

Introduction à C++ et Rcpp

Hervé Perdry

2020-03-10

Table des matières

1	À propos de ce document	5
2	Objets de R	7
2.1	Types	7
2.2	Attributs	8
2.3	Mieux examiner les objets	9
2.4	Pour les apprentis sorciers	10
3	Bases de C++	13
3.1	Nécessité des types	13
3.2	Les entiers	14
3.3	Les flottants	15
3.4	Constantes numériques	17
3.5	Opérateurs arithmétiques	17
3.6	Booléens	18
3.7	Tableaux de taille fixe	18
3.8	Contrôle du flux d'exécution : les boucles	19
3.9	Contrôle du flux d'exécution : les alternatives	21
3.10	Premiers objets Rcpp : les vecteurs	22
4	Manipuler les objets de R en C++	27
4.1	Objets génériques : SEXP	27
4.2	Vecteurs	27
4.3	Vecteurs nommés	30
4.4	Sucre syntaxique	32
4.5	Facteurs	35
4.6	Listes et Data frames	36
5	Exemple : Metropolis-Hastings	39
5.1	L'algorithme	39
5.2	Une version R	41
5.3	Première version en C++	41
5.4	Comparaisons	42
5.5	Une version améliorée	42

Chapitre 1

À propos de ce document

Ce document est issu d'un cours d'école doctorale donné en février/mars 2019. Un effort a été fait pour le rendre lisible sans les commentaires fait en cours, mais il y a certainement toujours des lacunes.

Les exemples de code proposés utilisent souvent le standard C++11. Pour pouvoir compiler avec ce standard, utilisez `Sys.setenv("PKG_CXXFLAGS" = "-std=c++11")`.

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 2

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.

2.1 Types

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

```
> a <- c("bonjour", "au revoir")
> b <- c(TRUE, FALSE, TRUE, TRUE)
> c <- c(1.234, 12.34, 123.4, 1234)
> d <- matrix( rpois(12, 2), 4, 3)
> e <- list(un = a, deux = b)
> f <- data.frame(b, c)
> g <- factor( c("F", "M", "F", "F") )
> typeof(a) # chaînes de caractères
[1] "character"
```

```
> typeof(b) # valeurs vrai / faux
[1] "logical"
```

```
> typeof(c) # nombres "à virgule"
[1] "double"
```

```
> typeof(d) # nombres entiers
[1] "integer"
```

```
> typeof(e) # liste
[1] "list"
```

```
> typeof(f) # data frame = liste
[1] "list"
```

```
> typeof(g) # facteur = entiers (!)
[1] "integer"
```

Il y a deux types de variables numériques : `double` (nombres « à virgule », en français 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(c > 100))
[1] "integer"

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

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

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

```
> str(a) # chaines de caractères
chr [1:2] "bonjour" "au revoir"

> str(b) # vrai / faux
logi [1:4] TRUE FALSE TRUE TRUE

> str(c) # "à virgule"
num [1:4] 1.23 12.34 123.4 1234

> str(d) # entiers (matrice 4x3)
int [1:4, 1:3] 2 1 4 2 4 2 2 1 4 1 ...

> str(e) # liste
List of 2
 $ un  : chr [1:2] "bonjour" "au revoir"
 $ deux: logi [1:4] TRUE FALSE TRUE TRUE

> str(f) # data frame
'data.frame':  4 obs. of  2 variables:
 $ b: logi  TRUE FALSE TRUE TRUE
 $ c: num   1.23 12.34 123.4 1234

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

2.2 Attributs

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

```
> names(c) <- c("elt1", "elt2", "elt3", "elt4")
> c
      elt1      elt2      elt3      elt4
 1.234   12.340   123.400 1234.000

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


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

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

Les data frames et les facteurs ont également des attributs :

```
> attributes(f)
$names
[1] "b" "c"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4
```

```
> attributes(g)
$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(d, "dim")
[1] 4 3

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

2.3 Mieux examiner les objets

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".
```

2.3.1 Cas de la matrice

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

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

2.3.2 Cas du data frame :

```
> dput(f)
structure(list(b = c(TRUE, FALSE, TRUE, TRUE), c = c(1.234, 12.34,
123.4, 1234)), class = "data.frame", row.names = c(NA, -4L))
```

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 »).

