PhUSE 2015

Paper TS04

EmpoweR your Application with R!

Sudipta Basu, Cytel Statistical Software & Services Pvt. Ltd., Pune, India
Pravin Holkar, Cytel Statistical Software & Services Pvt. Ltd., Pune, India

ABSTRACT

The coverage of any software program is limited to the scope of its specifications. Its features are fixed and frozen by the time it is released to the user. What if we can make it so dynamic that its scope can be customized by the user, in ways that the specifications might not have even imagined? One way of doing it is to allow user to execute custom scripts at specific points in the software and modifying its behavior based on the scripts' output.

R, one of the most widely recognized statistical applications, has an API which is available to C programmers. This paper will explain the basics of this interface and demonstrate its use through the C language with some examples. In particular, we will show how we can enrich a program that simulates a clinical trial using R functions.

INTRODUCTION

Consider the following three scenarios:

1. We have an application which simulates an oncology trial, which generates time to event data using the exponential distribution. But a user wants to generate data using some other distribution instead, say Weibull which incidentally is not available in the software. What if we had a way to do this without modifying the application code?

2. An application provides various types of output including some plots and tables. But a user is looking for some more plots that are not available in the application. How can the user do this?

3. An application provides output in some particular format. A user wants to produce custom reports from that output. An automated tool for creating such custom reports would be of great help. How can we give this flexibility to the user without much effort?

All these things are possible by harnessing the power of R through our application. This can be done using the API provided by R for the C programmers. It can thus be an answer to all these three scenarios and lot more. In this paper, we will explain how to use this API and demonstrate how these three scenarios can be handled through it.

DATA TYPES

First, we need to understand some basics. At the heart of every C function in the R API are conversions between R data types and C data types. Following are some of the most important data types in R.

R Data Type	Description	C Data Types
integer	Specifies integer vectors or scalars	int, int*, int[]
numeric	Specifies floating point vectors or scalars	double, double*
logical	Boolean variables taking one of the two values TRUE / FALSE	bool
character	Strings in R	char
matrix	Collection of data elements of the same type arranged in a two-dimensional rectangular layout	int[], int**, double[], double**, etc.
list	Generic vector containing other objects including other lists	User defined type (linked lists, struct, void*)
data.frame	A data table capable of storing different types of data	User defined type (void*)

R provides C users some interface functions for converting C data types to R data types and vice versa. Details of the entire interfacing with C is available in a standard document called "R Internals" that is available on the CRAN website. So, to achieve our objective, we need to first understand the basic workings of R internals.

R uses a fundamental data type called SEXP for handling all types of data internally. SEXP is short for “symbolic expression”. The API functions accept SEXP objects as input and return SEXP objects. SEXP is actually a pointer to an internal R structure with typedef SEXPREC. Note that, the internal structure of the SEXPREC is not made available to the users. Rather, SEXP is an opaque pointer, and the internals are known only to the R API.

As mentioned, everything is an SEXP in R. But there are different types of SEXPs like INTSXP, REALSXP, STRSXP, LGLSXP, VECSXP and more. Each SEXP subtype can be accessed from a C program using some data accessors. Here are some examples:

R Function	SEXP subtype	Data accessor
integer()	INTSXP	int *INTEGER(x)
numeric()	REALSXP	double *REAL(x)
logical()	LGLSXP	int *LOGICAL(x)
character()	STRSXP	CHARSXP STRING_ELT(x, i)
c()	VECSXP	SEXP VECTOR_ELT(x, i)
NULL	NILSXP	R_NilValue
externalptr	EXTPTRSXP	SEXP (accessor funcs)

In most cases, we will deal with converting to and from R vectors. In fact, R treats all scalars as vectors of size 1. One of the simplest ways to create a new R vector is to use the function **allocVector()**. It takes two arguments. The first one is the type of SEXP that we want to create. The second one is the length of the vector we want to create.

We will now demonstrate how to copy vectors to and from R using a sample C code.

