A crash course on Rcpp

01-28-2020

```r
library(tidyverse)
```

```
## -- Attaching packages --------- tidyverse 1.3.0 --

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

System requirements

To run c++ code in R, you will need to first install a workable toolchain.

- Windows
  - Install Rtools
- macOS
  - Run `which clang` in terminal to see if you have already installed it
  - If not, run `xcode-select --install` to install it.
- Linux
  - It depends

In R, we will need to install the package `Rcpp`.

The following sections will teach you the basics by translating simple R functions to their C++ equivalents

- No inputs, scalar output
- Scalar input and scalar output
- Vector input and scalar output
- Vector input and vector output
- Matrix input and vector output
- DataFrame input and DataFrame output

No inputs, scalar output

This function has no arguments and always returns the integer 1:

```r
oneR <- function() 1L
```

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

- You must declare the type of output the function returns.
- Every statement is terminated by a `;`
- R doesn't have the idea of scalar variables but scalars and vectors in C++ are different.

R type	Rcpp type
int scalar	int
double scalar	double
logicial scalar	bool
character scalar	String
int vector	IntegerVector
double vector	NumericVector
logicial vector	LogicalVector
character vector	CharacterVector

```r
oneR()
```

```
## [1] 1
```

```r
oneC()
```

```
## [1] 1
```

Scalar input, scalar output

The next example function implements a scalar version of the sign() function which returns 1 if the input is positive, and -1 if it's negative:

```r
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}
```

```cpp
// [[Rcpp::export]]
int signC(int x) {
  if (x > 0) {
    return 1;
```

```
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}
```

- We declare the type of each input in the same way we declare the type of the output.
- The `if` syntax is identical.
- The `while` loop syntax is almost the same, we use `break` to break the loop but use `continue` instead of `next` skip current iteration.

```
// [[Rcpp::export]]
int while_ex(int x) {
  while(x < 9) {
    x = x + 1;
    if (x == 7) {
      break;
    }
  }
  return x;
}
```

```
while_ex(1)
```

```
## [1] 7
```

```
while_ex(7)
```

```
## [1] 9
```

```
while_ex(10)
```

```
## [1] 10
```

Vector input, scalar output

It is where things get compicated.

```
# it is the inefficient version of the function `sum`
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

