R et C++

Romain François

romain@r-enthusiasts.com

@romain_francois

Mise en route

Que font ces fonctions C++ ?

Chacune correspond à une fonction R de base

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;
  for(int i = 0; i < n; i++) {
    y += x[i];
  }
  return y / n;
}
```

```
NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);
  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
}
```

Pourquoi C++ ?

Vectorisation

- La clé pour de bonnes performances en R est la vectorisation
- Vectorisation = Boucle écrite en C
- Vectorisation = Bon Vocabulaire
- ok d'écrire des boucles en C++

Probability you get a vaccination

Infected last year

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) +
    0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

```
# 🐈
vacc1a(40, TRUE, FALSE)
```

```
# 💔
vacc1a(c(20, 40),
  c(TRUE, FALSE), c(FALSE, FALSE))
```

Vectorisation "barbare" 🔨

```
vacc1 <- function(age, female, ily) {
   n <- length(age)
   res <- numeric(n)
   for( i in seq_len(n) ){
      res[i] <- vacc1a( age[i], female[i], ily[i] )
   }
   res
}


vacc2 <- function( age, female, ily ){
   mapply( vacc1a, age, female, ily )
}
```

Ecrivez une version vectorisée de vacc1a

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

```r
vacc3 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) +
    0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

version C++

```
// fonction non vectorisée
double vacc4a(double age, bool female, bool ily){
    double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
    p = p * (female ? 1.25 : 0.75) ;
    if( p < 0.0 ) p = 0.0 ;
    if( p > 1.0 ) p = 1.0 ;
    return p;
}

// [[Rcpp::export]]
NumericVector vacc4( NumericVector age, LogicalVector female, LogicalVector ily ){
    int n = age.size() ;
    NumericVector out(n) ;
    for( int i=0; i<n; i++){
        out[i] = vacc4a( age[i], female[i], ily[i] ) ;
    }
    return out;
}
```

Version R avec des astuces

```
vacc5 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * (0.5 * female + 0.75)
  p[p < 0] <- 0
  p[p > 1] <- 1
  p
}
```

version C++ bonus (avec mapply)

```
// fonction non vectorisée
double vacc4a(double age, bool female, bool ily){
    double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
    p = p * (female ? 1.25 : 0.75) ;
    if( p < 0.0 ) p = 0.0 ;
    if( p > 1.0 ) p = 1.0 ;
    return p;
}

// [[Rcpp::export]]
NumericVector vacc6( NumericVector age, LogicalVector female, LogicalVector ily ){
    return mapply( age, female, ily, vacc4a ) ;
}
```

```
bench_all <- function(n){
  age <- rnorm( n, mean = 35, sd = 5)
  ily <- sample( c(T,F), n, replace = TRUE )
  female <- sample( c(T,F), n, replace = TRUE )

  microbenchmark(
    barbare     = vacc1(age, female, ily),
    mapply      = vacc2(age, female, ily),
    vectorised  = vacc3(age, female, ily),
    astuce      = vacc5(age, female, ily),
    cpp         = vacc4(age, female, ily),
    cpp_mapply  = vacc6(age, female, ily)
  )
}

> bench_all( 1e4 )
Unit: microseconds
       expr       min          lq       mean     median          uq       max neval
    barbare 31361.995 39706.7245 43313.4645 44642.4555 46077.8700 83338.858   100
     mapply 31951.518 37762.6505 41977.2340 42974.0915 45073.6105 80358.871   100
 vectorised  1816.154  1863.4635  2018.0800  1920.9640  2153.7630  2924.513   100
     astuce   244.383   252.0670   284.8837   261.4580   275.3880   881.074   100
        cpp   144.543   149.3875   163.7788   156.4105   174.0765   255.989   100
 cpp_mapply   148.594   152.1825   162.1056   156.9830   169.0080   266.445   100
```

```
bench_fast <- function(n){
  age <- rnorm( n, mean = 35, sd = 5)
  ily <- sample( c(T,F), n, replace = TRUE )
  female <- sample( c(T,F), n, replace = TRUE )

  microbenchmark(
    vectorised  = vacc3(age, female, ily),
    astuce      = vacc5(age, female, ily),
    cpp         = vacc4(age, female, ily),
    cpp_mapply  = vacc6(age, female, ily)
  )
}
```

```
> bench_fast( 1e5 )
Unit: milliseconds
       expr       min         lq      mean    median        uq       max neval
 vectorised 21.506944 22.794844 25.456525 23.579265 24.793819 62.794472   100
     astuce  2.465745  3.271930  3.787820  3.730076  4.163524  6.829233   100
        cpp  1.443867  1.530507  1.768969  1.763629  1.820133  3.317670   100
 cpp_mapply  1.463295  1.522410  1.829126  1.778884  1.898599  3.480135   100
```