C code snippet

```
#include <Rinternals.h>

SEXP RpnValsArr;
// Allocating a vector of integers of size 2
RpnValsArr = Rf_allocVector(INTSXP, 2);
PROTECT(RpnValsArr);

// Storing integers in the vector
INTEGER(RpnValsArr)[0] = 4;
INTEGER(RpnValsArr)[1] = 7;

// Retrieving integers from the vector
int pnVal[2];
int pnLen;

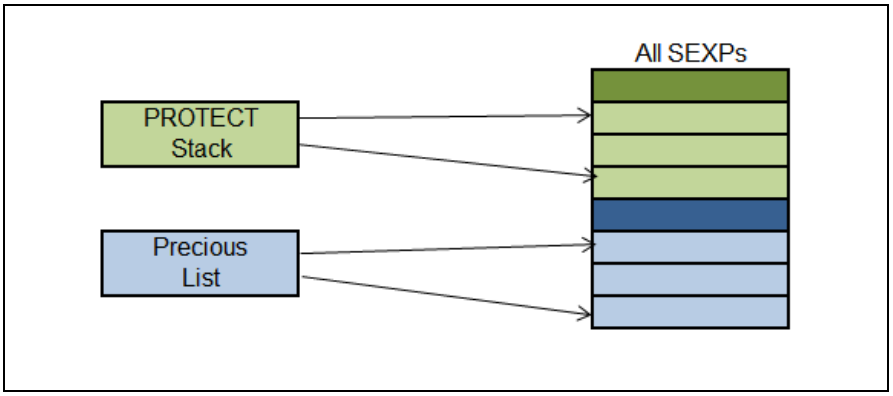
pnVal = INTEGER(RpnValsArr);
pnLen = LENGTH(RpnValsArr);

UNPROTECT(1);
```

GARBAGE COLLECTION

Note that, in the sample C code mentioned above we used two functions `PROTECT()` and `UNPROTECT()`. These functions are mainly used in the context of garbage collection in R.

The `PROTECT()` call tells R that the object passed to it is in use and shouldn't be deleted if garbage collection is activated. Please note that, this is a very important part of calling R from C. If objects are not properly protected, it leads to difficulty diagnosing errors, typically segfaults. But other errors are also possible. For instance, it may lead to unexpected behavior of that object including abnormal termination of the entire program. We don't need to protect objects that R already knows we are using, for instance, function arguments or loop indices within the corresponding loop. R keeps these objects in a list called **"Precious List"**. R maintains a protection stack to keep track of the objects we have protected. The `PROTECT()` call pushes the object into the protection stack.



We also need to unprotect every protected object once we are done using it. The way to do that is by using the function `UNPROTECT()`. It takes a single integer argument n and unprotects the last n objects that were protected by popping the last n items off the protection stack. The number of protects and unprotects must match. Otherwise, R will warn about a **"stack imbalance in .Call"**. Also, it is a good idea to release the memory from time to time by periodically unprotecting unused objects instead of unprotecting everything at the end. The pointer protection stack has a fixed size (default 10,000) and can become full.

The next step is calling an R function from C. Calling an R function involves six major steps:

1. Create an R instance
2. Load an R script into the R instance
3. Install an R function
4. Execute the R function
5. Extract outputs from the R function
6. Unload R

LOADING AND UNLOADING R

Now, let's discuss how to load an R instance from C. First of all, you have to make sure that the R executable install path e.g., **C:\Program Files\R\R-3.1.1\bin\i386** or **C:\Program Files\R\R-3.1.1\bin\x64** is in the windows **PATH** environment variable. Then, execute the following codes to load R in your environment. Also, after you are done using the connection to R, unload the R instance. Please note that, an R instance can be opened and closed only once during the lifetime of a process. So, make sure to load R and unload it only once in your code.

C code snippet

```
// This argument is required for invoking R silently in the background without
// the R GUI
char *localArgs[] = {"R", "--silent"};

// Invoke an instance of R
Rf_initEmbeddedR(sizeof(localArgs)/sizeof(localArgs[0]), localArgs);

// End the instance of R
Rf_endEmbeddedR(0);
```

We will now illustrate Steps 2 to 5 one by one in the context of the first scenario we mentioned in the introduction.

SCENERIO 1: GENERATING TIME TO EVENT DATA USING WEIBULL DISTRIBUTION

In this scenario, the user is trying to generate survival data using Weibull distribution. Suppose the user wants to use the following R code for this.

