**dotCall64: An Efficient Interface to Compiled C/C++ and Fortran Code Supporting Long Vectors**

by Florian Gerber, Kaspar Mösinger and Reinhard Furrer

**Abstract** The R functions `.C()` and `.Fortran()` can be used to call compiled C/C++ and Fortran code from R. This so-called foreign function interface is convenient, since it does not require any interactions with the C API of R. However, it does not support long vectors (i.e., vectors of more than $2^{31}$ elements). To overcome this limitation, the R package `dotCall64` provides `.C64()`, which can be used to call compiled C/C++ and Fortran functions. It transparently supports long vectors and does the necessary castings to pass numeric R vectors to 64-bit integer arguments of the compiled code. Moreover, `.C64()` features a mechanism to avoid unnecessary copies of function arguments, making it efficient in terms of speed and memory usage.

**Introduction**

The interpreted character of R makes it a convenient front-end for a wide range of applications. Although R provides a rich infrastructure, it can be advantageous to extend R programs with compiled code written in C/C++ or Fortran (Eubank and Kupresanin, 2011). According to Chambers (2008), reasons for such an extension are the access to new and trusted computations, the increase in computational speed, and the object referencing capabilities. For completeness, we also list the reasons against such an extension, which include an increased workload to write, maintain, and debug the software, platform dependencies, and a less readable source code.

R provides two types of interfaces to call compiled code documented in “Writing R Extensions” (R Core Team, 2016a). First, the modern interfaces to C/C++ code feature the R functions `.Call()` and `.External()`. It enables accessing, modifying, and returning R objects from C/C++ using the C API of R (Wickham, 2014). On one hand, this is convenient when the C/C++ code is specifically written to be used with R. In that case, the C API serves as a glue between R and C/C++, providing some R functionality and control over copying R objects on the C/C++ level. On the other hand, it requires the user to learn the C API of R. Especially, when an R interface is built on top of existing C/C++ code this constitutes an additional effort. Since R has no Fortran API, the modern interfaces to C/C++ code are not suitable to embed Fortran code into R. Second, the foreign function interface provides the R functions `.C()` and `.Fortran()`. This interface allows the compiled code to read and modify atomic R vectors, which are exposed as the corresponding C/C++ and Fortran types, respectively. Thus, no additional API is required, making it favorable for embedding C/C++ and Fortran code that is not specifically designed for R.

On top of these interfaces provided by R, R packages exist that simplify the integration of compiled code into R. One such R package is `inline` (Sklyar et al., 2016), which allows the user to dynamically define R functions and S4 methods with inlined compiled code. Other examples are `Rcpp` (Eddelbuettel et al., 2016a; Eddelbuettel and François, 2011; Eddelbuettel, 2013) and its extensions `RcppArmadillo` (Eddelbuettel et al., 2016b; Eddelbuettel and Sanderson, 2014), `RcppEigen` (Bates et al., 2016; Bates and Eddelbuettel, 2013), `RcppParallel` (Allaire et al., 2016), and `Rcpp11` (François and Ushey, 2014), which greatly simplify the extension of R with C++ code. Similar to the modern interfaces to C/C++ code, the Rcpp package family is designed to extend R with compiled code that is specifically written for that purpose.

Building R packages is a way to share compiled code across different platforms. (See, e.g., Plummer, 2011 for comments on including portable C++ code in R packages.) As of 09-02-2016, 2'303 of the 9'079 R packages on CRAN (http://www.cran.r-project.org/) include compiled C/C++ and/or Fortran code using both the foreign function interface and the modern interfaces to C/C++ code with a similar frequency. Figures 1 gives an overview of the number of packages using `.C()`, `.Fortran()`, `.Call()`, and `.External()`.

In the remainder of this article, we focus on the intention to embed compiled code into R without using its C API. An example of an R package using that type of interface is the SPArse Matrix package `spam` (Furrer, 2016; Furrer and Sain, 2010; Gerber and Furrer, 2015), which is built around the Fortran library SPARSKIT (Saad, 1994). Here, the R function `.Fortran()` from the foreign function interface seems to be suitable. Conversely, using the modern interfaces to C/C++ code is also possible but requires adding an additional layer of C code to enable communication between R and the compiled Fortran code. However, using `.Fortran()` is also not satisfying, since it lacks flexibility and
Figure 1: Number of R packages on CRAN using the indicated R functions to interface C/C++ or Fortran code (as of 2016-09-02). Note that R packages linking to Rcpp use `.Call()`.

performance, as also stated in its help page: “These functions [.C() and .Fortran()] can be used to make calls to compiled C and Fortran 77 code. Later interfaces are ‘.Call’ and ‘.External’ which are more flexible and have better performance.” Two of the missing features of the foreign function interface are:

- support of long vectors,
- a mechanism to avoid unnecessary copies of R vectors.

The latter is the reason for the lower performance of the foreign function interface compared to the modern interfaces to C/C++ code. Since the foreign function interface does not allow R vectors to be passed to compiled code by reference (without copying), it is especially impractical for big data application. The missing features of the foreign function interface motivated the development of the R package dotCall64 (Mösinger, Gerber, and Furrer, 2016), which is presented in this article.

Limitations of the foreign function interface

To set the scene for dotCall64, we first discuss some limitations of the foreign function interface and give insights into the R implementation of long vectors.

Long vectors

The foreign function interface does not support long vectors; see help("long vector"). To understand why extending it to support long vectors is a non-trivial task, we give more details on the long vector implementation of R. In R, vectors are one of the most basic object types underlying more complex objects, such as matrices and arrays. They can be thought of as strings of elements that can be indexed according to their relative positions. Prior to version 3.0.0, the length of vectors was limited to $2^{31} - 1$ elements and indexing thereof was exclusively based on R vectors of type integer. More precisely, the latter are signed 32-bit integer vectors having a value range of $[-2^{31} + 1, 2^{31} - 1]$. Starting from the release of version 3.0.0 in early 2013, support for so-called long vectors was supplied. That is, atomic (raw, logical, integer, numeric, complex, and character) vectors, lists, and expressions can now have up to $2^{52}$ elements. The introduction of long vectors was done with minimal changes in R and especially, without changing or adding a 64-bit integer data type. Vectors of lengths less than $2^{31} - 1$ remain unchanged and addressing elements thereof still uses R vectors of type integer. In contrast, long vectors use numeric vectors of type doubles to address elements, which are integer precise up to $2^{52}$. This implied changes in some R functions, such as length(), which returns an integer or a double type depending on whether the input vector is a long vector.

