <Rcpp.h>
#include "swap.h"
#include "show.h"
// [[Rcpp::export]]
void demonstrateSwap3(int a, int b) {
  if(a > b) swap(a,b); // on passe directement x et y
  SHOW(a);
  SHOW(b);
}
```

```
> demonstrateSwap3(12, 14)
a = 12
b = 14
```

```
> demonstrateSwap3(14, 12)
a = 12
b = 14
```

Note : n'utilisez pas ce template dans votre code, C++ a déjà un template `std::swap`, qui est mieux fait que celui-ci ! (Il évite la copie). Voir https://en.cppreference.com/w/cpp/algorithm/swap

7.5.2 Laisser le compilateur s'occuper des types

Une des utilités des templates est de permettre de ne pas trop se soucier de types « compliqués » comme ceux des fonctions. Ci-dessous on définit une fonction `numDerivation` qui donne une approximation de la dérivée de son premier argument.

```
// file: derivSquare.cpp
#include <Rcpp.h>
#include "show.h"
#include "numDerivation.h"

double square(double a) {
  return a*a;
}

//[[Rcpp::export]]
void derivSquare(double x) {
  double fx = square(x);
  double dfx = numDerivation(square, x);
  SHOW(fx);
  SHOW(dfx);
}
```

```
> derivSquare(3)
fx = 9
dfx = 6
```

On voit que `numDerivation` prend deux arguments, une fonction `f` et un point `a`, et donne une approximation de $f'(a)$ en calculant

$$\frac{f(a+\varepsilon)-f(a-\varepsilon)}{2\varepsilon}.$$

Quel est le type d'une fonction ?! Nous laissons au compilateur le soin de remplacer `FTYPE` par la bonne valeur. Dans notre exemple, lors d'instanciation du template, `TYPE` est remplacé par `double` et `FTYPE` par `double (*)(double)` (pointeur vers une fonction qui renvoie un `double` et dont l'argument est un `double`). Un code équivalent serait

```
// file: derivSquare.alt
#include <Rcpp.h>
#include "show.h"
#define epsilon 0.001
double numDerivation(double (*f)(double), double a) {
  return (f(a+epsilon) - f(a-epsilon))/(2*epsilon);
}

double square(double a) {
  return a*a;
}

//[[Rcpp::export]]
void derivSquare(double x) {
  double fx = square(x);
  double dfx = numDerivation(square, x);
  SHOW(fx);
  SHOW(dfx);
}
```

7.6 C++11

Le standard C++11 est maintenant accepté dans les packages R. Pour permettre la compilation avec les extensions offertes par C++11, il faut inclure dans `src/` un fichier `Makevars` contenant

```
PKG_CXXFLAGS = -std=c++11
```

7.6.1 Les boucles sur un vecteur et `auto`

Parmi les extensions offertes, les plus séduisantes sont sans doute : le mot-clef `auto`, qui laisse le compilateur deviner le type, quand le contexte le permet, de façon similaire à ce qui est fait dans les templates, et les boucles `for` qui parcourent un vecteur… comme en R.

```
// file: countZeroes.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
int countZeroes(Rcpp::IntegerVector x) {
  int re = 0;
  for(auto a : x) {
    if(a == 0) ++re;
  }
  return re;
}
```

```
> set.seed(1); a <- sample(0:99, 1e6, TRUE)
> countZeroes(a);
[1] 10017
```

Si on veut pouvoir modifier la valeur des éléments du vecteur, il suffit d'utiliser une référence :

```
// file: addOne.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void addOne(Rcpp::IntegerVector x) {
  for(auto & a : x) {
    a++;
  }
}
```

```
> a <- 1:10
> addOne(a)
> a
 [1]  2  3  4  5  6  7  8  9 10 11
```

NOTE En utilisant une référence, on évite la copie. Même si on ne désire pas modifier les éléments du vecteur, cela peut modifier énormément les performances de la boucle en question !

TODO ajouter exemple !!!

7.6.2 Nombres spéciaux

C++11 propose aussi des fonctions pour tester si des nombres flottants sont finis ou non, sont `NaN`. Il est aussi plus facile de renvoyer un `NaN` ou une valeur infinie (c'était possible avec `std::numeric_limits`).

```
// file: tests.cpp
#include <Rcpp.h>
#include <cmath>       // !!! CE HEADER EST NÉCESSAIRE
//[[Rcpp::export]]
void tests(double x) {
  if(std::isnan(x)) Rcpp::Rcout << "NaN\n";
  if(std::isinf(x)) Rcpp::Rcout << "infini\n";
  if(std::isfinite(x)) Rcpp::Rcout << "fini\n";
}
```

```
// file: specials.cpp
#include <Rcpp.h>
#include <cmath>
//[[Rcpp::export]]
Rcpp::NumericVector specials() {
  Rcpp::NumericVector x(2);
  x[0] = NAN;
  x[1] = INFINITY;
  return x;
}
```

```
> tests( NA )
NaN
```

```
> tests( -Inf )
infini
```

```
> tests( pi )
fini
```

```
> specials();
[1] NaN Inf
```

Chapitre 8

Créer des objets

All the examples are in the R package… Install it with

```
> devtools::install_github("introRcpp/introRcppObjets")
```

Load it with

```
library(introRcppObjets)
```

8.1 Une classe bavarde : le dodo

8.1.1 Déclaration de la classe

Le fichier `dodo.h` est dans `inst/include/introRcppObjets/` (voir aussi le `src/Makevars`)

```
// file: dodo.h
#include <Rcpp.h>
#include <vector>
#include "debug.h"

#ifndef _dodo_
#define _dodo_

class dodo {
public:
  int a;
  int b;
  std::vector<int> vec;