R code snippet

```
# NumSub : An integer giving the number of subjects for whom time to event is to be
# generated
# TreatmentID: A 0 based integer array indicating the trial arm (control / treatment)
# each subject belongs to.
# HazardRates: A 0 based double array containing the hazard rates for control
# and treatment arms
GenWeibull <- function(NumSub, TreatmentID, HazardRates)
{
  time <- c()

  for(m in 1:NumSub){
    j <- TreatmentID[m]
    time[m] <- rweibull(n = 1, shape = 2, scale = 1 / HazardRates[j+1])
  }

  return(list(SurvivalTime = as.double(time), ErrorCode = as.integer(0)))
}
```

Let us now see how to call this function from C.

LOADING THE R SCRIPT FROM C

R internals provide a function called `R_tryEval`. Using this function we can execute different R functions from C. We use the R command `source` to load the script file that contains the function `GenWeibull`. We use `R_tryEval` to execute the `source` command.

C code snippet

```
// sFileName is the name of the R file and R_GlobalEnv is the default global
// environment in R where we usually run our R codes
R_tryEval(Rf_lang2(install("source"), mkString(sFileName)), R_GlobalEnv, &nError);
```

INSTALLING THE R FUNCTION

Once we have loaded the R scripts, which may contain multiple functions, we need to specify the function we want to execute through R. We do that using the function `Rf_install`.
We then create a list of the function arguments in the order in which they are specified in the R function. We can also use named arguments so that the arguments can be passed in a different order.
First, we pass the function name to R

C code snippet

```
// Here, ptArgList is an object of type SEXP which contains the function name
SEXP ptNextArg = ptArgList = Rf_allocList(nCurArgCount + 1);
SET_TYPEOF(ptArgList, LANGSXP);
SETCAR(ptArgList, Rf_install("GenWeibull"));
```

Next we pass the arguments to our function. Execute the following functions to add them one by one.

C code snippet

```
// Creating R variables for holding the function arguments

// ptArgArr is a pointer to an already allocated SEXP
// array which stores all the arguments

// Creating function argument NumSub
Rf_protect(ptArgArr[0] = Rf_allocVector(INTSXP, 1));
INTEGER(ptArgArr[0])[0] = NumSub;

// Copying the name of the argument NumSub into an argument name array ptArgName
strcpy(ptArgName[0], "NumSub");
```

```
// Creating argument TreatmentID
Rf_protect(ptArgArr[1] = Rf_allocVector(INTSXP, NumSub));

for(int i = 0; i < NumSub; i++){
	INTEGER(ptArgArr[0])[i] = TreatmentID[i];
}

// Copying the name of the argument TreatmentID into ptArgName
strcpy(ptArgName[1], "TreatmentID");

// Creating argument HazardRates
Rf_protect(ptArgArr[2] = Rf_allocVector(REALSXP, 2));

for(int i = 0; i < 2; i++){
	REAL(ptArgArr[2])[i] = HazardRates[i];
}

// Copying the name of the argument HazardRates into ptArgName
strcpy(ptArgName[2], "HazardRates");


// Creating a linked list of function arguments (we have 3 arguments here)
for(int i = 0; i < 3; i++){
	ptNextArg = CDR(ptNextArg);
	SETCAR(ptNextArg, ptArgArr[i]);

	// Arguments are tagged with name, before passing to the function
	SET_TAG(ptNextArg, install(ptArgName[i]));
}

// We now have the R function argument list. We are ready to execute the R function!
```

EXECUTING R FUNCTION

Now that we have passed the function name and all its arguments to R, let us now execute it to generate outputs.

C code snippet

```
// Executing the function though R. ptRetVal is an SEXP variable containing the output
// and nErr contains the integer error code. If nErr is 0, then the execution of the
// function through R has been successful, otherwise, there is an error.
Rf_protect(ptRetVal = R_tryEval(ptArgList, R_GlobalEnv, &nErr));
```

EXTRACTING THE OUTPUTS FROM R

The next step is to extract the function output from R and process it into a format that the C code can understand and handle. In this example, the R function is returning a list. So, we read the list and retrieve the list elements one by one.