> typeof(length(integer(1)))
[1] "integer"
> typeof(length(integer(2^31)))
[1] "double"

Note that as.numeric() returns a double type and as.integer() returns an integer type, though both integer and double type are of class "numeric", see the "Note on names" section in the R help page help("is.double").

While the R implementation of long vectors favors backwards compatibility, care is needed when manipulating those with compiled code. We distinguish between passing long vectors and indexing long vectors. The former requires passing vectors of more than $2^{31} - 1$ elements to compiled code and
is trivial. The latter is challenging, since the indexing R vector is of type double, whereas the compiled code would naturally expect a 64-bit integer type. To overcome this discrepancy, one needs to cast the indexing vector from a double to a 64-bit integer type before calling the compiled code and back-cast it afterwards.

**Technical note:** This section gives technical insights into the underlying C implementation of long vectors in R and may be skipped without loss of the general idea. We refer to the source code of R version 3.3.1 in several places and show relevant parts thereof in the appendix. Information on the current and future directions of long vectors and 64-bit types in R can be found in “R Internals” (R Core Team, 2016b, Section 12).

In R, vectors are made out of a header of type VECSEXP that is followed by the actual data (Listing 1, line 272). The header contains a field length of type R_len_t, which is defined as signed int32_t (a 32-bit integer). Thus, that length field cannot capture the length of a long vector. Instead, it is set to -1 whenever the length of the vector is larger than $2^{31} - 1$, and an additional header of type R_long_vec_hdr_t is prefixed. The prefixed header has a field length of type R_xlen_t, which is defined as ptrdiff_t type (Listing 1, line 75) being “[... the signed integer type of the result of subtracting two pointers. This will probably be one of the standard signed integer types (short int, int or long int), but might be a nonstandard type that exists only for this purpose” (GNU C Library, 2016, Appendix A.4).

This implementation has the advantage that the existing code does not need to be changed and still works with vectors having less than $2^{31}$ elements. Hence, the C code of R can be changed successively to support long vectors throughout several R versions, as opposed to changing the entire C code in one step. To make C code compatible with long vectors, adaptations are needed. For example, the widely used C function R_len_t length(SEXP s) (Listing 2, line 124) returns the length of a SEXP (S expression) as a R_len_t. Thus, all instances of that function have to be replaced with calls to the 64-bit counterpart (i.e., the function R_xlen_t xlength(SEXP s) given in line 159 of Listing 2).

**Copying arguments**

The foreign function interface exposes pointers to R vectors to compiled code. In order to avoid any corruption of R vectors, they are copied and the compiled code receives pointers to copies of the R vectors. One exception is when the R vector has the named status 0 (i.e., the object is not bound to any symbol); see “Writing R Extensions” (R Core Team, 2016a, Section 5.9.10). This is the case when the passed R vector is an evaluated constructor (e.g., integer(1)). This is often used when the only purpose of the R vector is to capture results from the compiled code.

Another situation in which there is no need for copying R vectors is when the compiled code only reads an R vector without modifying it. However, the foreign function interface does not allow the user to avoid copying of R vectors (with named status 1 or 2), which leads to a significant computational overhead, especially for large vectors. Note that prior to R version 3.2.0, the copying of R vectors could be avoided by setting the argument DUP of .C() and .Fortran() to FALSE. In later R versions, this argument is depreciated and users are referred to the modern interfaces to C/C++ code as a more flexible interface; see help(".C") and “R NEWS” (R Core Team, 2016c).

**The R package dotCall64**

The limitations of the foreign function interface discussed above have motivated the development of the R package dotCall64. Its main function is .C64(), which can be used to interface compiled code. In contrast to .C() and .Fortran(), it supports long vectors and 64-bit integer arguments of compiled compiled functions/subroutines and provides a mechanism to control duplication of function arguments. Emphasis was put on providing a trustworthy implementation featuring structured R and C source code, documentation, examples, unit tests implemented with testthat (Wickham, 2011), and R scripts containing the later presented performance measurements.

**Usage of the R function .C64()**

The function .C64() can be used as an enhanced replacement of the foreign function interface and is equally easy to use; see also the documentation in the reference manual (Mösinger, Gerber, and Furrer, 2016). Its syntax resembles that of the function .C(), and both functions have common arguments as shown in Table 1.
The required arguments of `.C64()` are:

- `.NAME` The name of the compiled C/C++ function or Fortran subroutine.
- `...` Up to 65 R vectors to be accessed by the compiled code.
- `SIGNATURE` A character vector of the same length as the number of arguments of the compiled function/subroutine. Each string specifies the signature of one such argument. Accepted signatures are “integer”, “double”, and “int64”. The R, C/C++, and Fortran types corresponding to these specifications are given in Table 2.

With that, the following call to the compiled C function void get_c(double input, int index, double output) using `.C()` can be replaced by its `.C64()` counterpart. Therefore, for example,

```r
> .C("get_c", input = as.double(1:10), index = as.integer(9), output = double(1))
```

becomes

```r
> .C64("get_c", SIGNATURE = c("double", "integer", "double"),
+ input = 1:10, index = 9, output = 0)
```

While more detailed code examples are given later, this is enough to highlight some features of `.C64()`. First, `.C64()` does require the additional argument `SIGNATURE` specifying the argument types of the compiled function/subroutine. In return, it coerces the provided R vectors to the specified signatures making the `as.double()` and `as.integer()` statements unnecessary. Second, all provided arguments can be long vectors. Third, if one of the arguments of the compiled function is a 64-bit integer (`int64_t` in the case of C/C++ functions, and `integer (kind = 8)` types for Fortran subroutines), it is enough to set the corresponding `SIGNATURE` argument to “int64” to successfully evaluate the function. That is, `.C64()` does the necessary double to 64-bit integer and 64-bit integer to double castings before and after evaluating the compiled code, respectively.

