Rcpp and RInside: Easier R and C++ integration

UseR! 2009 Presentation

Dirk Eddelbuettel, Ph.D.
Dirk.Eddelbuettel@R-Project.org
edd@debian.org

Université Rennes II, Agrocampus Ouest
Laboratoire de Mathématiques Appliquées
8-10 July 2009

Overview
Agenda for today

- Brief discussion of how R is extended with compiled code
  - .C versus .Call interfaces
- Rcpp: Interface classes
  - Easier interfacing
  - Automatic type matching and dimension settings
  - Provides “object view” similar to R
- RInside: Embedding R in C++
  - Access to R analysis directly from C++ applications
  - Re-uses Rcpp C++ interface classes

R and Compiled Code

Or how do we combine the two?

Once we exhausted all other options (*e.g.* using vectorised expression and/or just-in-time compilation and/ or optimised libraries), the most direct speed gain for R programs comes from switching to compiled code.

We briefly go over two approaches:

- using the `.C` interface
- using the `.Call` interface

before introducing the `Rcpp` classes.

A different approach is to keep the core logic 'outside' but to *embed* R into the application—which we discuss later.

Compiled Code: A brief introduction

Gory details are provided in the *R Extensions* manual

R offers several functions to access compiled code: `.C` and `.Fortran` as well as `.Call` and `.External`. (*R Extensions*, sections 5.2 and 5.9; *Software for Data Analysis*). `.C` and `.Fortran` are older and simpler, but more restrictive in the long run.

The canonical example is the convolution function:

```
1  void convolve(double *a, int *na, double *b, int *nb, double *ab)
2  {
3    int i, j, nab = *na + *nb - 1;
4
5    for(i = 0; i < nab; i++)
6      ab[i] = 0.0;
7    for(i = 0; i < *na; i++)
8      for(j = 0; j < *nb; j++)
9        ab[i + j] += a[i] * b[j];
10 }
```

Compiled Code: The Basics

Continuing . . .

The convolution function is called from R by

```
conv <- function(a, b)
  .C("convolve",
     as.double(a),
     as.integer(length(a)),
     as.double(b),
     as.integer(length(b)),
     ab = double(length(a) + length(b) - 1))$ab
```

One must take care to coerce all the arguments to the correct R storage mode before calling .C. Mistakes in type matching can lead to wrong results or hard-to-catch errors.

Compiled Code: The Basics

Continuing . . .

Using the alternative `.Call` interface, the example becomes

```
1  #include <R.h>
2  #include <Rdefines.h>
3
4  SEXP convolve2(SEXP a, SEXP b)
5  {
6    int i, j, na, nb, nab;
7    double *xa, *xb, *xab;
8    SEXP ab;
9
10   PROTECT(a = AS_NUMERIC(a));
11   PROTECT(b = AS_NUMERIC(b));
12   na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
13   PROTECT(ab = NEW_NUMERIC(nab));
14   xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
15   xab = NUMERIC_POINTER(ab);
16   for(i = 0; i < nab; i++) xab[i] = 0.0;
17   for(i = 0; i < na; i++)
18     for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
19   UNPROTECT(3);
20   return(ab);
21 }
```

Compiled Code: The Basics

Continuing . . .

Now the call becomes easier by just using the function name and the vector arguments—all other handling is done at the C/C++ level:

```
conv <- function(a, b) .Call("convolve2", a, b)
```

In summary, we see that

- there are different entry points
- using different calling conventions
- leading to code that may need to do more work at the lower level.

Compiled Code: Rcpp

Overview

`Rcpp` has one goal: making it easier to interface C++ and R code.

Using the `.Call` interface, we can use features of the C++ language to automate the tedious bits of the macro-based C-level interface to R.

One major advantage of using `.Call` is that vectors (or matrices) can be passed directly between R and C++ without the need for explicit passing of dimension arguments. And by using the C++ class layers, we do not need to directly manipulate the SEXP objects.

