

# RcppGSL: Easier GSL use from R via Rcpp

Dirk Eddelbuettel

Romain François

Version 0.2.5 as of July 6, 2015

## Abstract

The GNU Scientific Library, or **GSL**, is a collection of numerical routines for scientific computing (Galassi et al., 2010). It is particularly useful for C and C++ programs as it provides a standard C interface to a wide range of mathematical routines such as special functions, permutations, combinations, fast fourier transforms, eigensystems, random numbers, quadrature, random distributions, quasi-random sequences, Monte Carlo integration, N-tuples, differential equations, simulated annealing, numerical differentiation, interpolation, series acceleration, Chebyshev approximations, root-finding, discrete Hankel transforms physical constants, basis splines and wavelets. There are over 1000 functions in total with an extensive test suite.

The **RcppGSL** package provides an easy-to-use interface between **GSL** and R, with a particular focus on matrix and vector data structures. **RcppGSL** relies on **Rcpp** (Eddelbuettel and François, 2011; Eddelbuettel, François, Allaire, Chambers, Bates, and Ushey, 2015; Eddelbuettel, 2013) which is itself a package that eases the interfaces between R and C++.

## 1 Introduction

The GNU Scientific Library, or **GSL**, is a collection of numerical routines for scientific computing (Galassi et al., 2010). It is a rigorously developed and tested library providing support for a wide range of scientific or numerical tasks. Among the topics covered in the GSL are complex numbers, roots of polynomials, special functions, vector and matrix data structures, permutations, combinations, sorting, BLAS support, linear algebra, fast fourier transforms, eigensystems, random numbers, quadrature, random distributions, quasi-random sequences, Monte Carlo integration, N-tuples, differential equations, simulated annealing, numerical differentiation, interpolation, series acceleration, Chebyshev approximations, root-finding, discrete Hankel transforms least-squares fitting, minimization, physical constants, basis splines and wavelets.

Support for C programming with the GSL is readily available: the GSL itself is written in C and provides a C-language Application Programming Interface (API). Access from C++ is therefore possible, albeit not at the abstraction level that can be offered by dedicated C++ implementations.<sup>1</sup>

The GSL is somewhat unique among numerical libraries. Its combination of broad coverage of scientific topics, serious implementation effort and the use of a FLOSS license have lead to a fairly wide usage of the library. As a concrete example, we can consider the the CRAN repository network for the R language and environment (R Development Core Team, 2013). CRAN contains over a dozen packages interfacing the GSL: **copula**, **dynamo**, **gsl**, **gstat**, **magnets**, **mvabund**, **QRMLib**, **RBrownie**, **RDieHarder**, **RHmm**, **segclust**, **surveillance**, and **topicmodels**. This is a clear indication that the GSL is popular among programmers using either the C or C++ language for solving problems applied science.

At the same time, the **Rcpp** package (Eddelbuettel and François, 2011; Eddelbuettel, 2013; Eddelbuettel et al., 2015) offers a higher-level abstraction between R and underlying C++ (or C) code. **Rcpp** permits R objects like vectors, matrices, lists, functions, environments, ..., to be manipulated directly at the C++

---

<sup>1</sup>Several C++ wrappers for the GSL have been written over the years yet none reached a state of completion comparable to the GSL itself. Three such wrapping library are <http://cholm.home.cern.ch/cholm/misc/gslmm/>, <http://gslwrap.sourceforge.net/> and <http://code.google.com/p/gslcpp/>.

level, alleviates the needs for complicated and error-prone parameter passing and memory allocation. It also permits compact vectorised expressions similar to what can be written in R directly at the C++ level.

The **RcppGSL** package discussed here aims to help close the gap. It tries to offer access to GSL functions, in particular via the vector and matrix data structures used throughout the GSL, while staying closer to the ‘whole object model’ familiar to the R programmer.

The rest of paper is organised as follows. The next section shows a motivating example of a fast linear model fit routine using **GSL** functions. The following section discusses support for **GSL** vector types, which is followed by a section on matrices.

