MFE Programming Workshop

Week 7

Brett Dunn and Mahyar Kargar

Fall 2017

Rcpp Interface and RStudio

Why would you want to use another language?

- R is an interpreted language.
- Low programmer time.
- A great tool for data munging, statistics, regressions, etc.
- However, certain tasks in R can be slow.
- You can greatly improve performance by rewriting key functions in another language, such as C++.
- A good workflow:
  1. Write your program in R.
  2. If the program is too slow, benchmark your code.
  3. Try to speedup any bottlenecks in R.
  4. Convert any remaining bottlenecks to C++.

Bottlenecks in R

Typical bottlenecks that C++ can address include:

- Loops that can't be easily vectorized because subsequent iterations depend on previous ones. Particularly nested loops.
- 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.

R Foreign Languange Interfaces

- R provides several built-in functions to interface to other languages:
  - `.C()`, for C and C++ code,
  - `.Call()`, also for C and C++ code,
  - `.Fortran()` for Fortran code,
  - and many more.
- Packages exist to call Java, Python, Julia, and other languages.
- Resources:
  - Writing R Extensions
  - Advanced R by Hadley Wickham
- These interfaces are time-consuming to learn, but the `Rcpp` package provides a much simpler interface.

Rcpp

- The Rcpp package is a fantastic tool written by Dirk Eddelbuettel and Romain François (with key contributions by Doug Bates, John Chambers, and J.J. Allaire).
- Rcpp makes it very simple to connect C++ to R.
- Rcpp makes it easy to pass R objects like vectors, matrices, and lists to C++ and back to R.
  - However, there is overhead in doing this.
  - If you are concerned about speed, consider using the simplest structure.

Installation of Rcpp

- Rcpp is a CRAN package, but it requires an additional tool chain of a compiler, linker and more in order to be able to create binary object code extending R.
- On Linux all required components are likely to be present. For macOS, you will need to install the "command line tools" from the Xcode developer package (free) and you may want to install a version of gfortran.
  - See Rcpp FAQ sections 2.10 and 2.16 if you encounter problems with macOS.
- On Windows, users need to install Rtools.
- After you have the tool chain, you can install the Rcpp package:

```
install.packages('Rcpp')
```

First Steps

- Three key functions:
  1. `evalCpp()`
  2. `sourceCpp()`
  3. `cppFunction()`

Basic Usage: `evalCpp()`

- `evalCpp()` evaluates a single C++ expression. Includes and dependencies can be declared.
- This allows us to quickly check C++ constructs.

```
library(Rcpp)
evalCpp("exp(2)") # simple test
```

```
## [1] 7.389056
```

```
evalCpp("std::numeric_limits<double>::max()")
```

```
## [1] 1.797693e+308
```

Basic Usage: cppFunction()

- cppFunction() creates, compiles and links a C++ file, and creates an R function to access it.

```
cppFunction("
  int simpleExample() {
    int x = 10;
    return 2.0*x;
}")
simpleExample() # same identifier as C++ function
```

```
## [1] 20
```

Basic Usage: cppFunction() with C++11

- C++11 is the ISO C++ standard ratified in 2011, and it provides many new features over C++98 or C++03.
- To use C++11 code in R, we add `plugins=c("cpp11")`.
- May need to run this first:
  `Sys.setenv("PKG_CXXFLAGS"="-std=c++11")`
- See First steps in using C++11 with Rcpp.

```
Sys.setenv("PKG_CXXFLAGS"="-std=c++11")
cppFunction("
  int exampleCpp11() {
    auto x = 10;
    return 2.0*x;
}",plugins=c("cpp11"))

exampleCpp11() # same identifier as C++ function
```

```
## [1] 20
```

Basic Usage: `sourceCpp()`

- `sourceCpp()` is the actual workhorse behind `evalCpp()` and `cppFunction()`.
- It is described in more detail in the package vignette Rcpp Attributes.
- `sourceCpp()` builds on and extends `cxxfunction()` from package inline, but provides even more ease-of-use, control and helpers—freeing us from boilerplate scaffolding.
- A key feature are the plugins and dependency options: other packages can provide a plugin to supply require compile-time parameters (`RcppArmadillo`, `RcppEigen`, `RcppGSL`)

Basic Usage: RStudio

- In R Studio, File / New File / C++ File, and this file is created:

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

// This is a simple example of exporting a C++ function to R. You can
// source this function into an R session using the Rcpp::sourceCpp
// function (or via the Source button on the editor toolbar)...

