RcppGSL: Easier GSL use from R via Rcpp

Dirk Eddelbuettel[a] **and Romain François**[b]

[a]https://dirk.eddelbuettel.com; [b]https://github.com/romainfrancois

This version was compiled on March 5, 2026

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, 2013; Eddelbuettel *et al.*, 2026; Eddelbuettel and Balamuta, 2018) 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 rigourously 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 available as the **GSL** itself is written in C, and provides a C-language Application Programming Interface (API). Access from C++ is possible, albeit not at an abstraction level that could be offered by dedicated C++ implementations. Several C++ wrappers for the **GSL** have been written over the years; none reached a state of completion comparable to the **GSL** itself.

The **GSL** combines broad coverage of scientific topics, serious implementation effort, and the use of the well-known GNU General Public License (GPL). This has lead to fairly wide usage of the library. As a concrete example, we can consider the Comprehensive R Archive Network (CRAN) repository network for the R language and environment (R Core Team, 2025). CRAN contains over three dozen packages interfacing the **GSL**. Of these more than half interface the vector or matrix classes as shown in Table 1. This provides empirical evidence indicating 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.*, 2026; Eddelbuettel and Balamuta, 2018) offers a higher-level interface between R and C++. **Rcpp** permits R objects like vectors, matrices, lists, functions,

Package	Any `gsl` header	`gsl_vector.h`	`gsl_matrix.h`
abn	⋆	⋆	⋆
BayesLogit	⋆		
BayesSAE	⋆	⋆	⋆
BayesVarSel	⋆	⋆	⋆
BH	⋆	⋆	
bnpmr	⋆		
BNSP	⋆	⋆	⋆
cghseg	⋆	⋆	⋆
cit	⋆		
diversitree	⋆		⋆
eco	⋆		
geoCount	⋆		
graphscan	⋆		
gsl	⋆	⋆	
gstat	⋆		
hgm	⋆		
HiCseg	⋆		⋆
igraph	⋆		
KFKSDS	⋆	⋆	⋆
libamtrack	⋆		
mixcat	⋆	⋆	⋆
mvabund	⋆	⋆	⋆
outbreaker	⋆	⋆	⋆
R2GUESS	⋆	⋆	⋆
RCA	⋆		
RcppGSL	⋆	⋆	⋆
RcppSMC	⋆		
RcppZiggurat	⋆		
RDieHarder	⋆	⋆	⋆
ridge	⋆	⋆	⋆
Rlibeemd	⋆	⋆	
Runuran	⋆		
SemiCompRisks	⋆		⋆
simplexreg	⋆	⋆	⋆
stsm	⋆	⋆	⋆
survSNP	⋆		
TKF	⋆	⋆	⋆
topicmodels	⋆	⋆	⋆
VBLPCM	⋆	⋆	
VBmix	⋆	⋆	⋆

Table 1. CRAN Package Usage of GSL outright, for vectors and for matrices.

Note: Data gathered in late July 2015 by use of `grep` searching (recursively) for inclusion of any GSL header, or the vector and matrix headers specifically, within the `src/` or `inst/include/` directories of expanded source code archives of the CRAN network. Convenient (temporary) shell access to such an expanded code archive via WU Vienna is gratefully acknowledged.

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

The **RcppGSL** package discussed here aims to close the gap. It offers 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' familar 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. The following two section discusses error handling, and then use of **RcppGSL** in your own package. This is followed by short discussions of how to use **RcppGSL** with **inline** and *Rcpp Attributes*, respectively, before a short concluding summary.

