Workshop: High-performance computing for economists

Lars Vilhuber[1] John M. Abowd[1] Richard Mansfield[1]
Hautahi Kingi[1] Flavio Stanchi[1] Sida Peng[1]
Kevin L. McKinney

[1]Cornell University, Economics Department,

August 17-19, 2015

Basic subroutine programming

Goal

- Show the basics of proper subroutine programming
- Advantages, pitfalls
- Examples in R
- Later: generalization and differences in other programming languages

Control structures in programming languages

Mostly generic

- `if, else`: testing a condition [R, SAS]
- `for`: execute a loop a fixed number of times [R, in SAS: `do`]
- `while`: execute a loop while a condition is true [R,SAS]
- `until`: execute a loop until a condition is true [SAS]
- `repeat`: execute an infinite loop [R]
- `break`: break the execution of a loop [R, SAS]
- `next`: skip an interation of a loop [R]
- `return`: exit a function [R]

Control structures: if

Control structures: if

... in R

```
if(<condition>) {
## do something
} else {
## do something else
}
if(<condition1>) {
## do something
} else if(<condition2>) {
## do something different
} else {
## do something different
}
```

Control structures: if

... in R

```r
if(<condition >) {
## do something
} else {
## do something else
}
if(<condition1 >) {
## do something
} else if(<condition2 >) {
## do something different
} else {
## do something different
}
```

... in SAS

```sas
if (<condition>) then do;
## do something
end; else do;
## do something else
end;
if (<condition1>) then do;
## do something
else if (<condition2>) then do;
## do something different
end; else do;
## do something different
end;
```

Control structures: for

Run through a fixed sequence of numbers (or in R, a sequence of vectors)

Control structures: for

Run through a fixed sequence of numbers (or in R, a sequence of vectors)

simple loop in R

```
for(i in 1:10) {
print(i)
}
```

Control structures: for

Run through a fixed sequence of numbers (or in R, a sequence of vectors)

simple loop in R

```
for(i in 1:10) {
print(i)
}
```

... in SAS

```
do i = 1 to 10;
put i;
end;
```

Control structures: for

Across programming languages, some flexibility:

Control structures: for

Across programming languages, some flexibility:

Equivalent loops in R

```
x <- c("a", "b", "c", "d")
for(i in 1:4) {
print(x[i])
}
for(i in x) {
print(i)
}
for(i in 1:4) print(x[i])
```

... in SAS

```
do i = 1 to 10;
put i;
end;
```

Interrupting loops

How a loop ends

- at the end
- by resetting the counter to the end value, or by setting the looping condition to its exit value explicitly
- when it encounters a `break` (for `repeat` loops in R)

What do we loop over?

Within the loop, something is done

```
for (i in 1:100000) {
## do something here
# stuff (1 line)
# stuff (2nd line)
# done
}
```

What do we loop over?

Within the loop, something is done

```
for (i in 1:100000) {
## do something here
# stuff (1 line)
# stuff (2nd line)
# done
}
```

What if that is really complicated?

```
for (i in 1:100000) {
## do something really complicated here (398 lines)
}
```

Subroutines

Breaking into discrete chunks

```
for (i in 1:100000) {
## do something really complicated here
## (398 lines)
}
```

Subroutines

Breaking into discrete chunks

```
for (i in 1:100000) {
## do something really complicated here
## (398 lines)
}
```

In the previous example, this was unwieldy (which loop are you closing on line 400?)

Subroutines

Breaking into discrete chunks

```
for (i in 1:100000) {
## do something really complicated here
## (398 lines)
}
```

In the previous example, this was unwieldy (which loop are you closing on line 400?)

We can start by breaking out the complicated stuff into a well-defined subroutine.

Subroutines

Breaking into discrete chunks

```
for (i in 1:100000) {
## do something really complicated here
## (398 lines)
}
```

We can start by breaking out the complicated stuff into a well-defined subroutine.

In the previous example, this was unwieldy (which loop are you closing on line 400?)

```
for (i in 1:100000) {
## new subroutine
do_something_complicated(args=something)
##
}
```

Subroutines, procedures, etc.

What do you call a subroutine?