## 2 Motivation: fastLm

Fitting linear models is a key building block of analysing data and modeling. R has a very complete and feature-rich function in `lm()` which can provide a model fit as well as a number of diagnostic measure, either directly or via the `summary()` method for linear model fits. The `lm.fit()` function provides a faster alternative (which is however recommend only for for advanced users) which provides estimates only and fewer statistics for inference. This may lead to user requests for a routine which is both fast and featureful enough. The `fastLm` routine shown here provides such an implementation. It uses the **GSL** for the least-squares fitting functions and provides a nice example for **GSL** integration with R.

```
#include <RcppGSL.h>

#include <gsl/gsl_multifit.h>
#include <cmath>

// [[Rcpp::export]]
Rcpp::List fastLm(Rcpp::NumericMatrix Xs, Rcpp::NumericVector ys) {
  RcppGSL::vector<double> y = Rcpp::wrap(ys);           // create gsl data structures
  RcppGSL::matrix<double> X = Rcpp::wrap(Xs);

  int n = X.nrow(), k = X.ncol();
  double chisq;
  RcppGSL::vector<double> coef(k);                      // to hold the coefficient vector
  RcppGSL::matrix<double> cov(k,k);                    // and the covariance matrix

  // the actual fit requires working memory we allocate and free
  gsl_multifit_linear_workspace *work = gsl_multifit_linear_alloc (n, k);
  gsl_multifit_linear (X, y, coef, cov, &chisq, work);
  gsl_multifit_linear_free (work);

  // assign diagonal to a vector, then take square roots to get std.error
  Rcpp::NumericVector std_err;
  std_err = gsl_matrix_diagonal(cov);                   // need two step decl. and assignment
  std_err = sqrt(std_err);                             // sqrt() is an Rcpp sugar function

  Rcpp::List res = Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                                       Rcpp::Named("stderr")      = std_err,
                                       Rcpp::Named("df.residual")   = n - k);

  // free all the GSL vectors and matrices -- as these are really C data structures
  // we cannot take advantage of automatic memory management
  coef.free(); cov.free(); y.free(); X.free();

  return res;     // return the result list to R
}
```

We first initialize a *RcppGSL* vector and matrix, each templated to the standard numeric type `double` (and the GSL supports other types ranging from lower precision floating point to signed and unsigned integers as well as complex numbers). We then reserve another vector and matrix to hold the resulting coefficient estimates as well as the estimate of the covariance matrix. Next, we allocate workspace using a GSL routine, fit the linear model and free the workspace. The next step involves extracting the diagonal element from the covariance matrix. We then employ a so-called iterator—a common C++ idiom from the Standard Template Library (STL)—to iterate over the vector of diagonal and transforming it by applying the square root function to compute our standard error of the estimate. Finally we create a named list with the return value before we free temporary memory allocation (a step that has to be done because the underlying objects are really C objects conforming to the **GSL** interface and hence without the automatic memory management we could have with C++ vector or matrix structures as used through the **Rcpp** package) and return the result to R.

We note that **RcppArmadillo** (François, Eddelbuettel, and Bates, 2014; Eddelbuettel and Sanderson, 2014) implements a matching `fastLm` function using the Armadillo library by Sanderson (2010), and can do so with more compact code due to C++ features. Moreover, **RcppEigen** (Bates, François, and Eddelbuettel, 2014; Bates and Eddelbuettel, 2013) provides a `fastLm` implementation with a comprehensive comparison of matrix decomposition methods.

### 3 Vectors

This section details the different vector representations, starting with their definition inside the **GSL**. We then discuss our layering before showing how the two types map. A discussion of read-only ‘vector view’ classes concludes the section.

#### 3.1 GSL Vectors

**GSL** defines various vector types to manipulate one-dimensional data, similar to R arrays. For example the `gsl_vector` and `gsl_vector_int` structs are defined as:

```
typedef struct{
    size_t size;
    size_t stride;
    double * data;
    gsl_block * block;
    int owner;
} gsl_vector;

typedef struct {
    size_t size;
    size_t stride;
    int * data;
    gsl_block_int * block;
    int owner;
} gsl_vector_int;
```

A typical use of the `gsl_vector` struct is given below:

```

int i;
gsl_vector *v = gsl_vector_alloc(3);    // allocate a gsl_vector of size 3

for (i = 0; i < 3; i++) {                // fill the vector
    gsl_vector_set(v, i, 1.23 + i);
}

double sum = 0.0 ;                       // access elements
for (i = 0; i < 3; i++) {
    sum += gsl_vector_set(v, i);
}

gsl_vector_free(v);                      // free the memory