2.3.3 Cas du facteur :

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

```
> levels(g)
[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 :

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

2.4 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 ))
@55b12384a4b0 13 INTSXP g0c0 [NAM(7)] 1 : 10 (compact)
```

```
> .Internal(inspect( c(0.1, 0.2) ))
@55b124f840b8 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2
```

```
> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@55b125dbfc88 14 REALSXP g0c2 [NAM(7)] (len=2, tl=0) 0.1,0.2
```

```
> names(a) <- c("A", "B")
> .Internal(inspect( a ))
@55b1277d5938 14 REALSXP g0c2 [NAM(1),ATT] (len=2, tl=0) 0.1,0.2
ATTRIB:
  @55b124f47348 02 LISTSXP g0c0 []
    TAG: @55b121b5df70 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x6000] "names" (has value)
    @55b1277d58f8 16 STRSXP g0c2 [NAM(7)] (len=2, tl=0)
      @55b123bc0d28 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "A"
      @55b1246a3378 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "B"
```

```

> .Internal(inspect( h ))
@55b128a9d6d8 13 INTSXP g0c2 [OBJ,NAM(7),ATT] (len=4, tl=0) 2,1,1,2
ATTRIB:
  @55b1248c26f8 02 LISTSXP g0c0 []
    TAG: @55b121b5e0c0 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x6000] "levels" (has value)
    @55b128a9d5d8 16 STRSXP g0c2 [NAM(7)] (len=2, tl=0)
      @55b123da0178 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "L1"
      @55b123d9ed20 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "L2"
    TAG: @55b121b5e440 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x6000] "class" (has value)
    @55b1297371f8 16 STRSXP g0c1 [NAM(7)] (len=1, tl=0)
      @55b121c05bb0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "factor"

```

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...

Chapitre 3

Bases de C++

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

```
#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 bibliothèques. La bibliothèque `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 bibliothèque `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.

3.1 Nécessité des types

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 chaînes 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 être déclarées, et ne pourrions 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.

3.2 Les entiers

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++).

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void int_types() {
  char a;
  short b;
  int c;
  long int d;
  Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcout << "sizeof(c) = " << sizeof(c) << "\n";
  Rcout << "sizeof(d) = " << sizeof(d) << "\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 parcimonie (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
```

Notez le `using namespace Rcpp` qui permet de taper simplement `Rcout` au lieu de `Rcpp::Rcout`.

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.

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`.

3.2.0.1 Notre première fonction « non void »

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

```
[[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.

```
[[Rcpp::export]]
int cast_to_int(int x) {
    return x;
}
```

3.2.1 Initialisation des variables

Il est nécessaire d'initialiser les variables.

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

Testons :

```
> 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;`

3.3 Les flottants

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

```
#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";
}
```

3.3.1 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 $a2^b$, où 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"
```

3.3.2 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*.

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

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

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

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



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

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

```
#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
```

3.4 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 faut taper 0. ou 0.0, etc. La plupart du temps le compilateur corrige ces petites erreurs.

3.5 Opérateurs arithmétiques

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

```
#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'incrémentement ++ et de décrémentation --.

```
#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

```

3.6 Booléens

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

```

// [[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.

```

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

```

3.7 Tableaux de taille fixe

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

```

#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 débogage « 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).

3.8 Contrôle du flux d'exécution : les boucles

3.8.1 Boucles for

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

```

#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 :

```

#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.

```
#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.

3.8.2 continue et break

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

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

```
> trois(12)
01.2.34.5.67.8.910.11.
```

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

```
#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)
AAAAAAAAAAAAAAAA
```

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

3.8.3 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.

```
#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.

```

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

```

```

> racine_carree(2)
[1] 1.414216

```

```

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

```

Cherchez sur le site cppreference.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.

3.9 Contrôle du flux d'exécution : les alternatives

3.9.1 `if` et `if else`

Cela fonctionne tout à fait comme en R ; la x

```

// [[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

```

3.9.2 switch

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

```

#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";
    }
}

```

3.10 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.

