

# Dynamic Loading on an SGI in Splus

B. Narasimhan  
Department of Statistics  
Stanford University  
Stanford, CA 94305

Version of May 20, 1997

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>The Problem</b>                      | <b>1</b> |
| 1.1      | Compiling the Fortran routine . . . . . | 2        |
| 1.2      | Creating the shared library . . . . .   | 2        |
| 1.3      | Shortcut . . . . .                      | 2        |
| 1.4      | Loading the shared library . . . . .    | 2        |
| 1.5      | Invoking the routine. . . . .           | 3        |
| <b>2</b> | <b>Tips</b>                             | <b>3</b> |
| <b>3</b> | <b>Troubleshooting</b>                  | <b>3</b> |
| <b>4</b> | <b>Remarks</b>                          | <b>4</b> |
| <b>5</b> | <b>Indices</b>                          | <b>5</b> |
| 5.1      | Code Chunks . . . . .                   | 5        |
| 5.2      | Index of Identifiers . . . . .          | 5        |

## Abstract

This little note tells you how to dynamically load Fortran and C programs in Splus on our SGI server where shared libraries are used.

## 1 The Problem

Consider the following fortran subroutine that takes three arguments, an array  $x$ , an integer  $n$  that indicates the length of the array and a storage area `result` where it should return the sum of the elements in the array.

```
1  <Fortran subroutine 1>≡
      subroutine sumvec(x,n,sum)
      real x(n), sum
      integer n
      sum = 0.0
      do 10 i=1,n
          sum = sum + x(i)
10  continue
      return
```

end

Defines:

sumvec, used in chunk 3.

Uses n 3 and x 3.

How can one call such a function from Splus? Perform the following steps.

- 2a *<Steps 2a>*≡  
*<Compile the fortran routine 2b>*  
*<Create a shared library 2c>*  
*<Load the shared library into Splus 2e>*  
*<Invoke the fortran routine 3>*

## 1.1 Compiling the Fortran routine

Assuming that the fortran program is stored in the file foo.f do the following.

- 2b *<Compile the fortran routine 2b>*≡ (2a)  
 rgmiller> f77 -c foo.f  
 Defines:  
 foo.o, used in chunk 2c.

## 1.2 Creating the shared library

Let us called the shared library libfoo.so. Create the shared library as follows.

- 2c *<Create a shared library 2c>*≡ (2a)  
 rgmiller> Splus SHLIB -o libfoo.so foo.o  
 Defines:  
 libfoo.so, used in chunks 2 and 3.  
 Uses foo.o 2b.

## 1.3 Shortcut

In recent versions of Splus, the two steps mentioned above can be combined into one.

- 2d *<Shortcut 2d>*≡  
 rgmiller> Splus SHLIB -o libfoo.so foo.f  
 Uses libfoo.so 2c.

## 1.4 Loading the shared library

Load the shared library into Splus as follows.

- 2e *<Load the shared library into Splus 2e>*≡ (2a)  
 > dyn.load.shared( "./libfoo.so" )  
 Uses libfoo.so 2c.

| Splus storage mode | C                            | Fortran          |
|--------------------|------------------------------|------------------|
| "logical"          | long *                       | LOGICAL          |
| "integer"          | long *                       | INTEGER          |
| "single"           | float *                      | REAL             |
| "double"           | double *                     | DOUBLE PRECISION |
| "character"        | char **                      | CHARACTER (*)    |
| "complex"          | struct {<br>double re, im;}* | DOUBLE COMPLEX   |
| "list"             | void **                      |                  |

Table 1: Data type correspondence between Splus, C and Fortran

## 1.5 Invoking the routine.

The function can now be invoked from Splus. However a few words of warning before we illustrate the use. Splus stores its variables in its own format and they must be coerced into a format palatable to fortran. One uses the `storage.mode` function of Splus for this purpose. Table 1 from Venables and Ripley[2] below shows the correspondence between Splus, C and Fortran variables.

So here is an example invocation.

```
3 <Invoke the fortran routine 3>≡ (2a)
> dyn.load.shared("./libfoo.so", symbols=symbol.F(sumvec))
> x <- 1:10
> storage.mode(x) <- "single"
> n <- as.integer(length(x))
> y <- 0
> storage.mode(y) <- "single"
> .Fortran("sumvec", x, n, result = y)
```

Defines:

n, used in chunk 1.

x, used in chunk 1.

Uses `libfoo.so 2c` and `sumvec 1`.

The result is a list with all the arguments you passed. You can refer to the result as usual via `$result`.

Note carefully how the storage modes are coerced to match the single precision reals that fortran expects. If some variables are double precision and others single precision, make sure that you coerce the storage modes appropriately.