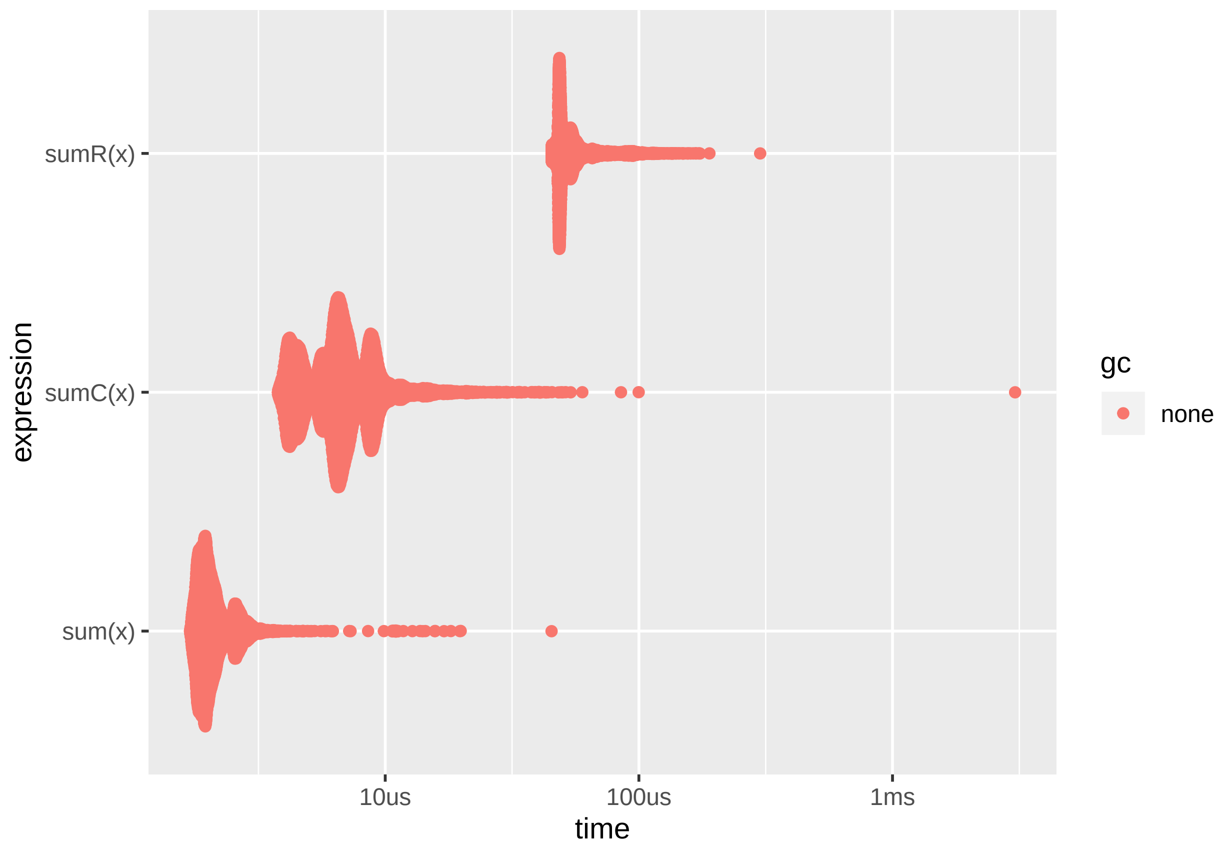
```
// The first two lines import the namespace Rcpp for things such as `NumericVector`
#include <Rcpp.h>
using namespace Rcpp;

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

- `NumericVector` is the Rcpp type of a double vector. To find its length, we could use the `.size()` method of `NumericVector`.
- We need to specify the type of each variable. `int n` and `double total`.
- The `for` loop starts with `i = 0` and ends if `i < n`.
- After each iteration, we increment the value of `i` by one, `i++` increases the value of `i` by 1
- In C++, VECTOR INDICES START AT 0!!! The first element is `x[0]` and the last element is `x[n - 1]`;
- Use `=` for assignment, not `<-`
- `total += x[i]` means `total = total + x[i]`. Similarly, there are `-=`, `*=`, and `/=`.

Let's benchmark the performances

```
x <- runif(1e3)
result <- bench::mark(
  sum(x),
  sumC(x),
  sumR(x)
)
ggplot2::autoplot(result)
```



Of course `sum` is the fastest because it is highly optimized.

Vector input, vector output

This is a function that computes the Euclidean distance between a value and a vector of values:

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
y <- runif(1e5)
pdistR(0.5, y)
```

```
##  [1] 0.255485640 0.229890789 0.054039210 0.270676364 0.122179186 0.462412476
##  [7] 0.135088415 0.194867675 0.435889564 0.051341986 0.066068572 0.452016024
## [13] 0.087847646 0.282962840 0.275337823 0.047548974 0.284258967 0.048361412
## [19] 0.467163491 0.074895364 0.087819253 0.195562997 0.087657041 0.180136611
## [25] 0.072682167 0.395440088 0.301997955 0.004604136 0.337813990 0.032160942
## [31] 0.011194430 0.176914385 0.204771344 0.157339890 0.066551447 0.003999129
## [37] 0.222183961 0.129084832 0.156721736 0.431577813 0.083635426 0.375309794
## [43] 0.118110161 0.145013505 0.369137626 0.198897033 0.455870834 0.234076293
## [49] 0.204533324 0.420571555
##  [ reached getOption("max.print") -- omitted 99950 entries ]
```

