A SIMPLE PROTOCOL FOR SIMULATIONS IN R

ROGER KOENKER

Abstract. It is easy to get sloppy with the organization of simulation exercises and this usually results in painful experiences at the latter stages of projects, and ultimately to the irreproducibility of results. If one could adhere to a simple, consistent protocol I believe many of these problems could be avoided.

1. The Four Commandments

I propose the following rules:

1: Always use `set.seed( )`.
2: Never run simulations interactively, always run them from source files.
3: Document the initial environment.
4: Save the final environment.

2. Some Implementation Details

There are undoubtedly many ways to implement these rules; I will briefly sketch one way that seems to be quite convenient. Suppose that we create a source file called `sim.R` that looks like this:

```
# toy MC experiment to test the Gossett binary response estimator

source("plink.R")
system("hostname")
date()
sessionInfo()

R <- 500
n <- 500
set.seed(1968)
x <- 5*rnorm(n)
dfs <- c(1,2,6)
A <- array(0,c(3,3,R))
for(i in 1:length(dfs)){
        df <- dfs[i]
        print(paste("i = ",i))
        for(j in 1:R){
```

Version August 5, 2014. This research was partially supported by NSF grant SES-02-40781.

```
                        y <- (1.0 * x + rt(n,df) > 0)
                        f <- pglm(y ~ x,link="Gossett")
                        A[,i,j] <- c(f$nuhat,f$nulo,f$nuhi)
                        }
                }
```

We begin by `source()`ing some functions from the file `plink.R`. These functions and any data residing in `plink.R` will be saved as part of the global environment at the end of the process and will therefore be available for post-mortem analysis. Next we record the name of the machine, the date and some basic information about the version of R and the versions of the packages that we have attached. Then we initialize the seed for the random number generator and get to work. The print statement is used to monitor progress of the job by occasionally peeking in the resulting `sim.Rout` file using, e.g. the shell command `tail`. In more complicated situations it would often be preferable to write a function that encapsulated a single iteration of the simulation.

The file `sim.R` would normally be invoked from the command line with,

```
R CMD BATCH sim.R
```

By so doing we automatically generate two new files: `sim.Rout` which contains a transcript of the executed session, and `.RData` which by default contains a binary version, in standard R format, of the global environment including all functions and data as it existed at the end of the session. It seems prudent to rename this file to something more meaningful. The process can be automated by the following shell script: These four rules produce a pair of new files `sim.Rout` and `sim.Rda`.

```
R CMD BATCH $1.R
mv .RData $1.Rda
chmod -w $1.R
```

This approach enforces a consistent naming convention. Calling the script `Rbatch`, one would simply invoke it with

```
Rbatch sim &
```

This automatically moves the standard output file `.RData` created by the simulation into a more specifically named file `sim.Rda` that can be preserved for posterity.

Remarks

(i) The use of `set.seed( )` is essential if one wants to ensure the reproducibility of results. The R random number generators allow one to restart the sequence by simply resetting the seed to the same value that was used previously.

(ii) The batch approach provides an explicit record of what was done. It also, conveniently, provides automatic timing information on the run at the end of the `sim.Rout` file.

(iii) The `sessionInfo( )` call documents the version of R and any included packages, thereby enabling restoration of the environment used to create the results. Note that prior versions of R and its packages are available from CRAN. This resolves a long-standing problem with proprietary software for which it was difficult, or even impossible, to restore prior versions of the software to repeat a prior analysis.

(iv) Use of `save.image( )` at the end of the session, which occurs automatically (by default with the `R CMD BATCH` command), ensures a complete record of what was produced by the originating batch file, and can be easily restored using `load( )`. This allows one to separate the relatively time consuming simulation phase of the project from the analysis phase. The latter typically requires relatively little computational effort but may involve considerable fine tuning to produce appropriate tables and graphics. It is advantageous to have separate files for each table and figure, each of which can rely on load to recreate the output of the simulation. For procedures that produce large `.Rda` files one could also add a compression step. The latter can be accomplished automatically by setting,

```
options(save.image.defaults = list(compress=TRUE))
```

either in the `sim.R` file or in the `.Rprofile` file that gets sourced automatically at startup.