Rcpp example

An illustration

We can consider the 'distribution of determinant' example of Venables and Ripley (and re-used by Milborrow to motivate Ra). The simplest version can be written as follows:

```
#include <Rcpp.hpp>

RcppExport SEXP dd_rcpp(SEXP v) {
  SEXP rl = R_NilValue;              // Use this when nothing is returned

  RcppVector<int> vec(v);            // vec parameter viewed as vector of doubles
  int n = vec.size(), i = 0;

  for (int a = 0; a < 9; a++)
    for (int b = 0; b < 9; b++)
      for (int c = 0; c < 9; c++)
        for (int d = 0; d < 9; d++)
          vec(i++) = a*b - c*d;

  RcppResultSet rs;                  // Build result set returned as list to R
  rs.add("vec", vec);                // vec as named element with name 'vec'
  rl = rs.getReturnList();           // Get the list to be returned to R.

  return rl;
}
```

but it is actually preferable to use the exception-handling feature of C++ as in the slightly longer next version.

Rcpp example
Continuing

```
1  #include <Rcpp.hpp>
2
3  RcppExport SEXP dd_rcpp(SEXP v) {
4    SEXP   rl = R_NilValue;         // Use this when there is nothing to be returned.
5    char* exceptionMesg = NULL;     // msg var in case of error
6
7    try {
8      RcppVector<int> vec(v);       // vec parameter viewed as vector of doubles.
9      int n = vec.size(), i = 0;
10     for (int a = 0; a < 9; a++)
11       for (int b = 0; b < 9; b++)
12         for (int c = 0; c < 9; c++)
13           for (int d = 0; d < 9; d++)
14             vec(i++) = a*b - c*d;
15
16     RcppResultSet rs;             // Build result set to be returned as a list to R.
17     rs.add("vec", vec);           // vec as named element with name 'vec'
18     rl = rs.getReturnList();      // Get the list to be returned to R.
19   } catch(std::exception& ex) {
20     exceptionMesg = copyMessageToR(ex.what());
21   } catch(...) {
22     exceptionMesg = copyMessageToR("unknown reason");
23   }
24
25   if (exceptionMesg != NULL)
26     error(exceptionMesg);
27
28   return rl;
29 }
```

Rcpp example
Continuing

We can create a shared library from the source file as follows:

```
PKG_CPPFLAGS=`r -e'Rcpp:::CxxFlags()'` \
     R CMD SHLIB dd.rcpp.cpp \
     `r -e'Rcpp:::LdFlags()'`

g++ -I/usr/share/R/include \
     -I/usr/lib/R/site-library/Rcpp/lib \
     -fpic -g -O2 \
     -c dd.rcpp.cpp -o dd.rcpp.o
g++ -shared -o dd.rcpp.so dd.rcpp.o \
     -L/usr/lib/R/site-library/Rcpp/lib \
     -lRcpp -Wl,-rpath,/usr/lib/R/site-library/Rcpp/lib \
     -L/usr/lib/R/lib -lR
```

Note how we let the `Rcpp` package tell us where header and library files are stored.

Rcpp example
Continuing

We can then load the file using `dyn.load` and proceed as in the `inline` example.

```
dyn.load("dd.rcpp.so")

dd.rcpp <- function() {
    x <- integer(10000)
    res <- .Call("dd_rcpp", x)
    tabulate(res$vec)
}

mean(replicate(100,system.time(dd.rcpp())["elapsed"]))
[1] 0.00047
```

Basic Rcpp usage

Scalar variables

`Rcpp` eases data transfer from R to C++, and back. We always convert to and from `SEXP`, and return a `SEXP` to R.

The key is that we can use this as a 'variant' type permitting us to extract data via appropriate C++ classes. We pass data from R via named lists that may contain different types:

```
list(intnb=42, fltnb=6.78, date=Sys.Date(),
     txt="some thing", bool=FALSE)
```

by initialising a `RcppParams` object and extracting as in

```
RcppParams param(inputsexp);
int      nmb = param.getIntValue("intnb");
double   dbl = param.getDoubleValue("fltnb");
string   txt = param.getStringValue("txt");
bool     flg = param.getBoolValue("bool";
RcppDate dt  = param.getDateValue("date");
```

Basic Rcpp usage

Vector variables

Similarly, we can constructs vectors and matrics of `double`, `int`, as well as vectors of types `string` and date and datetime. The key is that we *never* have to deal with dimensions and / or memory allocations — all this is shielded by C++ classes.