Additional arguments of `.C64()` are the following:

- `INTENT` A character vector of the same length as the number of arguments of the compiled function/subroutine. Each string specifies the intent of one such argument. Accepted intents are "rw" (read and write), "r" (read), and "w" (write).
- `NAOK` A logical flag specifying whether the R vectors passed though ‘...’ are checked for missing and infinite values.
- `PACKAGE` A character vector of length one restricting the search path of the compiled function/subroutine to the specified package.
- `VERBOSE` If 0 (default), no warnings are printed. If 1 and 2, then warnings for tuning and debugging purposes are printed.

A complete list of arguments including their default values is also given in Table 1.

### Table 1: Arguments and default values of the R function `.C()` from the foreign function interface and `.C64()` from `dotCall64`. The depreciated arguments of `.C()` are marked with “∗”.

| Argument | Type | Default |
|----------|------|---------|
| `.NAME`  |      |         |
| `...`    |      |         |
| `SIGNATURE` |      |         |
| `NAOK`   |      | FALSE   |
| `DUP`    |      | TRUE    |
| `PACKAGE` |      |         |
| `ENCODING` |     |         |

### Table 2: Supported `SIGNATURE` arguments of `.C64()` and the corresponding C/C++, Fortran, and R data types. The column “cast” indicates whether casting is necessary.

| `SIGNATURE` | C/C++ type | Fortran type | R type  | cast |
|-------------|------------|--------------|---------|------|
| “double”    | double     | double precision | double | no   |
| “int”       | int        | integer (kind = 4) | integer | no   |
| “int64”     | int64_t    | integer (kind = 8) | double  | yes  |
The argument `INTENT` influences the copying of R vectors and can be seen as an enhanced version of the depreciated `DUP` argument of `.C()`. By default, all intents are set to “read and write” implying that the compiled code receives pointers to copies of the R vector given to ‘...’. This behavior is desirable when the compiled function reads the corresponding R vectors and modifies (writes to) them. For arguments of the compiled function/subroutine that are only read and not modified, the intent can be set to “read.” With that, the compiled code receives pointers to the corresponding R vectors itself. While this avoids copying, it is absolutely necessary that the compiled code does not alter these vectors, as this corrupts the corresponding R vectors in the current R session. For arguments that are only used to write results into it, the intent “write” is suitable. To obtain the desired performance gain, the corresponding R vectors passed to ‘...’ have to be of class “vector_dc”. R objects of that class contain information on the type and length of the vectors. They can be constructed with the R function `vector_dc()`, taking the same arguments as `vector()` from the `base` R package. For example, instead of passing the R vector `vector(mode = "numeric", length = 8)`, the following R object should be passed.

```r
> vector_dc(mode = "numeric", length = 8)
$mode
[1] "numeric"

$length
[1] 8

attr(,"class")
[1] "vector_dc" "list"
```

Based on this information, `.C64()` allocates the corresponding vector (initialized with zeros). That vector is then exposed to the compiled function to write into it. Note that specifying the suitable intent may reduce computation time by avoiding unnecessary copying of R vectors and by avoiding unnecessary double to 64-bit integer and 64-bit integer to double castings for `SIGNATURE = "int64"` type arguments. More details on the other arguments are given in the package manual of `dotCall64` (Mösinger, Gerber, and Furrer, 2016).

**Implementation of the R function `.C64()`**

The function `.C64()` uses the function `.External()` from the modern interfaces to C/C++ code to directly pass all provided arguments to the C function `dC64()`. After basic checks of the provided arguments, the function proceeds as schematized in Figure 2. Note that the flowchart depicts the procedure for the case in which the compiled function/subroutine has only one argument. Otherwise, `dC64()` repeats the depicted scheme for all arguments.

One aspect to highlight is the castings of R vectors for `SIGNATURE = "int64"` arguments. For such arguments, the double to `int64_t` casting is done for the intents “read and write” and “read”; see the boxes labeled with (a). In that case, duplication is not necessary, as the implemented casting allocates a new vector anyway. The back-casting from `int64_t` to double is only done for the intents “read and write” and “write”; see the box labeled with (b).

Moreover, an argument of `SIGNATURE` different from “int64” with intent “read and write” is duplicated in any case; see boxes labeled with (c). If the intent is “read,” it is not duplicated, and if the intent is “write,” the argument is only duplicated when it has a reference status different from 0. R vectors increase their reference status when they are passed to an R function, and therefore a safe way to allocate a zero initialized vector without copying is to pass an R object of class "vector_dc".

As casting is an expensive operation in terms of computational time, we distribute this task to multiple threads using openMP, if available (Dagum and Menon, 1998; OpenMP architecture review board, 2016). Note that the number of used threads can be controlled with the R function `omp_set_num_threads()` from the package `OpenMPController` (Guest, 2013). The package `dotCall64` can also be compiled without the openMP feature by removing the flag `$\$(SHLIB_OPENMP_CFLAGS)` in the ‘src/Makevars’ file of the source code.

**Examples**

We showcase the function `.C64()` from the R package `dotCall64` with an example function implemented in C and Fortran. Besides the calls thereof via `.C64()`, the C and Fortran function definitions and the commands to compile and load the code are given. A direct comparison with `.C()` shows the limitations of the foreign function interface and that it is straightforward to overcome these with `.C64()`. Moreover, the similarities and differences in the syntax become visible. The considered
Figure 2: Flowchart of the involved processes when using `.C64()` to call a compiled function/subroutine with one argument. In the pre-process phase, the provided R vector passed through ‘...’ is checked and prepared according to the arguments NAOK, SIGNATURE, and INTENT. Then, the compiled function/subroutine specified with the argument .NAME is called. Finally, the vector is back-cast in the post-process phase if necessary.

example function takes the arguments ‘input’ (double), ‘index’ (integer), and ‘output’ (double) and writes the element of ‘input’ at the position specified with ‘index’ to ‘output’.