```

### 3.2 RcppGSL::vector

RcppGSL defines the template `RcppGSL::vector<T>` to manipulate `gsl_vector` pointers taking advantage of C++ templates. Using this template type, the previous example now becomes:

```

int i;
RcppGSL::vector<double> v(3);            // allocate a gsl_vector of size 3

for (i = 0; i < 3; i++) {                // fill the vector
    v[i] = 1.23 + i;
}

double sum = 0.0 ;                       // access elements
for (i = 0; i < 3; i++) {
    sum += v[i];
}

v.free();                                // free the memory

```

The class `RcppGSL::vector<double>` is a smart pointer, that can be used anywhere where a raw pointer `gsl_vector` can be used, such as the `gsl_vector_set` and `gsl_vector_get` functions above.

Beyond the convenience of a nicer syntax for allocation and release of memory, the `RcppGSL::vector` template facilitates interchange of **GSL** vectors with **Rcpp** objects, and hence **R** objects. The following example defines a .Call compatible function called `sum_gsl_vector_int` that operates on a `gsl_vector_int` through the `RcppGSL::vector<int>` template specialization:

```

// [[Rcpp::export]]
int sum_gsl_vector_int( RcppGSL::vector<int> vec){
    int res = std::accumulate(vec.begin(), vec.end(), 0);
    vec.free();    // we need to free vec after use
    return res;
}

```

Once the function has been created via `sourceCpp()` or `cppFunction()` from *Rcpp Attributes* (see section 8 for more on this), it can then be called from R :

```

> sum_gsl_vector_int(1:10)
[1] 55

```

A second example shows a simple function that grabs elements of an R list as `gsl_vector` objects using implicit conversion mechanisms of **Rcpp**

```

// [[Rcpp::export]]
double gsl_vector_sum_2(Rcpp::List data) {
  // grab "x" as a gsl_vector through the RcppGSL::vector<double> class
  RcppGSL::vector<double> x = data["x"] ;

  // grab "y" as a gsl_vector through the RcppGSL::vector<int> class
  RcppGSL::vector<int> y = data["y"] ;
  double res = 0.0 ;
  for (size_t i=0; i< x->size; i++) {
    res += x[i] * y[i] ;
  }

  // as usual with GSL, we need to explicitly free the memory
  x.free() ;
  y.free() ;

  // return the result
  return res ;
}

```

called from R:

```

> data <- list( x = seq(0,1,length=10), y = 1:10 )
> gsl_vector_sum_2(data)
[1] 36.66667

```

### 3.3 Mapping

Table 1 shows the mapping between types defined by the **GSL** and their corresponding types in the **RcppGSL** package.

gsl vector	RcppGSL
<code>gsl_vector</code>	<code>RcppGSL::vector&lt;double&gt;</code>
<code>gsl_vector_int</code>	<code>RcppGSL::vector&lt;int&gt;</code>
<code>gsl_vector_float</code>	<code>RcppGSL::vector&lt;float&gt;</code>
<code>gsl_vector_long</code>	<code>RcppGSL::vector&lt;long&gt;</code>
<code>gsl_vector_char</code>	<code>RcppGSL::vector&lt;char&gt;</code>
<code>gsl_vector_complex</code>	<code>RcppGSL::vector&lt;gsl_complex&gt;</code>
<code>gsl_vector_complex_float</code>	<code>RcppGSL::vector&lt;gsl_complex_float&gt;</code>
<code>gsl_vector_complex_long_double</code>	<code>RcppGSL::vector&lt;gsl_complex_long_double&gt;</code>
<code>gsl_vector_long_double</code>	<code>RcppGSL::vector&lt;long double&gt;</code>
<code>gsl_vector_short</code>	<code>RcppGSL::vector&lt;short&gt;</code>
<code>gsl_vector_uchar</code>	<code>RcppGSL::vector&lt;unsigned char&gt;</code>
<code>gsl_vector_uint</code>	<code>RcppGSL::vector&lt;unsigned int&gt;</code>
<code>gsl_vector_ushort</code>	<code>RcppGSL::vector&lt;insigned short&gt;</code>
<code>gsl_vector_ulong</code>	<code>RcppGSL::vector&lt;unsigned long&gt;</code>

Table 1: Correspondance between **GSL** vector types and templates defined in **RcppGSL**.