Standard C++ STL vectors can be created with an additional cast.

There is support for creating `data.frame` objects in C++: they are returned as lists, `as.data.frame` has to be performed at the R level.

To return values to R, we declare an object of type `RcppResultSet` and use the `add` methods to insert named elements before converting this into a list that is assigned to the returned `SEXP`.

Back in R, we access them as elements of a standard R list by position or name.

Faster linear models

Let us consider a comparison between `lm()` and `lm.fit()`. How fast could compiled code be? Let's wrap a GNU GSL function.

```
1  #include <cstdio>
2  extern "C" {
3  #include <gsl/gsl_multifit.h>
4  }
5  #include <Rcpp.h>
6
7  RcppExport SEXP gsl_multifit(SEXP Xsexp, SEXP Ysexp) {
8      SEXP rl=R_NilValue;
9      char *exceptionMesg=NULL;
10
11     try {
12         RcppMatrixView<double> Xr(Xsexp);
13         RcppVectorView<double> Yr(Ysexp);
14
15         int i,j,n = Xr.dim1(), k = Xr.dim2();
16         double chisq;
17
18         gsl_matrix *X = gsl_matrix_alloc (n, k);
19         gsl_vector *y = gsl_vector_alloc (n);
20         gsl_vector *c = gsl_vector_alloc (k);
21         gsl_matrix *cov = gsl_matrix_alloc (k, k);
22         for (i = 0; i < n; i++) {
23             for (j = 0; j < k; j++)
24                 gsl_matrix_set (X, i, j, Xr(i,j));
25             gsl_vector_set (y, i, Yr(i));
26         }
```