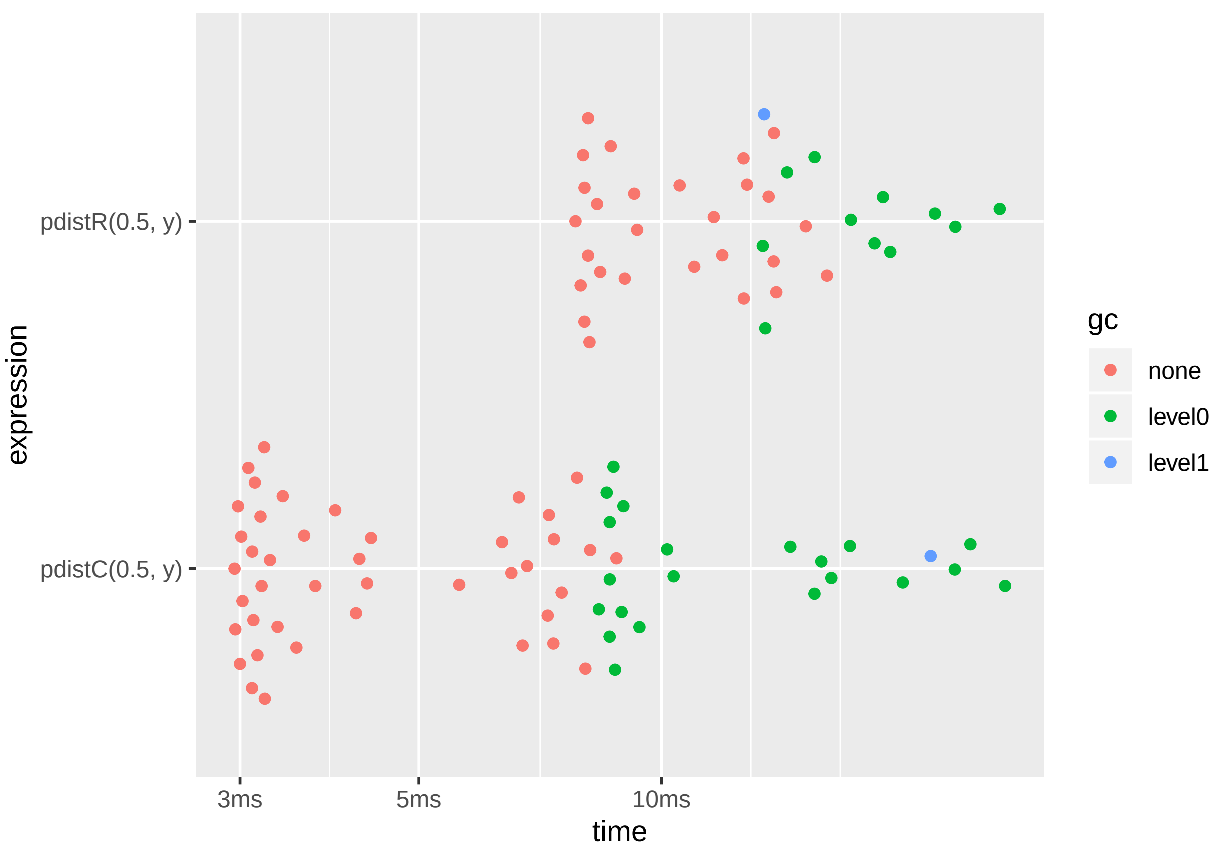
```
#include <Rcpp.h>
using namespace Rcpp;
```

```
// [[Rcpp::export]]
NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; i++) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}
```

- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n);`.
- Another useful way of making a vector is to copy an existing one: `NumericVector zs = clone(ys);`.
- C++ uses `pow()`, not `^`, for exponentiation.
- C++ is faster because it avoids the creation of temperary object like `x - ys` and `(x - ys)^2`.

```
y <- runif(1e6)
ggplot2::autoplot(bench::mark(
  pdistR(0.5, y),
  pdistC(0.5, y)
))
```



- We saved about `5ms` by rewriting the R function in C++. Assume it takes 5 mins for you to write that function, you'd need to run it ~60 000 times to make rewriting worthwhile.

Sugar functions

Rcpp provides a number of sugar functions (check https://teuder.github.io/rcpp4everyone_en/210_rcpp_functions.html#list-of-r-like-functions for a list of such functions) that make write C++ code easier.

