Writing Functions in Splus

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

1 Modifying an Existing Function

The easiest way to become familiar with working with functions is to look at some existing ones and to make some simple modifications. Since functions are treated like any other data object within Splus you can look at any function by typing its name. Note that the actual algorithm behind the function will not appear in the function definition; there will usually be a call to `.Internal()`, `.Fortran()` or `.C()` within the definition to take care of that; the function definitions themselves act as an interface. For example, the function "postscript", which is the postscript device driver looks like this:

```
function(file = "PostScript.out", command = "lpr -Pps",
         horizontal = F, width = 0, height = 0, rasters = 0,
         pointsize = 0, font = 1, preamble = PS.preamble)
{
        graphics.off()  # wrap-up for any existing graphics driver
        if(file!="")
          {     where <- file
                how <- 0 }
        else
           {    where <- command
                how <- 1 }
        z <- .C("postscript",
                as.character(where),as.integer(how),
                as.integer(horizontal),as.single(width),
                as.single(height),as.integer(rasters),
                as.single(pointsize),as.integer(font),
                as.character(preamble),as.integer(length(preamble)))
        Device.Default("postscript")
        invisible(z[[1]])
}
```

After closing any other active graphics devices (`graphics.off()`), this function checks to see if a file to receive the postscript output has been specified. If so, it passes the name of the file to the C function postscript; otherwise, it passes the command which has been specified. Note the method by which defaults are supplied to the program; the values after the equal sign in the argument list serve as a value to use if one was not specified by the caller of the function. Each argument to the C program has been coerced into either a character argument, a single precision real number (i.e. a `float` in C), or an integer. This is a good practice, as no argument type checking is done automatically within the function interface. The function `invisible()` suppresses the default behavior of returning the result of the last statement within a function.

Suppose that you wanted to modify this function so that it would by default pass its output directly to the printer, through the command `lpr -Plw4`. You would create a new function in your local Splus directory (usually `~/.Data`) called `postscript` using the `vi()` function. This function opens a vi editor window on a copy of its argument, and produces as its result the updated version of its arguments. (The `vi()` function works for data as well as functions.) The command

```
postscript <- vi(postscript)
```

would open an edit window on the system's version of the postscript function, allow you to modify it (in this case by eliminating the default `file` argument and modifying the `command` argument), and store the result in the function `postscript` in your local Splus directory.

Two words of caution are in order. First, once you create a copy of a function in this way, it will always be accessed when you use its name, and there will be no warning given that you are overriding the system's version of the command after the first time that you save it. Thus, if you inadvertently name a function "q", you may find it difficult to end your Splus session! Second, when an error occurs while editing a function using the `vi` command, changes which you have made may not be saved. You can re-edit the most recent version of the function by calling `vi` with no arguments, (i.e `vi()`); however, if you are writing an extensive function, you may want to save a copy of it in some convenient location (using the `:w filename` command from within vi) in case an error is found.

2 Writing a new Function

As an example of a function, consider a matrix with two columns; the first containing group numbers, and the second containing a numerical value. We wish to write a function called `find.means` which will take as its argument the name of this matrix, and which will return a list whose elements are the group numbers, number of observations and means of the numerical value for each distinct group number in the first column of the matrix. (This problem can more easily be solved with the Splus function `tapply`, but it will serve as a simple example.)

To begin editing a function, we create a null function to serve as an argument to `vi`; one way to do this is with the statement:

```
find.means<-vi(function(){})
```

This will present us with an edit window that looks like this:

```
function()
{
}
```

We can now fill in the arguments and function body. One implementation of the program is as follows:

```
function(z)
{
        zu <- unique(z[, 1])
        ans <- list(n = numeric(length(zu)),
                   value = numeric(length(zu)),
                   mean = numeric(length(zu)))
        i <- 1
        for(j in zu) {
                ans$value[i] <- j
                ans$mean[i] <- mean(z[, 2][z[, 1] == j])
                ans$n[i] <- length(z[, 1][z[, 1] == j])
                i <- i + 1
 ans
}
```

