A problem arises when building R interfaces to C/C++ libraries involves testing: how to go about replicating the existing C/C++ tests in R without undue effort. If the C/C++ tests are simple and small enough, they can be manually translated. However, when there are many tests, and each test initializes its own large data structures, the task becomes a chore.

This problem faced with a the ECOSolveR, a solver package crucial to our larger package CVXR. Until version 0.4, we had been content with including one small test and a larger one using saved RDA files in the R package. But with our work on CVXR moving towards a version 1.0 release, we wanted to batten down the hatch.

The ECOS C library has about 28 tests and many of them include large, initialized arrays as test data.

The initial thought was to parse out the arrays in the C source and write out R equivalents for testing. But that could be error-prone and if a test failed, we could never really be sure that our translation was not a problem.

So we looked around for solution that would let us create R data structures from within the C test code.

**Enter RInside**

RInside allows one to embed R inside C/C++.

To use RInside, one initializes a handle to the R process in the C/C++ program. This handle can be used to assign R variables to values held in C/C++ scalars, arrays, etc. One can also evaluate arbitrary strings using the R process, which is quite handy: we can use it to execute a saveRDS call to save R structures in a file. The C++ snippet below shows a simple example.

// test.cpp

#include

// Stuff a double array in a std::vector

std::vector< double > dVec(double \*data, int len) {

std::vector< double > result;

for (int i = 0; i < len; i++) result.push\_back(data[i]);

return(result);

}

#define DLEN(x) x? (int) (sizeof(x) / sizeof(double)) : 0

int main(int argc, char\* argv[]) {

int n = 1;

double y[] = {1.0, 2.0, 3.0};

RInside R(argc, argv);

R.assign(n, "n"); // assign R variable n to C scalar n

R.assign(dVec(y, DLEN(y)), "y"); // assign R variable y to C vector y

std::string rds\_file = "out.RDS";

R.assign(rds\_file, "rds\_file"); // assign R variable rds\_file the value "rds\_file"

R.parseEvalQ("saveRDS(list(n = n, y = y), file = rds\_file)");

return 0;

}

Note the use of some macros to determine lengths of initialized C arrays; in particular, we account for the fact that the array pointer could be NULL, common in the code we encounter. Not shown here is a similar macro that can be used for an initialized integer array. (Dynamically allocated arrays pose no difficulty since the lengths would be known.) Such macros are made accessible to the C/C++ code via included headers.

The following features of the ECOS C tests make it possible to exploit RInside.

* Each C test is in a single source file. One exception has five tests in a single file.
* Each C test has a pattern: a call to a *setup* function to set up the data, followed by an actual call to the solver.
* Most (but not all) arrays are initialized in the C source.
* The types of the variables in the setup function call are fixed.
* The tests are all invoked from a single test harness.

This suggests a following strategy.

1. Create an R process in the C library testing harness.
2. Modify each C test source file by inserting RInside calls *before* calling the setup function. These calls will save the C data in R data structures, and export the R data using a call saveRDS.
3. Rerun the C library tests to generate the RDS files.
4. Use the generated RDS files in R package tests.

**Example**

Taking as an example, the setup call (near the bottom of the file) has the following form:

mywork = ECOS\_setup(MPC01\_n, MPC01\_m, MPC01\_p, MPC01\_l, MPC01\_ncones, MPC01\_q, 0,

MPC01\_Gpr, MPC01\_Gjc, MPC01\_Gir,

MPC01\_Apr, MPC01\_Ajc, MPC01\_Air,

MPC01\_c, MPC01\_h, MPC01\_b);

And since the signature of ECOS\_setup is known, the parameter types (scalars or arrays) can be fixed.

var\_type <- c(n = "int", m = "int", p = "int", l = "int", ncones = "int",

q = "int\*", e = "int",

Gpr = "double\*", Gjc = "int\*", Gir = "int\*",

Apr = "double\*", Ajc = "int\*", Air = "int\*",

c = "double\*", h = "double\*", b = "double\*")

**Extracting C variable names**

A simple string match in R can easily identify the starting and ending lines of the C setup invocation in the test source file and return a list of the C variable names in each call.

get\_setup\_vars <- function(source, file\_source = TRUE) {

n <- length(lines <- if (file\_source) readLines(source) else source)

starts <- grep("ECOS\_setup", lines)

potential\_ends <- grep("\\);$", lines)

ends <- sapply(starts,

function(i) potential\_ends[potential\_ends >= i][1])

## starts[i]:ends[i] are chunks of interest.

lapply(seq\_along(starts),

function(i) {

lines[starts[i]:ends[i]] %>%

paste(collapse = "") %>%

stringr::str\_replace(pattern = "(.\*ECOS\_setup\\()", replacement = "") %>%

stringr::str\_replace(pattern = "(\\);)$", replacement = "") %>%

stringr::str\_split(pattern = ",") %>%

magrittr::extract2(1) %>%

stringr::str\_trim()

})

}

A test invocation.

source\_lines <- c('mywork = ECOS\_setup(MPC01\_n, MPC01\_m, MPC01\_p, MPC01\_l, MPC01\_ncones, MPC01\_q, 0,',

