Integration: R and C++

Programming for Statistical Science

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- `reticulate` vignette

Getting started

Install and load the `Rcpp` package.

```
library(Rcpp)
library(tidyverse)
library(bench)
```

You'll also need a working C++ compiler. A **compiler** is a software program that converts computer programming code written by a human programmer into binary code (machine code) that can be understood and executed by a CPU.

- Windows: install Rtools
- Mac: install Xcode
- Linux: `sudo apt-get install r-base-dev`

Everything is already installed and configured on `Rook`. Use that for today if you don't have a compiler on your machine.

When to consider using C++

- Loops that can't be easily vectorized in R. Computations must be done in a sequential fashion.
- Recursive functions.
- Tasks or problems that involve advanced data structures or algorithms. C++ has efficient implementations of core algorithms through the standard template library (STL).

Writing C++ functions in R

Scalar input, scalar output

Let's write a C++ function that computes the absolute value of a scalar.

```
cppFunction(
'double abs_c(double x) {
  if (x >= 0) {
    return x;
  } else {
    return -1 * x;
  }
}')
```

```
abs_c(x = 10L)
abs_c(x = 3.1)
abs_c(x = -11)
```

```
#> [1] 10
#> [1] 3.1
#> [1] 11
```

It seems to work as expected.

What's happening?

Using your computer's compiler, `Rcpp` will compile the C++ code you pass into `cppFunction()` and construct an R function that connects to the compiled C++ function.

```
abs_c
```

```
#> function (x)
#> .Call(<pointer: 0x7f11dd0f3450>, x)
#> <bytecode: 0x56224396b9b8>
```

This is an R function that uses `.Call()` to invoke the underlying C++ function.

Breaking down `abs_c()`

```
double abs_c(double x) {
  if (x >= 0) {
    return x;
  } else {
    return -1 * x;
  }
}
```

What differences do you notice between R and C++?

- We don't use the keyword `function()`.
- We must declare the output type and input type.
- An explicit return is required. Recall that this is optional in R.
- Each statement ends with a semicolon.

Performance comparison: `abs()` v. `abs_c()`

```
bench::mark(
  abs(10),
  abs_c(10),
  abs(-10),
  abs_c(-10),
  min_time = Inf,
  iterations = 10000
)[, 1:6]
```

```
#> # A tibble: 4 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 abs(10)     88.01ns 194.06ns  3256501.        0B      0
#> 2 abs_c(10)    1.66µs   3.45µs   288444.    2.49KB      0
#> 3 abs(-10)   143.89ns 171.94ns  3461447.        0B      0
#> 4 abs_c(-10)   1.72µs   2.45µs   329081.    2.49KB     32.9
```

Vector input, vector output

Let's vectorize our C++ absolute value function.

```
cppFunction(
  'NumericVector vabs_c(NumericVector x) {
    // get length of vector x
    int n = x.size();

    for (int i = 0; i < n; ++i) {
      if (x[i] < 0) {
        x[i] *= -1;
      }
    }
    return x;
  }'
)
```

```
x <- c(-4, 1, 0, -9.1, 102)
vabs_c(x)
```

```
#> [1]   4.0   1.0   0.0   9.1 102.0
```

```
vabs_c(rnorm(n = 5, mean = -10, sd = 1))
```

```
#> [1] 10.911869 10.345215  9.733824 11.116314  9.875868
```

Breaking down `vabs_c()`

```
NumericVector vabs_c(NumericVector x) {
  // get length of vector x
  int n = x.size();

  for (int i = 0; i < n; ++i) {
    if (x[i] < 0) {
      x[i] *= -1;
    }
  }
  return x;
}
```

What differences do you notice between R and C++?

- We must declare the type of new variables defined in our function.
- The `for` loop syntax is different - initialize `i` to be 0 with `int i = 0`, loop as long as `i < n`, increment `i` by 1 after each iteration.
- Indexing in C++ begins at 0, not 1 like in R.
- The = is used for assignment, not `<-`.
- Modify in-place operators such as `*=` exist.
- C++ is an object-oriented language (`x.size()`).

Performance comparison: `abs()` v. `vabs_c()`

```
x <- rnorm(n = 1e6)
bench::mark(
  abs(x),
  vabs_c(x),
  relative = TRUE
)[, 1:6]
```

```
#> # A tibble: 2 x 6
#>   expression   min median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
#> 1 abs(x)      5.92   5.91      1        3135.      Inf
#> 2 vabs_c(x)   1      1         5.87        1       NaN
```

Why do you think `vabs_c()` is so much faster?

Objects in R, `Rcpp`, and C++

Value	R vector	Rcpp vector	Rcpp matrix	Rcpp scalar	C++ scalar
Logical	logical	LogicalVector	LogicalMatrix	-	bool
Integer	integer	IntegerVector	IntegerMatrix	-	int
Real	numeric	NumericVector	NumericMatrix	-	double
Complex	complex	ComplexVector	ComplexMatrix	Rcomplex	complex
String	character	CharacterVector(StringVector)	CharacterMatrix(StringMatrix)	String	string
Date	Date	DateVector	-	Date	-
Datetime	POSIXct	DatetimeVector	-	Datetime	time_t

The `Rcpp` package provides these classes for collecting the various objects back to R and passing R objects to the compiled C++ function.