### 3.4 Vector Views

Several **GSL** algorithms return **GSL** vector views as their result type. **RcppGSL** defines the template class `RcppGSL::vector_view` to handle vector views using C++ syntax.

```

// [[Rcpp::export]]
List test_gsl_vector_view() {
  int n = 10 ;
  RcppGSL::vector<double> v(n) ;
  for (int i=0 ; i<n; i++) {
    v[i] = i ;
  }
  RcppGSL::vector_view<double> v_even =
    gsl_vector_subvector_with_stride(v,0,2,n/2);
  RcppGSL::vector_view<double> v_odd =
    gsl_vector_subvector_with_stride(v,1,2,n/2);

  List res = List::create(Named("even") = v_even,
                          Named("odd" ) = v_odd);
  v.free() ;    // we only need to free v, the views do not own data
  return res ;
}

```

As with vectors, C++ objects of type `RcppGSL::vector_view` can be converted implicitly to their associated **GSL** view type. Table 2 displays the pairwise correspondance so that the C++ objects can be passed to compatible **GSL** algorithms.

gsl vector views	RcppGSL
<code>gsl_vector_view</code>	<code>RcppGSL::vector_view&lt;double&gt;</code>
<code>gsl_vector_view_int</code>	<code>RcppGSL::vector_view&lt;int&gt;</code>
<code>gsl_vector_view_float</code>	<code>RcppGSL::vector_view&lt;float&gt;</code>
<code>gsl_vector_view_long</code>	<code>RcppGSL::vector_view&lt;long&gt;</code>
<code>gsl_vector_view_char</code>	<code>RcppGSL::vector_view&lt;char&gt;</code>
<code>gsl_vector_view_complex</code>	<code>RcppGSL::vector_view&lt;gsl_complex&gt;</code>
<code>gsl_vector_view_complex_float</code>	<code>RcppGSL::vector_view&lt;gsl_complex_float&gt;</code>
<code>gsl_vector_view_complex_long_double</code>	<code>RcppGSL::vector_view&lt;gsl_complex_long_double&gt;</code>
<code>gsl_vector_view_long_double</code>	<code>RcppGSL::vector_view&lt;long double&gt;</code>
<code>gsl_vector_view_short</code>	<code>RcppGSL::vector_view&lt;short&gt;</code>
<code>gsl_vector_view_uchar</code>	<code>RcppGSL::vector_view&lt;unsigned char&gt;</code>
<code>gsl_vector_view_uint</code>	<code>RcppGSL::vector_view&lt;unsigned int&gt;</code>
<code>gsl_vector_view_ushort</code>	<code>RcppGSL::vector_view&lt;insigned short&gt;</code>
<code>gsl_vector_view_ulong</code>	<code>RcppGSL::vector_view&lt;unsigned long&gt;</code>

Table 2: Correspondance between **GSL** vector view types and templates defined in **RcppGSL**.

The vector view class also contains a conversion operator to automatically transform the data of the view object to a **GSL** vector object. This enables use of vector views where **GSL** would expect a vector.

## 4 Matrices

The **GSL** also defines a set of matrix data types : `gsl_matrix`, `gsl_matrix_int` etc ... for which **RcppGSL** defines a corresponding convenience C++ wrapper generated by the `RcppGSL::matrix` template.

### 4.1 Creating matrices

The `RcppGSL::matrix` template exposes three constructors.

```

// convert an R matrix to a GSL matrix
matrix(SEXP x) throw(::Rcpp::not_compatible)

// encapsulate a GSL matrix pointer
matrix(gsl_matrix* x)

// create a new matrix with the given number of rows and columns
matrix(int nrow, int ncol)

```

## 4.2 Implicit conversion

RcppGSL::matrix defines implicit conversion to a pointer to the associated **GSL** matrix type, as well as dereferencing operators, making the class RcppGSL::matrix look and feel like a pointer to a **GSL** matrix type.

```

gsltype* data;
operator gsltype*() { return data; }
gsltype* operator->() const { return data; }
gsltype& operator*() const { return *data; }

```

## 4.3 Indexing

Indexing of **GSL** matrices is usually the task of the functions `gsl_matrix_get`, `gsl_matrix_int_get`, ... and `gsl_matrix_set`, `gsl_matrix_int_set`, ...