In this function, we are producing a list which contains three elements: value, the unique group values represented in the first column of the matrix which is passed to the function; mean, the mean for each group;

and n, the number of observations in each group. Note that, before we could refer to the elements of the list in the program, we had to let Splus know what the list will hold by using the `list()` function. In order for the list to be built, Splus must know the mode of each element in the list; here the `numeric()` function was used, since the results will be numbers. The argument to the `numeric()` function tells Splus the length of the element. Finally, since variables mentioned as targets for assignment statements within a function disappear by default after the function invocation, the final statement of the function, `ans`, will force the function to return as its value the list which contains the values, means and counts. (To override this default, and force a function to produce an object which will be stored in your `.Data` directory, use the "`<<-`" assignment operator.)

When you exit the `vi` function, Splus will make sure the function's syntax is valid, and, if so, store the function in your `.Data` directory. If not, it will tell you there is an error, and advise you to re-edit the function. In this case, the command `find.means <- vi()` will open a vi edit window on the version of the function that had the error.

3 Calling an external program from an Splus function

Splus allows you to call external programs written either in C or in FORTRAN. In either case, the program should not be written as a main program, but as a subroutine (in FORTRAN) or a function (in C) which Splus will be able to dynamically load into its already executing image. The value returned by a function is **not** available to Splus; all information between Splus and the external program must be passed through the arguments of the subroutine or function. To illustrate how to make external programs usable inside of Splus a simple example will be used, first with an external C program, then with the same program written in FORTRAN. Before writing the function, it is important to understand the representations of data within Splus and the correspondence to the data types in C and FORTRAN. These relationships are summarized in the following table:

S	**C**	**FORTRAN**
`"single"`	`float*`	`real`
`"double"`	`double*`	`double precision`
`"integer"`	`long*`	`integer`
`"character"`	`char**`	`character`
`"complex"`	`struct { double re,im;}*`	`double complex`
`"list"`	`char**`	`-`

Note that in each case, the **address** of the data object will be passed to and returned from the function, never the actual value itself. While this is the natural approach in FORTRAN (which uses call by address in its parameter lists), C programmers must take extra caution when writing functions to be used in Splus. This technique is necessary to allow objects altered in the external function to remain changed when they "return" to the Splus environment.

The first step in writing the function is to write the interface within Splus. To illustrate, suppose we want to write a function which will call an external program to calculate the cumulative sum of a vector of numbers. (The Splus function `cumsum()` already takes care of this, but we will use this simple task as an example.) We need only to pass the vector of values to the Splus function, but we must also pass the length of the vector to our external program. The interface for a C program could be written as follows:

```
function(x)
{
    dyn.load.shared("./cs.so")
    n <- length(as.vector(x))
    .C("csum",
             as.integer(n),
             result = as.double(x))$result
}
```

The following points should be noted:

- The function `dyn.load.shared` dynamically loads the named shared object into the already existing version of Splus. However, once a module is dynamically loaded, it remains available through the remainder of the Splus session. Thus, once the function is debugged and working, the `dyn.load.shared` call should be preceded by a check to see if the module was already loaded as follows:

  ```
  if(!is.loaded(C.symbol("csum")))dyn.load.shared("./cs.so")
  ```

- Since the object file stays loaded throughout the Splus session, it may be necessary to restart Splus if you need to reload a new version of your program into Splus.
- Only shared objects are suitable for dynamic loading into Splus. The details of producing a shared object are presented below.
- The name of the C source file, the resulting object file, and the Splus function name need not be the same. What is important is that the `dyn.load.shared` function be passed the name of the shared object file, and the `.C()` function be passed the function name which is defined in the C program. Notice that `dyn.load.shared` needs a complete pathname even if the file to be loaded is in the current directory.
- The data object "x" is coerced to a vector before its length is taken. This allows the function to handle matrices, although it will simply calculate the cumulative sum for all the elements of the matrix, taken in column-by-column order.
- The arguments "n" and "x" are coerced to an integer and a double precision vector, respectively, before being passed to the C routine. This will insure that the S interface converts the data to the appropriate type (if necessary) before calling your C routine.
- The `.C()` function actually returns a list, with each of its arguments representing the list elements. Elements in the list can be named (like `result` in the example) and referred to using the usual "$" notation. In this example, the result uses the same storage as the input vector x. The Splus interface insures that data stored in the calling frame will not be damaged using this technique, and it is encouraged.
- Since an Splus function returns the result of the last statement in the function, the technique outlined above causes the function to return the appropriate value. It should be emphasized that the original (calling frame) vector "`x`" is *not* overwritten; it has the same value after returning from the function as it did before the function was called.

