A crash course on Rcpp

01-28-2020

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

No inputs, scalar output

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

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

```
// [[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

```
oneR()
```

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

```
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:

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

```
// [[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
}
```

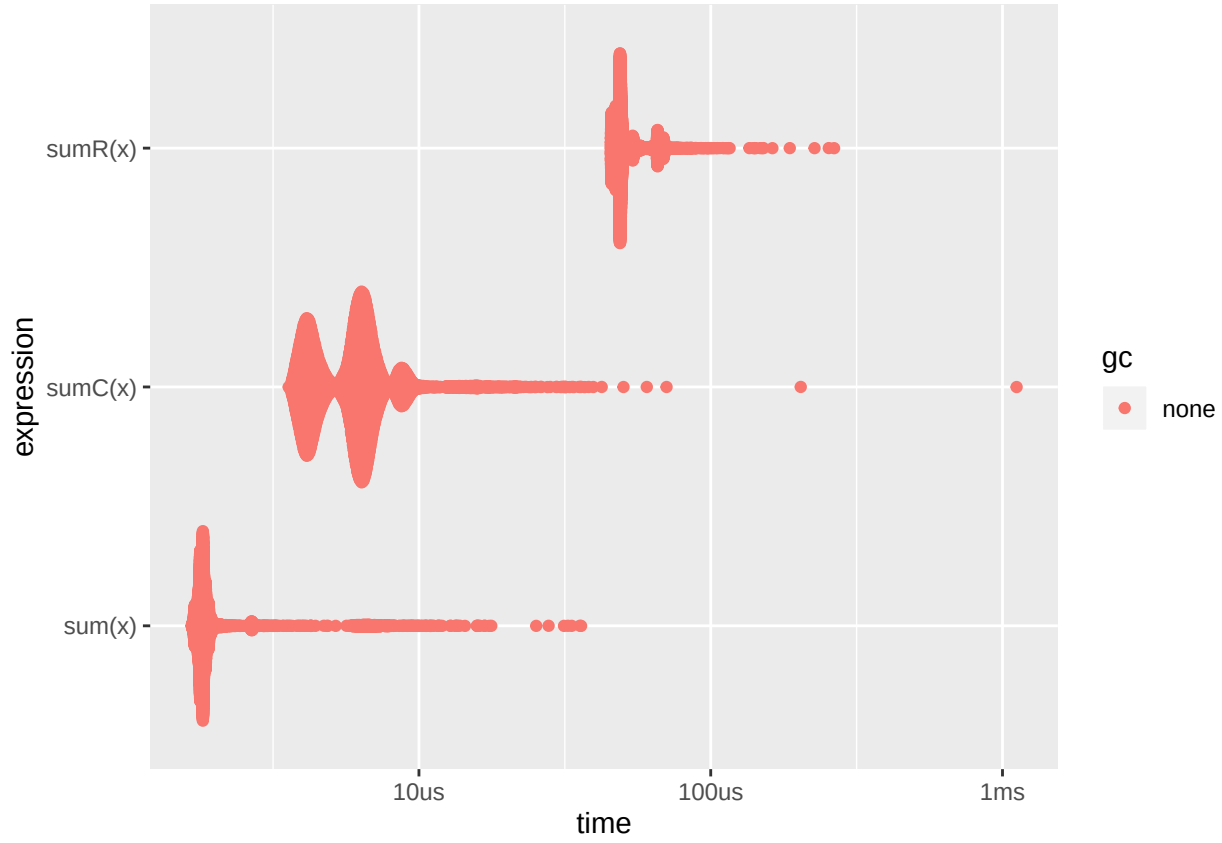
```
// The first two lines import the namespace Rcpp for things such as `NumericVector`
#include <Rcpp.h>
using namespace Rcpp;

// [[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)
```

```
## Loading required namespace: tidyr
```



