Calling R functions from C

Phil Spector (spector@stat.berkeley.edu)
Statistical Computing Facility
Department of Statistics
University of California, Berkeley

A sometimes useful capability of functions in R is their ability to pass data to and from programs that are written in C. While the actual workings of the interface are quite complex, a simple example should provide a model for the process.

Suppose we wish to write a numerical integration program, using Simpson's rule, which subdivides an interval into a number of "sections", and then adds together the estimated areas of the sections. The numerical details need not get in the way of the example; the basic point is that, since looping may be inefficient in R we would like to perform the integration in a C program, but still have the convenience of defining the function to be integrated from within the R environment.

The interface between C and R is a subroutine known as `call_R`. This subroutine takes as one of its arguments a pointer to a function, which is passed into the C environment as a list. The number and type of the arguments is also passed to `call_R`, with the results being passed back through the argument list, not as a return value. It is generally easiest to divide the C program into three parts: the interface, which is called by R and which calls the algorithm, a "shell" which uses `call_R` to have R evaluate the function, and finally the algorithm itself, isolated from the issues of the interface. This strategy can be demonstrated in the following C program:

```
static char *sfunction;
dosimp(char** funclist,double *start,
       double *stop,long *n,double *answer)
{
  double sfunc(double);
  double simp(double(*)(),double,double,long);
  sfunction = funclist[0];
  *answer = simp((double(*)())sfunc,*start,*stop,*n);
}
double sfunc(double x)
{
  char *modes[1];
  char *arguments[1];
  double *result;
  long lengths[2];
  lengths[0] = (long)1;
  arguments[0] = (char*)&x;
  modes[0] = "double";
  call_R(sfunction,(long)1,arguments,modes,lengths,
         (char*)0,(long)1,arguments);
  result = (double*)arguments[0];
  return(*result);
}
double simp(double(*func)(),double start,double stop,long n)
```

```
{
  double mult,x,t,t1,inc;
  long i;
  inc = (stop - start) / (double)n;
  t = func(x = start);
  mult = 4.;
  for(i=1;i<n;i++)
     {x += inc;
      t += mult * func(x);
      mult = mult == 4. ? 2. : 4.; }
  t += func(stop);
  return(t * inc / 3.);
}
```

The first function, dosimp, simply extracts the function pointer from the list passed into the C environment, and loads it into a static pointer to make it available to the function sfunc, and then calls the function which actually does the integration. It is the function dosimp that will be called from R. The function sfunc uses the call_R function to pass arguments to, and receive results from, R. Its first argument is the function pointer dereferenced from the list pointer which was passed to dosimp from R. The second argument is the number of arguments to be passed to the R function, in this case 1. Next is an array of pointers to the arguments themselves, each cast as char*. Note that this is an array of pointers, i.e. a pointer to a pointer, and not just a pointer. Similarly, the modes argument is an array of pointers to character strings which tell the R function how to treat each argument which is passed; the most common modes are double and character. As in all functions external to R the arguments themselves are pointers to the values they represent, not the values themselves. The next argument is an array of integers (longs) giving the length of each of the arguments passed; in this case, lengths[0] has been set to 1. The next argument, which is cast as a null pointer in this example, is a pointer to an array of names of the arguments being passed to the R function; usually an explicit pointer to names will not be required. The next argument is the number of results which the R function will return; by returning a list, an R function can return more than one value into the C environment. The final argument is an array of pointers to where the results will be returned. Note that by passing the same pointer for both results and arguments, no additional space needs to be allocated for results within the C environment. While not required, this technique is encouraged, and will not affect any of the variables which are defined in the R environment.

The simp function is a straight forward implementation of Simpson's 1/3 rule, and has no special provisions for use with R. This will usually be the case when the strategy outlined above is used. To compile these programs, assuming they are stored in a file called simp.c, use the following command:

```
R CMD SHLIB simp.c
```

Finally, the R function to call these routines is as follows:

```
function(func, start, stop, n = 10)
{
   if(trunc(n/2) - n/2!=0)
     stop("Number of points must be even.")
   if(!is.loaded("dosimp"))dyn.load("./simp.so")
   t = 0
   .C("dosimp",list(func),as.double(start),
                as.double(stop),as.integer(n),
                ans = as.double(t))$ans
}
```

This function accepts three argument; the first is the name of the R function to be numerically integrated (passed without quotes); the next two arguments are the limits of the integration, and the final argument is the number of intervals to use in the Simpson's rule algorithm.

Note the use of the `list` function to pass the function pointer to the C routine. As in previous examples, the arguments are coerced to the appropriate types using `as.integer` and `as.double`, with the returned value from the C function being passed back as the final argument.