The C function for this example is as follows:

```
csum(long *n,double *x)

{  long i;

   for(i=1;i<*n;i++)
      x[i] += x[i - 1];

}
```

Note that both the vector x and the length n were passed to the C program as addresses. As mentioned previously, this is a requirement of the Splus interface.

To compile the function, assuming it is stored in the file `filename.c`, the following UNIX commands would be used:

```
    cc -KPIC -c filename.c
    cc -G -o filename.so filename.o
```

The `-KPIC` option is necessary to create an object file that is appropriate for making a shared object suitable for dynamic loading into Splus; the `-G` option in the second command produces the actual shared object.

To achieve the same results with a Fortran program, the only change necessary in the interface is to change the `.C()` call to a `.Fortran()` call. The following Fortran program would then be compiled:

```
       subroutine csum(n,x)
       integer i,n
       double precision x(1)
       do 10 i=2,n
   10  x(i) = x(i) + x(i - 1)
       return
       end
```

The compilation command required to produce the object file is:

```
f77 -c -PIC filename.f
f77 -G -o filename.so filename.o
```

where `filename.f` is the name of the file in which the FORTRAN code is stored.

4 Using External Subroutine Libraries

As a more complex example of a function, suppose we wish to write a random number generator for the Weibull distribution, using the NAG subroutine library routine `g05dpf`. We will model our function after existing random number generators, and name it `nagrweibull`, since there is an existing `rweibull` function in Splus. The arguments to the function will be the number of random variables generated, and optionally, shape and scale parameters, which will both default to 1.0 if not specified.

For more complex functions, the interface plays a more important role. It is here where defaults are set and parameters should be checked; if illegal or out-of-range data is sent to an external program, the results will be unpredictable, to say the least. Thus there is a great reward for paying attention to these details in the interface, before a problem arises in the external function.

If your object file requires external libraries, you must include them in the final compile which produces the shared object, so that Splus can find the necessary libraries when the program is run. Thus, if your program needed the NAG libraries, for example, you would use UNIX commands like the following to create an appropriate shared object:

```
   cc -KPIC -c prog.c
   cc --f77 -G -o prog.so prog.o -lnag
```

By default, the C library, accessed by `-lc`, is always included, so you don't need to specify it explicitly.) If additional object files were needed to create the shared object for dynamic loading, they would simply be compiled using the `-KPIC` option and included on the command line of the final compile command.

The interface for the `nagrweibull` function could be written as follows:

```
function(n, shape = 1, scale = 1)
{
   dyn.load.shared("./rw.so")
   if(shape <= 0)
      stop("Shape parameter must be > 0.")
   if(scale <= 0)
      stop("Scale parameter must be > 0.")

   result <- double(n)
   .C("rweib",
            as.integer(n),
            as.double(shape),
            as.double(scale),
            result = result)$result
}
```

The same considerations of the previous example apply here, especially regarding the check using `is.loaded` once initial testing has shown that the C program is working properly. The function `stop()` is used to print an appropriate error message and terminate execution before too much damage is done. The C program which calls the NAG routine is as follows:

```
rweib(long *n,double *a,double *b,double *x)

{  double g05dpf_(double*, double*,long*);
   long i,ifail;

   for(i=0;i<*n;i++)
      *(x++) = g05dpf_(a,b,&ifail);
}
```

