Software Manual
Institute of Bioinformatics, Johannes Kepler University Linz

Rchemcpp: An R package for computing the similarity of molecules
— *Technical Report* —

Michael Mahr and Günter Klambauer
Institute of Bioinformatics, Johannes Kepler University, Linz, Austria

March 13, 2013

Institute of Bioinformatics
Johannes Kepler University Linz
A-4040 Linz, Austria

Tel. +43 732 2468 8880
Fax +43 732 2468 9511
http://www.bioinf.jku.at

Contents

1 Overview

Rchemcpp is an R-package which wraps the main functionality of the C++ class-library 'chemcpp'. 'chemcpp' is available under an LGPL 2 license from sourceforge.net. It allows comparing molecule files against itself or against other molecule files by different algorithms whilst creating comparison matrices. Rchemcpp wraps this functionality by making it available as R functions. The R-package is simple to install, since the 'chemcpp' library is built as part of the package. It is portable in that it can also be built on Windows-systems by using the 'Rtools' installable. The binary Windows-package is automatically built at CRAN and readily available from there.

2 Design considerations - Wrapping an existing portable C or C++ library into an R-package

There are several possibilities of how to wrap a library into an R-package. Some of them are explained below:

2.1 Pre-installed executable

The simplest way to wrap a software with a command-line interface is to force the user to install the software on the computer. This might just be as simple as letting the user specify the location of the software-binary at the R function-call and writing/reading the input/output files to a temporary directory. Since this is not very simple, flexible, fast or portable, we will not discuss this possibility further.

2.2 Configuring and building a source-tree inside an R package during its configuration

Running a standard GNU configure-script is theoretically possible, however fails due to some shortcomings of 'Rtools', because of it being an incomplete subset of 'cygwin' executables mixed with a compiler from 'MinGW'. This means that on Unix, configuring a software inside an R-package will work fine, however configuring the same software inside an R-package on Windows+'Rtools' will most likely not work.

E.g. the 'chemcpp' './configure'-script fails, because it requires the 'chmod' executable, which 'Rtools' does not include. Also the 'Rtools' commands do not correctly handle all unix-style paths, e.g. as needed for writing to the '/tmp' directory.

The solution would be to install 'MSYS' and put the path to the 'Rtools' '/gcc'-directory when the 'MSYS'-installer asks for a 'MinGW' installation. (e.g. 'C:/Rtools/gcc-4.6.3') Almost every standard Unix software compiles marvelously fine this way on Windows. The 'configure' script of the R-package can simply invoke './configure' on a subdirectory of the package which contains an unchanged Unix software library (e.g. 'chemcpp'). Then the file 'makevars' can invoke 'make' on the subdirectory to build the software inside the package. Finally the standard R->c interfaces can be used to make the functionality of the C/C++ libraries available to R. With this constellation, R-packages can compile most Unix-software inside of them(!)

The disadvantage of this attempt is that the 'MSYS'-tools are designed to run inside an 'MSYS' shell. (not the standard Windows 'cmd' shell.)

The resulting problems are:

1. The packages cannot be compiled outside the 'MSYS' shell on Windows.
2. Building packages under the 'MSYS' shell is not explicitly supported by 'Rtools'. (Many packages do not build well with R CMD INSTALL that way.)
3. 'R for Windows' and 'Rtools' would need to be adjusted to compile packages properly when run from inside the 'MSYS' shell.
4. CRAN does not auto-compile packages to Windows-binaries with this constellation ('MSYS' shell).

Since CRAN has a policy to not host Windows-binaries which have not been built by themselves, this is the show-stopper.

This means: It is a solution to easily integrate Unix-software into an R-package with Windows-compatibility. Currently, however, this is not an option if the package is supposed to be made available via CRAN and the package is intended to support the Windows platform too.

(Hopefully in the far future, R for Windows will have an option to install packages into a 'MinGW' + 'MSYS' shell environment. This would allow deprecating the 'Windows-binary-package' concept. However this would probably also introduce cross-platform problems between old and new packages, as well as inconveniences, such as always having unix-style path-names in the package-environment.)