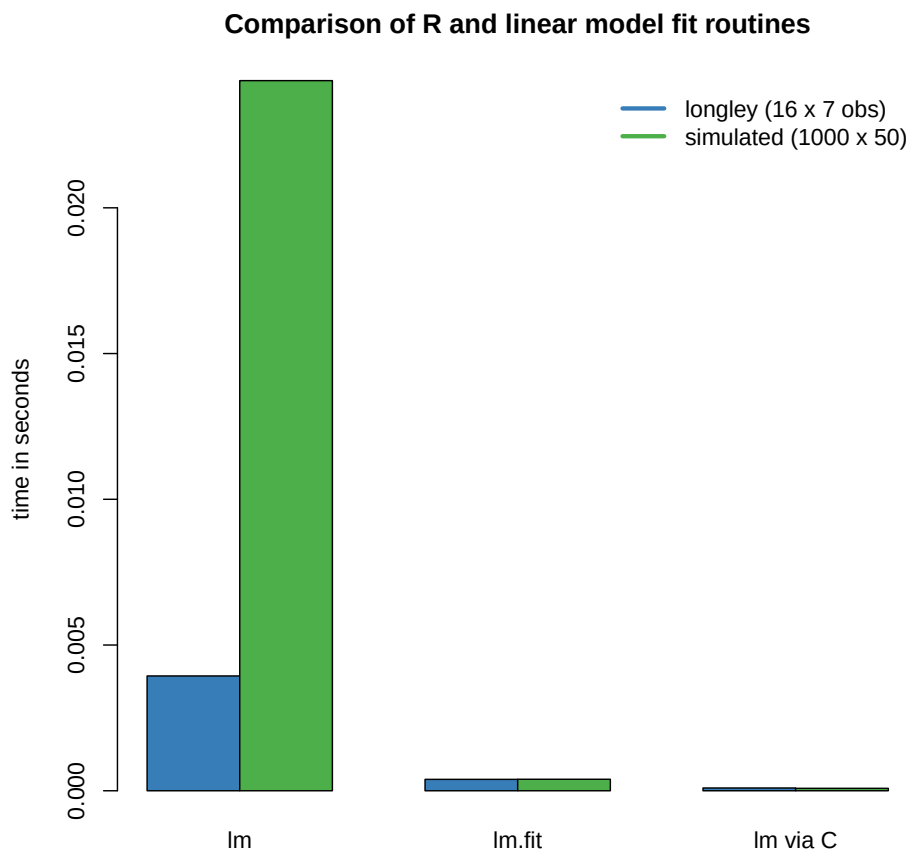
Another Rcpp example

Continuing

```
27          gsl_multifit_linear_workspace *work = gsl_multifit_linear_alloc (n, k);
28          gsl_multifit_linear (X, y, c, cov, &chisq, work);
29          gsl_multifit_linear_free (work);
30
31          RcppMatrix<double> CovMat(k, k);
32          RcppVector<double> Coef(k);
33          for (i = 0; i < k; i++) {
34              for (j = 0; j < k; j++)
35                  CovMat(i,j) = gsl_matrix_get(cov,i,j);
36              Coef(i) = gsl_vector_get(c,i);
37          }
38          gsl_matrix_free (X);
39          gsl_vector_free (y);
40          gsl_vector_free (c);
41          gsl_matrix_free (cov);
42
43          RcppResultSet rs;
44          rs.add("coef", Coef);
45          rs.add("covmat", CovMat);
46
47          rl = rs.getReturnList();
48
49      } catch(std::exception& ex) {
50          exceptionMesg = copyMessageToR(ex.what());
51      } catch(...) {
52          exceptionMesg = copyMessageToR("unknown reason");
53      }
54      if(exceptionMesg != NULL)
55          Rf_error(exceptionMesg);
56      return rl;
57  }
```

Another Rcpp example

Continuing



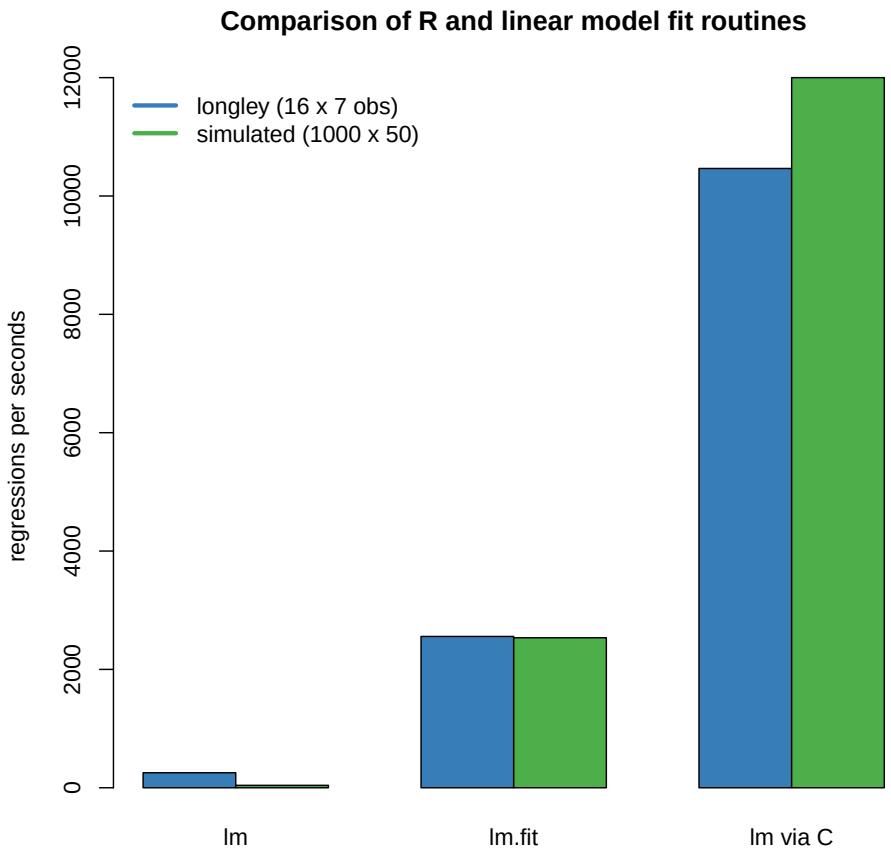
Source: Our calculations

The small `longley` example exhibits less variability between methods, but the larger data set shows the gains more clearly.

The `lm.fit()` approach appears unchanged between `longley` and the larger simulated data set.