C code snippet

```
char sName[256];
SEXP names = Rf_getAttrib(ptRetVal, R_NamesSymbol); // Retrieving variable names

for(int i = 0; i < length(ptRetVal); i++){
	SEXP data = VECTOR_ELT(ptRetVal, i);
	strncpy(sName, CHAR(STRING_ELT(names, i)), 256);

	if(_stricmp(sName, "SurvivalTime") == 0){
		memcpy(pdValsArr, REAL(data), length(data) * sizeof(double));
	}
	else if(_stricmp(sName, "ErrorCode") == 0){
		memcpy(pnValsArr, INTEGER(data), length(data) * sizeof(int));
	}
}

// C array pdValsArr now contains the data generated from Weibull distribution using
// the R function.
```

SCENARIO 2: INVOKING GRAPHS FROM R

As we mentioned in the introduction, one way to enhance the quality of output is through graphs. It is visually enriching and much clearer for the user to understand the output. But giving such charts in our application may not be within its scope. R is famous for its graphics. Why not tap into that huge repertoire of plots and let the user decide which plot to use? That way the user can access all these charts even if our application does not directly produce them. Let us demonstrate how to produce such a custom chart using the R API for C programmers with an example.

In Oncology trials, which use time to event data, one very popular plot is the Kaplan Meier plot. The following R code can be used for generating a Kaplan Meier plot given a dataset.

R code snippet

```
// survdata dataframe contains three columns
// time: survival time of the patients
// status: indicates whether the patient is an event or not
// trtid: indicates whether the patient was administered control or active treatment
KMPlot <- function(survdata, sFileName)
{
  library(survival)
  png(sFileName, width = 1000, height = 480)
  surv <- survfit(Surv(time, status) ~ trtid, data = survdata)

  par(mfrow = c(1,2))
  colors <- c("red", "blue")

  plot(surv, lwd = 2, lty = 1:2, xlab = "Months", ylab = "Cumulative Survival", col =
    colors, mark = c(1,2))

  title("Kaplan-Meier Curves\nSurvival")
  legend(80, .9, c("Treatment", "Control"), lty = 2:3)

  lsurv2 <- survfit(Surv(time, status) ~ trtid, data = survdata, type = 'fleming')

  plot(lsurv2, lwd = 2, lty = 1:2, fun = "cumhaz", xlab = "Months", ylab = "Cumulative
   Hazard", col = colors, mark = c(1,2))

  title("Kaplan-Meier Curves\nHazard")
  legend(75, 2.7, c("Treatment ", "Control "), lty = 2:3)

  dev.off()
 }
```

Now, to call this R function from C, we will follow the same steps as before. We will store this function in a file, pass the function name and arguments as before, execute the function through R and retrieve outputs. But, there are two main differences in this case.

1. First, here we are passing a data frame as an argument to R.
2. Secondly, here we are not retrieving output from the return value as before. Instead, we are passing the name of a png file which will be the desired output in this case. The R function will save the plot in the specified png file.

Let us first see how to pass a data frame to R from C. Here, we will show how to create a data frame which contains the data provided from the C code. Then, we will pass the data frame as an argument to our function in R. Note that, we can create and add lists as function arguments to R functions in a similar way as data frames.

PASSING A DATA FRAME TO R

Suppose that, in the above example, we have time, status and trtid information for `NumSub` patients. We will first create a data frame with a single column corresponding to Time. Then we will add the rest of the columns to that data frame sequentially. We will use the built-in R function `data.frame` for this.

C code snippet

```
int nErr = 0;

// Memory allocation for SEXP integers corresponding to C integer array.
SEXP RpdTimeArr = Rf_allocVector(REALSXP, nNumVals);
SEXP RpnStatusArr = Rf_allocVector(INTSXP, nNumVals);
SEXP RpnTrtidArr = Rf_allocVector(INTSXP, nNumVals);
SEXP RpdDataFrame, dimnames ;
```

```
// Copying values of time in the corresponding variable
memcpy(REAL(RpdTimeArr), pdTimeArr, nNumVals * sizeof(double));

// Time is made as first column in the Dataframe
Rf_protect(RpdDataFrame =  R_tryEval(lang2(install("data.frame"), (SEXP)RpdTimeArr),
       R_GlobalEnv, &nErr));

// Copying values of status in the corresponding variable
memcpy(INTEGER(RpnStatusArr), pnStatusArr, nNumVals * sizeof(int));

// Adding status column to the existing data frames
Rf_protect(RpdDataFrame =  R_tryEval(lang3(install("data.frame"), RpdDataFrame,
       (SEXP)RpnStatusArr), R_GlobalEnv, &nErr));

// Copying values of trtid in the corresponding variable
memcpy(INTEGER(RpnTrtidArr), pnTrtIdArr, nNumVals * sizeof(int));

// Adding trtid column to the existing data frames
Rf_protect(RpdDataFrame =  R_tryEval(lang3(install("data.frame"), RpdDataFrame,
       (SEXP)RpnTrtidArr), R_GlobalEnv, &nErr));

// Setting column names
dimnames = allocVector(STRSXP, 3);
SET_STRING_ELT(dimnames, 0, mkChar("time"));
SET_STRING_ELT(dimnames, 1, mkChar("status"));
SET_STRING_ELT(dimnames, 2, mkChar("trtid"));
Rf_setAttrib(RpdDataFrame, R_NamesSymbol, dimnames);

// Adding the dataframe as an argument to R Function. nCurArgCount
ptArgArr[0] = RpdDataFrame;

// Copying the name of the argument into a Argument Name array
strcpy(ptArgName[0], psName);
```

