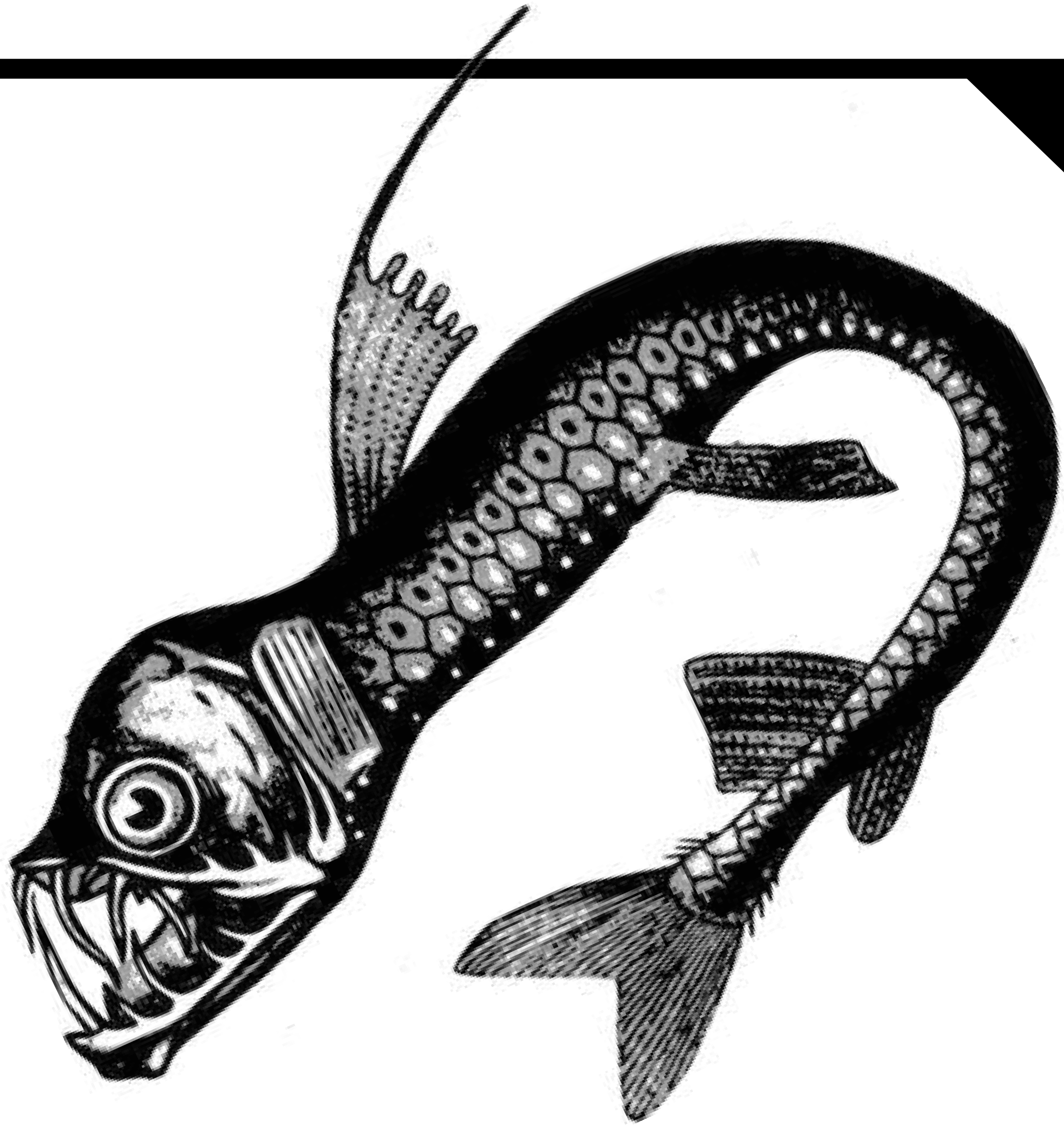
Version 0.1.0



Guide to the SEXPtools Package

A Simplified C interface for R

Drew Schmidt

Guide to the **SEXPtools** Package

A simplification to R's native C interface

June 25, 2014

Drew Schmidt
`wrathematics@gmail.com`

Version 0.1.0

Acknowledgement

We thank Wei-Chen Chen for his numerous contributions and improvements to the package configuration/build, particularly with respect to the Windows platform.