Interface C/C++ code

A C implementation of the described example function is given next.

```c
void get_c(double *input, int *index, double *output) {
    output[0] = input[index[0] - 1];
}
```

We write the function into ‘get_c.c’ and compile it with the command line command ‘R CMD SHLIB get_c.c’. The resulting dynamic shared object (‘get_c.so’ on our Linux platform) must be loaded into R before the compiled function can be called. Note that, in the following R code, the extension of the shared object is replaced with .Platform$dynlib.ext to make the code platform independent.

```r
> dyn.load(paste0("get_c", .Platform$dynlib.ext))
```

One can use the foreign function interface to call this function. We use the R functions as.double() and as.integer() to ensure that the types of the passed R vectors match the signature of the C function get_c().

```r
> .C("get_c", input = as.double(1:10), index = as.integer(9), output = double(1))$output [1] 9
```
Next, we try to use the same call with a long vector `x_long` passed to the argument 'input' of `get_c()`.

```r
> x_long <- double(2^31); x_long[9] <- 9; x_long[2^31] <- -1
> .C("get_c",
      input = as.double(x_long), index = as.integer(9), output = double(1))$output
Error: long vectors (argument 1) are not supported in .Fortran
```

As expected, `.C()` throws an error because it does not support long vectors. The error—and the confusing error message referring to `.Fortran()` instead of `.C()` with `.C64()`—can be avoided by replacing `.C()` with `.C64()`. This allows the evaluation of the C function `get_c()` with the long vector `x_long`. Additionally, `.C64()` requires the argument `SIGNATURE` encoding the signatures of the arguments of `get_c()`. This information is used to coerce all provided R vectors to the specified signatures. Thus, it is no longer necessary to reassure that the types of the passed R vectors match the signature of the compiled function.

```r
> install.packages("dotCall64")
> library("dotCall64")
> .C64("get_c", SIGNATURE = c("double", "integer", "double"),
      input = x_long, index = 9, output = double(1))$output
[1] 9
```

In contrast to the call using `.C()`, the ninth element of the long vector `x_long` is returned. However, the argument `index` of `get_c()` is of type int (a 32-bit integer), and hence, elements at positions beyond $2^{31} - 1$ cannot be extracted. To overcome this, we adapt the definition of the C function `get_c()` and replace the `int` type in the declaration of the argument `index` with the `int64_t` type, which is defined in the C header file `stdint.h`.

```c
#include <stdint.h>
void get64_c(double *input, int64_t *index, double *output) {
  output[0] = input[index[0] - 1];
}
```

We write the function into `get64_c.c` and compile it with `R CMD SHLIB get64_c.c` to obtain the dynamic shared object (`get64_c.so` on our platform). Because of the `int64_t` argument, it is not possible to call this function with `.C()`. On the other hand, `.C64()` can interface this function when the second element of the `SIGNATURE` argument is set to "int64".

```r
> dyn.load(paste0("get64_c", .Platform$dynlib.ext))
> .C64("get64_c", SIGNATURE = c("double", "int64", "double"),
      input = x_long, index = 2^31, output = double(1))$output
[1] -1
```

In the call above, the function `.C64()` casts the argument `index` from `double` (the R representation of 64-bit integers) into a `int64_t` type vector before calling `get64_c()`, and back-casts it from `int64_t` to `double` afterwards.

### Interface Fortran code

The function `.C64()` can also be used to interface compiled Fortran code. To highlight some Fortran specific features, we translate the C function `get_c()` into the Fortran subroutine `get_f()`.

```fortran
subroutine get_f(input, index, output)
  double precision :: input(*), output(*)
  integer :: index
  output(1) = input(index)
end
```

Note that we only use lower case letters in the Fortran function and variable names to avoid unnecessary symbol-name translations. We write the function into the `get64.f` and compile it with `R CMD SHLIB get64.f` to obtain the dynamic shared object (`get64.so` on our platform). In contrast to `.Fortran()`, `.C64()` allows passing pointers to long vectors.

```r
> dyn.load(paste0("get_f", .Platform$dynlib.ext))
> .C64("get_f", SIGNATURE = c("double", "integer", "double"),
      input = x_long, index = 9, output = double(1))$output
[1] 9
```

Again, elements with positions beyond $2^{31} - 1$ cannot be accessed, since the argument `index` is of type integer and compiled as a 32-bit integer by default. To make `get_f()` compatible with 64-bit
integers, we can either change the declaration of ‘index’ to ‘integer (kind = 8) index’ in ‘get_f.f’ or leave the Fortran code unchanged and set the following compiler flag to compile integers as 64-bit integers.

```
MAKEFLAGS="PKG_FFLAGS=-fdefault-integer-8" R CMD SHLIB get_f.f
```

Note that both the ‘kind = 8’ declaration and the ‘-fdefault-integer-8’ flag are valid for the GFortran compiler (GNU Fortran compiler, 2014) and may not have the intended effect using other compilers. The resulting dynamic shared object from the command above (‘get_f.so’ on our platform) can be called from R as follows.

```
> dyn.load(paste0("get_f", .Platform$dynlib.ext))
> .C64("get_f", SIGNATURE = c("double", "int64", "double"),
+    input = x_long, index = 2^31, output = double(1))$output
[1] -1
```

### Extend R packages to support long vectors

Extending R packages to support long vectors allows developers to distribute compiled code featuring 64-bit integers with an R user interface. Given the popularity of R, this is a promising approach to make such software available to many users. With the function `.C64()`, the workload of extending an R package to support long vectors is reduced to the following tasks:

- replace the R function to call compiled code with `.C64()`,
- replace the 32-bit integer type declarations in the compiled code with a 64-bit integer declaration.

The latter task implies replacing all `int` type declarations in C/C++ code with `int64_t` type declarations and replacing all `integer` type declarations in Fortran code with ‘integer (kind = 8)’. In both cases, the replacements can be automatized (e.g., with the stream editor GNU sed, 2010). If the considered Fortran code does not explicitly declare the bits of the integers, an alternative approach is to set the compiler flag ‘-fdefault-integer-8’ to compile integers as 64-bit integers using GFortran compilers. This is convenient because the Fortran code does not need to be changed at all in that case.

