Rcpp examples

Paolo Bosetti

Rcpp: wrapping C/C++ functions

Thanks to the `Rcpp` library it is particularly easy to interface R with C/C++ code. This is called *wrapping*, i.e. making a C++ function available to R as if it were an R function. This is particularly useful when performance is an issue, as C++ code is generally faster than R code.

The `Rcpp` library is designed to make it easy to interface with C++ functions and classes. Being a valid subset of C++, C can be used as well, although if you use external libraries compiled in C, you ought to remember to declare all functions as `extern "C"` in the C headers.

Resources

I am linking here useful documentation and resources for `Rcpp`:

- Rcpp for everyone
- Rcpp in Hadley Wickham's Advanced R
- Rcpp website
- Rcpp Gallery
- Rcpp cheat sheet
- RcppEigen intro

Examples

Hello, World!

C++ code can be embedded in R code as strings or read from source files:

```
code <- r"(
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
void hello_world() {
  Rcout << "Hello, World!" << std::endl;
}
)"

sourceCpp(code=code)
hello_world()
```

```
Hello, World!
```

Note the special comment `// [[Rcpp::export]]` that tells the `sourceCpp` function to make the function available to R. *Without that comment, the function is compiled but not exposed to R.*

Deal with vectors

Vectors, lists and matrices can be passed to and returned from C++ functions:

```
code <- r"(
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector add_one(NumericVector x) {
  return x + 1;
}

// [[Rcpp::export]]
NumericVector scale(NumericVector x, double factor) {
  return x * factor;
}

// [[Rcpp::export]]
NumericVector add(NumericVector x, NumericVector y) {
  return x + y;
```

```
}

// [[Rcpp::export]]
NumericVector operate(NumericVector x, NumericVector y) {
  if (x.size() != y.size()) {
    stop("Vectors must have the same length");
  }
  NumericVector res(x.size());
  for (int i = 0, j = x.size()-1; i < x.size(); i++, j--) {
    res[i] = x[i] + y[j];
  }
  return res;
}

// [[Rcpp::export]]
NumericMatrix outer_product(NumericVector x, NumericVector y) {
  NumericMatrix res(x.size(), y.size());
  for (int i = 0; i < x.size(); i++) {
    for (int j = 0; j < y.size(); j++) {
      res(i, j) = x[i] * y[j];
    }
  }
  return res;
}

// [[Rcpp::export]]
DataFrame data_frame(NumericVector x, NumericVector y) {
  if (x.size() != y.size()) {
    stop("Vectors must have the same length");
  }
  return DataFrame::create(
    _["x"] = x,
    _["y"] = y
  );
}

)"

sourceCpp(code=code)
add_one(1:10)
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

```
scale(1:10, 2)
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
add(1:10, 11:20)
```

```
 [1] 12 14 16 18 20 22 24 26 28 30
```

```
operate(1:10, 11:20)
```

```
 [1] 21 21 21 21 21 21 21 21 21 21
```

```
outer_product(1:3, 1:4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    4    6    8
[3,]    3    6    9   12
```

```
data_frame(1:3, (1:3)^2)
```

```
  x y
1 1 1
2 2 4
3 3 9
```

Look in Rcpp for everyone for the documentation of the Rcpp data classes and their methods.

Functionals

R functions can be passed to and executed by C++ code, enabling functional programming:

```
code <- r"(
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector apply(NumericVector x, Function f) {
  NumericVector res(x.size());
  for (int i = 0; i < x.size(); i++) {
    res[i] = as<double>(f(x[i]));
  }
  return res;
}
)"

sourceCpp(code=code)
apply(1:10, \(x){ x + 1 })
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

If the passed function takes named arguments, it can be called in C++ as `f(_["x"] = 10, _["y"] = 20)`. Look in Hadley Wickham's *Advanced R* for more information.

RcppEigen

Eigen (or Armadillo) can be used to perform linear algebra operations:

```
code <- r"(
#include <RcppEigen.h>
using namespace Rcpp;
using namespace Eigen;

// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::export]]
NumericVector eigen_example(NumericMatrix x) {
  Map<MatrixXd> m(as<Map<MatrixXd> >(x));
  SelfAdjointEigenSolver<MatrixXd> es(m);
  return wrap(es.eigenvalues());
}
)"
```

```
sourceCpp(code=code)
eigen_example(matrix(1:4, 2, 2))
```

```
[1] 0 5
```

Wrapping Classes

C++ classes can be wrapped as *modules* and used in R as S4 objects. In this case, there is no need to mark the functions with `// [[Rcpp::export]]`, you rather define a class mapping with the `RCPP_MODULE` macro:

```
code <- r"(
#include <Rcpp.h>
using namespace Rcpp;

class Counter {
public:
  Counter() : count(0) {}
  void increment() { count++; }
  int get() { return count; }
private:
  int count;
};

RCPP_MODULE(counter_module) {
  class_<Counter>("Counter")
  .constructor()
  .method("increment", &Counter::increment)
  .method("get", &Counter::get);
}

)"

sourceCpp(code=code)
counter <- new(Counter)
counter$increment()
counter$increment()
counter$get()
```

```
[1] 2
```