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}

// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.
//

/*** R
timesTwo(42)
*/
```

Basic Usage: RStudio (Cont'ed)

In R, we ask `Rcpp` to source the function:

```
sourceCpp("./examples/timesTwo.cpp")
```

```
##
## > timesTwo(42)
## [1] 84
```

Now the function is available to use in R:

```
timesTwo(c(12,24))
```

```
## [1] 24 48
```

Basic Usage: RStudio (Cont'ed)

So what just happened?

- We defined a simple C++ function.
- It operates on a numeric vector argument.
- We asked `Rcpp` to 'source it' for us.
- Behind the scenes `Rcpp` creates a wrapper.
- `Rcpp` then compiles, links, and loads the wrapper.
- The function is available in R under its C++ name `timesTwo`.

Rcpp Data Types

Rcpp defines several C++ data types that correspond with R data types:

- `IntegerVector` for vectors of type integer.
- `NumericVector` for vectors of type numeric.
- `LogicalVector` for vectors of type logical.
- `CharacterVector` for vectors of type character.
- `List` for `lists`.
- `DataFrame` for `data.frames`.
- `Function` for functions.
- `Environment` for environments.

Example of Rcpp::DataFrame

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

// [[Rcpp::export]]
DataFrame timesTwoDF(NumericVector x) {
  // create a new vector y the same size as x
  NumericVector y(x.size());
  // loop through, double x, assign to y
  for(int i = 0; i < x.size(); i++) {
    y[i] = x[i]*2.0;
  }
  // return a data.frame
  return DataFrame::create(
    Named("x")=x,
    Named("y")=y);
}
```

- **Note**: IN C++, VECTOR INDICES START AT **0**! This is a very common source of bugs.

Example of DataFrame (Cont'ed)

Now, we can source the code in R and use the function:

```
sourceCpp("./examples/timesTwoDF.cpp")
x <- 1:5
df <- timesTwoDF(x)
str(df)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ x: num  1 2 3 4 5
##  $ y: num  2 4 6 8 10
```

Random Number Generation

Rcpp simplifies using R's random number generators in C++.

```
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector rcppNormals(int n) {
    return rnorm(n);
}
```

In R:

```
sourceCpp("./examples/rcppNormals.cpp")
set.seed(1234)
rcppNormals(5)
```

```
## [1] -1.2070657  0.2774292  1.0844412 -2.3456977  0.4291247
```

Benchmark with built-in rnorm

```
library(microbenchmark)
n <- 1e5
microbenchmark(rnorm(n),rcppNormals(n))
```

```
## Unit: milliseconds
##             expr      min       lq     mean   median       uq      ma
##         rnorm(n) 6.307224 7.265441 7.826056 7.316797 8.014512 17.8408
##  rcppNormals(n) 4.324387 4.989832 5.709363 5.008140 5.510521 15.3593
##  neval
##    100
##    100
```

More Rcpp Examples

Example: Vector input, scalar output

- One big difference between R and C++ is that the cost of loops is much lower in C++.
- Let's compare

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

Use cppFunction()

- cppFunction() allows you to write C++ functions in R.
- When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function.

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

C++ vs. R Syntax

- The C++ version is similar, but:
  - To find the length of the vector, we use the `.size()` method, which returns an integer. C++ methods are called with `.` (i.e., a full stop).
  - The `for` statement has a different syntax: `for(init; check; increment)`. This loop is initialised by creating a new variable called `i` with value 0. Before each iteration we check that `i < n`, and terminate the loop if it's not. After each iteration, we increment the value of `i` by one, using the special prefix operator `++` which increases the value of `i` by 1.
  - Use `=` for assignment, not `<-`.
  - C++ provides operators that modify in-place: `total += x[i]` is equivalent to `total = total + x[i]`. Similar in-place operators are `-=`, `*=`, and `/=`.

Bechmarking: In C++ loops are much faster than R

- In this example C++ is much more efficient than R:
  - `sumC()` is competitive with the highly optimized built-in `sum()`,
  - `sumR()` is several orders of magnitude slower.

```
library(microbenchmark)
x <- runif(2000)
microbenchmark(sum(x), sumC(x), sumR(x))
```

```
## Unit: microseconds
##      expr    min     lq       mean median     uq       max neval
##    sum(x)  1.551  1.862    1.93968  1.862  1.863     8.068   100
##   sumC(x)  2.482  2.793   12.02773  2.793  3.103   861.403   100
##   sumR(x) 94.953 95.263 130.26247 95.264 95.884 3518.529   100
```

Using `sourceCpp`

- So far, we've used inline C++ with `cppFunction()`.
- For real problems, it's usually easier to use stand-alone C++ files and then source them into R using `sourceCpp()`.
- Your stand-alone C++ file should have extension .cpp, and needs to start with:

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

- And for each function that you want available within R, you need to prefix it with:

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

- In RStudio File/New File/C++ File does these steps for you.

Compile the C++ code

- To compile the C++ code, use `sourceCpp("path/to/file.cpp")`.
- This will create the matching R functions and add them to your current session.
- Note that these functions can not be saved in a `.Rdata` file and reloaded in a later session; they must be recreated each time you restart R.
- You can embed R code in special C++ comment blocks.

```
/*** R
# This is R code
*/
```

Example: meanC vs. the built-in mean():

```
#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))
*/
```

Resources

- Main Rcpp page: `Rcpp` for Seamless R and C++ Integration.
- `Rcpp` book by Dirk Eddelbuettel. Can download it for free from SpringerLink on the UCLA network.
- Advanced R by Hadley Wickham: Rcpp and R's C interface chapetrs.
- `Rcpp` Gallery: Articles and code examples for the `Rcpp` package.
- Writing R Extensions from CRAN.
- Extending R by John M. Chambers.