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

// [[Rcpp::export]]
NumericVector slr_cpp(NumericVector x, NumericVector y) {
  double mux = mean(x);
  double muy = mean(y);
  double sxy = sum((x - mux)*(y - muy));
  double sxx = sum(pow(x - mux, 2));
  double slope = sxy / sxx;
  double intercept = muy - slope * mux;
  return NumericVector::create(intercept, slope);
}
```

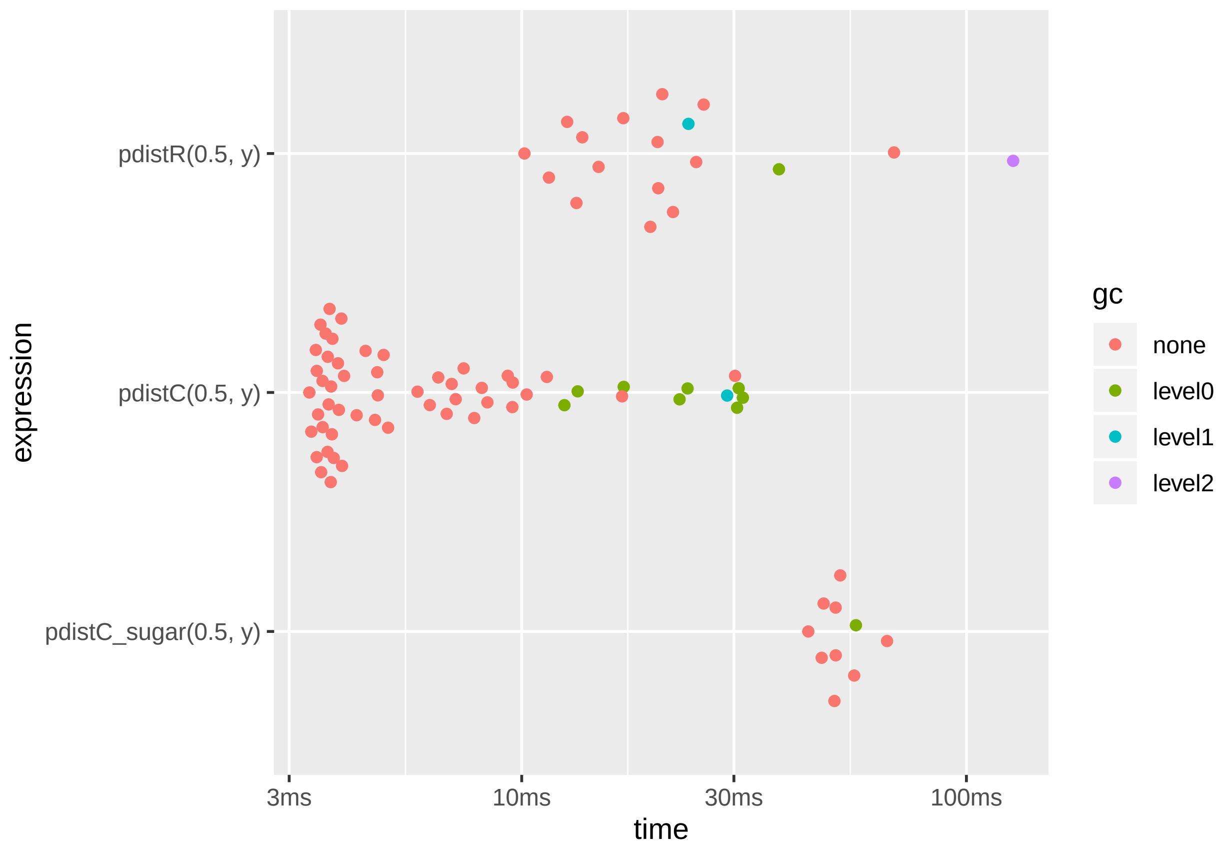
Caution

Using sugar functions could be even slower than R functions.

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

// [[Rcpp::export]]
NumericVector pdistC_sugar(double x, NumericVector ys) {
  return sqrt(pow(ys - x, 2.0));
}
```

```
y <- runif(1e6)
ggplot2::autoplot(bench::mark(
  pdistR(0.5, y),
  pdistC(0.5, y),
  pdistC_sugar(0.5, y)
))
```



`pdistC_sugar` is the slowest among the three versions. The slowness is due to the creation of temporary objects such as `ys-x` and `pow(ys - x, 2.0)`.

More on Vector Subsetting

Rcpp vectors could be extracted in "almost" the same way as R vectors.