Problèmes difficiles avec R

- Pas possible de vectoriser parce que l'itération dépend des résultats précédents
- Appels récursifs ou nombreux à des fonctions
- Structures de données avancées

Pourquoi C++

- Moderne, haute performances
- Contrôle précis de l'allocation mémoire
- Bonnes libraries standard (stl, boost, ...)
- Plus facile que le C ou le Fortran
- Pas trop compliqué à apprendre

Pourquoi pas C++

- Apprendre un autre langage
- Temps de développement plus long
- Finalement, c'est peut être plus simple d'augmenter son vocabulR

Ce que Rcpp fait pour vous

```
library(Rcpp)
cppFunction('int one() {
  return 1;
}', verbose = TRUE)
one()
```

Fichier .cpp généré

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int one() {
   return 1;
}
```

Code généré d'interfaçage avec R:

```
#include <Rcpp.h>

// one
int one();
RcppExport SEXP sourceCpp_51404_one() {
BEGIN_RCPP
    Rcpp::RObject __result;
    Rcpp::RNGScope __rngScope;
    __result = Rcpp::wrap(one());
    return __result;

END_RCPP
}
```

Compilation

```
/Library/Frameworks/R.framework/Resources/bin/R CMD SHLIB -o
'sourceCpp_567.so'  'one.cpp'
clang++ -I/Library/Frameworks/R.framework/Resources/include -
DNDEBUG  -I/usr/local/include -I/usr/local/include/freetype2 -
I/opt/X11/include  -I"/Library/Frameworks/R.framework/
Versions/3.2/Resources/library/Rcpp/include" -I"/private/tmp"
-fPIC  -O3 -c one.cpp -o one.o
clang++ -dynamiclib -Wl,-headerpad_max_install_names -
undefined dynamic_lookup -single_module -multiply_defined
suppress -L/Library/Frameworks/R.framework/Resources/lib -L/
usr/local/lib -o sourceCpp_567.so one.o -F/Library/Frameworks/
R.framework/.. -framework R -Wl,-framework -Wl,CoreFoundation
```

Chargement de la DLL

```
`.sourceCpp_51404_DLLInfo` <-
dyn.load('/var/folders/2p/
w41bln195jv507lc9mffd0fr0000gn/T//
RtmptA49Ij/sourcecpp_111a33418ed6/
sourceCpp_567.so')
```

Création de la fonction R:

```
one <-
Rcpp::sourceCppFunction(function() {},
FALSE, `.sourceCpp_51404_DLLInfo`,
'sourceCpp_51404_one')
rm(`.sourceCpp_51404_DLLInfo`)
```

Comment en bénéficier:

- install.packages("Rcpp")
  - Sur windows, installer Rtools
  - Sur OSX, installer Xcode et les outils
- Gérez votre PATH, ou utilisez RStudio

Fonctions

C++

Environnement de base

- Les fonctions C++ vivent dans un fichier .cpp
- On les "source" avec sourceCpp
- Rcp s'occupe du reste

```cpp
#include <Rcpp.h>
using namespace Rcpp ;

// [[Rcpp::export]]
int one(){
  return 1 ;
}
```

} Vrai code

```
> sourceCpp( "one.cpp" )
> one()
[1] 1
```

Ecrire une fonction two() qui retourne 2

Défi 1 : Pas de copier/coller

Défi 2 : De mémoire

Types de base

R	Vector	Scalar
character	CharacterVector	String
double	NumericVector	double
integer	IntegerVector	int
logical	LogicalVector	bool

Scalaires en entrée et en sortie

```
#include <Rcpp.h>
using namespace Rcpp ;