A more elaborate extension could feature two versions of the compiled code: one with 32-bit integers and the other one with 64-bit integers. Then, the R function can dispatch to either version according to the sizes of the involved vectors. This avoids double to 64-bit integer castings when only vectors with less than \(2^{31} - 1\) elements are involved. It is convenient to manage two versions of compiled code by putting them into two separate R packages. The first package includes the compiled code with 32-bit integers together with the R code and the documentation. This package can be used independently as long as no long vectors are involved. The second package can be seen as an add-on package and includes only the compiled code with integers declared as 64-bit integers. Thus, loading both packages enables long vector support. This separation into two packages has the advantage that the compiled functions featuring 32-bit integers and their 64-bit counterparts can have the same name. The desired function is then specified by setting the appropriate `PACKAGE` argument of `.C64()`.

As a proof of concept, we extended the sparse matrix algebra R package `spam` to handle sparse matrices with more than \(2^{31} - 1\) non-zero elements. From the user perspective, the syntax to manipulate such matrices remains the same. In fact, `spam` users may not even notice the extension. In the case, in which the number of non-zero entries of a matrix exceeds \(2^{31} - 1\) and the add-on package `spam64` is loaded, `spam` automatically dispatches to the compiled code with 64-bit integers. The new capabilities of `spam` and `spam64` were illustrated with a parametric model of a non-stationary spatial covariance matrix fitted to satellite data. More information on `spam64` and the data example is given by Gerber, Mösinger, and Furrer (2016).

### Performance

There are different settings in which the elapsed time to interface compiled code is relevant. One of those is when the compiled code is interfaced often and takes only a short time to evaluate. Here, the overhead of the interface becomes relevant, which is in the order of a few microseconds for `.C64()`. Another such setting is when large and possibly long vectors are passed through `.C64()`. In that case, the overhead is negligible, as other services of the interface and the execution of the compiled code take up several orders of magnitude more time. When `.C64()` is used to interface 64-bit integer arguments of the compiled code, the largest share of the elapsed time is caused by the double to 64-bit integer and 64-bit integer to double castings. Since castings are implemented with openMP, the elapsed time thereof also depends on the number of used threads. Besides that, copying objects and checking them for missing/infinite values are also time-consuming operations.
Another performance aspect is peak memory usage. Using the default arguments of `.C64()`, its peak memory usage is about twice the size of the R vectors passed through `...`, and hence, is similar to `.C()`. An exception where the peak memory usage is reduced is indicated below.

### Performance relevant arguments of `.C64()`

Further, `.C64()` provides arguments to optimize calls to compiled code, one of which is the argument `INTENT`, which is set to “read and write” by default. Since many compiled functions/subroutines only read or write to certain arguments, it is safe to avoid copying in some cases. For example, the C function `get64_c()`, as defined above, only reads the arguments ‘input’ and ‘index’ and only writes to the argument ‘output’. Thus, we can set the `INTENT` argument of `.C64()` to `c("r", "r", "w")` and pass the argument with intent “write” as objects of class “vector_dc” to reduce the copying of R vectors to a minimum. Another significant performance gain is obtained by setting the argument `NAOK` to `TRUE`. This avoids checking the R vectors passed through ‘...’ for `NA`, `NaN`, and `Inf` values. Small-scale performance gains can be achieved by setting the `PACKAGE` argument, which reduces the time to find the compiled code, and by setting `VERBOSE = 0`, which avoids the execution of `getOptions("dotCall64.verbose")`. Similar speed considerations that are partially applicable to `.C64()` are given in “Writing R Extensions” (R Core Team, 2016a, Section 5.4.1). An optimized version of the call to the C function `get64_c()`, taking the discussed performance considerations into account, is given next.

```r
> .C64("get64_c", SIGNATURE = c("double", "int64", "double"),
+ input = x_long, index = 2^31, output = numeric_dc(1),
+ INTENT = c("r", "r", "w"), NAOK = TRUE, PACKAGE = "dotCall64", VERBOSE = 0)
```

### Timing measurements

In the following, we present detailed timing measurements and benchmark `.C64()` against `.C()`, where possible. We consider the following C function contained in the R package `dotCall64`.

```c
void BENCHMARK(void *a) {
}
```

This function takes one pointer ‘a’ to a variable of an unspecified data type and does no operations with it. Thus, the elapsed time to call `BENCHMARK()` from R is dominated by the performance of the used interface. We measure the time to call this function with different `NAOK` and `INTENT` settings of `.C64()` and benchmark it against `.C()` using `microbenchmark` (Mersmann et al., 2015). To get an estimate of the measurement uncertainty, we repeated the measurements between 100 and 10’000 times and report the median elapsed time as well as the interquartile range (IQR) of the replicates. Naturally, timing measurements are platform dependent. We produced the presented results on Intel Xeon CPU E7-2850 2.00 GHz processors using a 64-bit Linux environment where R was installed with default installation flags. When not indicated differently, the measurements were produced using a single thread.

First, we consider the situation in which a pointer to an R vector of length one is passed to the compiled C function `BENCHMARK()`. The following truncated R code illustrates how the measurements were performed. The complete R scripts implementing all presented performance measurements are available in the ‘benchmark’ directory in the source code of `dotCall64`.

```r
> library("microbenchmark")
> int <- integer(1)
> microbenchmark(
+ .C("BENCHMARK", a = int, NAOK = FALSE, PACKAGE = "dotCall64"),
+ .C64("BENCHMARK", SIGNATURE = "integer", a = int, INTENT = "rw"),
+ NAOK = FALSE, PACKAGE = "dotCall64", VERBOSE = 0),
+ .C64("BENCHMARK", SIGNATURE = "integer", a = int, INTENT = "r"),
+ NAOK = FALSE, PACKAGE = "dotCall64", VERBOSE = 0),
```

Since the R vector ‘`int`’ is very short, a large part of the elapsed time in this experiment is caused by the overhead of the interfaces. Table 3 presents the resulting timing measurements in microseconds. They indicate that `.C()` is more than two times faster compared to `.C64()`. However, this is not surprising, since `.C64()` is more flexible and therefore has a larger overhead. The arguments `NAOK` and `INTENT` have little influence on the elapsed times. The IQRs of around one microsecond indicate a relatively large variability of the elapsed time, which is typical for short timing measurements.