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

// [[Rcpp::export]]
NumericVector positives(NumericVector x) {
    return x[x > 0];
}

// [[Rcpp::export]]
List first_three(List x) {
    IntegerVector idx = IntegerVector::create(0, 1, 2);
    return x[idx];
}

// [[Rcpp::export]]
List with_names(List x, CharacterVector y) {
    return x[y];
}
```

```r
x <- -5:5
positives(x)
```

```
## [1] 1 2 3 4 5
```

```r
l <- as.list(1:10)
first_three(l)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

```r
l <- l %>% set_names(letters[1:10])
with_names(l, c("a", "e", "g"))
```

```
## $a
## [1] 1
##
## $e
## [1] 5
##
## $g
## [1] 7
```

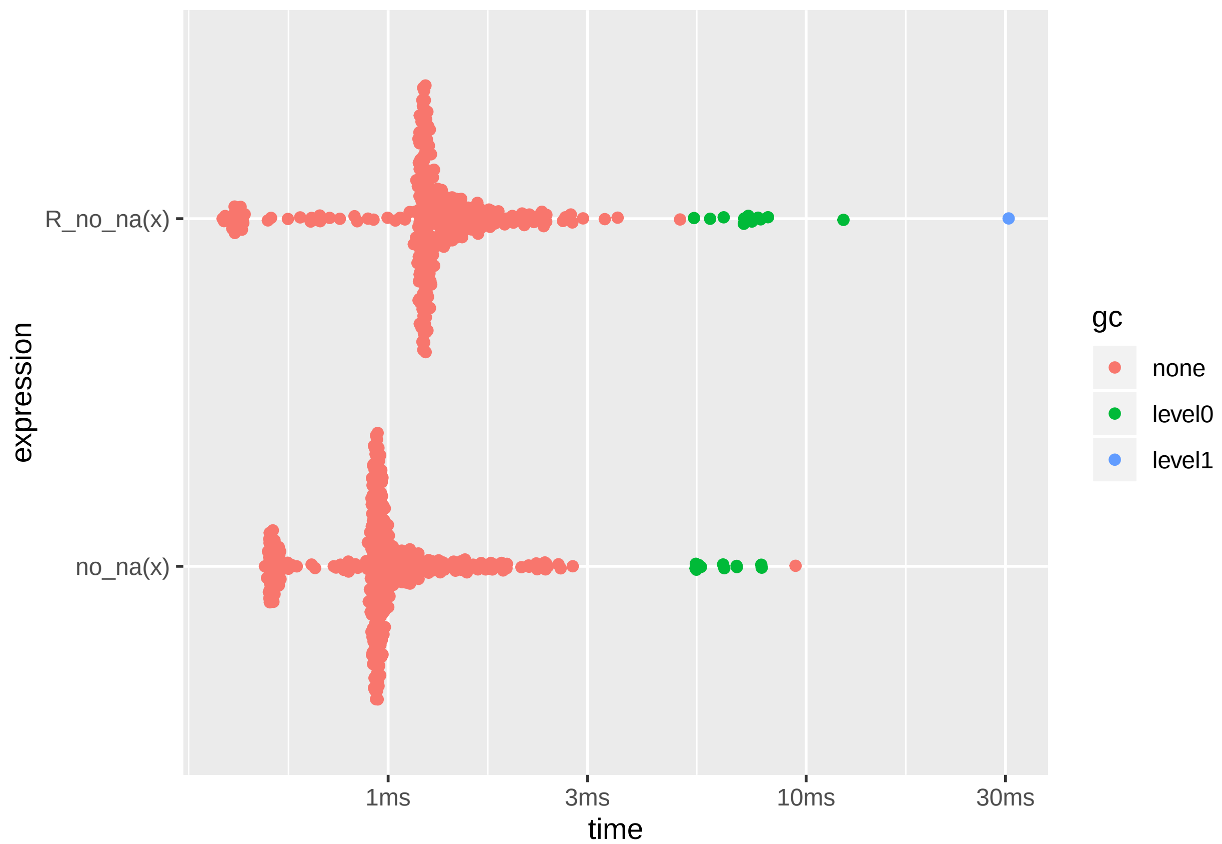
Most excitingly, the subset mechanism works well with Rcpp sugar.