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] 0.409192913 0.324755816 0.122622891 0.436150201 0.457855274 0.141511727
##  [7] 0.325373766 0.318207346 0.013928196 0.230353751 0.175008078 0.006365499
## [13] 0.397802182 0.111964540 0.308311638 0.458547175 0.290118958 0.013165661
## [19] 0.054805304 0.187466768 0.333443925 0.375695311 0.352637260 0.105043641
## [25] 0.412029443 0.491653150 0.357971296 0.063388968 0.054828393 0.286746749
## [31] 0.171087583 0.058495082 0.435441438 0.381039871 0.352866920 0.304080855
## [37] 0.226415675 0.155297819 0.462133820 0.201726839 0.095233805 0.434277482
## [43] 0.285727031 0.343767312 0.068805930 0.444862053 0.490096141 0.224091182
## [49] 0.436751630 0.263407111
##  [ 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;
}
```

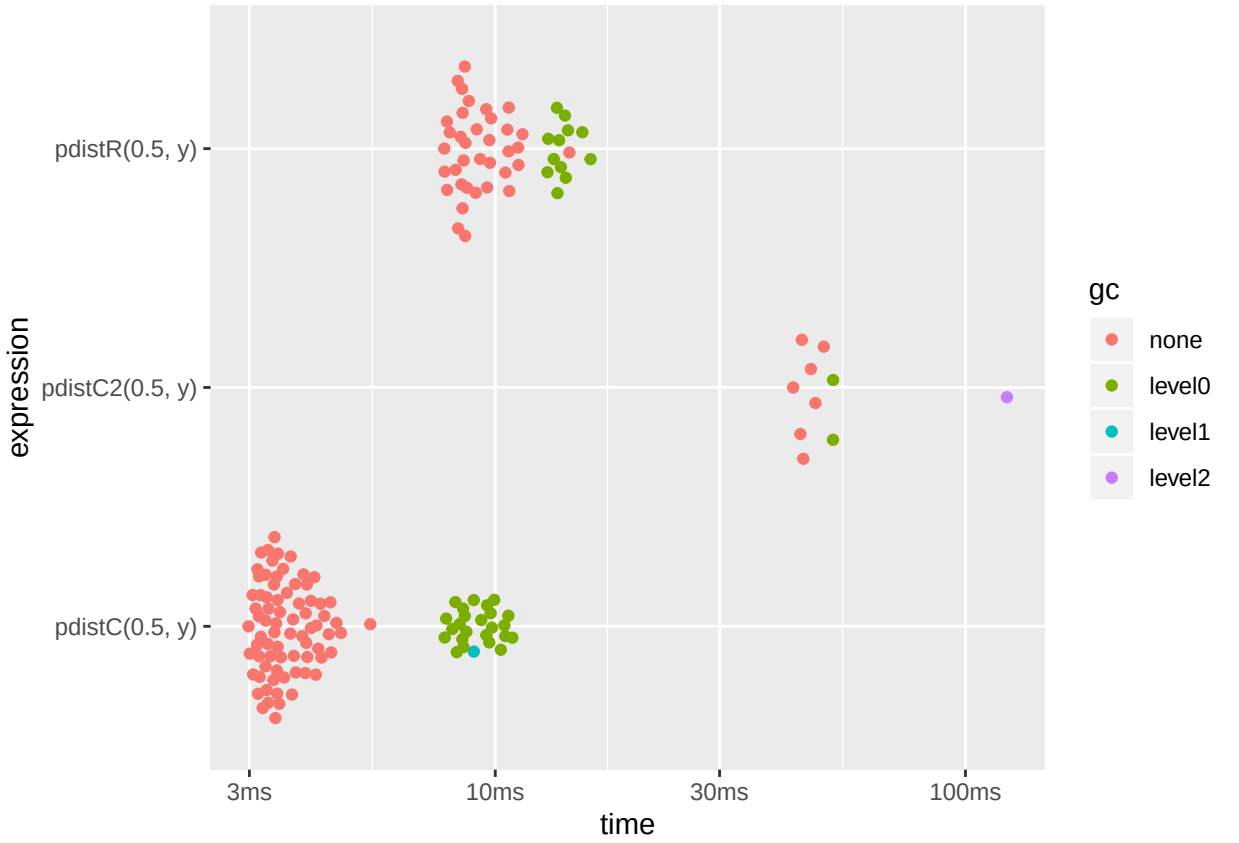
- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n);`.
- Another useful way of making a vector is to copy an existing one: `NumericVector zs = clone(ys);`.
- C++ uses `pow()`, not `^`, for exponentiation.
- C++ is faster because it avoids the creation of temperary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

Rcpp provides sugar functions, but its performance could be much worse.

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

// [[Rcpp::export]]
NumericVector pdistC2(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),
  pdistC2(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.

Reference

- Rcpp chapter of Advanced R https://adv-r.hadley.nz/rcpp.html
- Rcpp Quick Reference Guide http://dirk.eddelbuettel.com/code/rcpp/Rcpp-quickref.pdf