// [[Rcpp::export]]
double add( double x, double y){
  return x + y ;
}
```

Exercice

- Ecrire une fonction norme: f(x,y) = sqrt(x^2 + y^2)
- En utilisant pow(,2.0) and pow(, 0.5)
- Avec sqrt et une fonction square

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double norm1( double x, double y){
    return pow( pow(x,2) + pow(x,2), 0.5 ) ;
}
```

```
#include <Rcpp.h>
using namespace Rcpp;


inline double square( double x){
    return x*x ;
}

// [[Rcpp::export]]
double norm2( double x, double y){
     return sqrt( square(x) + square(y) ) ;
}
```

Vecteurs en entrée

Scalaires en sortie

```
sumR <- function( x ){
    res = 0
    for( i in seq_along( x ) ){
        res = res + x[i]
    }
    res
}
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sumCpp( NumericVector x){
    double res = 0.0;
    int n = x.size();
    for( int i=0; i<n; i++){
      res += x[i] ;
    }
    return res ;
}
```

En C++ les vecteurs commencent à 0

En C++ les vecteurs commencent à 0

En C++ les vecteurs commencent à 0

Exercice

Ecrire une fonction weighted_mean

Comparez la avec la version base weighted.mean et à cette version inefficace :

```
weighted_mean <- function(x, w) {
   total <- 0
   total_w <- 0
   for (i in seq_along(x)) {
     total <- total + x[i] * w[i]
     total_w <- total_w + w[i]
   }
   total / total_w
 }
```

```
#include <Rcpp.h>
using namespace Rcpp ;

// [[Rcpp::export]]
double weighted_mean_cpp( NumericVector x, NumericVector w){
    int n = x.size() ;
    double total = 0.0 ;
    double total_w = 0.0 ;
    for( int i=0; i<n; i++){
       total += x[i] * w[i] ;
       total_w += w[i] ;
    }
    return total / total_w ;
}
```

RcppParallel

(tbb)

Appeler une fonction simple en parallèle

```
x <- rnorm(1e6 )
y <- sqrt(x)
```

Version séquentielle avec Rcpp

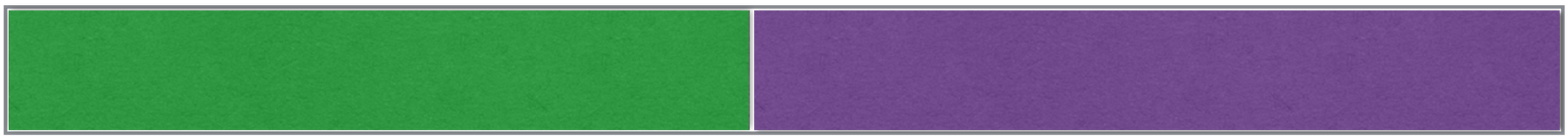
```
#include <Rcpp.h>
using namespace Rcpp ;

// [[Rcpp::export]]
NumericVector cpp_sqrt( NumericVector x ){
    int n = x.size();
    NumericVector y = no_init(n) ;
    for( int i=0; i<n; i++){
       y[i] = sqrt(x[i]) ;
    }
    return y ;
}
```

x

sqrt

sqrt

y



```
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp ;
using namespace RcppParallel ;

// [[Rcpp::depends(RcppParallel)]]
struct SquareRoot : public Worker {
   public:
      SquareRoot( const NumericVector& x_, NumericVector& y_ ) :  x(x_), y(y_){}
      void operator()( std::size_t begin, std::size_t end ){
          for( std::size_t i=begin; i<end; i++){
              y[i] = sqrt( x[i] ) ;
          }
      }
   private:
      const NumericVector& x ;
      NumericVector& y ;

};

// [[Rcpp::export]]
NumericVector cpp_sqrt_parallel_1( NumericVector x ){
    int n = x.size();
    NumericVector y = no_init(n) ;
    SquareRoot obj( x, y ) ;
    parallelFor( 0, n, obj ) ;
    return y ;
}
```