Exercise

Write a C++ function that computes the sum of the squared deviations about a specific value. An R function to do this can be defined as follows.

```
ssd <- function(x, x0) {
  sum((x - x0) ^ 2)
}
```

Assume `x` is a vector and `x0` is a "scalar". Compare the performance between `ssd()` and your C++ function.

Hint: `pow()` in C++ is used for exponentiation.

What happens if we pass in the wrong type?

```
cppFunction(
  'double ssd_c_new(IntegerVector x, double x0) {
    int n = x.size();
    double result = 0;
    for (int i = 0; i < n; ++i) {
      result += pow(x[i] - x0, 2);
    }
    return result;
  }'
)
```

```
x <- rnorm(10)
z <- ssd_c_new(x, x0 = 0)
z
typeof(z)
ssd_c_new(x, x0 = pi)
```

```
#> [1] 2
#> [1] "double"
#> [1] 88.12967
```

Source C++ into R

Source C++

For more complicated problems you'll probably want to write a separate C++ script file for your functions. You can then compile the C++ code and make the functions available in your R environment with `sourceCpp()`.

Start your `.cpp` script file with the following.

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

Before each function include `// [[Rcpp::export]]`. This will ensure it gets exported to your R environment.

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

// [[Rcpp::export]]
```

Example script

Write a function that performs a linear search for a target value `x0`. Save this in a script file named `search.cpp`.

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

// [[Rcpp::export]]
int search(NumericVector x, double x0) {
  int n = x.size();

  for (int i = 0; i < n; ++i) {
    if (x[i] == x0) {
      return i + 1;
    }
  }
  return 0;
}
```

Export `search()` so it is available in R.

```
sourceCpp(file = "search.cpp")
```

Comparison: `search()` versus `which()` and `==`

```
x <- sample(1:10000)
search(x, x0 = 9124)
which(x == 9124)
```

```
#> [1] 9624
#> [1] 9624
```

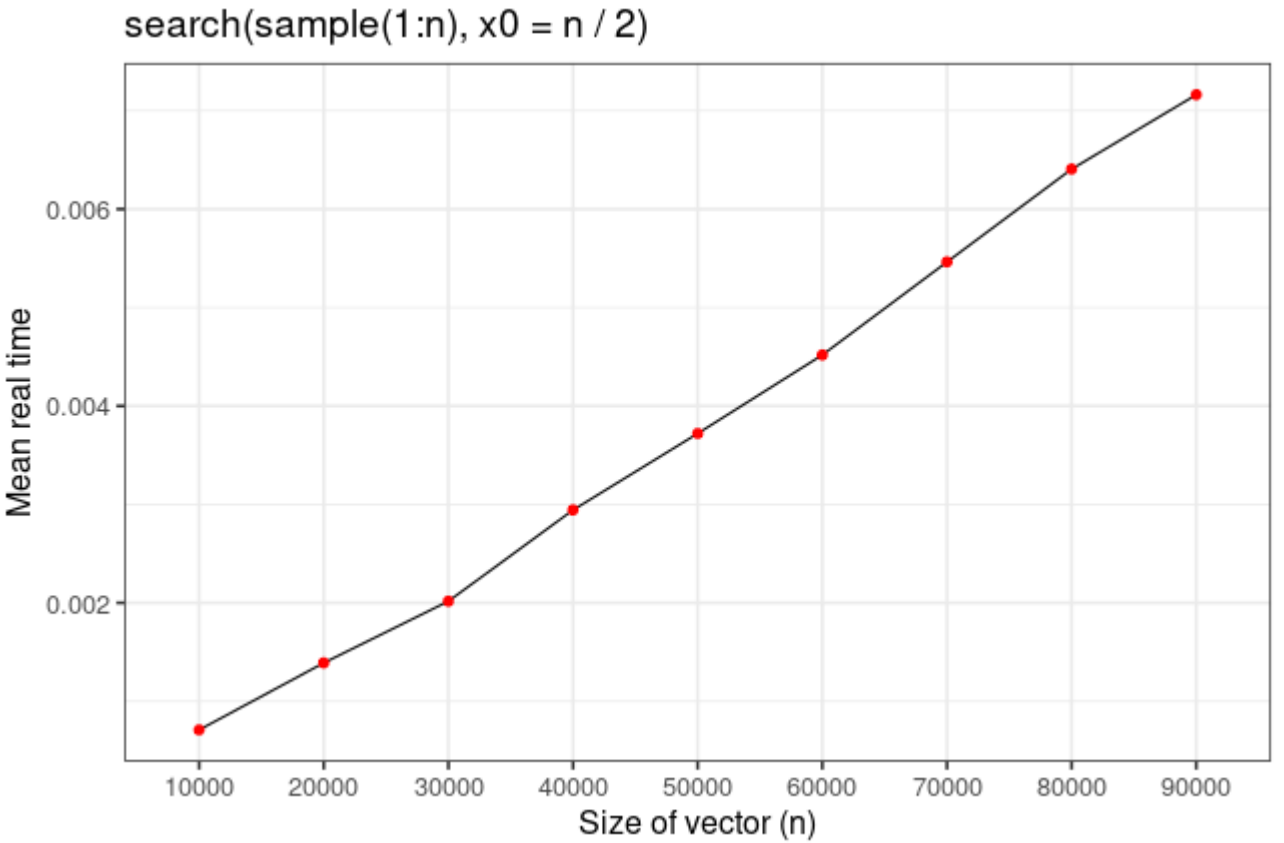
```
mark(
  search(x, x0 = 9124),
  which(x == 9124),
)[, 1:6]
```

```
#> # A tibble: 2 x 6
#>   expression                min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>           <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 search(x, x0 = 9124)   21.5µs     26µs    34420.    85.2KB     79.3
#> 2 which(x == 9124)       29.5µs   30.8µs    30932.    78.2KB     52.2
```