2.3 Copying all library source-code to '/src'

If the license of the software allows it, one might copy all source-files into the '/src' folder of the R- package. This will most likely require changes to the code-files, because of the missing source-code configuration and therefore is a one-way trip: If a future version of the software appears, all steps need which were done to get it into the '/src' directory need to be repeated. This is especially complicated, if the software source-code consists of cascaded directories which would need to be arranged to a single directory.

2.4 Creating a custom configuration-script or makefile for a source-tree inside an R package

Since executing configure-scripts from foreign software does not always work on Windows+'Rtools', one can attempt to drop the configure-script and makefile of the software and write a custom one. The configure-script and makefile are basically rewritten into the 'makevars' file of the R-package. Again care must be taken that the script runs under Unix as well as Windows+'Rtools'. The standard package-skeleton which R generates shows a simple way of how to use the 'config' and 'config.win' script to substitute expressions in the 'makevars' file. (This information is written in the comments of the 'makevars' file when generating a package skeleton.)

The lack of documentation around the possibilities of 'makevars' in this case is the biggest problem. The most simple way to overcome this shortcoming is to look at the 'makevars'-file of existing package, such as the 'Matrix' package.

Nevertheless this option of integrating 'chemcpp' was chosen.

Since 'Rtools' includes a version of 'make', a makefile was created in the source-subdirectory of the 'chemcpp' software source which compiles the sources to a .so or .dll. The Makefile of 'chemcpp' is invoked by the code in the 'makevars' file as shown below:

```
PKG_LIBS = `$(R_HOME)/bin/Rscript -e "Rcpp:::LdFlags()"` -L./ -lchemcpp
  -Wl,-rpath,$(R_PACKAGE_DIR)/libs

PKG_CPPFLAGS = -I./chemcpp/src

all: sublib

sublib:
      @(cd chemcpp/src && CFLAGS="$(CFLAGS)" CXXFLAGS="$(CXXFLAGS)"
  MkInclude="$(MkInclude)" $(MAKE) all) || exit 1;
```

'chemcpp' specific source-code adjustments

It was necessary to adjust the 'chemcpp' library code in order to make it suitable for using it inside an R-package:

1. 'chemcpp' contains a data-directory which is defined as a C preprocessor macro. This macro is usually set by the user before compiling by editing the master header-file of the software. Statically patching this directory is not an option, because the location of the binary R package might change (especially on Windows systems). So the solution was to patch 'chemcpp' by replacing the preprocessor-macro with a different preprocessor-macro which is a function call. This function call uses the standard C++ library to read an environment-variable. This environment-variable is set in R-package 'zzz.R' before the actual loading of the shared library takes place.
2. 'chemcpp' contains a trivial bug which occurs on some compilers. The solution was simply to add an include-macro at the top of one source-file of 'chemcpp'.

Both modifications, as well as copying the custom Makefile into the 'chemcpp' source-tree and stripping 'chemcpp' from all unneeded files are performed by three small scripts in the 'source_prep' folder of the R package.

- del_chemcpp.sh: deletes the 'src/chemcpp'-directory and the data-files (cleans the R-package from chemcpp)
- prep_chemcpp_1.0.2.sh: downloads 'chemcpp', patches it and copies the data-files into the right folders

- strip_chemcpp_1.0.2.sh: removes all unnecessary files from 'chemcpp' which are not required by the R-package

3 Linking / Packaging

Possibility 2 and 4 create a .so or .dll file which is moved or copied to the '/src' directory after the build. This library is then automatically packaged by R. The compiler-options in the 'makevars' file have to be adjusted to show the linker where it finds the .so or .dll file. (Just like as linking with the Rcpp library)

4 R user interface for 'chemcpp'

'chemcpp' is special in that the source-code of it contains a shared C++ library and a large set of C++ command-line executables ('tools') which use this library.

Most users of 'chemcpp' only use these command-line tools in order to work with 'chemcpp'. This is why the main focus of the R package was to give the R user access to the funtionality of these tools. Allowing the R user to use the 'chemcpp' library directly was only an optional goal.

