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) 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, 2017). 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., 2017) offers a higher-level interface between R and C++. Rcpp permits R objects like vectors, matrices, lists, functions, environments, …, to be ma-

```
<table>
<thead>
<tr>
<th>Package</th>
<th>Any gsl header</th>
<th>gsl_vector.h</th>
<th>gsl_matrix.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>abn</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BayesLogit</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BayesSAE</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>BayesVarSel</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>BH</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>bnpmmr</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNNS</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>cghseg</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>cit</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diversiTree</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eco</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>geoCount</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>graphscan</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gsl</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gstat</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hgm</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HiCseg</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>igraph</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KFKSDS</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>libamtrack</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>mixcat</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mvabund</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outbreaker</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2QUGESS</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCA</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RcppGSL</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RcppSMC</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RcppZiggurat</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDieHarder</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>ridge</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Rifibeemd</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runuran</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SemiCompRisks</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>simplexreg</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>stsm</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>survSNP</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TKF</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>topicmodels</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>VBLPCM</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBmix</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>
```

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.
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’ 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. The following two sections discuss 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.

```cpp
#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 dimensions, 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., 2016; 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., 2016; 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-dimensionnal data, similar to R arrays. For example the `gsl_vector` and `gsl_vector_int` structs are defined as:

```c
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:

```c
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
free(v);
```

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:

```c
// [[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:

```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`

```c
// [[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 res;
}
```
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.

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.


4.2. 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.

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.

4.4. Methods. The RcppGSL::matrix type also defines the following member functions:

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 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.

```c
// 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: `gsl_set_error_handler_off()` and `gsl_reset_error_handler()` 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 `gsl_set_error_handler_off()` 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 of multi-core BLAS variant which compute the vector norm in parallel. On all ‘reasonable’ matrix sizes, however, the performance difference should be negligible.

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:

```bash
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 is compiled, linked and loaded on-the-fly.

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:

```bash
#!/bin/sh

GSL_CFLAGS="$(R_HOME)/bin/Rscript -e "RcppGSL:::CFLAGS()"
GSL_LIBS="$(R_HOME)/bin/Rscript -e "RcppGSL:::LdFlags()"
```

```
# 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)
```

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.

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

```c
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:

```bash
# 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.

6. Using RcppGSL with inline

The inline package (Sklyar et al., 2015) 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.

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

```c
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
}
```

```bash
GSL_CFLAGS="$(R_HOME)/bin/Rscript -e "RcppGSL:::CFLAGS()"
GSL_LIBS="$(R_HOME)/bin/Rscript -e "RcppGSL:::LdFlags()"
```

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

```bash
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()"
```
### require (inline)

```r
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")
```

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

#### 7. Using RcppGSL with Rcpp Attributes

**Rcpp Attributes** (Allaire et al., 2017) 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.

```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:

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

The function `cppFunction()` is also available to convert a simple character string argument containing a valid C++ function into an eponymous R function. And like `sourceCpp()`, it can also use plugins. See the vignette “Rcpp-attributes” (Allaire et al., 2017) of the **Rcpp** package (Eddelbuettel et al., 2017) 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 algorithms provided by the **GSL** with greater ease.
References