"The same things in different [programming] languages can have different names. Programs, Procedures, Functions, **Subroutines**, Subprograms, Subqueries ... these words all have very similar meanings. [...] We can call an object, it executes and performs a complex process. Whether that object is called any one of the above list depends on what programming language it is written in, whether a human being can call it, or whether it has to be called by another program or one of the other names on the list."

Source

An practical overview of subroutine programming

We will use R...

... but expand to cover what this might look like in SAS (macros), Stata (programs, ado files), Matlab (functions) tomorrow.

... use this as a building block to scale to HPC.

You already use them!

Much functionality in SAS, Stata, Matlab, R is implemented as a subroutine:

- `proc import` in SAS (compare call to log file)
- Most statistical functions in Stata (`regress` used earlier is actually in `../ado/base/r/regress.ado`)

You can create them too!

Functions in R

R objects of class ”function”

```
f <- function(<arguments>) {
## do something here
}
```

- Functions can be arguments to other functions
- Functions can be nested (even recursive)
- Functions can have named arguments with default values, or missing values

We draw on Peng’s ”Computing for Data Analysis, Week 2” for this section.

Basic examples

Let's start with a simple example

```
# This is a very simple function
# It simply draws a draws from a normal with mean=b
f <- function(a=1,b=0) {
        rnorm(a,mean = b)
}
```

Basic examples

Let's start with a simple example

```
# This is a very simple function
# It simply draws a draws from a normal with mean=b
f <- function(a=1,b=0) {
        rnorm(a,mean = b)
}
```

The function is defined, has two arguments with default values, and a really short name.

Basic examples

Let's start with a simple example

```
# This is a very simple function
# It simply draws a draws from a normal with mean=b
f <- function(a=1,b=0) {
        rnorm(a,mean = b)
}
```

The function is defined, has two arguments with default values, and a really short name.

Results

```
> source("3-1-simple-function.R")
> f
function(a=1,b=0) {
        rnorm(a,mean = b)
}
> f()
[1] 0.1709822
> set.seed(10)
> f(a=10,b=5)
 [1] 5.018746 4.815747 3.628669 4.400832 5.294545 5.389794 3.791924 4.636324
 [9] 3.373327 4.743522
```

Defining function arguments

Functions have *named* arguments which potentially have *default* values.

- The formal arguments are the arguments included in the function definition
- Not every function call in R makes use of all the formal arguments
- Function arguments can be missing, or can have default values

Function arguments at invocation

Function arguments can be passed by position (*positional*) or by explicit name (in which case, position doesn't matter).

Listing 1: Equivalent calls

```
> set.seed(10)
> mean(f(b=5,a=10))
[1] 4.509343
> set.seed(10)
> mean(f(a=10,b=5))
[1] 4.509343
> set.seed(10)
> mean(f(10,5))
[1] 4.509343
```

However, it is **good practice** to use *named* arguments, as they make use of the function (especially when more complex) easier and more transparent.

Lazy evaluation

Arguments can remain unused:

```
f <- function(a, b) {
a^2
}
> f(2)
[1] 4
```

`b` was never used.

Lazy evaluation

Arguments can remain unused:

```
f <- function(a, b) {
a^2
}
> f(2)
[1] 4
```

`b` was never used.

Arguments can be faulty, but don't lead to problems until used

```
f <- function(a, b) {
print(a)
print(b)
}
> f(45)    # note no value provided for b
[1] 45
Error in print(b) : argument "b" is missing, with no default
>
```

The absence of a value for `b` only led to an error at the time it was called.

Scoping

Scope of functions

```
f <- function(a,b) {
   print(a)
   print(c)
   c <- paste(a,b)
   print (c)
}
```

Try it out:

Scoping

Scope of functions

```
f <- function(a,b) {
    print(a)
    print(c)
    c <- paste(a,b)
    print (c)
}
```

Try it out:

```
> f(c('a'),c('b'))
[1] "a"
function (..., recursive = FALSE)  .Primitive("c") <== ERROR!
[1] "a b"
```

Scoping

What happened?

`c` was not defined, leading to the error on line 4. Line 5 reports what happens to `c` once it is defined.

Understanding the scope

```
5 > c <- c('Nothing')
6 > f(c('a'),c('b'))
7 [1] "a"
8 [1] "Nothing"
9 [1] "a b"
```

So what value does `c` now have?