(v) The array `A` that is produced by the simulation can usually be easily manipulated using R's `apply` function to produce tables and or figures as appropriate. For tables intended for $\LaTeX$ documents there are several very useful tools for semi-automatically generating $\LaTeX$ code available from the `Hmisc` package of Frank Harrell. The `quantreg` package also contains a `latex.table` function adapted from an early version of Harrell's function of the same name.

3. The `foreach` Wrinkle

The `doMC` package provides a convenient way to use multicore machines to do simply parallelized simulations, but it also offers some opportunities to destroy the reproducibility of what seemed to be a simple simulation exercise. I think that I now understand how to rectify this. The basic issue is: How does one insure that each of the pieces of a `foreach` loop starts with the same seed? If this behavior is not what is desired, then I have no advice.[1] An important caveat – thanks to Tom Parker for this – is that one needs to be sure that the number of cores used in the foreach doesn't exceed the number of cores available on the machine being used, if it does then seeds are updated when cores are recycled.

Here is a some test code to illustrate how this works:

```
# Test of set.seed of foreach command
   require(doMC) #loads multicore and foreach automatically.
```

[1]It may seem weird to have each instance of the loop starting from the same seed, but the way that I usually do simulations it seems natural: each instance of the foreach typically represents a different distributional assumption, or sample size, or some other feature of the model, so starting each at the same seed means that any one of the instances can be reproduced by simply doing it individually using the original seed.

```
    registerDoMC(5) # intention to use 5 cores
 # Now the setup for the simulation
 date()
 system("hostname")
 sessionInfo()
 set.seed(1968)
 J <- 5
 opt <- list(set.seed = FALSE)
 AJ <- foreach(j = 1:J, .options.multicore = opt) %dopar% {
     A <- sum(rnorm(10))
     }
```

As we can see from examining the AJ object produced, each component of the list is identical. In this code we have to set the options for multicore in a somewhat unintuitive way in order to get each of the processes to use the same initial seed. But once this is done things seem to go as desired.

Of course if we don't want to use the same seed in each foreach instance then we can simplify this somewhat.

```
# Test of set.seed of foreach command
    require(doMC) #loads multicore and foreach automatically.
    registerDoMC(5) # intention to use 5 cores
    # Now the setup for the simulation
    date()
    system("hostname")
    sessionInfo()
    J <- 5
    seeds <- 1:J
    AJ <- foreach(j = 1:J) %dopar% {
                 set.seed(seeds[j])
         A <- sum(rnorm(10))
     }
```

In this case we know how to assign seeds to each instance and all the seeds and results are available in the final `sim.Rda` object.

4. The Cluster

The next stage of complication is expanding the use of `foreach` from a desktop machine like my mac pro to a cluster like UIUC's new LAS cluster. For the moment I have a rather primitive elaboration of the foregoing that seems to work. I would appreciate any suggestions of refinements. Essentially this requires replacing the parallel backend `doMC` by the cluster version `doMPI`.

The easy part of the cluster transition was getting R installed and in my case installing Mosek and some related packages, this was all quite straightforward and could be done without any administrative privileges within my own home directory.

Once this was done I created a sbatch file `foo.s` that looked like this:

```
#!/bin/tcsh

#SBATCH --job-name=foo
#SBATCH --partition=e
#SBATCH -n 21
#SBATCH --time=48:00:00
#SBATCH --mem-per-cpu=2048
#SBATCH --mail-type=FAIL
#SBATCH --mail-type=END
#SBATCH --mail-user=rkoenker@illinois.edu

mpirun -np 21  R CMD BATCH foo.R
```