2. Motivation: fastLm

Fitting linear models is a key building block of analysing and modeling data. R has a very complete and feature-rich function in `lm()` which provides 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 as part of the **RcppGSL** package. 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>

// declare a dependency on the RcppGSL package;
// also activates plugin (but not needed when
// 'LinkingTo: RcppGSL' is used with a package)
//
// [[Rcpp::depends(RcppGSL)]]

// tell Rcpp to turn this into a callable
// function called 'fastLm'
//
// [[Rcpp::export]]
Rcpp::List fastLm(const RcppGSL::Matrix & X,
                  const RcppGSL::Vector & y) {

    // row and column dimension
    int n = X.nrow(), k = X.ncol();
    double chisq;
    // to hold the coefficient vector
    RcppGSL::Vector coef(k);
    // and the covariance matrix
    RcppGSL::Matrix cov(k,k);


    // the actual fit requires working memory
    // which we allocate and then 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;
    // need two step decl. and assignment
    std_err = gsl_matrix_diagonal(cov);
    // sqrt() is an Rcpp sugar function
    std_err = Rcpp::sqrt(std_err);

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

The function interface defines two **RcppGSL** variables: a matrix and a vector. Both use the standard numeric type `double` as discussed below. The **GSL** supports other types ranging from lower precision floating point to signed and unsigned integers as well as complex numbers. The vector and matrix classes are templated for use with all these C / C++ types—though R uses only `double` and `int`. For these latter two, we offer a shorthand definition via a `typedef` which allows a shorter non-template use. Having extracted the row and column dimentions, 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 just-allocated workspace. The next step involves extracting the diagonal element from the covariance matrix, and taking the square root (using a vectorised function from **Rcpp**). Finally we create a named list with the return values.

In earlier version of the **RcppGSL** package, we also explicitly called `free()` to return temporary memory allocation to the operating system. This step had to be done as the underlying objects are managed as C objects. They conform to the **GSL** interface, and work without any automatic memory management. But as we provide a C++ data structure for matrix and vector objects, we can manage them using C++ facilities. In particular, the destructor can free the memory when the object goes out of scope. Explicit `free()` calls are still permitted as we keep track the object status so that memory cannot accidentally be released more than once. Another more recent addition permits use of `const &` in the interface. This instructs the compiler that values of the corresponding variable will not be altered, and are passed into the function by reference rather than by value.

We note that **RcppArmadillo** (Eddelbuettel *et al.*, 2025; Eddelbuettel and Sanderson, 2014) implements a matching `fastLm` function using the Armadillo library by Sanderson (2010), and can do so with even more compact code due to C++ features. Moreover, **RcppEigen** (Bates *et al.*, 2024; Bates and Eddelbuettel, 2013) provides a `fastLm` implementation with a comprehensive comparison of matrix decomposition methods.

3. Vectors

This section details the different vector represenations, 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-dimensionnal 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;
// allocate a gsl_vector of size 3
gsl_vector *v = gsl_vector_alloc(3);

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

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

// free the memory
gsl_vector_free(v);
```

Note that we have to explicitly free the allocated memory at the end. With C-style programming, this step is always the responsibility of the programmer.

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

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

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

// (optionally) free the memory
// also automatic when out of scope
v.free();
```

The class `RcppGSL::vector<double>` is a smart pointer which can be deployed 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 of course the managed release of memory either via `free()` or when going out of scope), 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(const
                       RcppGSL::vector<int> &
                       vec) {
    int res = std::accumulate(vec.begin(),
                              vec.end(), 0);
    return res;
}
```

Here we no longer need to call `free()` explicitly as the `vec` allocation is returned automatically at the end of the function body when the variable goes out of scope.

Once the function has created via `sourceCpp()` or `cppFunction()` from *Rcpp Attributes* (see section 7 for more on this), it can then be called from R :

```
fx <- Rcpp::cppFunction("
int sum_gsl_vector_int(RcppGSL::vector<int> vec) {
    int res = std::accumulate(vec.begin(),
                              vec.end(), 0);
    return res;
}", depends="RcppGSL")
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(const Rcpp::List & data) {
    // grab "x" as a gsl_vector through the
    // RcppGSL::vector<double> class
    const RcppGSL::vector<double> x = data["x"];

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

called from R:

```
Rcpp::cppFunction("
double gsl_vector_sum_2(Rcpp::List data) {
    RcppGSL::vector<double> x = data[\"x\"];
    RcppGSL::vector<int> y = data[\"y\"];
    double res = 0.0;
    for (size_t i=0; i< x->size; i++) {
        res += x[i] * y[i];
    }
    return res;
}", depends= "RcppGSL")
data <- list( x = seq(0,1,length=10), y = 1:10 )
gsl_vector_sum_2(data)
#  [1] 36.6667
```

3.3. Mapping. Table 2 shows the mapping between types defined by the **GSL** and their corresponding types in the **RcppGSL** package.