Scoping

What happened?

`c` was not defined, leading to the error on line 4. Line 5 reports what happens to `c` once it is defined.

Understanding the scope

```
12  > c <- c('Nothing')
13  > f(c('a'),c('b'))
14  [1] "a"
15  [1] "Nothing"
16  [1] "a b"
```

So what value does `c` now have?

```
17  > print(c)
18  [1] "Nothing"
```

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.
- R took the “global” value of `c` ...

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.
- R took the “global” value of `c` ...
- until the function defined it, at which point it took a new value internal to the function

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.
- R took the "global" value of `c` ...
- until the function defined it, at which point it took a new value internal to the function
- ... which was only used until the function ended. The "global" value had not changed.

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.
- R took the "global" value of `c` ...
- until the function defined it, at which point it took a new value internal to the function
- ... which was only used until the function ended. The "global" value had not changed.

Scoping

Understanding the scope of variables

- In the example, `c` was defined both inside the function and outside.
- R took the “global” value of `c` ...
- until the function defined it, at which point it took a new value internal to the function
- ... which was only used until the function ended. The “global” value had not changed.

Other programming languages

Each programming language has a different way of handling these. Take care to read up on it.

Searching for variables (and functions)

- The global environment or the user?s workspace is always the first element of the search list and the **base** package is always the last.
- The order of the packages on the search list matters!
- User?s can configure the order of packages as they get loaded on startup ...
- When a user loads a package with library the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.

Scoping (name resolution) rules

Lexical scoping

Lexical (or **static**) resolution can be determined at compile time, and is also known as **early binding**

Dynamic scoping

Dynamic resolution can in general only be determined at run time, and thus is known as **late binding**.

Most modern languages use lexical scoping for variables and functions, though de facto dynamic scoping is common in macro languages, which do not directly do name resolution. [1]

Scoping rules

Back to R

```
f <- function(a,b) {
    print(paste(a,b))
    print(c)
}
```

- Named (formal) arguments (`a,b`) are always local
- Free variables (not defined in the call) (`c`) are subject to scoping rules

Lexical scoping in R

Lexical scoping in R means that

> *the values of free variables are searched for in the environment in which the function was defined.*

What is an environment?

- *An environment is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value.*
- *Every environment has a parent environment; it is possible for an environment to have multiple ?children?*
- *A function + an environment = a closure or function closure.*

Safe use of scoping

Define all variables as arguments

To avoid confusion about where variables are defined, unless you have a good reason to deviate from this

- Define all variables as arguments to the function
- In other languages, explicitly define the scope of a variable

Scoping in SAS

Scope of macro variables in SAS

Define a similar program in SAS

```
%macro myprogram(a=,b=);
%let c=&a.&b.;
%put &c.;
%mend;
```

Scoping in SAS

Scope of macro variables in SAS

Define a similar program in SAS

```
%macro myprogram(a=,b=);
%let c=&a.&b.;
%put &c.;
%mend;
```

Call it and assess where `c` comes from:

```
%myprogram(a=a,b=b);
%put &c.;
%let c=Nothing;%put &c.;
%myprogram(a=a,b=b);
%put &c.;
```

Scoping in SAS

with results

```
1     %macro myprogram(a=,b=);
2     %let c=&a.&b.;
3     %put &c.;
4     %mend;
6     %myprogram(a=a,b=b);
ab
WARNING: Apparent symbolic reference C not resolved.
7     %put &c.;
&c.
8     %let c=Nothing;%put &c.;
Nothing
9     %myprogram(a=a,b=b);
ab
10    %put &c.;
ab
```

Scoping in SAS

Better:

```
%macro myprogram(a=,b=);
%local c;
%let c=&a.&b.;
%put &c.;
%mend;
%let c=Nothing;
%myprogram(a=a,b=b);
ab
%put &c.;
Nothing
```

Efficient use of scoping

Differences that need to be taken into account

- Scoping rules can be leveraged to improve optimization (see Peng's Coursera course and others)
- The use of scoping differs across languages (what is feasible in R cannot be simply translated into Java or SAS or Stata)