3.10.1 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.**

```

#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;
}
```

3.10.2 Exemple : compter les zéros

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros(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++) {
    re += (x[i] == 0)?1:0;
  }
  return re;
}
```

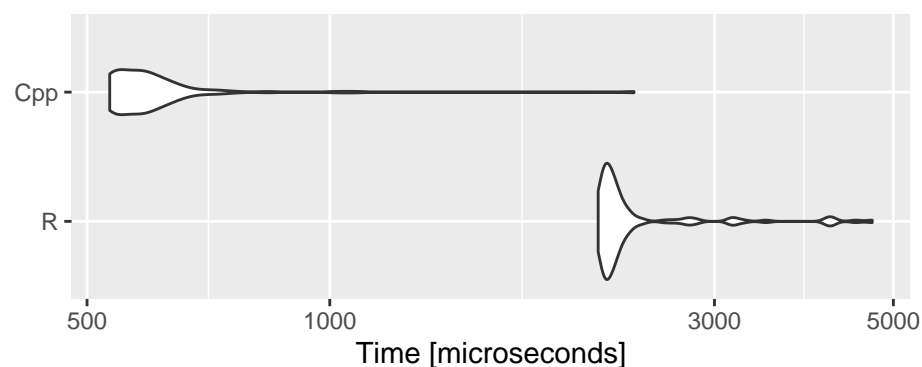
On a utilisé ici l'opérateur ternaire `test?a:b`, qui vaut `a` si `test` est true, `b` sinon. Comment les performances de cette fonction se comparent-elles avec le code R `sum(a == 0)` ?

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

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

```
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp = count_zeros(a))
> mbm
Unit: microseconds
expr      min       lq      mean    median      uq     max neval
R    2152.210 2195.0575 2441.0652 2238.436 2332.963 4707.585   100
Cpp    533.476  545.7295  616.9864  582.958  617.092 2384.363   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.

La solution suivante a été proposée en cours.

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros2(Rcpp::IntegerVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    (x[i] == 0)?re++:0;
  }
  return re;
}
```

Mais en fait, comme en R, ajouter un booléen à un int produit une conversion `false : 0` et `true : 1`.

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros3(Rcpp::IntegerVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += (x[i] == 0);
  }
  return re;
}
```

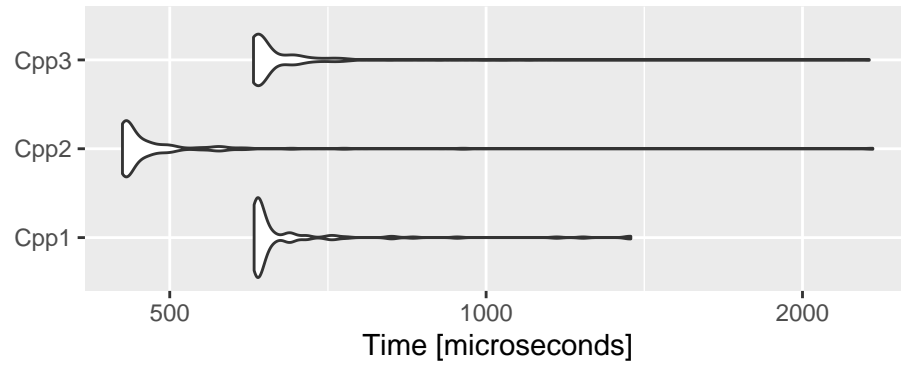
Une comparaison s'impose.

```
> count_zeros2(a)
[1] 10017
```

```
> count_zeros3(a)
[1] 10017
```

```
> mbm <- microbenchmark::microbenchmark( Cpp1 = count_zeros(a),
+     Cpp2 = count_zeros2(a),
+     Cpp3 = count_zeros3(a) )
> mbm
Unit: microseconds
expr      min       lq      mean    median       uq      max neval
Cpp1 601.507 604.2005 656.4529 610.4535 630.5535 1373.105   100
Cpp2 450.797 452.9220 500.0337 457.9355 480.3500 2333.357   100
Cpp3 600.700 604.5395 658.4637 608.8460 650.2260 2314.989   100
```

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

Selon les options de compilation ou les architectures, il peut y avoir ou non une différence entre ces solutions.