This provides a jobname, specifies that the job is destined for the economics nodes, a maximum time limit of 48 hours, maximum memory of 2G, etc. The `mpirun` command runs a R batch job specified by the file `foo.R`. A simple example `foo.R` looks like this:

```
# Pseudo simulation exercise for testing script to automate cluster job control
# The specifics are obviously silly since this could be done more efficiently.

require(doMPI)
cl <- startMPIcluster()
registerDoMPI(cl)

date()
system("hostname")
sessionInfo()
opt <- list(set.seed = FALSE)

set.seed(22)
R <- 50
trims <- (1:7)/20
ns <- c(100,500,1000)
dfs <- 1:5
G <- expand.grid(ns,dfs)
files <- paste("f",1:nrow(G), sep = "")
A <- array(0, c(length(trims), R))
```

```
AK <- foreach(k = 1:nrow(G), .options.multicore = opt) %dopar% {
    n <- G[k,1]
    df <- G[k,2]
    file <- files[k]
    for(j in 1:R){
       x <- rt(n, df)
       for(i in 1:length(trims)){
          A[i,j] <- mean(x, trim = trims[i])
       }
    cat(j, "\n",  file = file) # Monitor progress
    }
    A
}
save.image()
closeCluster(cl)
mpi.quit()
```

Now provided that both `foo.R` and `foo.s` are in the same directory on the cluster I can simply invoke:

```
sbatch foo.s
```

to run the job, which will then get distributed to the various nodes of the cluster.

The job runs producing the usual log file called in this case `foo.Rout` and an output file called `.RData` containing the results of the job. Apparently, the R packages are found and promulgated to the nodes. As I understand it, on the LAS cluster we have 12 core machines so I suppose if one wanted 48 cores, one could rescale the "-n" line in Rbatch, but I've not yet tested this. There are a variety of open questions about how to deal with scheduler options, that I also hope to sort out soon.

The downside of initializing the job on the cluster is that it creates lots of opportunities for file name confusion since there are multiple copies of files on one's desktop machine and on the cluster. And the file `foo.s` seems a bit redundant; since it would seem to be easily constructed automatically, thereby avoiding clashes of filenames. In this spirit, I've tried to construct a shell script that would generate a `foo.s` file send it and the associated `foo.R` file to the cluster, execute the job, and hopefully return the output files to my desktop machine. This is a work in progress, but the first stages seem to do what was intended. I will update this as I learn more about how to do things. (This involves relearning what little I once knew about shell programming from ancient times at Bell Labs.) The following (Bourne) shell script lives in my bin directory as `Cbatch`.

```
#!/bin/sh

pwd=$PWD
while :
do
```

```
    case "$1" in
-n) shift; n="$1";;
-*) usage "bad argument $1";;
*) break;;
    esac
    shift
done
job=$1

cat > $job.s <<%
#!/bin/tcsh

#SBATCH --job-name=$job
#SBATCH --partition=e
#SBATCH -n $n
#SBATCH --time=48:00:00
#SBATCH --mem-per-cpu=2048
#SBATCH --mail-type=FAIL
#SBATCH --mail-type=END
#SBATCH --mail-user=rkoenker@illinois.edu

mpirun -np $n  R CMD BATCH $job.R
mv .RData $job.Rda
scp $job.Rda $job.Rout roger@yzzy.econ.illinois.edu:$pwd
%
scp $job.R $job.s rkoenker@keeling:SimBin
ssh rkoenker@keeling "cd SimBin; sbatch $job.s"
```

Obviously, this assumes that loginless access to the cluster has been setup for ssh, and the short pathname for keeling is known. Equally obviously, it assumes that access is permitted the other direction so the output files can be returned to the desktop machine. This is what is happening at the very end of the so-called "Here document," i.e. in the scp line before the trailing percent sign. The `$pwd` variable assures that the output files go back to the directory from whence they were generated. Finally, any files that the job expects to be able to source have to already be in the directory `SimBin`.

This script is invoked on my desktop like this:

```
Cbatch -n 15 foo
```

I suppose that it should go without saying, but since I seem to have a difficult time adhering to this procedure I'll say it anyway: Always run a prototype simulation with a trivial number of replications on a noncluster machine for debugging purposes. It is very difficult to track down bugs on the cluster since various pieces of the job are running on different nodes and cores. Running the simulation in a simply nested for loop allows one to use `traceback` and other debugging tools. Another useful strategy is to combine outer loops using `expand.grid( )`, as illustrated in the example above. I've also found it useful to include the file progress monitor strategy adopted there, in case there is a crash and one would like to reconstruct the data for which it occurred. At the very least running a small version of the simulation permits one to get a rough estimate of the time to completion for the full version.

DEPARTMENT OF ECONOMICS, UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN