Introduction to Rcpp - A Tutorial by Aditi Garg

1 Introduction

The Rcpp package provides C++ classes that greatly facilitate interfacing C or C++ code in R packages using the `.Call()` interface provided by R. It provides a powerful API on top of R, permitting direct interchange of rich R objects (including S3, S4 or Reference Class objects) between R and C++.

Sometimes R code just isn't fast enough. We've used profiling to figure out where our bottlenecks are, and we've done everything we can in R, but our code still isn't fast enough. In this tutorial we'll learn how to improve performance by rewriting key functions in C++.

Typical bottlenecks that C++ can address include:

- Loops that can't be easily vectorised because subsequent iterations depend on previous ones.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than that in R.
- Problems that require advanced data structures and algorithms that R doesn't provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double-ended queues.

2 Installation

Install the latest version of Rcpp from CRAN with `install.packages("Rcpp")`.

A working C++ compiler is also needed. To get it:

- On Windows, install Rtools.
- On Mac, install Xcode from the app store.
- On Linux, `sudo apt-get install r-base-dev` or similar.

3 Usage

3.1 Exporting C++ Functions

We'll verify that Rcpp is installed and working correctly with a "Hello, World!" program [1]. Create a blank file called `hello.cpp` and enter the code:

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 // [[Rcpp::export]]
5 void hello()
6 {
7 Rprintf("Hello, World!\n");
8 }
```

Line 1 includes `Rcpp.h` which contains the definitions used by the Rcpp package. The comment on Line 4 is an Rcpp attribute. Attributes are annotations that are added to C++ source files to indicate that C++ functions should be made available as R functions. Finally, the routine `Rprintf`, which is part of the R API, prints to the R console. The syntax is the same as `printf`.

Now you can go over to the R prompt and type in the following to test your code:

```
1 library(Rcpp)
2 sourceCpp("hello.cpp")
3 hello()
```

The `sourceCpp` function parses a C++ file and looks for functions marked with the `Rcpp::export` attribute. A shared library is then built and its exported functions are made available as R functions in the specified environment.

Congratulations! We can now write C++ functions using built-in C++ types and Rcpp wrapper types and then source them just as we would an R script.

3.2 Including C++ Inline

Maintaining C++ code in it's own source file provides several benefits including the ability to use C++ aware text-editing tools and straightforward mapping of compilation errors to lines in the source file. However, it's also possible to do inline declaration and execution of C++ code.

There are several ways to accomplish this, including passing a code string to `sourceCpp` or using the shorter-form `cppFunction` or `evalCpp` functions. Go to the R prompt and execute:

```
1 library(Rcpp)
2 cppFunction('int add(int x, int y, int z) {
3   int sum = x + y + z;
4   return sum;
5 }')
6 # add works like a regular R function
```

```
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x7f2f4aa933d0>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

When we run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function.

4 The Rcpp Interface

4.1 Data Structures

In the example shown in 3.2 the function returns an int (a scalar integer). Scalars and vectors are different. In R we need vectors for maximum functionality to be enabled. For that reason, Rcpp's functionality is provided through a set of C++ classes that wrap R data structures. A few of them are:

- Basic vector classes: NumericVector, IntegerVector, CharacterVector, and LogicalVector.
- Scalar equivalents: int, double, bool, String.
- Matrix Equivalents: IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix.
- List, DataFrame.

There are also classes for many more specialised language objects: Environment, ComplexVector, RawVector, DottedPair, Language, Promise, Symbol, WeakReference, and so on.

Memory management is handled automatically by the class constructors and destructors.

4.2 Rcpp Sugar

Rcpp provides a lot of syntactic "sugar" to ensure that C++ functions work very similarly to their R equivalents. In fact, Rcpp sugar makes it possible to write efficient C++ code that looks almost identical to its R equivalent.

If there's a sugar version of the function you're interested in, you should use it: it'll be both expressive and well tested.

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 (for vectors) `var()`.