Chapitre 4

Manipuler les objets de R en C++

4.1 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.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
std::string le_type(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";
    }
}
```

Ça n'est généralement pas une bonne idée d'écrire des fonctions génériques comme celle-ci. Utiliser les types définis par Rcpp est généralement plus facile et plus sûr. Cependant nous verrons plus loin qu'à l'intérieur des fonctions Rcpp ils peuvent être utiles, par exemple dans le cas où une fonction renvoie soit un NILSXP, soit un objet d'un autre type.

4.2 Vecteurs

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).

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector zeros(int n) {
  IntegerVector R(n);
  return R;
}
//[[Rcpp::export]]
IntegerVector whatever(int n, int a) {
  IntegerVector R(n, a);
  return R;
}
//[[Rcpp::export]]
IntegerVector uninitialized(int n) {
  IntegerVector R = no_init(n);
  return R;
}
//[[Rcpp::export]]
IntegerVector nombres_fetiches() {
  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] 702500360      21937 702500304      21937 664899432
```

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

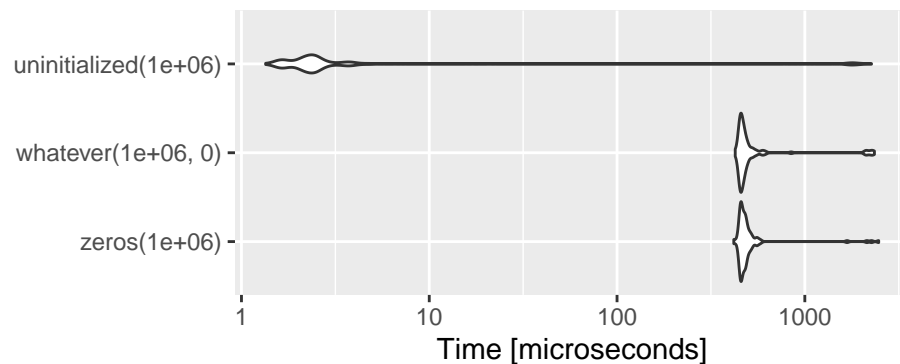
Comparons les performances des trois premières fonctions (comme à chaque fois, les résultats peuvent varier d'une architecture à l'autre).

```
> mbm <- microbenchmark::microbenchmark(zeros(1e6), whatever(1e6, 0), uninitialized(1e6))
> mbm
```

Unit: microseconds

	expr	min	lq	mean	median	uq	max	neval
	<code>zeros(1e+06)</code>	418.446	456.660	543.8650	472.4280	491.5995	2474.626	100
	<code>whatever(1e+06, 0)</code>	425.927	453.673	601.0897	467.7760	494.8815	2347.276	100
	<code>uninitialized(1e+06)</code>	1.345	2.045	124.5295	2.3615	2.7635	2264.899	100

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



4.2.1 Gestions des valeurs manquantes

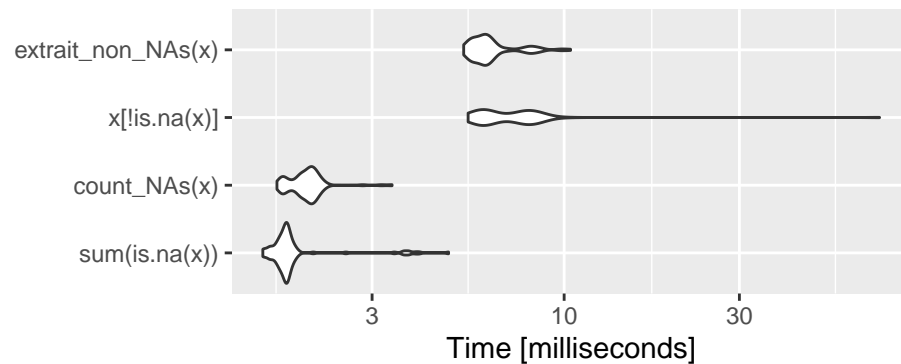
Cette fonction utilise `IntegerVector::is_na` qui est la bonne manière de tester si un membre d'un vecteur entier est NA.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int count_NAs(NumericVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += NumericVector::is_na(x[i]);
  }
  return(re);
}
// [[Rcpp::export]]
NumericVector extrait_non_NAs(NumericVector x) {
  int nb_nas = count_NAs(x);
  int n = x.size();
  NumericVector R(n - nb_nas);
  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)), count_NAs(x), x[!is.na(x)], extrait_non_NAs(x) )
> mbm
Unit: milliseconds
      expr      min       lq      mean    median      uq      max  neval
sum(is.na(x)) 1.512640 1.686837 1.905988 1.745065 1.789692 4.846301   100
count_NAs(x)   1.652371 1.878562 1.997777 2.007327 2.076477 3.400882   100
x[!is.na(x)]   5.477273 6.009724 7.689034 6.805464 8.061266 72.229495   100
extrait_non_NAs(x) 5.327972 5.725945 6.415352 6.105628 6.439961 10.412195   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.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector creer1() {
  NumericVector x = NumericVector::create(Named("un") = 10, Named("deux") = 20);
  return x;
}
```

Application :