As shown, we also define two convenient shortcuts for the very common case of `double` and `int` vectors. First, `RcppGSL::Vector` is a short-hand for the `RcppGSL::vector<double>` template instantiation. Second, `RcppGSL::IntVector` does the same for integer-valued vectors. Other types still require explicit templates.

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

    return Rcpp::List::create(
        Rcpp::Named("even") = v_even,
        Rcpp::Named("odd" ) = v_odd);
}
```

As with vectors, C++ objects of type `RcppGSL::vector_view` can be converted implicitly to their associated **GSL** view type. Table 3 displays the pairwise correspondance so that the C++ objects can be passed to compatible **GSL** algorithms.

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. And as before, `double` and `int` types can be accessed via the `typedef` variants `RcppGSL::VectorView` and `RcppGSL::IntVectorView`, respectively.

Lastly, in order to support `const &` behaviour, all `gsl_vector_XXX_const_view` variants are also supported (where `XXX` stands for any of the atomistic C and C++ data types).

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)

// 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 an implicit conversion to a pointer to the associated **GSL** matrix type, as well as dereferencing operators. This makes 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.

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

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

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 (also called via destructor)

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.

4.6. Error handler. When input values for **GSL** functions are invalid, the default error handler will abort the program after printing an error message. This leads R to an abortion error. To avoid this behaviour, one needs to avoid it first by using `gsl_set_error_handler_off()`, and then detect error conditions by checking whether the result is `NAN` or not.

GSL vector	RcppGSL
`gsl_vector`	`RcppGSL::vector<double>` as well as `RcppGSL::Vector`
`gsl_vector_int`	`RcppGSL::vector<int>` as well as `RcppGSL::IntVector`
`gsl_vector_float`	`RcppGSL::vector<float>`
`gsl_vector_long`	`RcppGSL::vector<long>`
`gsl_vector_char`	`RcppGSL::vector<char>`
`gsl_vector_complex`	`RcppGSL::vector<gsl_complex>`
`gsl_vector_complex_float`	`RcppGSL::vector<gsl_complex_float>`
`gsl_vector_complex_long_double`	`RcppGSL::vector<gsl_complex_long_double>`
`gsl_vector_long_double`	`RcppGSL::vector<long double>`
`gsl_vector_short`	`RcppGSL::vector<short>`
`gsl_vector_uchar`	`RcppGSL::vector<unsigned char>`
`gsl_vector_uint`	`RcppGSL::vector<unsigned int>`
`gsl_vector_ushort`	`RcppGSL::vector<insigned short>`
`gsl_vector_ulong`	`RcppGSL::vector<unsigned long>`

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

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

Table 3. Correspondance between GSL vector view types and templates defined in RcppGSL.

```
// close the GSL error handler
gsl_set_error_handler_off();

// call 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;
}
```

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.

5. Using RcppGSL in your package

The **RcppGSL** package contains a complete example package 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 neglible.

5.1. The `configure` script.

5.1.1. 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 "${GSL_CONFIG}" != ""; then
    # Use gsl-config for header and linker args
    # (without BLAS which we get from R)
    GSL_CFLAGS=`${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
AC_SUBST(GSL_CFLAGS)
AC_SUBST(GSL_LIBS)

AC_OUTPUT(src/Makevars)
```

A source file such as this `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.

5.1.2. 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
```

This allows for a simpler and more direct way to just set the compile and link options, taking advantage of the installed **RcppGSL** package. See the **RcppZiggurat** package for an example.

5.2. 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(const RcppGSL::Matrix & G) {
    int k = G.ncol();
    Rcpp::NumericVector n(k);           // results
    for (int j = 0; j < k; j++) {
        RcppGSL::vector_view<double> colview =
             gsl_matrix_const_column(G, j);
        n[j] = gsl_blas_dnrm2(colview);
    }
    return n;                           // return
}
```

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`.