The R Journal Vol. XX/YY, AAAA 20ZZ

ISSN 2073-4859
Table 3: Elapsed times in microseconds to pass double, integer, and 64-bit integer pointers to vectors of length one from R to C using \texttt{.C()} and \texttt{.C64()}. The used INTENT arguments of \texttt{.C64()} are indicated in brackets. Reported are median elapsed times of 100,000 replicates. The corresponding IQRs are indicated in parentheses.

We repeated the same experiment with vectors of length $2^{28}$. Now, the elapsed times are dominated by services of the interfaces (i.e., checking for missing/infinite values, copying, and casting). The timings in seconds are presented in Table 3. They indicate that \texttt{.C64()} with argument INTENT = "rw" and \texttt{.C()} showed similar elapsed times. When the intent is set to “read” (INTENT = "r"), the elapsed times were reduced and dropped to 0.00 seconds for some configurations. Moreover, not checking for missing/infinite values (NAOK = TRUE) decreases the elapsed times across all considered cases. The castings of SIGNATURE = "int64" arguments seems to be the most time-consuming task. Note that the IQRs are now smaller relative to the measured timings, because the measured times are larger.

Table 4: Elapsed times in seconds to pass double, integer, and 64-bit integer pointers to vectors of length $2^{28}$ from R to C using \texttt{.C()} and \texttt{.C64()}. The used INTENT arguments of \texttt{.C64()} are indicated in brackets. Reported are median elapsed times of 100 replicates. The corresponding IQRs are indicated in parentheses.

In a second series of timing measurements, we consider the situation in which a pointer to a vector is passed to the compiled code to write into the vector. We measure the elapsed times of this task as shown in the following truncated R code.

```r
> microbenchmark(
+ \texttt{.C("BENCHMARK", a = integer(2^{28}), NAOK = TRUE, package = "dotCall64")}
+ \texttt{.C64("BENCHMARK", SIGNATURE = "integer", a = integer(2^{28}), INTENT = "rw",}
+ NAOK = TRUE, package = "dotCall64", VERBOSE = 0)
+ \texttt{.C64("BENCHMARK", SIGNATURE = "integer", a = integer_dc(2^{28}), INTENT = "w",}
+ NAOK = TRUE, package = "dotCall64", VERBOSE = 0),
...) 
```

Note the usage of \texttt{integer_dc()}, which creates a list containing the length and class of the vector. This information is then used by \texttt{.C64()} to create the corresponding vector in C. Table 5 shows the timing measurements for the described setting. As expected using \texttt{.C64()} with INTENT = "w" reduces the elapsed times compared to INTENT = "rw" substantially. Furthermore, \texttt{.C()} and \texttt{.C64()} with INTENT = "w" have similar elapsed times. While \texttt{.C()} relies on the reference counting mechanism of R objects to avoid copying ("Writing R Extensions," R Core Team, 2016a), \texttt{.C64()} uses the "vector_dc" class. The latter has the advantage that one double to 64-bit integer casting can be avoided in the SIGNATURE = "int64" case.

Table 5: Elapsed times in seconds to pass double, integer, and 64-bit integer pointers to vectors of length $2^{28}$ initialized with zeros from R to C using \texttt{.C()} and \texttt{.C64()}. The used INTENT arguments of \texttt{.C64()} are indicated in brackets. Reported are median elapsed times of 100 replicates. The corresponding IQRs are indicated in parentheses.
The function `.C64()` features an openMP implementation of the double to 64-bit integer and 64-bit integer to double castings of `SIGNATURE = "int64"` arguments. Hence, the computational workload of the castings can be distributed to several threads running in parallel. To quantify the performance gain related to using openMP, we control the number of used threads to be between 1 and 10 with the R package `OpenMPController` and measure the elapsed times of the following call:

```r
> .C64("BENCHMARK", SIGNATURE = "int64", a = a, INTENT = "rw", NAOK = TRUE,
+ PACKAGE = "dotCall64", VERBOSE = 0)
```

We let ‘a’ be double vectors of length $2^{16}$, $2^{22}$, $2^{28}$, and $2^{34}$ and performed five replicated timing measurements for each configuration. The results are summarized in Figure 3. The reduction in computation time due to using multiple threads is greatest for the vectors of length $2^{34}$, where using 10 threads reduced the elapsed times by about 70%. Conversely, for the vector of length $2^{16}$ no reduction was observed.

**Summary**

This paper presents the R package `dotCall64`, which provides an alternative to `.C()` and `.Fortran()` from the foreign function interface. In the first section, we introduce R’s interfaces to embed compiled C/C++ and Fortran code. We argue that, in some situations, a `.C()` type interface is more convenient compared to using the C API of R in conjunction with the modern interfaces to C/C++ code. In section two, we motivate the development of `dotCall64` with a discussion of missing features of the foreign function interface and an overview of the R implementation of long vectors. Then, we present the usage and the implementation of the `.C64()` function from the R package `dotCall64`. This is followed by examples demonstrating the capabilities of the new interface—also in comparison with the foreign function interface. Furthermore, we discuss strategies to extend entire R packages with compiled code supporting long vectors. In the last section, we present performance measurements of the `.C64()` interface and benchmark it against `.C()`. This highlights the speed gains achieved by avoiding unnecessary copies of R vectors and by using openMP for casting R vectors. In conclusion, the interface provided by the R package `dotCall64` is an up-to-date version of the foreign function interface including tools to conveniently embed compiled code manipulating long vectors.