- Vector summaries: `cumsum()`, `diff()`, `pmin()`, and `pmax()`.
- Finding values: `match()`, `self_match()`, `which_max()`, `which_min()`.
- Dealing with duplicates: `duplicated()`, `unique()`.
- `dnorm`, `qnorm`... for all standard distributions.

Finally, `noNA(x)` asserts that the vector x does not contain any missing values, and allows optimisation of some mathematical operations.

5 Examples

5.1 Row Maximums

Suppose we want to compute the maximum element of each row in a matrix. To achieve this, we loop over each row of the matrix and use the sugar routine `max`:

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

// [[Rcpp::export]]
NumericVector row_max(NumericMatrix m) {
int nrow = m.nrow();
NumericVector max(nrow);

for (int i = 0; i < nrow; i++)
        // Get row i with m(i, _).
        max[i] = Rcpp::max( m(i, _) );

return max;
}
```

Notice that the matrix classes in Rcpp use parentheses `( )` as the subset operator rather than square brackets `[ ]`. This is due to limitations in C++.

5.2 Element Sum

One big difference between R and C++ is that the cost of loops is much lower in C++. For example, we could implement the sum function in R using a loop. If you've been programming in R a while, you'll probably have a visceral reaction to this function!

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

In C++, loops have very little overhead, so it's fine to use them.

```
#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;
}
```

This is a good example of where C++ is much more efficient than R.

```
library(microbenchmark)

x <- runif(1e3)
microbenchmark(
  sum(x),
  sumC(x),
  sumR(x)
)
#> Unit: microseconds
#>     expr    min     lq   mean median     uq      max neval
#>   sum(x)   1.96   2.52   3.08   2.89   3.12     9.57   100
#>  sumC(x)   4.73   5.77   7.39   6.48   7.81    29.10   100
#>  sumR(x) 590.00 635.00 743.96 665.00 777.00 2,820.00   100
```

As shown by the following microbenchmark, `sumC()` is competitive with the built-in (and highly optimised) `sum()`, while `sumR()` is several orders of magnitude slower.

5.3 Mean

We will implement `mean` in C++ and then compares it to the built-in `mean()` and the purpose of this example is to also demonstrate how you can embed R code in special C++ comment blocks. This is really convenient if you want to run some test code::

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

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

```

```
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}

/*** R
library(microbenchmark)
x <- runif(1e5)
microbenchmark(
  mean(x),
  meanC(x)
)
*/
```

The microbenchmark code will automatically execute in the R prompt when this file is loaded using a call to `sourceCpp` in the prompt. On running this you'll notice that `meanC()` is much faster than the built-in `mean()`. This is because it trades numerical accuracy for speed.

5.4 Fibonacci

The standard definition of the Fibonacci sequence is,

$$Fn = Fn - 1 + Fn - 2 \tag{1}$$

With initial values F0 = 0 and F1 = 1.

This leads this intuitive (but slow) R implementation:

```
fibR <- function(n) {
if (n == 0) return(0)
if (n == 1) return(1)
return (fibR(n - 1) + fibR(n - 2))
}
```

We can write an easy (and very fast) C++ version, that has a six-hundred fold increase for no real effort or setup cost:

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

// [[Rcpp::export]]
int fibonacci(const int x) {
if (x == 0)
        return(0);
if (x == 1)
        return(1);
return (fibonacci(x - 1)) + fibonacci(x - 2);
}
```

6 Resources

For a definitive reference on Rcpp, see Dirk Eddelbuettel's book, Seamless R and C++ Integration with Rcpp [1].

You can also go on to his website and look at the official documentation along with demos and examples.

References

[1] Nick Ulle, *Getting Started with Rcpp*, http://heather.cs.ucdavis.edu/Rcpp.pdf

[2] Eddelbuettel, Dirk, et al. "Rcpp: Seamless R and C++ integration." Journal of Statistical Software 40.8 (2011): 1-18.