We thank Christian Heckendorf for helping with this package's general configuration issues.

Contents

1 Introduction

In this vignette, we will introduce the **SEXPtools** package, including its motivations and purpose, how to install it, its interface specifications, and a detailed example.

1.1 Purpose

This package is intended to serve somewhat the same purpose as the very (deservedly!) popular package **Rcpp**. However, this package is not meant to be a competitor to **Rcpp**. Rather, it is meant to fill a very small niche that **Rcpp** does not fill.

1.2 License

The **SEXPtools** package is licensed under the very permissive 2-clause BSD license, commonly referred to as the FreeBSD license. For a copy of the license, see the file named `LICENSE` in the root directory of the package source.

1.3 Installation

The package should install without issue from the command line via the usual commands:

Shell Command

```
R CMD INSTALL SEXPtools_0.1-0.tar.gz
```

2 Linking with SEXPtools

2.1 Configuring a Package to use SEXPtools

In your `configure.ac` and/or `src/Makevars` file(s), you can get the package linking and include information via:

```
R_SCMD="${R_HOME}/bin/Rscript -e"

SEXPTOOLS_LDFLAGS=`${R_SCMD} "SEXPtools:::ldflags()"`
SEXPTOOLS_CPPFLAGS=`${R_SCMD} "SEXPtools:::cppflags()"`
```

and adding `$(SEXPTOOLS_LDFLAGS)` to `PKG_LIBS` and `$(SEXPTOOLS_CPPFLAGS)` to `PKG_CPPFLAGS`. See the Writing R Extensions manual for more information. You can also see the **pbdBASE** and **pbdDMAT** packages as examples.

2.2 Testing the Configuration

To ensure that the package configuration is correct, you can use this test code. Include this C file:

sexptools_test.c

```
#include <SEXPtools.h>

SEXP sexptools_test(SEXP mat)
{
        PRINT(mat);

        return RNULL;
}
```

and this R file:

sexptools_test.r

```
sexptools_test <- function() .Call("sexptools_test",
   matrix(1:30, 10))
```

Then build your package with the usual `R CMD INSTALL` and test by loading R and running:

```
library(<my package>)

sexptools_test()
```

3 Example Usage

3.1 Basic Example

Suppose you have a C function `int fib(int n)` which produces the n'th Fibonacci number:

Fibonacci

```
int fib(n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-1);
}
```

To wrap this for R with **SEXPtools**, you

Fibonacci Wrapper

```c
#include <SEXPtools.h>

SEXP fib_wrap(SEXP n)
{
  R_INIT;
  SEXP ret;
  newRvec(ret, 1, "int");

  INT(ret) = fib(INT(n));

  R_END;
  return ret;
}
```

This may look verbose — and really it is — but for complicated examples, the tools begin to shine. So far, what we have seen isn't really "better", just different (although I obviously prefer it). The convenience offered by **SEXPtools** begins to show itself when we deal with managing lists, however.

Suppose we wanted, for example, to return both n and the n'th Fibonacci number in a list. In R's native C interface, this is a labyrinthine nightmare, but managing return lists is very simple with **SEXPtools**:

Fibonacci Wrapper 2

```c
#include <SEXPtools.h>

SEXP fib_wrap(SEXP n)
{
  R_INIT;
  SEXP R_list, R_list_names, nth_fib;
  newRvec(nth_fib, 1, "int");

  INT(nth_fib) = fib(INT(n));

  R_list_names = make_list_names(2, "n", "nth.fib");
  R_list = make_list(R_list_names, 2, n, nth_fib);

  R_END;
  return R_list;
}
```

You probably know that really I should be instituting a copy of n here; this is true of **Rcpp** and the pure native C interface as well.