'MPC01\_Gpr, MPC01\_Gjc, MPC01\_Gir,',

'MPC01\_Apr, MPC01\_Ajc, MPC01\_Air,',

'MPC01\_c, MPC01\_h, MPC01\_b);')

get\_setup\_vars(source\_lines, file\_source = FALSE)

[[1]]

[1] "MPC01\_n" "MPC01\_m" "MPC01\_p" "MPC01\_l"

[5] "MPC01\_ncones" "MPC01\_q" "0" "MPC01\_Gpr"

[9] "MPC01\_Gjc" "MPC01\_Gir" "MPC01\_Apr" "MPC01\_Ajc"

[13] "MPC01\_Air" "MPC01\_c" "MPC01\_h" "MPC01\_b"

**Inserting C code**

The next task is to generate the code to be inserted before the call to setup. The following code does the job, taking the types of each of the variables in the setup call and making use of macros shown in the C++ example above.

#' Assign a variable in R

#' @param x C variable name

#' @param r\_name R variable name

set\_val <- function(x, r\_name) paste0('R.assign(', x, ', "', r\_name, '");')

set\_ivec <- function(x, r\_name) paste0('R.assign(iVec(', x, ', ILEN(', x, ')), "', r\_name, '");')

set\_dvec <- function(x, r\_name) paste0('R.assign(dVec(', x, ', DLEN(', x, ')), "', r\_name, '");')

#' Generate C++ code lines for insertion into C test source

#' @param c\_name a vector of C variable names that should be saved in R

#' @param a named list of variable types

#' @param rds\_file a string naming the rds file for saving the list object

gen\_cpp <- function(c\_name, var\_type, rds\_file) {

r\_name <- names(var\_type)

result <- sapply(seq\_along(var\_type),

function(i) {

if (var\_type[i] == "int" || var\_type[i] == "double") {

set\_val(c\_name[i], r\_name[i])

} else if (var\_type[i] == "int\*") {

set\_ivec(c\_name[i], r\_name[i])

} else if (var\_type[i] == "double\*") {

set\_dvec(c\_name[i], r\_name[i])

} else {

stop("Unknown variable type")

}

})

## Create list to save:

output <- paste("foo <- list(",

paste(sapply(r\_name, function(x) paste(x, "=", x)), collapse=", "),

")")

## Set output file name and insert call to R saveRDS

c(result,

paste0("std::string fname = ", paste0('"', rds\_file, '"'), ";"),

set\_val('fname', 'fname'),

paste0('R.parseEvalQ("', output, '");'),

paste0('R.parseEvalQ("saveRDS(foo, file=fname)");'))

}

**Does it work?**

We can check that the appropriate C code is generated for inserting into the file.

c\_name <- get\_setup\_vars(source\_lines, file\_source = FALSE)[[1]]

gen\_cpp(c\_name, var\_type, "foo.rds")

[1] "R.assign(MPC01\_n, \"n\");"

[2] "R.assign(MPC01\_m, \"m\");"

[3] "R.assign(MPC01\_p, \"p\");"

[4] "R.assign(MPC01\_l, \"l\");"

[5] "R.assign(MPC01\_ncones, \"ncones\");"

[6] "R.assign(iVec(MPC01\_q, ILEN(MPC01\_q)), \"q\");"

[7] "R.assign(0, \"e\");"

[8] "R.assign(dVec(MPC01\_Gpr, DLEN(MPC01\_Gpr)), \"Gpr\");"

[9] "R.assign(iVec(MPC01\_Gjc, ILEN(MPC01\_Gjc)), \"Gjc\");"

[10] "R.assign(iVec(MPC01\_Gir, ILEN(MPC01\_Gir)), \"Gir\");"

[11] "R.assign(dVec(MPC01\_Apr, DLEN(MPC01\_Apr)), \"Apr\");"

[12] "R.assign(iVec(MPC01\_Ajc, ILEN(MPC01\_Ajc)), \"Ajc\");"

[13] "R.assign(iVec(MPC01\_Air, ILEN(MPC01\_Air)), \"Air\");"

[14] "R.assign(dVec(MPC01\_c, DLEN(MPC01\_c)), \"c\");"

[15] "R.assign(dVec(MPC01\_h, DLEN(MPC01\_h)), \"h\");"

[16] "R.assign(dVec(MPC01\_b, DLEN(MPC01\_b)), \"b\");"

[17] "std::string fname = \"foo.rds\";"

[18] "R.assign(fname, \"fname\");"

[19] "R.parseEvalQ(\"foo <- list( n = n, m = m, p = p, l = l, ncones = ncones, q = q, e = e, Gpr = Gpr, Gjc = Gjc, Gir = Gir, Apr = Apr, Ajc = Ajc, Air = Air, c = c, h = h, b = b )\");"

[20] "R.parseEvalQ(\"saveRDS(foo, file=fname)\");"

This C code can be inserted before the setup call in each test file in an automated way.

**Summary**

RInside can be part of the solution for generating R package tests based on underlying C/C++ library tests. The above approach, with some minor modifications, enabled us to reprogram all 28 C tests for our R package.

Once the modifications were inserted into the C source files, the C tests were re-run to generate RDS files now included in ECOSolveR version 0.5.