RcppGSL takes advantage of both operator overloading and templates to make indexing a **GSL** matrix much more convenient.

```

RcppGSL::matrix<int> mat(10,10);           // create a matrix of size 10x10

for (int i=0; i<10; i++) {                 // fill the diagonal
    mat(i,i) = i;                          // no need for setter function
}

```

## 4.4 Methods

The RcppGSL::matrix type also defines the following member functions:

```

nrow  extracts the number of rows
ncol  extract the number of columns
size  extracts the number of elements
free  releases the memory

```

## 4.5 Matrix views

Similar to the vector views discussed above, the RcppGSL also provides an implicit conversion operator which returns the underlying matrix stored in the matrix view class.

## 5 Error handler

When input values for **GSL** functions are invalid, the default error handler will abort the program after printing an error message, which will lead R to an abortion error. To avoid this, we need to close it first by using `gsl_set_error_handler_off()` and then detect whether the result is NAN or not.

```

// close the GSL error handler
gsl_set_error_handler_off();

// feed GSL function with some invalid values
double res = gsl_sf_hyperg_2F1(1, 1, 1.1467003, 1);

// detect the result is NAN or not
if (ISNAN(res)) {
    Rcpp::Rcout << "Invalid input found!" << std::endl;
}

```

See <http://thread.gmane.org/gmane.comp.lang.r.rcpp/7905> for a longer discussion of the related issues.

Starting with release 0.2.4, two new functions are available: `gslSetErrorHandlerOff()` and `gslResetErrorHandler()` which allow to turn off the error handler (as discussed above), and to reset to the prior (default) value.

In addition, the package now also calls `gslSetErrorHandlerOff()` when being attached, ensuring that the GSL error handler is turned off by default.

## 6 Using RcppGSL in your package

The **RcppGSL** package contains a complete example providing a single function `colNorm` which computes a norm for each column of a supplied matrix. This example adapts a matrix example from the GSL manual that has been chosen primarily as a means to showing how to set up a package to use **RcppGSL**.

Needless to say, we could compute such a matrix norm easily in R using existing facilities. One such possibility is a simple `apply(M, 2, function(x) sqrt(sum(x^2)))` as shown on the corresponding help page in the example package inside **RcppGSL**. One point in favour of using the **GSL** code is that it employs a BLAS function so on sufficiently large matrices, and with suitable BLAS libraries installed, this variant could be faster due to the optimised code in high-performance BLAS libraries and/or the inherent parallelism a multi-core BLAS variant which compute compute the vector norm in parallel. On all 'reasonable' matrix sizes, however, the performance difference should be negligible.

### 6.1 The configure script

#### 6.1.1 Using autoconf

Using **RcppGSL** means employing both the **GSL** and R. We may need to find the location of the **GSL** headers and library, and this done easily from a `configure` source script which `autoconf` generates from a `configure.in` source file such as the following:



```

AC_INIT([RcppGSLExample], 0.1.0)

## Use gsl-config to find arguments for compiler and linker flags
##
## Check for non-standard programs: gsl-config(1)
AC_PATH_PROG([GSL_CONFIG], [gsl-config])
## If gsl-config was found, let's use it
if test "${GSL_CONFIG}" != ""; then
    # Use gsl-config for header and linker arguments (without BLAS which we get from R)
    GSL_CFLAGS='${GSL_CONFIG} --cflags'
    GSL_LIBS='${GSL_CONFIG} --libs-without-cblas'
else
    AC_MSG_ERROR([gsl-config not found, is GSL installed?])
fi

# Now substitute these variables in src/Makevars.in to create src/Makevars
AC_SUBST(GSL_CFLAGS)
AC_SUBST(GSL_LIBS)

AC_OUTPUT(src/Makevars)

```

Such a source `configure.in` gets converted into a script `configure` by invoking the `autoconf` program.

We note that many other libraries use a similar (but somewhat newer and by-now fairly standard) scripting frontend called `pkg-config` which be deployed in a very similar by other packages. Calls such as the following two can be used from `configure` in a very similar manner:

```

pkg-config --cflags libpng
pkg-config --libs libpng

```

where `libpng` (for the png image format) is used just for illustration.