## 2 Tips

- Debug your Fortran or C programs *before* calling them from Splus. As statisticians, we would like to avoid confounding!
- You cannot use `print` statements in your Fortran routines without some hacks (and hacks they really are). See the next section for more information. This restriction does not apply to C programs. (No, `f2c` will not help here.)
- Note how subroutines rather than functions are used.

## 3 Troubleshooting

Most errors are likely to be due to lax attention to storage aspects. So the first question you should ask yourself is whether you've coerced the storage modes of your arguments correctly.

## Frequently Asked Questions

1. I get an error message such as:

```
Error in .C("S_QPE_shobjlist_load",: Can't load (dlopen) library ./libfoo.so:
21859:/usr/local/lib/S-PLUS/cmd/Sqpe: rld: Fatal Error: unresolvable
symbol in ./libfoo.so: bar
```

This is due to the fact that your program references other functions or routines that could not be found. Make sure that you have all of them available. In the above example, the function `bar` could not be located. If the file `bar.c` contains that function, compile it and put both `foo.o` and `bar.o` into the library. The short way to do this is of course to use:

```
rgmiller> Splus SHLIB -o libfoo.so foo.c bar.c
```

2. I get an error message such as: `Error in .C("S_QPE_shobjlist_load",:...)`, even when I see that there is a file `libfoo.so` in my directory.

Use the full pathname of the file such as `dyn.load.shared("./libfoo.so")`. One can set environment variables like `LD_LIBRARY_PATH` but I have good reasons to maintain that the former is the easiest and cleanest solution.

3. A friend gave me his Fortran code for doing *blah*. I want to call the routine from `Splus`. If I do the straightforward thing, I get errors. How should I proceed?

Follow these steps.

- (a) Is the code giving you a function or a subroutine? If the former, convert it into the latter. This is trivial to do if you know fortran.
- (b) Is the routine using `print` or `write` statements? If so, comment them out. If the program is small enough, this is usually easy to do. However, such statements might provide important diagnostic information. In other situations, the program might be so large that it might be impossible to comment all of them out safely without clobbering the program. Then you must use the routines `DBLEPR`, `REALPR`, or `INTPR` described in [2] for printing the output. If the `print` or `write` statements provide no useful information that can be obtained otherwise, you can trick `Splus` by writing some dummy routines. Again, refer to [2].

4. I made a change to my file `foo.c`, recompiled and reloaded the dynamic library using `dyn.load.shared` and yet my program behaves exactly as if the old version of `foo.c` is in effect. What gives?

Read the help page for `dyn.load.shared` carefully. Use the `symbols` argument to force `dyn.load.shared` to read in the redefined routines.

The dynamic loading mechanism loads the definition of a symbol by first checking if it is already defined. If so, no loading of the symbol is done. Thus, if you repeatedly load a shared library using the bare `dyn.load.shared` function, you'll be effectively doing nothing. The `symbols` argument forces reloading of the symbol definition. *It is therefore prudent to use the `symbols` argument until you have completely debugged your dynamically loaded routines.*

## 4 Remarks

In my opinion, the whole issue of dynamic loading is made unnecessarily complicated in `Splus` by having functions like `dyn.load`, `dyn.load2`, and `dyn.load.shared`. There should be one function called `dyn.load` that does the appropriate thing for the platform. For example, `Lisp-Stat[1]`, which borrowed some of the dynamic concepts

ideas from *S* has done this very thing: just one function called `dyn-load` does different things on different platforms. I hope this gets cleaned up soon.

Dynamically loading C routines is really no different, except that the symbols can be specified to the `dyn.load.shared` function via `symbol.C` instead of `symbol.F`.

## 5 Indices

### 5.1 Code Chunks

This index is generated automatically. The numeral is that of the first definition of the chunk.

*<Compile the fortran routine 2b>*

*<Create a shared library 2c>*

*<Fortran subroutine 1>*

*<Invoke the fortran routine 3>*

*<Load the shared library into Splus 2e>*

*<Shortcut 2d>*

*<Steps 2a>*

### 5.2 Index of Identifiers

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

`foo.o`: 2b, 2c

`libfoo.so`: 2c, 2d, 2e, 3

`n`: 1, 3

`sumvec`: 1, 3

`x`: 1, 3

## References

- [1] Luke Jon Tierney. *Lisp-Stat. An object-Oriented Environment for Statistical Computing and Dynamic Graphics*. John Wiley & Sons, 1990.
- [2] W. N. Venables and Brian D. Ripley. *Modern Applied Statistics with S-Plus*. Springer Verlag, 1994.