5.3. 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.

5.4. Input and Output. Because **RcppGSL** vectors are really C vectors, a little care is needed when using C++ **iostream** output. Vector elements needs to be accessed explicitly via accessors as *e.g.* `gsl_vector_get(x, 0)` to display the initial element of a vector `x`.

6. Using RcppGSL with inline

The **inline** package (Sklyar *et al.*, 2025) 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 has been 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='
  // create data structures from SEXP
  RcppGSL::matrix<double> M = sM;
  int k = M.ncol();
  // to store results
  Rcpp::NumericVector n(k);

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

foo <- cxxfunction(
    signature(sM="numeric"),
    body=bodytxt, inc=inctxt, plugin="RcppGSL")

# 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)
```

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

```
package.skeleton("mypackage", foo)
```

7. Using RcppGSL with Rcpp Attributes

Rcpp Attributes (Allaire *et al.*, 2026) 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 is `sourceCpp()` is meant for interactive work at the R command-prompt, but is 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
colNorm(const RcppGSL::Matrix & M) {
    int k = M.ncol();
    Rcpp::NumericVector n(k);            // results

    for (int j = 0; j < k; j++) {
        RcppGSL::VectorView colview =
            gsl_matrix_const_column (M, j);
        n[j] = gsl_blas_dnrm2(colview);
    }
    return n;                            // return
}

/*** 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), "*")
colNorm(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.*, 2026) of the **Rcpp** package (Eddelbuettel *et al.*, 2026) for full details.

8. 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 widely-used and well-understood 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 algorithmns provided by the **GSL** with greater ease.

References

Allaire JJ, Eddelbuettel D, François R (2026). *Rcpp Attributes*. `doi:10.32614/CRAN.package.Rcpp`. Vignette included in R package Rcpp.

Bates D, Eddelbuettel D (2013). "Fast and Elegant Numerical Linear Algebra Using the RcppEigen Package." *Journal of Statistical Software*, **52**(5), 1–24. `doi:10.18637/jss.v052.i05`.

Bates D, Eddelbuettel D, François R, Qiu Y (2024). *RcppEigen: Rcpp Integration for the Eigen Templated Linear Algebra Library*. `doi:10.32614/CRAN.package.RcppEigen`. R package version 0.3.4.0.2.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Use R! Springer, New York. ISBN 978-1-4614-6867-7.

Eddelbuettel D, Balamuta JJ (2018). "Extending R with C++: A Brief Introduction to Rcpp." *The American Statistician*, **72**(1). `doi:10.1080/00031305.2017.1375990`.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. `doi:10.18637/jss.v040.i08`.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2026). *Rcpp: Seamless R and C++ Integration*. `doi:10.32614/CRAN.package.Rcpp`. R package version 1.1.1.

Eddelbuettel D, François R, Bates D, Ni B, Sanderson C (2025). *RcppArmadillo: Rcpp Integration for the Armadillo Templated Linear Algebra Library*. `doi:10.32614/CRAN.package.RcppArmadillo`. R package version 15.2.3-1.

Eddelbuettel D, Sanderson C (2014). "RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics and Data Analysis*, **71**, 1054–1063. `doi:10.1016/j.csda.2013.02.005`.

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F (2010). *GNU Scientific Library Reference Manual*, 3rd edition. Version 1.14. ISBN 0954612078, URL https://www.gnu.org/software/gsl/.

R Core Team (2025). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. `doi:10.32614/R.manuals`. URL https://www.R-project.org/.

Sanderson C (2010). "Armadillo: An open source C++ Algebra Library for Fast Prototyping and Computationally Intensive Experiments." *Technical report*, NICTA. URL https://arma.sourceforge.net/.

Sklyar O, Murdoch D, Smith M, Eddelbuettel D, François R, Soetaert K, Ranke J (2025). *inline: Functions to Inline C, C++, Fortran Function Calls from R*. `doi:10.32614/CRAN.package.inline`. R package version 0.3.21.