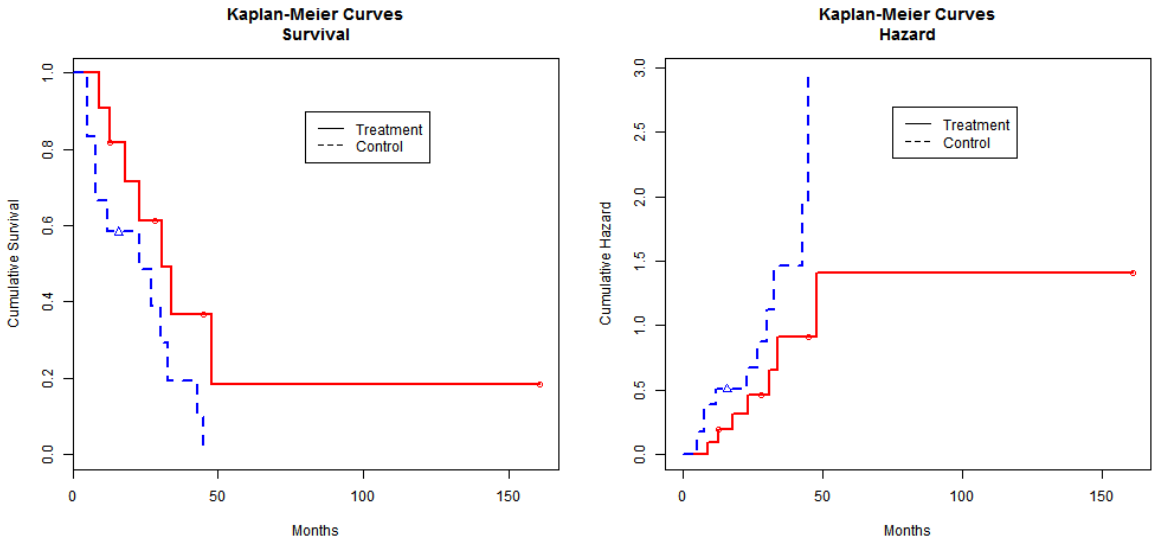
To illustrate this let us use the following data. Note that, we have simulated this data by adding some noise to the data `aml` from the package `survival` and changing some of the variables.

Data

	time	status	trtid
1	9.163479	1	1
2	12.897576	1	1
3	12.907921	0	1
4	18.038967	1	1
5	23.194903	1	1
6	28.096265	0	1
7	30.828752	1	1
8	33.993019	1	1
9	45.106563	0	1
10	48.038187	1	1
11	160.894407	0	1
12	5.077662	1	2
13	5.066969	1	2
14	7.875396	1	2
15	7.850228	1	2
16	11.807962	1	2
17	15.844911	0	2
18	23.104907	1	2
19	26.926804	1	2
20	30.149965	1	2
21	32.842019	1	2
22	42.836921	1	2
23	45.051172	1	2

On running the R function KMPlot() on this data, the following **png** file is generated:

Kaplan Meier Plots



SCENARIO 3: GENERATING CUSTOMISED REPORTS THROUGH R

We had earlier discussed the merits of allowing the user to generate customized reports with the outputs provided by the application. An automated tool will enable us to generate such custom reports automatically for different set of inputs.