**Acknowledgments**

We thank Rafael Ostertag for contributions to Figure 1. We acknowledge the support of the University of Zurich Research Priority Program (URPP) on “Global Change and Biodiversity.”
Bibliography

J. Allaire, R. François, K. Ushey, G. Vandenbrouck, M. Geelnard, and Intel. RcppParallel: Parallel programming tools for ‘Rcpp’, 2016. URL https://CRAN.R-project.org/package=RcppParallel. R package version 4.3.20. [p]

D. Bates and D. Eddelbuettel. Fast and elegant numerical linear algebra using the RcppEigen package. Journal of Statistical Software, 52(5):1–24, 2013. ISSN 1548-7660. doi: 10.18637/jss.v052.i05. URL https://www.jstatsoft.org/index.php/jss/article/view/v052i05. [p]

D. Bates, D. Eddelbuettel, R. François, Y. Qu, and the authors of Eigen for the included version of Eigen. RcppEigen: ‘Rcpp’ integration for the ‘Eigen’ templated linear algebra library, 2016. URL https://CRAN.R-project.org/package=RcppEigen. R package version 0.3.2.9.0. [p]

J. M. Chambers. Software for Data Analysis: Programming with R. Springer, 2008. ISBN 978-0-387-75936-4. URL http://statweb.stanford.edu/~jmc4/Rbook/. [p]

L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. IEEE Computing in Science & Engineering, 5(1):46–55, 1998. ISSN 1070-9924. doi: 10.1109/99.660313. [p]

D. Eddelbuettel. Seamless R and C++ integration with Rcpp. Springer, New York, 2013. ISBN 978-1-4614-6867-7. URL http://www.rcpp.org/book/. [p]

D. Eddelbuettel and R. François. Rcpp: Seamless R and C++ integration. Journal of Statistical Software, 40(8):1–18, 2011. ISSN 1548-7660. doi: 10.18637/jss.v040.i08. URL https://www.jstatsoft.org/index.php/jss/article/view/v040i08. [p]

D. Eddelbuettel and C. Sanderson. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. Computational Statistics and Data Analysis, 71:1054–1063, 2014. doi: 10.1016/j.csda.2013.02.005. [p]

D. Eddelbuettel, R. François, J. Allaire, K. Ushey, Q. Kou, D. Bates, and J. Chambers. Rcpp: Seamless R and C++ integration, 2016a. URL https://CRAN.R-project.org/package=Rcpp. R package version 0.12.6. [p]

D. Eddelbuettel, R. François, and D. Bates. RcppArmadillo: ‘Rcpp’ integration for the ‘Armadillo’ templated linear algebra library, 2016b. URL https://CRAN.R-project.org/package=RcppArmadillo. R package version 0.7.200.2.0. [p]

R. L. Eubank and A. Kupresanin. Statistical Computing in C++ and R. Chapman & Hall/CRC, 2011. ISBN 9781420066500. [p]

R. François and K. Ushey. Rcpp11: R and C++11, 2014. URL https://CRAN.R-project.org/package=Rcpp11. R package version 3.1.2.0. [p]

R. Furrer. spam: SP Arse Matrix, 2016. URL https://CRAN.R-project.org/package=spam. R package version 1.4-0. [p]

R. Furrer and S. R. Sain. spam: A sparse matrix R package with emphasis on MCMC methods for Gaussian Markov random fields. Journal of Statistical Software, 36(10):1–25, 2010. ISSN 1548-7660. doi: 10.18637/jss.v036.i10. URL https://www.jstatsoft.org/index.php/jss/article/view/v036i10. [p]

F. Gerber and R. Furrer. Pitfalls in the implementation of Bayesian hierarchical modeling of areal count data: An illustration using BYM and Leroux models. Journal of Statistical Software, 63(1):1–32, 2015. ISSN 1548-7660. doi: 10.18637/jss.v063.c01. URL https://www.jstatsoft.org/index.php/jss/article/view/v063c01. [p]

F. Gerber, K. Mösinger, and R. Furrer. Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS NDVI3g data. Comput. Geosci., 2016. doi: 10.1016/j.cageo.2016.11.015. [p]

GNU C Library. Reference manual, 2016. URL http://www.gnu.org/software/libc/manual/. [p]

GNU Fortran compiler. Reference manual for GCC version 4.9.2, 2014. URL https://gcc.gnu.org/ondlinedocs/gcc-4.9.2/gfortran/. [p]

GNU sed. Reference manual, 2010. URL https://www.gnu.org/software/sed/manual/. [p]
S. Guest. OpenMPController: Control number of OpenMP threads dynamically, 2013. URL http://CRAN.R-project.org/package=OpenMPController. R package version 0.1-2. [p]

O. Mersmann, C. Beleites, R. Hurling, and A. Friedman. microbenchmark: Accurate timing functions, 2015. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-21. [p]

K. Mössinger, F. Gerber, and R. Furrer. dotCall64: Foreign function interface with long vector support, 2016. URL https://CRAN.R-project.org/package=dotCall64. R package version 0.9-04. [p]

OpenMP architecture review board. OpenMP application program interface, version 4.5, 2016. URL http://www.openmp.org. [p]

M. Plummer. Portable C++ for R packages. The R Journal, 3(2):60–63, 2011. URL http://journal.r-project.org/archive/2011-2/RJournal_2011-2_Plummer.pdf. [p]

R Core Team. Writing R Extensions. R Foundation for Statistical Computing, Vienna, Austria, 2016a. URL http://cran.r-project.org/doc/manuals/R-exts.html. R version 3.3.1. [p]

R Core Team. R Internals. R Foundation for Statistical Computing, Vienna, Austria, 2016b. URL https://cran.r-project.org/doc/manuals/r-devel/R-ints.html. R version 3.3.1. [p]

R Core Team. R News. R Foundation for Statistical Computing, Vienna, Austria, 2016c. URL https://cran.r-project.org/doc/manuals/r-release/NEWS.html. [p]

Y. Saad. Parskit: A basic tool kit for sparse matrix computations, 1994. URL http://www-users.cs.umn.edu/~saad/software/SPARSKIT/index.html. [p]

O. Sklyar, D. Murdoch, M. Smith, D. Eddelbuettel, and R. François. inline: Inline C, C++, Fortran function calls from R, 2016. URL http://CRAN.R-project.org/package=inline. R package version 0.3.14. [p]

H. Wickham. testthat: Get started with testing. The R Journal, 3:5–10, 2011. URL http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf. [p]

H. Wickham. Advanced R. Chapman & Hall/CRC, 2014. ISBN 9781466586963. [p]

Florian Gerber
Department of Mathematics
University of Zurich
8057 Zurich
Switzerland
florian.gerber@math.uzh.ch

Kaspar Mössinger
Department of Mathematics
University of Zurich
8057 Zurich
Switzerland
kaspar.moesinger@gmail.com

Reinhard Furrer
Department of Mathematics and
Department of Computational Science
University of Zurich
8057 Zurich
Switzerland
reinhard.furrer@math.uzh.ch