Notice that, when calling a routine from a FORTRAN library from within a C program, an underscore must be appended to the routine name, thus the routine is referred to as `g05dpf_` This is true whether or not the function will be called from Splus As in the previous example, **all** arguments to the C function must be pointers — likewise, all arguments to the FORTRAN function must be pointers. This is why the address of ifail is passed to the routine instead of ifail itself. Since Splus does not have access to values returned by functions, the necessary values are loaded into a vector and then passed back through the argument list of the function. To create the necessary shared object file, the following commands could be used, assuming that the program was stored under the filename `rweib.c`:

```
      cc -KPIC -c rweib.c
      cc --f77 -G -o rw.so -lnag
```

An equivalent FORTRAN program which calls the NAG routine is as follows:

```
      subroutine rweib(n,a,b,x)

      integer i,n,ifail
      double precision a,b,g05dpf,x(1)

      ifail = 0
      do 10 i=1,n
  10  x(i) = g05dpf(a,b,ifail)

      return
      end
```

Assuming this program is stored in a file called `rw.f`, the program would be compiled, and a shared object created with the following commands:

```
    f77 -PIC -c rw.f
    f77 -G -o rw.so rw.o -lnag
```

and the `.C()` call in the interface would be replaced with a `.Fortran()` call.

5 Dealing with Matrices

If your C or Fortran function will deal with matrices, recall that Splus, like Fortran, stores it matrices by columns as a single-dimensioned array. If you don't store matrices that way in a C program, you will have to write one conversion program which takes the matrix arguments from Splus and converts them to the form that you expect in your C program, and another which takes your results from the C program and converts them to column-by-column format.

An additional complication when working with matrices is that, by default, the matrix character of an object is lost in the C or Fortran interface when it returns to the S environment. To make sure this doesn't happen, you can use the `storage.mode` function of Splus to permanently change the storage mode of the object, rather than temporarily coercing it into the appropriate type (with `as.double()` or `as.integer()` for example). For example, suppose we have a C function called `matfunc` which accepts a pointer to an $n \times p$ matrix of doubles, and two pointers to integer representing the dimensions n and p. If we call such a function with a Splus statement like:

```
newx <- .C("matfunc",x=as.double(x),as.integer(n),as.integer(p))$x
```

we would find that, upon returning from the `.C()` function, `newx` would be a vector of length $n*p$, and not a matrix, as it was when it was passed to the `.C()` function. The solution is as follows:

```
storage.mode(x) <- "double"
newx <- .C("matfunc",x=x,as.integer(n),as.integer(p))$x
```

Notice that when you set the storage mode in this way you should not call the `as.double` function for the matrix in question.

An alternative solution would be to reconstruct the matrix after the call to `.C()`; since the arrangement of the elements doesn't change, all that is required is a call to the `matrix()` function:

```
newx <- .C("matfunc",x=as.double(x),as.integer(n),as.integer(p))$x
newx <- matrix(newx,n,p)
```

6 Communicating with S Functions

An additional capability of functions in Splus is their ability to pass data to and from user written programs. One restriction of this capability is that the program must be 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 is inefficient in Splus 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 Splus environment.

The interface between C and Splus is a subroutine known as `call_S`. This subroutine takes as one of its arguments a pointer to a function, which is passed into the C environment as a list. (Recall the table of data types in Section 3.) The number and type of the arguments is also passed to `call_S`, 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 Splus and which calls the algorithm, a "shell" which uses `call_S` to have Splus 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_S(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 Splus. The function `sfunc` uses the `call_S` function to pass arguments to, and receive results from, Splus. Its first argument is the function pointer dereferenced from the list pointer which was passed to `dosimp` from Splus. The second argument is the number of arguments to be passed to the Splus 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 Splus function how to treat each argument which is passed; the most common modes are `double` and `character`. As in all functions external to Splus the arguments themselves are pointers to the values they represent, not the values themselves. The next argument is an array of integers (`long`s) 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 Splus function; usually an explicit pointer to names will not be required. The next argument is the number of results which the Splus function will return; by returning a list, an Splus 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 Splus environment.

The `simp` function is a straight forward implementation of Simpson's 1/3 rule, and has no special provisions for use with S. 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 commands:

```
    cc -KPIC -c simp.c
    cc -G -o simp.so simp.o
```

Finally, the Splus 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(C.symbol("dosimp")))dyn.load.shared("./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 Splus function to be numerically integrated; 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.