### 6.1.2 Using functions provided by RcppGSL

**RcppGSL** provides R functions (in the file `R/inline.R` that allow us to retrieve the same information. Therefore the `configure` script can also be written as:

```

#!/bin/sh

GSL_CFLAGS='${R_HOME}/bin/Rscript -e "RcppGSL::CFlags()"'
GSL_LIBS='${R_HOME}/bin/Rscript -e "RcppGSL::LdFlags()"'

sed -e "s|@GSL_LIBS@|${GSL_LIBS}|" \
    -e "s|@GSL_CFLAGS@|${GSL_CFLAGS}|" \
    src/Makevars.in > src/Makevars

```

Similarly, the `configure.win` for windows can be written as:

```

GSL_CFLAGS='${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe -e "RcppGSL::CFlags()"'
GSL_LIBS='${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe -e "RcppGSL::LdFlags()"'

sed -e "s|@GSL_LIBS@|${GSL_LIBS}|" \
    -e "s|@GSL_CFLAGS@|${GSL_CFLAGS}|" \
    src/Makevars.in > src/Makevars.win

```

## 6.2 The src directory

The C++ source file takes the matrix supplied from R and applies the **GSL** function to each column.

```
#include <RcppGSL.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

// [[Rcpp::export]]
Rcpp::NumericVector colNorm(Rcpp::NumericMatrix M) {
  // this conversion involves an allocation
  RcppGSL::matrix<double> G = Rcpp::as< RcppGSL::matrix<double> >(M);
  int k = G.ncol();
  Rcpp::NumericVector n(k);           // to store results
  for (int j = 0; j < k; j++) {
    RcppGSL::vector_view<double> colview = gsl_matrix_column (G, j);
    n[j] = gsl_blas_dnorm2(colview);
  }
  G.free();
  return n;                          // return vector
}
```

The Makevars.in file governs the compilation and uses the values supplied by configure during build-time:

```
# set by configure
GSL_CFLAGS = @GSL_CFLAGS@
GSL_LIBS   = @GSL_LIBS@

# combine with standard arguments for R
PKG_CPPFLAGS = $(GSL_CFLAGS)
PKG_LIBS = $(GSL_LIBS)
```

The variables surrounded by `will` be filled by configure during package build-time. As discussed above, this can either rely on `autoconf` or a possibly-simpler `Rscript`.

## 6.3 The R directory

The R source is very simply: it contains a single file created by the `Rcpp::compileAttributes()` function implementing the wrapper to the `colNorm()` function.

## 7 Using RcppGSL with inline

The **inline** package (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2013) is very helpful for prototyping code in C, C++ or Fortran as it takes care of code compilation, linking and dynamic loading directly from R. It is being used extensively by **Rcpp**, for example in the numerous unit tests.

The example below shows how **inline** can be deployed with **RcppGSL**. We implement the same column norm example, but this time as an R script which is compiled, linked and loaded on-the-fly. Compared to standard use of **inline**, we have to make sure to add a short section declaring which header files from **GSL** we need to use; the **RcppGSL** then communicates with **inline** to tell it about the location and names of libraries used to build code against **GSL**.

```

> require(inline)
> inctxt='
  #include <gsl/gsl_matrix.h>
  #include <gsl/gsl_blas.h>
  ,
> bodytxt='
  RcppGSL::matrix<double> M = sM;      // create gsl data structures from
SEXP
  int k = M.ncol();
  Rcpp::NumericVector n(k);            // to store results

  for (int j = 0; j < k; j++) {
    RcppGSL::vector_view<double> colview = gsl_matrix_column (M, j);
    n[j] = gsl_blas_dnorm2(colview);
  }
  M.free() ;
  return n;                            // return vector
  ,
> foo <- cxxfunction(
+   signature(sM="numeric"),
+   body=bodytxt, inc=inctxt, plugin="RcppGSL")
> M <- outer(sin(0:9), rep(1,10), "*") + outer(rep(1, 10), cos(0:9), "*")
> foo(M)
[1] 4.314614 3.120504 2.193159 3.261141 2.534157 2.572810
[7] 4.204689 3.652017 2.085236 3.073134

```

The RcppGSL inline plugin supports creation of a package skeleton based on the inline function.

```

> package.skeleton("mypackage", foo)