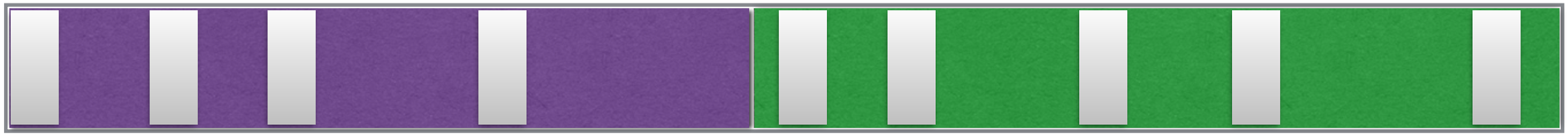
```
> x <- runif( 1e6 )
> microbenchmark( R = sqrt(x), serial = cpp_sqrt(x), parallel = cpp_sqrt_parallel_1(x) )
Unit: milliseconds
     expr      min       lq     mean   median       uq        max neval
        R 5.097073 7.023715 7.271958 7.140185 7.433031  10.053800   100
   serial 4.228400 6.961709 9.325789 7.134970 7.303951 225.694180   100
 parallel 1.205141 2.979181 3.156141 3.249106 3.535080   4.562999   100
```

Compter le nombre de positifs

```
x <- rnorm(1e6 )
n <- sum(x > 0)
```

```
#include <Rcpp.h>
using namespace Rcpp ;

// [[Rcpp::export]]
int count_positive_serial( NumericVector x ){
    int n = x.size();
    int res = 0 ;
    for( int i=0; i<n; i++){
        if( x[i] > 0 ) res++ ;
    }
    return res ;
}

/*** R
   x <- rnorm( 1e6 )
   n <- count_positive_serial( x )
   n
*/
```

x



count

count

4

5

9

```
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp ;
using namespace RcppParallel ;

// [[Rcpp::depends(RcppParallel)]]
struct Counter : public Worker {
    Counter( const NumericVector& x_ ) : x(x_), count(0){}
    Counter( const Counter& counter, Split) : x(counter.x), count(0) {}
    void operator()(std::size_t begin, std::size_t end) {
      int res=0 ;
      for( std::size_t i=begin; i<end; i++){
        if( x[i] > 0 ) res++ ;
      }
      count = res ;
    }
    void join(const Counter& rhs) {
      count += rhs.count;
    }

    const NumericVector& x ;
    int count ;

};



// [[Rcpp::export]]
int count_positive_parallel( NumericVector x ){
    int n = x.size();
    Counter counter(x) ;
    parallelReduce(0, n, counter, 10000);
    return counter.count ;
}
```

```
>   microbenchmark(
+     R = sum(x>0),
+     serial = count_positive_serial(x),
+     parallel = count_positive_parallel(x)
+   )
```

```
Unit: microseconds
     expr      min         lq       mean    median        uq      max neval
        R 2665.843 3943.6685 4017.3036 4041.5860 4225.2470 5872.889   100
   serial  362.405  405.9945  455.4635  431.8625  489.7240  780.104   100
 parallel  116.066  165.7270  197.7761  197.3605  226.6915  482.575   100
```

vacc

version C++

```
// fonction non vectorisée
double vacc4a(double age, bool female, bool ily){
    double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
    p = p * (female ? 1.25 : 0.75) ;
    if( p < 0.0 ) p = 0.0 ;
    if( p > 1.0 ) p = 1.0 ;
    return p;
}

// [[Rcpp::export]]
NumericVector vacc4( NumericVector age, LogicalVector female, LogicalVector ily ){
    int n = age.size() ;
    NumericVector out(n) ;
    for( int i=0; i<n; i++){
        out[i] = vacc4a( age[i], female[i], ily[i] ) ;
    }
    return out;
}
```

```
struct Vacc : Worker {
  NumericVector& out ;
  const NumericVector& age ;
  const LogicalVector& female ;
  const LogicalVector& ily ;

  Vacc( NumericVector& out_, const NumericVector& age_, const LogicalVector&
female_, const LogicalVector& ily_ ) :
    out(out_), age(age_), female(female_), ily(ily_){}

  void operator()(std::size_t begin, std::size_t end){
    for(std::size_t i=begin; i<end; i++){
      out[i] = vacc4a( age[i], female[i], ily[i]) ;
    }
  }

};

// [[Rcpp::export]]
NumericVector vacc7( NumericVector age, LogicalVector female, LogicalVector ily ){
  int n = age.size() ;
  NumericVector out = no_init(n) ;

  Vacc obj(out, age, female, ily) ;
  parallelFor( 0, n, obj) ;
  return out ;
}
```