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

// [[Rcpp::export]]
NumericVector no_na(NumericVector x) {
    return x[!is_na(x)];
}
```

```r
R_no_na <- function(x) {
    return( x[!is.na(x)] )
}
set.seed(1)
x <- rnorm(1e5)
x[sample(1e5, 1e4)] <- NA

ggplot2::autoplot(bench::mark(
  R_no_na(x),
  no_na(x)
))
```



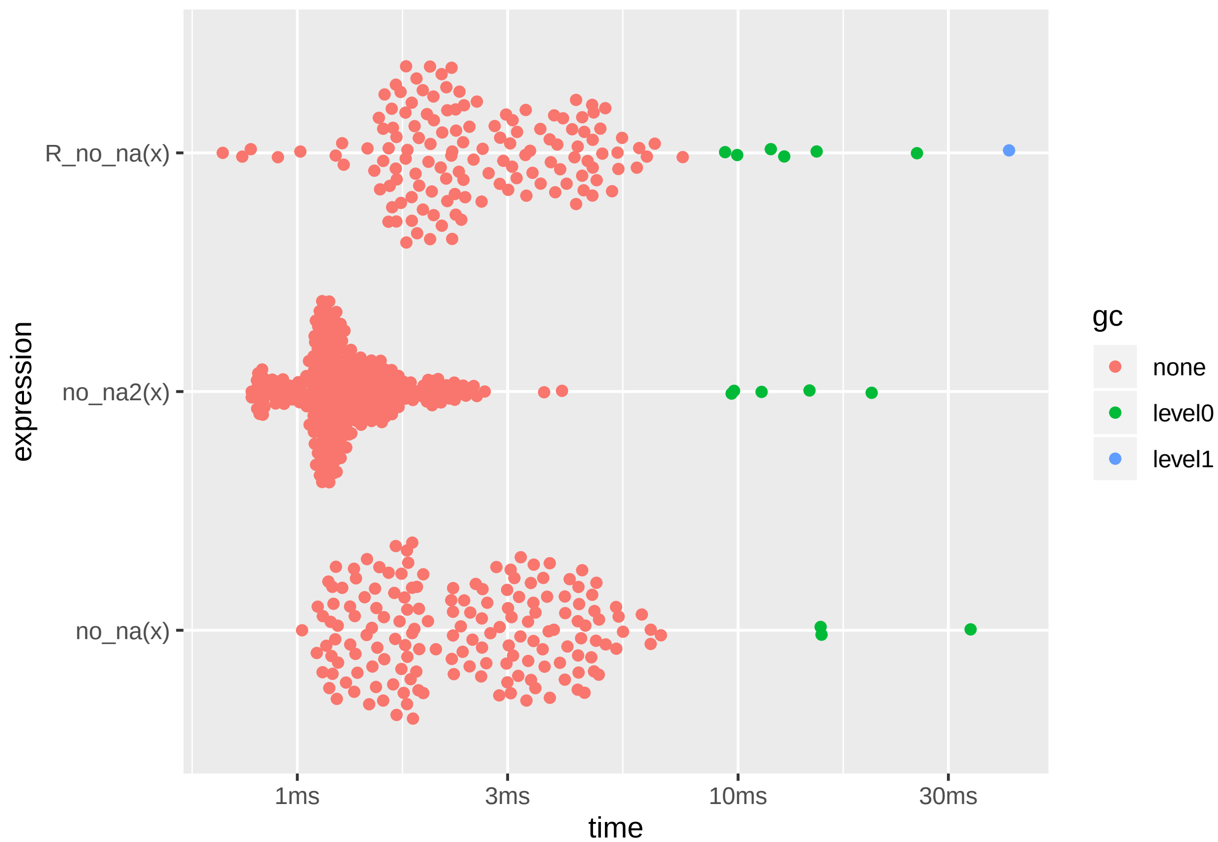
I'm curiously the performance of a native Rcpp.

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

// [[Rcpp::export]]
NumericVector no_na2(NumericVector x) {
  int n = x.size();
  int m = 0;
  int i, j = 0;
  for (i = 0; i < n; i++) {
    m += !R_IsNA(x[i]);
  }
  NumericVector y = NumericVector(m);
  for (i = 0; i < n; i++) {
    if (R_IsNA(x[i])) continue;
    y[j] = x[i];
    j++;
  }
  return y;
}
```

```
ggplot2::autoplot(bench::mark(
  R_no_na(x),
  no_na(x),
  no_na2(x)
```

```
))
```



Again, we see that a naive for loop is faster.