Another Rcpp example

Continuing



Source: Our calculations

By inverting the times to see how many 'regressions per second' we can fit, the merits of the compiled code become clearer.

One caveat, measurements depends critically on the size of the data as well as the cpu and libraries that are used.

Rcpp and package building

Locating headers and libraries

Two tips for easing builds with `Rcpp`:

- For command-line use, a shortcut is to copy `Rcpp.h` to `/usr/local/include`, and `libRcpp.so` to `/usr/local/lib`. The earlier example reduces to

  ```
  R CMD SHLIB dd.rcpp.cpp
  ```

  as header and library will be found in the default locations.

- For package building, we can have a file `src/Makevars` with

  ```
  # compile flag providing header directory
  PKG_CXXFLAGS=`Rscript -e 'Rcpp:::CxxFlags()'`
  # link flag providing libary and path
  PKG_LIBS=`Rscript -e 'Rcpp:::LdFlags()'`
  ```

See `help(Rcpp-package)` for more details.

RInside and bringing R to C++

Making embedded R easier since 2008

Sometimes we may want to go the other way and add R to and existing C++ project.

This can be simplified using `RInside`. Consider this *Hello World* example:

```
1  #include "RInside.h"                    // for the embedded R via RInside
2  #include "Rcpp.h"                       // for the R / Cpp interface
3
4  int main(int argc, char *argv[]) {
5
6      RInside R(argc, argv);              // create an embedded R instance
7
8      std::string txt = "Hello, world!\n";// assign a standard C++ string to 'txt'
9      R.assign( txt, "txt");              // assign string var to R variable 'txt'
10
11     std::string evalstr = "cat(txt)";
12     R.parseEvalQ(evalstr);              // eval the init string, ignoring any returns
13
14     exit(0);
15 }
```

RInside and bringing R to C++

Discussion the first example

This stylized example showed that

- we instantiate an `RInside` object, and we are passing the command-line parameters along which permits R specific initializations
- we use the `assign` member function to send C++ variables to named variables in the R interpreter
- we pass R code for evaluation as a string
- we use the `parseEvalQ` member function to *quietly* parse and evaluate the R expression in the string; the `parseEval` function is similar but returns a value.

The next example shows some more computation.

RInside and bringing R to C++

Another example

```
#include "RInside.h"                    // for the embedded R via RInside
#include "Rcpp.h"                       // for the R / Cpp interface used for transfer

std::vector< std::vector< double > > createMatrix(const int n) {
    std::vector< std::vector< double > > mat;
    for (int i=0; i<n; i++) {
        std::vector<double> row;
        for (int j=0; j<n; j++) row.push_back((i*10+j));
        mat.push_back(row);
    }
    return(mat);
}

int main(int argc, char *argv[]) {
    const int mdim = 4;
    std::string evalstr = "cat('Running ls()\n'); print(ls()); \
        cat('Showing M\n'); print(M); cat('Showing colSums()\n'); \
        Z <- colSums(M); print(Z); Z";  ## returns Z
    RInside R(argc, argv);
    SEXP ans;
    std::vector< std::vector< double > > myMatrix = createMatrix(mdim);

    R.assign( myMatrix, "M");               // assign STL matrix to R's 'M' var
    R.parseEval(evalstr, ans);              // eval the init string -- Z is now in ans
    RcppVector<double> vec(ans);            // now vec contains Z via ans
    vector<double> v = vec.stlVector();     // convert RcppVector to STL vector

    for (unsigned int i=0; i< v.size(); i++)
        std::cout << "In C++ element " << i << " is " << v[i] << std::endl;
    exit(0);
}
```

Summary

What did we cover?

We have seen that

- Rcpp allows easy data transfer from R to C++ and back
- Rcpp uses C++ features for an 'object-view' that should be a natural match for an R user and programmer. Transfer to and from STL containers is supported.
- A few other niceties are available: lighter-weight 'view-only' classes, full support for millisecond-resolution `Datetime` objects at the C++ level and more.
- RInside permits us to 'take R inside' existing or new applications

Neither package has its interface set in stone.

Contributions are welcome and even encouraged.