How to start using R on Hammer and LionX.

Extended Notes for Statistical Computing on Hammer & LionX Machines
Research Computing and Cyberinfrastructure Group (RCC)
Pennsylvania State University

1 Instruction

If you have already done any type of computation with R on our clusters, the chances are that you don't need to read this extended note. Just to have your click/download worthwhile please check our new FAQ page. You'll find few interesting points in this page even if you've been using our systems for a year or two. I've also put few FAQ for R on our clusters in section 3 of this extended note.
If you don't know what is Hammer or what is RCC Group, please read section Two thoroughly, especially the set up in step-3.
I hope you find this material useful.

2 R Workshop Basics

If you have never used RCC resources, make sure you follow these steps once before the workshop:

1- To set up your psu wifi please visit:
psu wifi link
2- To create an account on Hammer please fill out the request form from here, one or two days in advance:
Account Request link
3- To download and set up Exceed on demand please visit:
Exceed on demand link
4- While you have an account on Hammer and Exceed installed on your laptop please follow the steps here to connect to Hammer:
connect to hammer link
5- To get rstudio loaded and running please read the instructions from here:
rstudio launch link will be added soon
for now, in a new terminal, just type in:

```
module load rstudio
rstudio
```

3 Extended FAQ for R

One major question that non-R users usually ask first is about "how validated" R is as an open source computing package. For a quick answer, see R: Regulatory Compliance and Validation Issues A Guidance Document for the Use of R in Regulated Clinical Trial Environments.
With this being said, here is a short list of FAQ for our R package on Hammer and LionX, which is aimed to be beginner-friendly.

3.1 Question: What is a good GUI for R?

rstudio, all small letters, is the available de-facto GUI on our clusters for R. It loads the latest installed version of R while you load it.

3.2 Question: How can I manage R packages?

R is an open source statistical software made up of many user-written packages. The base version of R that is downloaded allows the user to get started in R, but anyone performing data analysis will need to install additional packages at some point. Please send us the library name, and the machine you are running your jobs on and we will get them installed so that you don't have to install it every time on your local temporary folders.

3.3 Question: Which packages do I already have?

To see what packages are installed, use the installed.packages() command. Below, we look at the first few libraries.

```
> installed.packages()[1:5,]
        Package   LibPath                        Version
bitops  "bitops"  "C:/Users/win-library/2.15" "1.0-5"
caTools "caTools" "C:/Users/win-library/2.15" "1.14"
digest  "digest"  "C:/Users/win-library/2.15" "0.6.2"
foreign "foreign" "C:/Users/win-library/2.15" "0.8-52"
Hmisc   "Hmisc"   "C:/Users/win-library/2.15" "3.10-1"
        Priority      Depends
bitops  NA            NA
caTools NA            "R (>= 2.2.0)"
digest  NA            "R (>= 2.4.1)"
foreign "recommended" "R (>= 2.14.0), stats"
Hmisc   NA            "R (>= 2.4.0), methods, survival"
        Imports                        LinkingTo
bitops  NA                             NA
caTools "bitops"                       NA
digest  NA                             NA
foreign "methods, utils"               NA
Hmisc   "lattice, cluster, survival" NA
        Suggests
bitops  NA
caTools "MASS, rpart"
digest  NA
```

```
foreign NA
Hmisc   "lattice, grid, nnet, foreign, chron, acepack,
TeachingDemos,\nrms, cluster, mice, subselect, tree"
        Enhances OS_type License      Built
bitops  NA       NA      "GPL (>= 2)" "2.15.2"
caTools NA       NA      "GPL-3"      "2.15.2"
digest  NA       NA      "GPL-2"      "2.15.2"
foreign NA       NA      "GPL (>= 2)" "2.15.2"
Hmisc   NA       NA      "GPL (>= 2)" "2.15.2"
```

From this output, we will first focus on the Package and Priority columns. The Package column gives the name of the package and the Priority column indicates what is needed to use functions from the package.
If Priority is "base", then the package is already installed and loaded, so all of its functions are available upon opening R.
If Priority is "recommended", then the package was installed with base R, but not loaded. Before using the commands from this package, the user will have to load it with the library command, e.g. library(boot).
If Priority is NA, then the package was installed by the user, but not loaded. Before using the commands from this package, the user will have to load it with the library command, i.e., library(car).

3.4 Question: How can I add or delete packages?

Answer: Any package that does not appear in the installed packages matrix must be installed and loaded before its functions can be used. A package can be installed using install.packages(" < package name> "). A package can be removed using remove.packages(" < package name> "). However, since as a student user, you don't have the root access, you may not be able to install the package in the directory you want. Please contact us and we will take care of it for you.