Values could be modified in place!!

With cloning the object, any changes in C++ may modify the values in R.

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

// [[Rcpp::export]]
void modifyVec(NumericVector x) {
  x[0] = 42;
}
```

```
x <- c(3, 5)
modifyVec(x)
x
```

```
## [1] 42  5
```

However, this side effect does not work if there is implict conversion.

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

// [[Rcpp::export]]
void modifyVec2(IntegerVector x) {
  x[0] = 42;
}
```

```r
x <- c(3, 5)  # it is double
modifyVec2(x)  # a converted vector is passed to C++
x
```

```
## [1] 3 5
```

Matrix input and vector output

R type	Rcpp type
int matrix	IntegerMatrix
double matrix	NumericMatrix
logicial matrix	LogicalMatrix
character matrix	CharacterMatrix

Our next example is to compute column sum of a matrix. (This example is for demonstration only, in practice, we use `colSums` directly)

```r
colSumsR <- function(x) {
  m <- ncol(x)
  n <- nrow(x)
  output <- double(m)
  for (j in seq_len(m)) {
    output[j] = 0
    for (i in seq_len(n))
      output[j] = output[j] + x[i, j]
  }
  output
}
```

```r
(x <- matrix(rnorm(9), nc = 3))
```

```
##            [,1]       [,2]      [,3]
## [1,] -0.1858388  1.3094334 -1.958362
## [2,]  0.3434321 -0.9157131  1.533542
## [3,]  0.4462492  0.6140196  1.204067
```

```r
colSumsR(x)
```

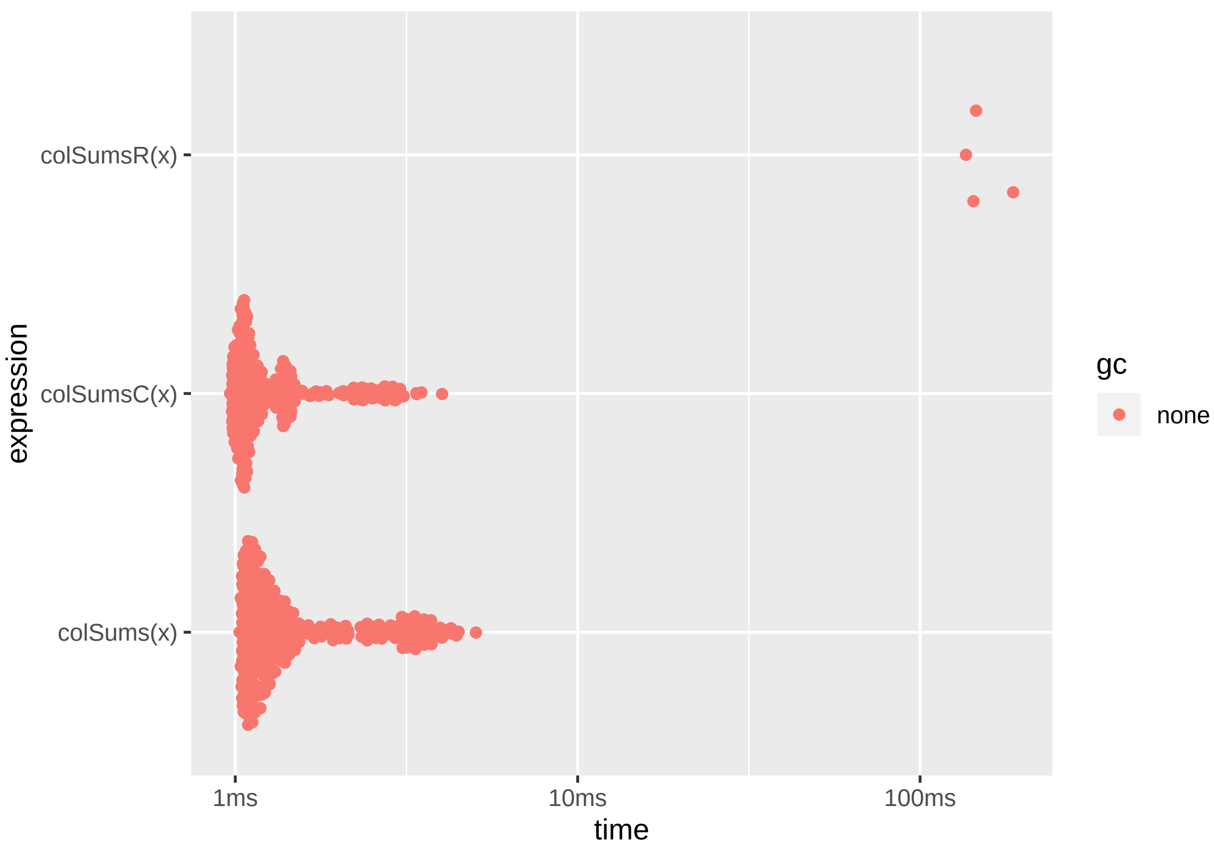
```
## [1] 0.6038425 1.0077399 0.7792470
```