Naming rules

Naming is important

- Naming is important - both of functions and of arguments: Compare the following two functions:

```
f <- function(a,b,c) {
    x <- sample(y,c)
    lm(a ~ b , data=x )
}
```

and

```
sample_reg <- function(lhs,rhs,samplesize=10,data=) {
    subset <- sample(data,samplesize)
    lm(lhs ~ rhs , data=subset )
}
```

- Give functions and variables meaningful names

Documentation

Provide some source-code level documentation

```
# Author: Lars Vilhuber
# This program defines a regression on a sample.
#
# Usage: sample_reg(lhs,(list of rhs),samplesize=10,data=data frame)
#
# Samplesize defaults to 10 observations, and will sample without replacement
# Regression is simple linear regression with no options
# Function returns an object of type lm

sample_reg <- function(lhs,rhs,samplesize=10,data=) {
    subset <- sample(data,samplesize)
    lm(lhs ~ rhs , data=subset )
}
```

- More sophisticated documentation: see package documentation

Naming rules

Robustness is important

- Think about backward compatibility (and your program library). Say you first used this:

```
sample_reg <- function(lhs,rhs,samplesize=10,data=) {
    subset <- sample(data,samplesize)
    lm(lhs ~ rhs , data=subset )
}
sample_reg(earnings,education,5,data=cps)
```

Naming rules

Robustness is important

- Think about backward compatibility (and your program library). Say you first used this:

```
sample_reg <- function(lhs,rhs,samplesize=10,data=) {
    subset <- sample(data,samplesize)
    lm(lhs ~ rhs , data=subset )
}
sample_reg(earnings,education,5,data=cps)
```

- Now you extend your model

```
sample_reg <- function(lhs,rhs,samplesize=10,data=) {
    subset <- sample(data,samplesize)
    lm(log(lhs) ~ rhs , data=subset )
}
sample_reg(earnings,education,5,data=cps)
```

Naming rules

Robustness is important

- You should either give your function a different name, or make the call robust to both the “old” way and the “new” way:

```
sample_reg <- function(lhs,rhs,samplesize=10,data=,logs = false ) {
    subset <- sample(data,samplesize)
    if ( logs ) {
     _lhs <- log(lhs)
     }
     else {
     _lhs <- lhs
     }
    lm(_lhs ~ rhs , data=subset )
}
```

which will work for both calls...

The power of functions

The power of functions

Why bother with functions?

- Initial example: putting 398 command lines into separate file (ease of use)
- Expansion on that: re-using the function across multiple projects (function library)
- Your function is a complete specification. You know want to vary or perturb all 25 parameters slightly, for robustness checks.

```
for (i in 1:1000) {
  for (j in 1:1000) {
   my_regression(model=base, xi=i, xj=j)
  }
}
```

The power of functions

Why bother with functions?

- You want to database the results:

```
for (i in 1:1000) {
  for (j in 1:1000) {
   results_db[i,j]=my_regression(model=base,xi=i,xj=j)
  }
}
```

The power of functions

The ultimate power of functions: scaling

- You want to speed the whole thing up by parallelizing:

```
library(doMC)
registerDoMC()
foreach (i = 1:1000, .combine=cbind) %dopar% {
  for (j in 1:1000) {
   results_db[i,j]=my_regression(model=base,xi=i,xj=j)
  }
}
```

which will run 1000 parallel threads, on as many cores as you can. (see doMC vignette or the help in your Rstudio installation)

Take-away

Major points

- Subroutines are a powerful tool to write clean, understandable code
- Subroutines (functions, macros, programs) are present in some form in all statistical programming languages
- Use consistent, clear naming
- Use robust subroutines, expanding them into a library that you can use and share across projects
- Clean subroutines are a critical component to scaling your analysis (parallelization)

Take-away

Additional items

- learn how to debug (different in each language, also critical to scaling)
- subroutines don't magically make your code efficient - they allow you to figure out which portions are not

Take-away

Additional items

- consider publishing your routines
  - R packages `http://r-pkgs.had.co.nz/` with publication in R Journal
  - Stata packages (user-contributed commands, ado files) from the Boston College Statistical Software Components (SSC) archive with publication in Stata Journal
  - Journal of Statistical Software
  - Github... (Stata, R devtools)