Note that `INT(n)` is a shorthand for `INT(n, 0)`, which is itself shorthand for R's `INTEGER(n)[0]`. Other values may be substituted, e.g. `INT(n, i)` for `INTEGER(n)[i]`. See Section 4 for more information.

A native version of this simple example (without **SEXPtools**) might look like:

Fibonacci Wrapper 2 - Native Interface

```c
#include <R.h>
#include <Rinternals.h>

SEXP fib_wrap(SEXP n)
{
  SEXP R_list, R_list_names, nth_fib;
  PROTECT(nth_fib = allocVector(INTSXP, 1));

  INTEGER(nth_fib)[0] = fib(INTEGER(n)[0]);

  PROTECT(R_list = allocVector(VECSXP, 2));
  PROTECT(R_list_names = allocVector(STRSXP, 2));

  SET_VECTOR_ELT(R_list, 0, n);
  SET_VECTOR_ELT(R_list, 1, nth_fib);

  SET_STRING_ELT(RET_NAMES, 0, mkChar("n"));
  SET_STRING_ELT(RET_NAMES, 1, mkChar("nth.fib"));

  UNPROTECT(3)
  return R_list;
}
```

4 Specification

If you are familiar with R's native C interface already, then you may simply wish to view `src/SEXPtools/SEXPtools.h` of the **SEXPtools** source tree.

4.1 GC and Allocation

Instead of explicitly calling `UNPROTECT` on the number of `PROTECT`'d objects, start every `SEXP` function with `R_INIT` and end every `SEXP` function with `R_END`:

GC Counter

```c
R_INIT; // Initialize the GC counter

R_END; // Call UNPROTECT on the appropriate number
```

Additionally, you don't need to call `PROTECT` anymore on new allocations; simply call `newR*` for the type of R object you want to construct:

Allocation

```
SEXP x;
int m, n;

// Construct R list object
newRlist(x, n);                                      // SEXPtools
PROTECT(x = allocVector(VECSXP, n));                 // Native equiv.

// Construct numeric R vector with C-type 'type'
newRvec(x, n, type);                                 // SEXPtools
PROTECT(x = allocVector(<SEXPTYPE>, n));             // Native equiv.

// Construct numeric R matrix
newRmat(x, m, n, type);                              // SEXPtools
PROTECT(x = allocMatrix(<SEXPTYPE>, m, n));          // Native equiv.
```

Allocation Examples

```
// A length 10 integer vector
newRvec(x, 10, "int");                               // SEXPtools
PROTECT(x = allocVector(INTSXP, 10));                // Native equiv.

// A 5 by 2 integer matrix
newRmat(x, 5, 2, "int");                            // SEXPtools
PROTECT(x = allocMatrix(INTSXP, 5, 2));             // Native equiv.
```

If you need to protect something you aren't allocating, use `PT()` instead of `PROTECT()`; this will automatically increment the counter which keeps track of the number of `PROTECT`'d objects.

4.2 Accessing SEXP Data

SEXPtools Data Accessors

```
SEXP x;
int i, j;

// Pointer to data
INTP(x)
DBLP(x)

// Vector data
INT(x)
INT(x, i)
DBL(x)
DBL(x, i)
STR(x)
STR(x, i)

// Matrix data
MatINT(x, i, j)
MatDBL(x, i, j)
```

Native Data Accessors

```
SEXP x;
int i, j;

// Pointer to data
INTEGER(x)
REAL(x)

// Vector Data
INTEGER(x)[0]
INTEGER(x)[i]
REAL(x)[0]
REAL(x)[i]
(char*)CHAR(STRING_ELT(x,0))
(char*)CHAR(STRING_ELT(x,i))

// Matrix data
INTEGER(x)[i+nrows(x)*j]
REAL(x)[i+nrows(x)*j]
```

4.3 Lists and Dataframes

For this section, only the **SEXPtools** version is provided, because the native equivalent is too horrifying to mention.