In R, there are quite a few packages which allow users to create customized reports. One of them is the `RMarkdown` package. `RMarkdown` enables the user to generate custom reports in **text**, **docx** and **pdf** formats through an **rmd** file. In this file, user can write texts in different tags to customize their reports. Additionally, they can also write embedded R codes in that file, which allows them to seamlessly integrate R outputs with other texts in different fonts and formats. We will now demonstrate how to use the techniques described above to execute an **rmd** file containing embedded R codes from C and produce a custom report in a **docx** format.

In Scenario 2, we had demonstrated how to generate custom plots from a dataset. Now, suppose we want to generate a report which will contain a snapshot and a short summary of the data, a few lines explaining what we are doing and then display the plots at the end. So, let us write the following **rmd** file which will do all these.

Rmd (R Markdown) code snippet

````
---
author: "Pravin Holkar, Sudipta Basu"
title: Report
output: word_document
---

This is an R Markdown document. Markdown is a simple formatting syntax for authoring
HTML, PDF, and MS Word documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

Here, we have got the time to event data for 23 patients.
A snapshot of the data gives us a glimpse into it's variable values.

```{r, echo = FALSE}
  opts_chunk$set(comment = NA)

```

```{r pander, echo = FALSE}
  emphasize.cols(1)
  pandoc.table(head(Data), caption = "Time-to-event Data", style = "grid")

```
````

```
This data has 3 columns namely, time, status and trtid.
A better idea about these variables can be obtained from the summary of the data.

```{r, echo = FALSE}

  pandoc.table(summary(Data), caption = "Variable Summary of Time-to-event Data",
style = "grid")

```

With this data, we have generated 2 Kaplan Meyer plots.
One is for Events, while the other is for Survival corresponding to the control and
the active treatment.


```{r, results="asis", echo = FALSE}

  library(survival)
  surv <- survfit(Surv(time, status) ~ trtid, data = Data)

  colors <- c("red", "blue")

  plot(surv, lwd = 2, lty = 1:2, xlab = "Months", ylab = "Cumulative Survival", col =
         colors, mark = c(1,2))

  title("Kaplan-Meier Curves\nSurvival")
  legend(80, 0.9, c("Treatment", "Control"), lty = 2:3)

  lsurv2 <- survfit(Surv(time, status) ~ trtid, data = Data, type = 'fleming')

  plot(lsurv2, lwd = 2, lty = 1:2, fun = "cumhaz", xlab = "Months", ylab = "Cumulative
   Hazard", col = colors, mark = c(1,2))

  title("Kaplan-Meier Curves\nHazard")
  legend(75, 2.7, c("Treatment ", "Control "), lty = 2:3)

```
```

But, one problem with using an **rmd** file is that, unlike our previous examples, we cannot put the entire code in that file within a function. So, we cannot pass the outputs provided by our application to it directly. Let us see how to bypass this issue.

Consider the following R Code:

R code snippet

```
// survdata contains three variables
// time: survival time of the patients
// status: indicates whether the patient is an event or not
// trtid: indicates whether the patient was administered control or active treatment
// sFileName is the name of the Rmd file to be rendered
GenReport <- function(survdata, sFileName){

  // Storing input values in global variables
  Data <- survdata
  FileName <- sFileName

  // loading the libraries rmarkdown and pander required for rendering the rmd file
  library(rmarkdown)
  library(pander)

  // rendering the input rmd file
  rmarkdown::render(FileName)
}
```

Note that, here we are storing the input parameters in global variables `data` and `FileName` which can be accessed directly from the same scope. Since, we are rendering the **rmd** file from the same function, the scope remains the same. So, as it can be seen from the code written in the **rmd** file, we are accessing and subsequently using these global parameters directly in the embedded R code.

On executing the **rmd** code by calling the wrapper R code from our application, we create the following report:

Customised Report

Report

Pravin Holkar, Sudipta Basu

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

Here, we have got the time to event data for 23 patients. A snapshot of the data gives us a glimpse into it's variable values.