  // constructeurs
  dodo() : a(1), b(2), vec(0) {
    Rcpp::Rcout << "Construction d'un dodo avec dodo()\n";
  }
  dodo(int a0, int b0) : a(a0), b(b0), vec(0) {
    Rcpp::Rcout << "Construction d'un dodo avec dodo(a, b)\n";
  }
```

```
  dodo(int a0, int b0, bool z) : a(a0), b(b0), vec(0) {
    Rcpp::Rcout << "Construction d'un dodo avec dodo(a, b, z)\n";
    if(z) {
      for(int i = a; i < b; i++) {
        vec.push_back(i);
      }
    }
  }

  // constructeur par copie
  dodo(const dodo & x) : a(x.a), b(x.b), vec(x.vec) {
    Rcpp::Rcout << "Copie d'un dodo (a = " << a << ", b = " << b << ")\n";
  }

  // operateur "move"
  dodo(dodo && x) : a(std::move(x.a)), b(std::move(x.b)), vec(std::move(x.vec)) {
    Rcpp::Rcout << "Move d'un dodo (a = " << a << ", b = " << b << ")\n";
  }

  void print() {
    Rcpp::Rcout << "Dodo a = " << a << ", b = " << b << ", ";
    SHOWVEC(vec);
  }

  // destructeur
  ~dodo() {
    Rcpp::Rcout << "Destruction d'un dodo a = " << a ;
    Rcpp::Rcout << ", b = " << b << "\n";
  }
};

#endif
```

8.1.2 Construction et destruction

```
// file: dodo_construit_detruit.cpp
#include <Rcpp.h>
#include "dodo.h"
#include "debug.h"
using namespace Rcpp;

// [[Rcpp::export]]
void dodo_construit_detruit() {
  Rcout << "Création des dodos x, y, z\n";
  dodo x;
  x.print();

  dodo y(1, 5);
  y.print();

  dodo z(2, 7, true);
```

```
  z.print();
  Rcout << "Fin de la fonction, et destruction des dodos\n";
}
```

Voyons ce que ça donne:

```
dodo_construit_detruit()
## Création des dodos x, y, z
## Construction d'un dodo avec dodo()
## Dodo a = 1, b = 2, vec = ()
## Construction d'un dodo avec dodo(a, b)
## Dodo a = 1, b = 5, vec = ()
## Construction d'un dodo avec dodo(a, b, z)
## Dodo a = 2, b = 7, vec = (2 3 4 5 6 )
## Fin de la fonction, et destruction des dodos
## Destruction d'un dodo a = 2, b = 7
## Destruction d'un dodo a = 1, b = 5
## Destruction d'un dodo a = 1, b = 2
```

8.1.3 Copie et move...

blabla

8.1.4 Création de vecteur

blabla

8.1.5 Passer un objet à R !

Il faut éviter la destruction de l'objet quand il sort du scope.

```
// file: RcppDodo.cpp
#include <Rcpp.h>
#include "dodo.h"
using namespace Rcpp;

// Ceci renvoie bien un pointeur mais le dodo est détruit
//[[Rcpp::export]]
XPtr<dodo> mkDodo1(int a, int b) {
  Rcout << "Création du dodo x : ";
  dodo x(a, b);
  // création du pointeur
  XPtr<dodo> p(&x, false);
  Rcout << "On renvoie un pointeur vers x ; mais x sort du scope !\n";
  return p;
}

// [[Rcpp::export]]
XPtr<dodo> mkDodo2(int a, int b) {
  // new alloue de la mémoire et appelle le constructeur
```

```
  dodo * p_Dodo = new dodo(a, b);
  // Création du pointeur XPtr
  XPtr<dodo> xp_Dodo(p_Dodo);
  return xp_Dodo;
}

// [[Rcpp::export]]
void printDodo1(XPtr<dodo> xpDodo) {
  // création d'un pointeur vers le dodo
  dodo * p_Dodo = xpDodo.get();
  p_Dodo->print();
}

// [[Rcpp::export]]
void printDodo2(XPtr<dodo> xpDodo) {
  // création d'une référence vers le dodo
  dodo & MonDodo( *xpDodo.get() );
  MonDodo.print();
}
```

La fonction `mkDodo1` renvoie bien un pointeur mais l'objet est détruit. Utiliser le pointeur créerait une catastrophe !

```
a <- mkDodo1(12, 14)
## Création du dodo x : Construction d'un dodo avec dodo(a, b)
## On renvoie un pointeur vers x ; mais x sort du scope !
## Destruction d'un dodo a = 12, b = 14
```

```
a
## <pointer: 0x7ffe12a50f60>
```

```
# surtout ne pas exécuter cette ligne
# printDodo1(a)
```

La solution est :

```
a <- mkDodo2(12, 14)
## Construction d'un dodo avec dodo(a, b)
```

```
a
## <pointer: 0x5581d8270fb0>
```

```
printDodo1(a)
## Dodo a = 12, b = 14, vec = ()
```

```
printDodo2(a)
## Dodo a = 12, b = 14, vec = ()
```

La classe `XPtr` a été construite de façon à ce que quand un pointeur externe est détruit, le destructeur de l'objet pointé est appelé. En pratique, ici, quand on efface `a`, le destructeur du dodo est appelé – pas forcément immédiatement, il faut appeler le garbage collecteur, qui sinon passe de lui-même régulièrement.

```
rm(a)
gc()   # appel du garbage collector
## Destruction d'un dodo a = 12, b = 14
##           used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  738617 39.5    1196660   64  1196660 64.0
## Vcells 1354719 10.4    8388608   64  2955626 22.6
```