Appendix: R source code

In the following, we show parts of the C source code of R version 3.3.1 to support the understanding of the long vector implementation. More precisely, the lines 26–377 from the file ‘Rinternals.h’ and the lines 124–191 from the file ‘Rinlinedfuns.h’ are shown in Listing 1 and Listing 2, respectively. The indicated line numbers in the code refer to the actual line numbers of the corresponding file.
Listing 1: ‘R-3.3.1/src/include/Rinternals.h’
# define FREESXP 31 /* node released by GC */
# define FUNSXP 99 /* Closure or Builtin or Special */

#error NOT YET

typedef enum {  
    NILSXP = 0, /* nil = NULL */
    SYMSXP = 1, /* symbols */
    LISTSXP = 2, /* lists of dotted pairs */
    CLOSXP = 3, /* closures */
    ENVSXP = 4, /* environments */
    PROMSXP = 5, /* promises: [un]evaluated closure arguments */
    LANGSXP = 6, /* language constructs (special lists) */
    SPECIALSXP = 7, /* special forms */
    BUILTINSXP = 8, /* builtin non-special forms */
    CHARSP = 9, /* "scalar" string type (internal only)*/
    LISTSXP = 10, /* lists of dotted pairs */
    DOTSP = 11, /* logical vectors */
    REALSXP = 13, /* integer vectors */
    REALSXP = 14, /* real variables */
    CPLXSXP = 15, /* complex variables */
    STRSXP = 16, /* string vectors */
    DOTSXP = 17, /* dot-dot-dot object */
    ANYSP = 18, /* make "any" arg work */
    VECSXP = 19, /* generic vectors */
    EXPSP = 20, /* expressions vectors */
    BCODESXP = 21, /* byte code */
    EXTPTRSXP = 22, /* external pointer */
    NEWSXP = 30, /* fresh node created in new page */
    FREESXP = 31, /* node released by GC */
    FUNSXP = 99 /* Closure or Builtin */
} SEXPTYPE ;

/* These are also used with the write barrier on, in attrib.c and util.c */
#define TYPE_BITS 5
#define MAX_NUM_SEXPTYPE (1<< TYPE_BITS )

/* Flags */

struct sxpinfo_struct {  
    SEXPTYPE type : TYPE_BITS ;/* == > ( FUNSXP == 99) %% 2^5 == 3 == CLOSXP  
            * -> warning: 'type' is narrower than values  
            * of its type  
            * when SEXPTYPE was an enum */
    unsigned int obj : 1;
    unsigned int named : 2;
    unsigned int gp : 16;
    unsigned int mark : 1;
    unsigned int debug : 1; /* functions and memory tracing */
    unsigned int trace : 1; /* functions and memory tracing */
    unsigned int spare : 1; /* currently unused */
    unsigned int gcgen : 1; /* old generation number */
    unsigned int gccls : 3; /* node class */
} ; /* Tot : 32 */

struct vecsxp_struct {  
    R_len_t length ;
    R_len_t truelength ;
} ;

struct primsxp_struct {  
    int offset ;
} ;

struct symsxp_struct {  
    struct SEXPREC * pname ;
    struct SEXPREC * value ;
    struct SEXPREC * internal ;
} ;

struct listsxp_struct {  
    struct SEXPREC * carval ;
    struct SEXPREC * cdrval ;
    struct SEXPREC * tagval ;
} ;

struct envsxp_struct {  
    struct SEXPREC * frame ;
    struct SEXPREC * enclos ;
    struct SEXPREC * hashtab ;
} ;

struct closxp_struct {  
    struct SEXPREC * formals ;
    struct SEXPREC * body ;
    struct SEXPREC * env ;
} ;

struct promsxp_struct {  
    struct SEXPREC * value ;
    struct SEXPREC * expr ;
    struct SEXPREC * env ;
} ;

/* Every node must start with a set of sxpinfo flags and an attribute  
field. Under the generational collector these are followed by the  
fields used to maintain the collector’s linked list structures. */

/* Define SWITH_TO_REFCNT to use reference counting instead of the  
NAMED mechanism. This uses the R-devel binary layout. The two  
‘named’ field bits are used for the REFCNT, so REFCNTMAX is 3. */

The R Journal Vol. XX/YY, AAAA 20ZZ  
ISSN 2073-4859
The R Journal Vol. XX/YY, AAAA 20ZZ
ISSN 2073-4859
# define IS_LONG_VEC (x) 0
#define IS_SCALAR (x, type) (TYPEOF (x) == (type) && LENGTH(x) == 1)
#endif

Listing 2: ‘R-3.3.1/src/include/Rinlinedfuns.h’

INLINE_FUN R_len_t length(SEXP s)
{
    switch (TYPEOF(s)) {
    case NILSXP:
        return 0;
    case LGLSXP:
    case INTSXP:
    case REALSXP:
    case CPLXSXP:
    case STRSXP:
    case CHARSXP:
    case VECSXP:
    case EXPRSXP:
    case RAWSXP:
        return LENGTH(s);
    case LISTSXP:
    case LANGSXP:
    case DOTSPX:
    { int i = 0;
        while (s != NULL && s != R_NilValue) {
            i = 0;
            s = CDR(s);
        }
        return i;
    }
    case ENVXSP:
        return RF_envlength(s);
    default:
        return 1;
    }
}

R_xlen_t Rf_envxlength(SEXP rho);

INLINE_FUN R_xlen_t xlength(SEXP s)
{
    switch (TYPEOF(s)) {
    case NILSXP:
        return 0;
    case LGLSXP:
    case INTSXP:
    case REALSXP:
    case CPLXSXP:
    case STRSXP:
    case CHARSXP:
    case VECSXP:
    case EXPRSXP:
    case RAWSXP:
        return XLENGTH(s);
    case LISTSXP:
    case LANGSXP:
    case DOTSPX:
    { R_xlen_t i = 0;
        while (s != NULL && s != R_NilValue) {
            i = 0;
            s = CDR(s);
        }
        return i;
    }
    case ENVXSP:
        return Rf_envxlength(s);
    default:
        return 1;
    }
}