interfacing to other languages in R

The following discusses the basic concepts of interfacing between R and other programming languages. Specifically, the examples below are in the context of calling C/C++ from R and calling R from Python.

writing C/C++ fucntions to be called from R

A general reason to write certain code in C to then be run within the R environment is often to increase performance. Another reason to drop down to C would be to utilize a specialized C I/O. For example, R used **TCP** (discussed earlier) protocols the 3rd layer of the standard internet communication system; C uses **UDP** which is fast in certain circumstances. R has wo base C/C++ interfaces with the **.C( )** and **.Call( )** functions.

The **.Call( )** function is often more versatile, equally requiring knowledge of R's internal structures.

```
R to C/C++ interfacing preliminaries for the illustrations used:
C stores 2D arrays in row-major Order; R stores 2D arrays in column-major order
C uses 0-based indexing subscripts; R uses 1-based indexing subscripts
Arguments passed from R to C are received as pointers
C functions must return void; values returned must be communicated through function arguments
```

The following example uses C code to extract subdiagonals from a square matrix.

(*attributed code written by Min-Yu Huang and published by Norman Matloff*).

```c
#include <R.h> // required

// arguments:
//    m: a square matrix
//    n: number of rows/columns of m
//    k: the subdiagonal index--0 for main diagonal, 1 for first
//       subdiagonal, 2 for the second, etc.
//    result: space for the requested subdiagonal, returned here

void subdiag(double *m, int *n, int *k, double *result)
{
  int nval = *n, kval = *k;
  int stride = nval + 1;
  for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
     result[i] = m[j];
}
```

The **stride** variable is a reference to parallel processing methods. Assuming a **1000**-colum matrix $\mathbb{R}^{n\times1000}$ with the C code looping through each element in a given column; top to bottom. Remembering that C uses **row-major** order, consecutive elements in a column are **1,000** elements apart as a single, long vector. In effect, the long vector is being **traversed** with a **stride** of **1,000** (accessing every 1000th element).

The code is compiled, for this example, in a Linux terminal window (Cygwin package for Windows):

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include     -fpic -g -O2 -c sd.c -o sd.o
gcc -std=gnu99 -shared -o sd.so sd.o    -L/usr/lib/R/lib -lR
```

The above ultimately creates a dynamic file library named ***sd.so***. The library is subsequently loaded into R and the C function is called as follows:

```r
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
```

The output is returned in R as follows:

```
result=double(dim(m)[1]-k))
[[1]]
 [1] 1 6 11 16 21 2 7 12 17 22 3 8 13 18 23 4 9 14 19 24 5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

Note the **result** name is assigned to both the formal argument (in C) and the actual argument (in R). Also note that space was allocated for the **result** in R code. The return values are displayed as a list of arguments.

debugging R/C code in R

Extra challenges arise when debugging R/C code in the context of using **GDB** (discussed previously). The following is the former example of R/C debugging using **GDB** on the ***sd.c*** code in the illustration.

```
$ R -d gdb                            #launch gdb from a terminal window command line
GNU gdb 6.8-debian
...
(gdb) run                             #call gdb to run R
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")                   #load compiled C code into R
>   # hit ctrl-c here                 #interrupt to pause R and return to gdb prompt
Program received signal SIGINT, Interrupt.
0xb7ffa430 in __kernel_vsyscall ()
(gdb) b subdiag                       #breakpoint set at entry to subdiag()
Breakpoint 1 at 0xb77683f3: file sd.c, line 3.
(gdb) continue                        #resume execution R (hitting enter-key twice)
Continuing.

Breakpoint 1, subdiag (m=0x92b9480, n=0x9482328, k=0x9482348, result=0x9817148)
    at sd.c:3
3          int nval = *n, kval = *k;
(gdb)
```

The C code is then executed as follows:

```
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
+ result=double(dim(m)[1]-k))

Breakpoint 1, subdiag (m=0x942f270, n=0x96c3328, k=0x96c3348, result=0x9a58148)
    at subdiag.c:46
46 if (*n < 1) error("n < 1\n");
```

The **GDB** session for debugging then continues as per usual. For a proper reference to understanding **GDB**:

Common GDB commands:

l	List code lines	n	Step to next statement	c	Continue
b	Set breakpoint	s	Step into function call	h	Help
r	Run/rerun	p	Print variable or expression	q	Quit

using R from Python

The **RPy** module in Python allows access to R from Python. It is often used in tandem with **NumPy** for the best experience. Loading **Rpy** from Python is achieved through executing the following code:

```
from rpy import *
```

The variable **r** is then loaded as a Python class instance.

The following illustrates the basic principles of running R from Python:

```
>>> r.hist(r.rnorm(100))
```

The subsequent action calls the **rnorm( )** function in R to product 100 standard normal variates in a histogram using the **hist( )** function in R. Note that the R names are prefixed with a **r.**; the Python wrappers for R functions are members of the class instance **r** in Python. However, the above call can produce noisy results within the *title of the graph* and the *x-axis label*. The latter is avoided as follows:

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

Problems will arise for R users when R and Python syntax experience collision. For example, calling the linear model function **lm( )** in R to predict variable **b** from variable **a**:

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm('v2 ~ v1',data=r.data_frame(v1=a,v2=b))
```

The above execution in Python requires more syntax than if performed in an R environment. Notably, Python syntax does not make use of the **tilde ~** character to execute as a function of; the model was specified as a string. Additionally, a dataframe was needed to contain the data; created through the **data.frame( )** function. Note that the **period "."** within the function is applied as an **underscore "_"** in Python. Lastly, the columns in the dataframe were named **V1** and **V2** prior to using them in the model formula.

The output object is a Python dictionary (analog of R's list type):

```
>>> lmout
{'qr': {'pivot': [1, 2], 'qr': array([[ -1.73205081, -17.32050808],
[ 0.57735027, -6.164414 ],
[ 0.57735027, 0.78355007]]), 'qraux':
```

The attributes of the **lm( )** function differ in Python; the coefficients of the fitted regression line (**lmout$coefficients** in R) are presented as **lmout['coefficients']** in Python and accessed as follows:

```
>>> lmout['coefficients']
{'v1': 2.5263157894736841, '(Intercept)': -2.5964912280701729}
>>> lmout['coefficients']['v1']
2.5263157894736841
```

Additionally, R commands can be executed to work on variables in R's namespace through the **r( )** function. The following executes the **wireframe( )** function illustration presented previously in R; below in **RPy**:

```
>>> r.library('lattice')
>>> r.assign('a',a)              #assign a copy variable from Python's namespace to R
>>> r.assign('b',b)              #assign a copy variable from Python's namespace to R
>>> r('g <- expand.grid(a,b)')   #note the '.' in expand.grid; due to R namespace
>>> r('g$Var3 <- g$Var1^2 + g$Var1 * g$Var2')
>>> r('wireframe(Var3 ~ Var1+Var2,g)')
>>> r('plot(wireframe(Var3 ~ Var1+Var2,g))')  #plot() called; not automatically displayed
```

The above is useful in the context of multiple syntax collisions between Python and R.