There are several ways to interface with C/C++ code from R:

4.1 .C calls

This is the most basic possibility of how to interface with C. Since 'chemcpp' is C++ and heavily uses classes, it would have been necessary to write wrapper- functions in c which entirely encapsulate the functionality of the tools. Since the tools directly parse the command-line, they would need to be rewritten (also in c), so that they can be called from R via '.Call'. Since the original executables write the output to a file for further processing, this output part would need to be rewritten to pass back the matrices to R. This nevertheless would have been a very good option, however this would also have meant that the user cannot use the functionality of the 'chemcpp' library directly.

4.2 Rcpp Modules - function calls

Rcpp Modules allow the programmer to avoid 'glue code' by specifying prototypes of c functions that are to be called from R in a special format. This makes implementation a bit cleaner, however does not give the user of the R-package an advantage over the '.Call' version.

4.3 Rcpp Modules - exposing classes

The second possibility of Rcpp Modules is to specify prototypes of entire C++ classes for exporting them to R. This allows the user to instantiate C++ objects from inside R via the 'new' operator. Unfortunately this is not possible for every C++ class: If the class contains functions which return a C+ + pointer, then Rcpp Modules does not know how it should deal with it. It is however possible to write a wrapper around a C++ function to make this possible:

```
void setComparisonSetCopy(SEXP s)
{
  delete comparisonSet;
  std::string rtypename("Rcpp_Rmoleculeset");
  Rcpp::S4 s4obj( s );
  if ( !s4obj.is( rtypename.c_str() ) ) {
    Rf_error((std::string("object is not of type ")+rtypename).c_str());
  }
  Rcpp::Environment env( s4obj );
  Rcpp::XPtr<Rmoleculeset> xptr( env.get(".pointer") );
  Rmoleculeset *o = static_cast<Rmoleculeset*> (R_ExternalPtrAddr(xptr));

  //Note: It could already be assigned here,
  //  however it is cloned to avoid scope/garbage-coll. issues
  //MoleculeSet::setComparisonSet(o);

  //invoke copy-constructor to clone object
  MoleculeSet::setComparisonSet(new Rmoleculeset(o));
}

SEXP getComparisonSetPointer()
{
  //do NOT mark as finalizable!
  Rcpp::XPtr<Rmoleculeset> xp((Rmoleculeset*)comparisonSet, false);
  Rcpp::Function maker =
    Rcpp::Environment::Rcpp_namespace()["cpp_object_maker"];
  return maker ( typeid(Rmoleculeset).name() , xp );
}
```

This however should only be done with utmost care: If an R program which uses this function stores a pointer to a C++ object which in the meantime has been deleted by C++, accessing it in R will lead to unpredictable results or segmentation faults. This brings C++ paradigms to R programs which force the user to consider such situations. This should be avoided.

All this made it necessary to derive C++ classes from the original 'chemcpp' classes. The functions in the derived classes overload or wrap the original functions. They are then exposed as member functions to R via Rcpp Modules.

In order to avoid introducing C++ paradigms, wrapper-functions which take references to objects as arguments were changed so that they duplicate the object before storing it into a member variable. Functions which return a pointer to an object were renamed in order to make it obvious that the function returns a pointer which might not be valid for long.

The tools which are most likely to be invoked by the user were reimplemented in R. These functions use the Rcpp Modules classes of Rchemcpp to access the functionality of the 'chemcpp' library behind. If the user requires it, he/she however can access the functionality of the 'chemcpp' directly from R. Please note however that only a small portion of the functionality of the 'chemcpp'

library was wrapped yet, such as the functions which are required by the five comparison-tools, as well as the functions which are necessary for creating a molecule from a vector of atoms and a matrix of bonds.

5 Usage

The package can be installed from CRAN by issuing the following commands from inside an R session:

```
install.packages("Rchemcpp")
library("Rchemcpp")
```

Please refer to the documentation inside the package for information on its functionality. The functionality that most users want to use are the five functions for generating kernel matrices which have their name starting with 'sd2gram'.