time	status	trtid
9.163	1	1
12.9	1	1
12.91	0	1
18.04	1	1
23.19	1	1
28.1	0	1

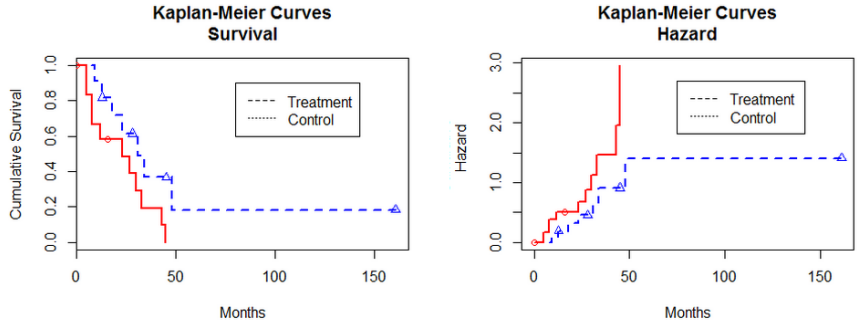
Table: Time-to-event Data

This data has 3 columns namely, time, status and trtid. A better idea about these variables can be obtained from the summary of the data.

time	status	trtid
Min. : 0.00	Min. :0.00	Min. :0.000
1st Qu.: 11.15	1st Qu.:0.75	1st Qu.:1.000
Median : 23.15	Median :1.00	Median :1.500
Mean : 28.23	Mean :0.75	Mean :1.458
3rd Qu.: 33.13	3rd Qu.:1.00	3rd Qu.:2.000
Max. :160.89	Max. :1.00	Max. :2.000

Table: Variable Summary of Time-to-event Data

With this data, we have generated 2 Kaplan Meyer plots. One is for Events, while the other is for Survival corresponding to the control and the active treatment.



LIMITATIONS

As amazing as the techniques described in this paper are, they do have a few limitations:

1. The basic idea behind this paper is to give more control to the user via R. This makes our application much more dynamic and flexible. But it also means that we have no control over what the user does in their code. There is no check whether the output returned by the user's code is correct or not. In fact, the code can be so faulty that it may even lead the entire application to crash. One way to avoid crash is to create a separate exe for the module which calls R and call that exe from the main application. That way the crash will remain localized and won't affect the entire application. But that's damage control at best and the basic issue remains.

2. The API provided by R for C programmers cannot be used in a multithreaded environment. So more than one thread cannot execute R functions in parallel using this technique.

3. Since a lot of steps are involved in calling an R function, the speed of the application may get somewhat affected compared to if the R function logic were coded directly inside the application. The performance degradation is not noticeable if the R function is called only a handful of times. But it can get significant if the R function is called a large number of times. E.g. when the function is called in every run of the simulation and hundreds of thousands of runs are performed.

4. The entire procedure is dependent on the R version the user is using. And R keeps releasing new versions very frequently. So, one has to keep monitoring the new versions released by R, especially to check if there are any changes in R internals that may break the code.

CONCLUSION

Despite the above limitations, from all the 3 examples demonstrated in the paper we see that the technique to call R from C has a huge potential to make our application significantly more powerful, flexible and dynamic. It gives more choice and power to the user by tapping into the gigantic repertoire of existing functionalities in R. Lastly, the best thing about these changes are that, without spending any extra resources or time, we are allowing the user to provide seemingly limitless number of functionalities through R.

REFERENCES

1. **R Internals**: R-ints.pdf, R-ints.html kept in RHome\doc\manual
2. **Advanced R** - Hadley Wickham: http://adv-r.had.co.nz/C-interface.html
3. **Native Interfaces for R** - Seth Falcon, 2010

ACKNOWLEDGEMENTS

We would like to thank our colleagues at Cytel (India) for their support, co-ordination and valuable comments. Special thanks to Mr. Ajay Sathe, Mr. Aniruddha Deshmukh, Ms. Namrata Deshpande and Mr. Sameer Bamnote for their help and guidance.

CONTACT INFORMATION

Author 1:

Author Name:	Sudipta Basu
Company:	Cytel Statistical Software & Services Pvt. Ltd. (Subsidiary of Cytel Inc.)
Address:	6th Floor, Lohia-Jain IT Park, A Wing, Chandani Chowk, Paud Road, Kothrud Pune - 411038, Maharashtra, India
Work Phone:	+9120-67090228
Email:	sudipta.basu@cytel.com
Web:	www.cytel.com

Author 2:

Author Name:	Pravin Holkar
Company:	Cytel Statistical Software & Services Pvt. Ltd. (Subsidiary of Cytel Inc.)
Address:	6th Floor, Lohia-Jain IT Park, A Wing, Chandani Chowk, Paud Road, Kothrud Pune - 411038, Maharashtra, India
Work Phone:	+9120-67090230
Email:	pravin.holkar@cytel.com
Web:	www.cytel.com