Lists

```
SEXP R_list, R_list_names;
const int num.items = 3;

// pretened we creates SEXP's item1, item2, and item3

R_list_names = make_list_names(num.items, "name1", "name2",
   "name3"); // as many as you like
R_list = make_list(R_list_names, num.items, item1, item2, item3);
```

This dataframe example uses R default row and column names. You can probably fill in the details from the above for the named case.

Dataframes

```
SEXP R_df;
const int num.cols = 2;
```

```
// pretened we creates SEXP's col1 and col2

R_df = make_dataframe(RNULL, RNULL, num.names, col1, col2);
```

4.4 Utility Functions

Testers

```
int is_Rnull(SEXP x);
int is_Rnan(SEXP x);
int is_Rna(SEXP x);
int is_double(SEXP x);
int is_integer(SEXP x);
```

Misc

```
RNULL                                               // SEXPtools
R_NilValue                                          // Native equiv.

// Print any SEXP with R's printing
SEXP x;
PRINT(x)
```

5 Q&A

5.1 Why make this?

Probably my biggest motivator was fun; I wanted to. Another, more pragmatic reason is that part of my workload prevents me from using **Rcpp**. I deal a lot with C and Fortran; trying to integrate C++ into that mess is not fun, and so for me, **Rcpp** is more of a burden than a savior. This leaves me stuck with the native C interface for R. And I don't like R's native C interface. This package is my attempt to make that interface (slightly) more friendly and convenient to work with.

5.2 Why the strange name?

Every R object (underneath, in the C interface) is an `SEXP` (short for S-expression) object, which is a struct pointer. This is explained in the R Internals manual. This package is a collection of tools for more easily managing SEXP objects.

5.3 Is this now, or will this ever be a competitor to Rcpp?

In terms of features, no. **Rcpp** is very good at aiding the writing of new code, or translating R code into compiled code. That is not a goal of this package.

5.4 How does this differ from Rcpp?

Each of these packages makes an attempt at solving a serious problem with utilizing compiled code from R: the native interface for C code in R sucks. There are huge differences between the two packages, however. In short, **Rcpp** is *much* a much more comprehensive solution. If you are new to using compiled code with R, frankly this package probably is not for you; you would likely be much better served by **Rcpp**.

SEXPtools is primarily meant for those who are working in pure C, writing C/Fortran code and wrapping it back into R. This makes the wrapping a bit simpler. It does not make the translation of R code into compiled code any simpler, which is one of **Rcpp**'s main goals.

SEXPtools is a pure C solution. **Rcpp** is a C++ solution. In the author's opinion, C++ brings a lot of needless complications and headaches if you aren't actually using C++; that is, if you are only writing C and Fortran, then bringing in C++ **will** create problems.

Another important difference is that **SEXPtools** is more permissively licensed than **Rcpp** (BSD rather than GPL). If this issue is important to you but you live in the C++ world, you may be interested in Romain Francois' new **Rcpp11** package.

5.5 Why would I want to use this package?

If spend a lot of time bringing code to R (especially if you deal with lists and dataframes), this package offers a lot of convenient shorthand for doing so.

5.6 Is this really easier than R's native interface?

It is for me, no question; notably, returning lists and dataframes is *much* less painful through some `stdarg` sorcery. Most of the rest of the package is minor cosmetic things; but the package is very permissively licensed, so feel free to pick and choose what you want, however you want.

5.7 How would I use SEXPtools in a package?

Assuming that you have some compiled code you have or want to create to use with a package, you simply link with **SEXPtools** and then wrap that compiled code with the utilities provided by **SEXPtools**. For the former, see Section 2, and for the latter, see Section 4 and Section 3. For actual package examples using **SEXPtools**, see **pbdBASE** version $\geq$ 0.3-0, **pbdDMAT** version $\geq$ 0.3-0, and **memuse** version $\geq$ 2.0.

Philosophically, you should never have the bulk of the work of a function (of any importance) be handled by the R interface (including **SEXPtools**'s version of it). If you do, then your code can never (easily) have a life outside of R. That may sound fine to you now, but if you ever decide to take some of your work outside of R, then you can't easily take your compiled code with you. This is just bad practice and shortsightedness.