```

## 8 Using RcppGSL with Rcpp Attributes

*Rcpp Attributes* (Allaire, Eddelbuettel, and François, 2013) builds on the features of the **inline** package described in previous section, and streamlines the compilation, loading and linking process even further. It leverages the existing plugins for **inline**. We already showed the corresponding function in the previous section. Here, we show it again as a self-contained example used via `sourceCpp()`. We stress that usage of `sourceCpp()` is meant for interactive work at the R command-prompt, but not the recommended practice in a package.

```

#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

#include <RcppGSL.h>

// declare a dependency on the RcppGSL package; also activates plugin
//
// [[Rcpp::depends(RcppGSL)]]

// declare the function to be 'exported' to R
//
// [[Rcpp::export]]
Rcpp::NumericVector foo(Rcpp::NumericMatrix sM) {
  RcppGSL::matrix<double> M = Rcpp::as<RcppGSL::matrix<double>>(sM);
  int k = M.ncol();
  Rcpp::NumericVector n(k);           // to store results

  for (int j = 0; j < k; j++) {
    RcppGSL::vector_view<double> colview = gsl_matrix_column (M, j);
    n[j] = gsl_blas_dnrm2(colview);
  }
  M.free();
  return n;                           // return vector
}

/** R
## see Section 8.4.13 of the GSL manual:
## create M as a sum of two outer products
M <- outer(sin(0:9), rep(1,10), "*") +
      outer(rep(1, 10), cos(0:9), "*")
foo(M)
*/

```

With the code above stored in a file, say, “gslNorm.cpp” one can simply call `sourceCpp()` to have the wrapper code added, and all of the compilation, linking and loading done — including the execution of the short R segment at the end:

```
> sourceCpp("gslNorm.cpp")
```

The function `cppFunction()` is also available to convert a simple character string argument containing a valid C++ function into a eponymous R function. And like `sourceCpp()`, it can also use plugins. See the vignette “Rcpp-attributes” (Allaire et al., 2013) of the **Rcpp** package (Eddelbuettel et al., 2015) for full details.

## 9 Summary

The GNU Scientific Library (GSL) by Galassi et al. (2010) offers a very comprehensive collection of rigorously developed and tested functions for applied scientific computing under a common Open Source license. This has lead to widespread deployment of **GSL** among a number of disciplines.

Using the automatic wrapping and converters offered by the **RcppGSL** package presented here, R users and programmers can now deploy algorithms provided by the **GSL** with greater ease.

## References

- J. J. Allaire, Dirk Eddelbuettel, and Romain François. *Rcpp Attributes*, 2013. URL <http://CRAN.R-Project.org/package=Rcpp>. Vignette included in R package Rcpp.
- Douglas Bates and Dirk Eddelbuettel. Fast and elegant numerical linear algebra using the RcppEigen package. *Journal of Statistical Software*, 52(5):1–24, 2013. URL <http://www.jstatsoft.org/v52/i05/>.
- Douglas Bates, Romain François, and Dirk Eddelbuettel. *RcppEigen: Rcpp integration for the Eigen templated linear algebra library*, 2014. URL <http://CRAN.R-Project.org/package=RcppEigen>. R package version 0.3.2.3.0.
- Dirk Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Use R! Springer, New York, 2013. ISBN 978-1-4614-6867-7.
- Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL <http://www.jstatsoft.org/v40/i08/>.
- Dirk Eddelbuettel and Conrad Sanderson. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics and Data Analysis*, 71:1054–1063, March 2014. doi: 10.1016/j.csda.2013.02.005. URL <http://dx.doi.org/10.1016/j.csda.2013.02.005>.
- Dirk Eddelbuettel, Romain François, JJ Allaire, John Chambers, Douglas Bates, and Kevin Ushey. *Rcpp: Seamless R and C++ Integration*, 2015. URL <http://CRAN.R-Project.org/package=Rcpp>. R package version 0.11.4.
- Romain François, Dirk Eddelbuettel, and Douglas Bates. *RcppArmadillo: Rcpp integration for Armadillo templated linear algebra library*, 2014. URL <http://CRAN.R-Project.org/package=RcppArmadillo>. R package version 0.4.600.0.
- Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, and Fabrice Rossi. *GNU Scientific Library Reference Manual*, 3rd edition, 2010. URL <http://www.gnu.org/software/gsl>. Version 1.14. ISBN 0954612078.
- R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL <http://www.R-project.org/>.
- Conrad Sanderson. Armadillo: An open source C++ algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010. URL <http://arma.sf.net>.
- Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, and Romain François. *inline: Inline C, C++, Fortran function calls from R*, 2013. URL <http://CRAN.R-Project.org/package=inline>. R package version 0.3.13.