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

// [[Rcpp::export]]
NumericVector colSumsC(NumericMatrix x) {
  int m = x.ncol();
  int n = x.nrow();
  NumericVector output = NumericVector(m);
  for (int j = 0; j < m; j++) {
    output[j] = 0;
    for (int i = 0; i < n; i++)
      output[j] += x(i, j);
  }
  return output;
}
```

- `.ncol()` and `.nrow()` return the number of columns and rows of the matrix
- we use `x(i, j)` instead of `x[i, j]` to retrieve the i, j th element of the matrix

```r
x <- matrix(runif(1e6), 1e3)
ggplot2::autoplot(bench::mark(
  colSums(x),
  colSumsR(x),
  colSumsC(x)
))
```



Again, we see that `colSumsC` is much faster than the pure R implementation `colSumsR`. The built-in `colSums` is the fastest because it is highly optimized.

DataFrame input and DataFrame output

The following example modifies some values of a dataframe.

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

// [[Rcpp::export]]
DataFrame modifyDataFrame(DataFrame df) {
  // access the columns
  // it is very important to clone `a` and `b`!!
  // since `clone()` does not know what `df["a"]` is, we need to specify the target type
  NumericVector a = clone<NumericVector>(df["a"]);
  CharacterVector b = clone<CharacterVector>(df["b"]);
  // make some changes
  a[1] = 42;
  b[1] = "foo";
  // return a new data frame
  return DataFrame::create(_["a"]= a, _["b"]= b);
}
```

```
df <- tibble(
    a = c(1, 2, 3),
    b = c("x", "y", "z"))
modifyDataFrame(df)
```

```
##    a   b
## 1  1   x
## 2 42 foo
## 3  3   z
```

List input

This is again not a good example because the same task could be easily done in R. It calcualtes the mean percentage error (`mpe()`) of a linear model.

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

// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  // we don't have to clone because the values are not altered
  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
```

```
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}
```

```r
mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod)
```

```
## [1] -0.01541615
```

Pass user function to Rcpp

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

// [[Rcpp::export]]
RObject callWithTwo(Function f) {
  return f(2);
  // if you want to use named argument
  // return f(_["x"] = 2);
}
```

```r
f <- function(x) x + 1
g <- function(x) 3^x
callWithTwo(f)
```

```
## [1] 3
```

```r
callWithTwo(g)
```

```
## [1] 9
```

More advanced topics

- C++ Standard Template Library

  The standard template library (STL) provides a set of extremely useful data structures and algorithms. However, it requires some background in C++ in order to disclose its full potential. (I will let you do your own reserach.)

- armadillo and eigen library

  If you want to do linear algebra opertions in C++, we should take a look at Rcpparmadillo and RcppEigen.

Reference

- Rcpp for everyone https://teuder.github.io/rcpp4everyone_en/
- Rcpp chapter of Advanced R https://adv-r.hadley.nz/rcpp.html
- Rcpp gallery https://gallery.rcpp.org/
- Rcpp Quick Reference Guide http://dirk.eddelbuettel.com/code/rcpp/Rcpp-quickref.pdf