What is the computation complexity of our `search()` function?

```
time_search <- function(n) {
  system_time({
    search(sample(1:n), x0 = n / 2)
  })
}
```

```
result <- map(10000 * c(1:9), ~replicate(1000, time_search(.x)))
```



R code in `.cpp` files

Using special C++ comment blocks, R code can be embed in the C++ file. For example, consider the file `search2.cpp`.

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

// [[Rcpp::export]]
int search2(NumericVector x, double x0) {
  int n = x.size();

  for (int i = 0; i < n; ++i) {
    if (x[i] == x0) {
      return i + 1;
    }
  }
  return 0;
}

/*** R
x <- sample(1:10000)
search2(x, x0 = 9124)
which(x == 9124)
*/
```

```
sourceCpp("search2.cpp")
```

```
#>
#> > x <- sample(1:10000)
#>
#> > search2(x, x0 = 9124)
#> [1] 8600
#>
#> > which(x == 9124)
#> [1] 8600
```

Exercise

Convert the R `diff()` function into C++. Only consider a numeric vector as an input and the lag argument. Write this function in a `.cpp` file and source it into R.

Some examples of R's `diff()` function:

```
set.seed(24581)
(x <- sample(1:10))
```

```
#>  [1]  7  4  9 10  1  2  8  5  3  6
```

```
diff(x, lag = 1)
```

```
#> [1] -3  5  1 -9  1  6 -3 -2  3
```

```
diff(x, lag = 3)
```

```
#> [1]  3 -3 -7 -2  4  1 -2
```

```
diff(x, lag = 4)
```

```
#> [1] -6 -2 -1 -5  2  4
```

Rcpp **Sugar**

Rcpp Sugar provides a convenient way to work with C++ functions in a similar way as to how R offers vectorized operations. Some functions have identical R equivalents.

Sugar functions can be roughly broken down into:

- Arithmetic and logical operators: +, *, -, /, pow, <, <=, >, >=, ==, !=, !
- Logical summary functions: any(), all(), is true(), is false()
- Vector views: they provide a "view" of a vector, head(), tail(), and so on
- Math functions: abs(), acos(), asin(), atan(), beta(), ceil(), ceiling(), choose(), cos(), exp(), factorial(), floor() and so on
- Scalar summaries: mean(), min(), max(), sum(), sd(), and var()
- Vector summaries: cumsum(), diff(), pmin(), and pmax()
- Finding values: match(), self match(), which max(), which min()
- Dealing with duplicates: duplicated(), unique()
- Standard statistical distribution functions

Source: http://heather.cs.ucdavis.edu/~matloff/158/RcppTutorial.pdf

Unofficial Rcpp API documentation

Example with Rcpp sugar

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

// [[Rcpp::export]]
NumericMatrix bootstrap_c(NumericVector data, int b) {
  // create a b x 2 matrix
  NumericMatrix boot_stats(b, 2);
  // data sample size
  int n = data.size();

  for (int i = 0; i < b; ++i) {
    NumericVector boot_sample = sample(data, n, true);

    // matrices also start indexing at 0
    boot_stats(i, 0) = mean(boot_sample);
    boot_stats(i, 1) = sd(boot_sample);
  }

  return boot_stats;
}
```

Finishing up

- Homework 6 is due Monday, November 16 at 11:59pm ET. Don't forget to complete Task 3.
- The project is due Monday, November 23 at 11:59am ET. Not late submissions will be accepted. I'll begin grading shortly after noon that day.
- The final lab is on Monday.
- TAs will have regularly scheduled office hours for the rest of the semester.
- In addition to Monday and Friday office hours, I'll add the following for next week:
    - Tuesday, 11/17 from 9:00am - 10:00am ET
    - Thursday, 11/19 from 11:00am - 12:00pm ET
- **Please submit a course evaluation.**

References

1. Francois, D. (2020). Rcpp: Seamless R and C++ Integration. http://www.rcpp.org/.

2. Wickham, H. (2019). Advanced R. https://adv-r.hadley.nz/.

3. Tsuda, M. (2020). Rcpp for everyone. https://teuder.github.io/rcpp4everyone_en/.