3.5 Question: What packages are available?

Answer: The list of available R packages is constantly growing. The actual list can be obtained using available.packages(). This returns a matrix with

a row for each package.

```
> P_avail<- available.packages()
> di(P_avail)
Error: could not find function "di"
> dim(P_avail)
[1] 4502   13
> P_avail[1:5,]
```

These first five (of 4,502) available packages illustrate that the package names are often acronyms and rarely reveal what the package functions do. A list of the packages available through CRAN including a short package description can be found at CRAN's Contributed Packages page. Question

3.6 What functions and data-sets are in a package?

Answer: It is easy to access some quick documentation for a package from R with the help command. The MASS package is used as an example here.

Listing 1: getting information on packages

```
help(package="MASS")
```

Once a package is loaded, the help command can also be used with all functions and data-sets listed here, e.g. help(Null).

3.7 Question: How can I time my code?

Answer: If you are using a Batch system, e.g. LionX machines. just add time at the beginning of the line where you call R. Also, adding few more lines to your PBS also could do the same. See below:

```
#   This is a PBS script. Here for example it will request
#   1 processor #on 1 node  for 24 hours.
#   Request 1 processors on 1 node
#
```

```
#PBS -l nodes=1:ppn=1
#
#   Request 4 hours of walltime
#
#PBS -l walltime=4:00:00
#
#   Request 4 gigabyte of memory per process
#
#PBS -l pmem=4gb
#
# Request that regular output and terminal output go to the same
#
#PBS -j oe
#
#   Add the following batch commands prior to your code execution.

echo " "
echo " "
echo "Job started on `hostname` at `date`"

make
# some more lines where you execute your code

# Add these few more lines to also record the ending of the
#execution.
echo " "
echo "Job Ended at `date`"
echo " "
```

Alternatively there are two R commands for timing code: proc.time and system.time. We want to add 1 to each of the values of a 100,000 element vector. We will do this with and without looping, timing both for comparison. The proc.time command essentially works as a stop-watch: you initialize it to a starting time, run all the code desired, and then stop it by subtracting the starting time from the ending time. First we use the slow, looping method to add 1 to each value in our vector:

```
> g <- rnorm(100000)
```

```
> h <- rep(NA, 100000)
>
> # Start the clock!
> ptm <- proc.time()
>
> # Loop through the vector, adding one
> for (i in 1:100000){
+     h[i] <- g[i] + 1
+ }
>
> # Stop the clock
> proc.time() - ptm
   user  system elapsed
   0.45    0.02    0.47
Alternatively, we can use a non-looping approach:
> ptm <- proc.time()
> h <- g + 1
> proc.time() - ptm
   user  system elapsed
      0       0       0
```

The values presented (user, system, and elapsed) will be defined by your operating system, but generally, the user time relates to the execution of the code, the system time relates to your CPU, and the elapsed time is the difference in times since you started the stopwatch (and will be equal to the sum of user and system times if the chunk of code was run altogether). Note the difference in their proportions! Its significant! The system.time command takes a single R expression as its argument. Thus, to repeat the steps above using system.time, we wrote function wrappers around our fast and slow methods. Again we see that the looping method is much slower. Note that system.time is simply calling proc.time! So the better one to use just depends on the nature of the code you wish to time.

Listing 2: system.time command in R

```
> slowadd <- function(g){
+     h <- rep(NA, length(g))
+     for (i in 1:length(g)){
+         h[i] <- g[i] + 1
```

```
+     }
+     return(h)
+ }
>
> system.time(a <- slowadd(g))
   user  system elapsed
   0.41    0.00    0.41

> system.time(a <- slowadd(g))
   user  system elapsed
   0.39    0.00    0.39
> system.time(a <- quickadd(g))
   user  system elapsed
      0       0       0
```

3.8 Question: How to implement C codes into R

On the process to achieve this C creates something called shared library. When R as an interpeting language reads it, a gsl library is needed as well. You want a -lgsl on your compile line to link against the library, and will need to have the library loaded when running your code in R. Also, Make sure, you have an explicit object file name on your compilation line.

This is a sample compiling options that you could use directly

```
gcc -Wall -lgsl  -fPIC -I/usr/global/gsl/1.13/include
-I$/gpfs/home/yulXX/work/CtoR -L/usr/global/gsl/1.13/lib
-std=c99 -fopenmp -c hmmutils.c
```

3.9 Do certain nodes run much slower than the rest?

Not necessarily. Depends on the cluster. On Cyberstar for example there are three different node types in CyberSTAR: Intel Nehalem nodes, AMD Shanghai nodes, and Intel Westmere nodes. The Westmere nodes are the fastest and the Shanghai nodes are the slowest. You can request a particular node type by adding 'westmere', 'nehalem', or 'shanghai' as a feature request in your PBS node request. For example:

```
#PBS -l nodes=1:westmere:ppn=1
```

Note that requesting a specific node type will probably increase the wait time of your jobs since a smaller portion of nodes will available to your jobs. All information regarding our cluster node types, etc. is found on our website for your review.

References

[1] RCC webiste: rcc.its.psu.edu