```
> a <- creer1()
> a
  un deux
  10  20
```

Une syntaxe plus dense est possible :

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector creer2() {
  NumericVector x = NumericVector::create(_["un"] = 10, _["deux"] = 20);
  return x;
}
```

Cela produit le même résultat.

```
> creer2()
  un deux
  10  20
```

4.3.2 Obtenir les noms d'un vecteur

Et voici comment obtenir les noms d'un vecteur.

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

Utilisons cette fonction:

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

> b <- seq(0,1,length=6)
> noms(b)
Error in noms(b): 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`. La solution est d'attraper le résultat de `x.names()` dans un `SEXP`, et de tester son type avec `TYPEOF`.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector noms2(NumericVector x) {
  SEXP R = x.names();
  if( TYPEOF(R) == STRSXP )
    return R;
  else
    return CharacterVector(0);
}
```

```
> noms2(a)
[1] "un" "deux"
```

```
> noms2(b)
character(0)
```

4.3.3 Accéder aux éléments par leurs noms

On utilise toujours la syntaxe `x[]` :

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
double get_un(NumericVector x) {
  if(x.containsElementNamed("un"))
    return x["un"];
  else
    stop("Pas d'élément 'un'");
}
```

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

```
> get_un(a)
[1] 10

> get_un(b)
Error in get_un(b): Pas d'élément 'un'
```

4.3.4 Énumérer 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).

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void enumere(NumericVector x) {
  SEXP r0 = x.names();
  if(TYPEOF(r0) != STRSEXP) {
    Rcout << "Pas de noms\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";
  }
}

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

4.4 Sucre syntaxique

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 intervenir trois boucles ; c'est un peu fastidieux. Le sucre syntaxique ajouté par les fonctions dites *Rcpp sugar* permet d'éviter de les écrire.

```
#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

```

4.4.1 Efficacité

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

```

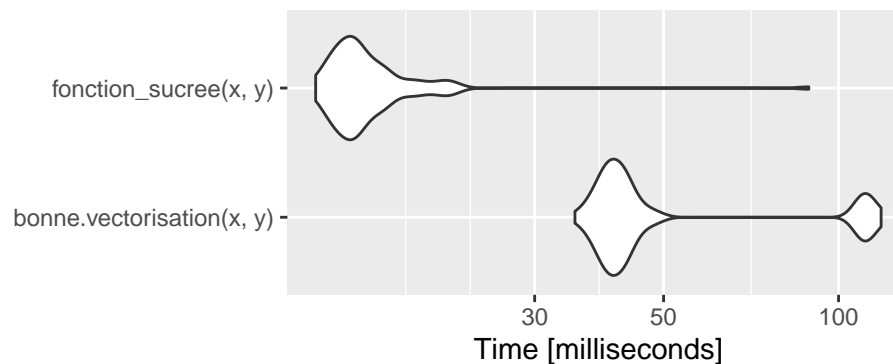
> x <- rnorm(1e6)
> y <- rnorm(1e6)
> mbm <- microbenchmark::microbenchmark( bonne.vectorisation(x,y), fonction_sucree(x,y) )
> mbm
Unit: milliseconds
      expr      min       lq      mean    median      uq      max  neval
bonne.vectorisation(x, y) 35.19087 39.94886 55.36508 42.34802 46.13692 118.36738   100
fonction_sucree(x, y) 12.59885 13.83564 16.11614 14.71586 16.43126  88.91227   100

```

```

> ggplot2::autoplot(mbm)

```



Revenons cependant à notre exemple de comptage de 0.

```

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

```

```

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

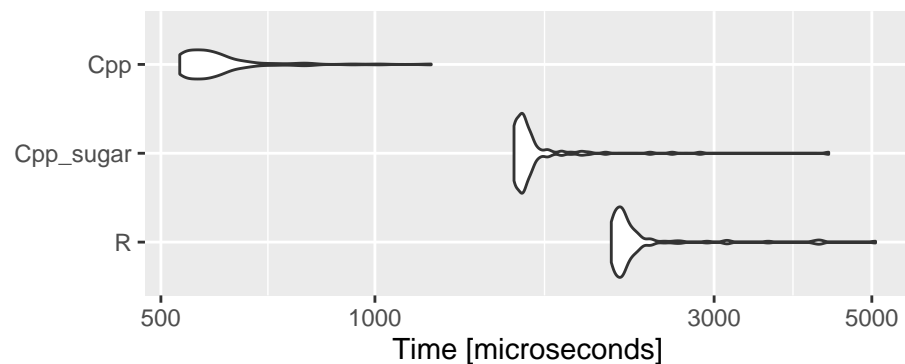
```

```

> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp_sugar = count_zeros_sugar(a), Cpp = count...
> mbm
Unit: microseconds
      expr      min       lq      mean    median      uq      max  neval
      R 2150.310 2188.0430 2374.2858 2230.608 2300.8305 5061.218   100
Cpp_sugar 1569.114 1590.7920 1700.2937 1617.358 1656.0245 4348.591   100
      Cpp  531.489  550.4395  604.9134  577.385  613.1225 1200.108   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.

4.4.2 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.

```
#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;
  }
  return (4.0 * (double) S / (double) N);
}
```

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

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

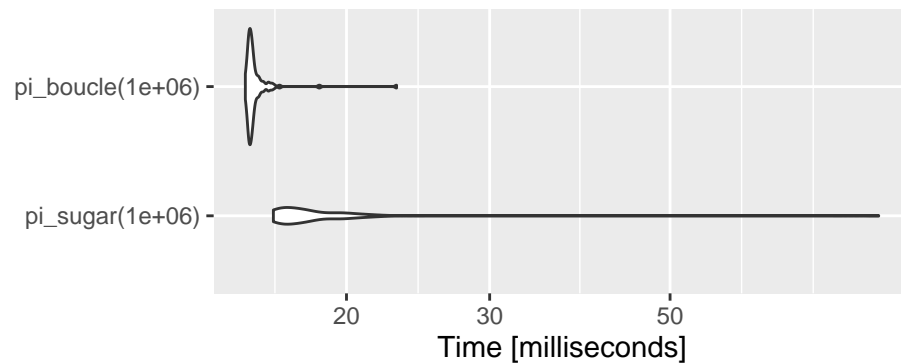
```
> mbm <- microbenchmark::microbenchmark(pi_sugar(1e6), pi_boucle(1e6))
```

```
> mbm
```

```
Unit: milliseconds
```

expr	min	lq	mean	median	uq	max	neval
pi_sugar(1e+06)	16.25889	16.69511	19.33209	17.59814	19.07943	90.28429	100

```
pi_boucle(1e+06) 15.01804 15.18473 15.46390 15.27342 15.43018 23.04769 100
> ggplot2::autoplot(mbm)
```



4.5 Facteurs

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector niveaux(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("Pas d'attribut 'levels'");
    default:
      stop("Attribut 'levels' de type inconvenant");
  }
}
```

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

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

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

```
> un_facteur()
[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.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector get_alpha_list(List x) {
  if( x.containsElementNamed("alpha") ) {
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("elt alpha n'est pas un 'NumericVector'");
    return R;
  } else
    stop("Pas d'elt alpha");
}
```

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

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void increment_alpha(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
```

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

$beta
```

```
[1] 1 2 3 4
```

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

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
List cree_liste() {
  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;
}
```

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 : il suffit la plupart de transformer le résultat en liste (avec `as.data.frame`) une fois qu'on l'a obtenu.

Chapitre 5

Exemple : Metropolis-Hastings

5.1 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.

5.1.1 Marche aléatoire

Une suite de valeurs aléatoires $x_1, x_2, \dots \in \mathbb{R}^d$ est une marche aléatoire¹ 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 loi normale de variance $\sigma^2 I_d$ et d'espérance x_t . Cela revient à dire que

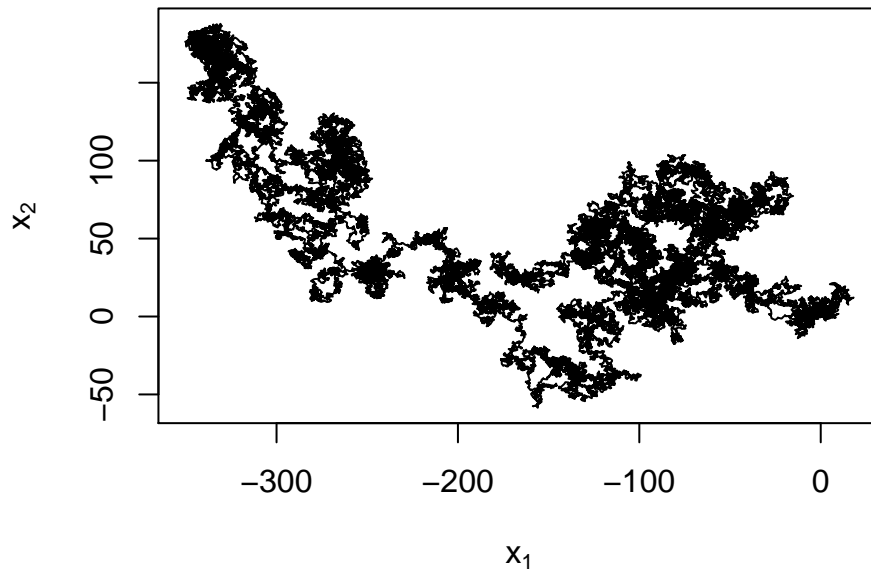
$$x_{t+1} = x_t + z$$

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 .

```
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")
```

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 ».



5.1.2 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 $\rho \geq 1$, 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 $x_{t+1} = x_t$.

5.1.2.1 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.

5.1.2.2 Les propriétés du résultat

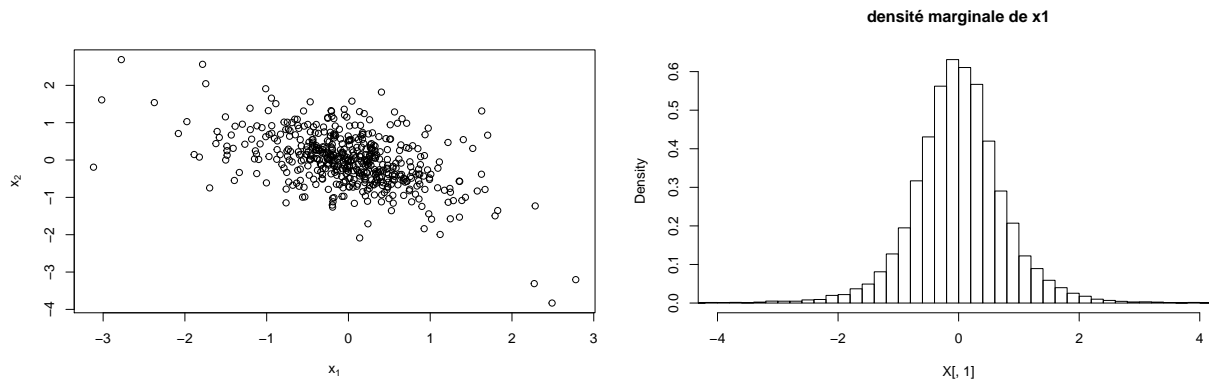
Si t est assez grand, alors x_t est approximativement de loi de densité $\pi(x)$ (ou proportionnelle à $\pi(x)$).

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*.

5.1.3 Application

On prend pour $x = (x_1, x_2)$, $\pi(x) = (1 + x_1^2 + x_1 x_2 + x_2^2)^{-3}$, 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. On doit obtenir quelque chose de similaire à ceci :



5.2 Une version R

```
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);
}
```

5.3 Première version en C++

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

```

#include <Rcpp.h>
using namespace Rcpp;

double Pi(double x1, double x2) {
  return pow((1 + x1*x1 + x1*x2 + x2*x2), -3.0);
}

//[[Rcpp::export]]
NumericMatrix MH_cpp(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;
}

```

5.4 Comparaisons

TBC

5.5 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.

```

#include <Rcpp.h>
using namespace Rcpp;

double Pi(double x1, double x2) {
  return pow((1 + x1*x1 + x1*x2 + x2*x2), -3.0);
}

//[[Rcpp::export]]
NumericMatrix MH_cpp_1(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 = 0;
  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 calculer !

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).