Advanced Topics in R Programming

Batch Jobs
Garbage Collection
Memory Management
Debugging

Duncan Temple Lang
duncan@wald.ucdavis.edu

Topics

- In the 2 lectures I will present, we'll try to cover:
  - General questions (R, ad hoc networks, programming, etc.)
  - Batch & Background jobs.
  - Garbage collection.
  - Managing memory.
  - Debugging.
  - Recursive functions.
- Notes at http://eeyore.ucdavis.edu/stat133/

"Batch" Jobs

- Usually run R commands interactively.
- But if they take a long time, you want to leave them and come back when they are finished.
- Can lock the screen - BAD
- Instead, use a batch or background job using the shell
- Important part of Scientific Computing.

Batch Jobs

- Put the R commands into a file, say code.R.
- Run R reading commands from that file
  put output into another file
- R --no-save < code.R >& output.Rout
- --no-save just tells R not to bother saving the work space when it finishes
- Other possible options are --vanilla, --save, --no-environ, etc.
  See documentation for R shell command, ?Startup

```
R --no-save < myCode.R >& output.Rout
```

- What does the < mean?
- The shell "redirects input" to R using the contents of the file myCode.R
- Very similar to typing the lines one at a time at the R prompt
  Not quite the same as source("myCode.R"), but close.

5

```
R --vanilla < myCode.R >& output.Rout
```

- The >& means "redirect both output and errors" to the file output.Rout
- If we just had
  ```
  > output.Rout
  ```
  the errors would go to the console/terminal.
- The >& is specific to the C shell (csh/tcsh)
  For the Bourne shell, bash/sh use
  R --no-save < myCode.R 2>1 > output.Rout

6

Background Jobs

- We still had to wait for the
  R --no-save < myCode.R ...
  command to finish before we start new command (in that terminal)
- If we logout, the process will terminate!
- We want to get a new prompt so we can do other things, including logging out.

7

```
nohup nice +18 R --no-save < myCode.R >&
              output.Rout &
```

- The second & tells the shell to put this process in the background and return the a new prompt.
  No connection to the >&.
- nice says "schedule my job when others aren't using the computer".
- +18 is the maximum amount of niceness
- Prefix command with nohup - no hangup.
  On many machines this is not needed, but it never hurts and guarantees the job keeps running when you logout.

8

Things to Remember

- Can logout and return later to see if the job is finished.
- First, remember which machine you used.
  Often people check on the wrong machine
- Has the task finished?
- If you arrange for your code to generate output at different points, you can look at the output file and look for those markers, e.g. print something at the end of iteration of a loop.
- To look at the file,
  `cat output.Rout` or `tail -f output.Rout`

9

General Job Monitoring

- Each job or "process" has a unique identifier - a number.
- kestrel>/app/bin/R --no-save < long.R >& out &
  [1] 19766
  The 19766 is the process identifier.
- Use the commands `top` and `ps` find status of machine, and job.
- Use kill to force a job to finish
  `kill -9 19766`

10

More to remember

- When creating plots, explicitly open graphics devices and close them.
- pdf("myPlot.pdf")
  hist(x)
  dev.off()
- This avoids them going to one big file, on different pages.

11

And more

- If your job stops unexpectedly, you will have to start again from the beginning.
- Sometimes useful to save results as you go along, i.e. at different stages/parts of the script.
- save(a, b, c, file = "myFile.rda")
- Then you can come back and reload them and continue on from that point or do additional computations.

12

Debugging

- If you get an error in your script, the job will stop and there will be a message in output.Rout.
- Hopefully the message will make it clear how to fix the problem.
- Often we need to examine the state of the session to figure out why things failed.
- So we need to be able interactively explore the values of the different variables

Post-morten Debugging

- First of all, test code on smaller datasets.
- But if it does happen in a batch job, we don't have interactive access!!
  Can't use `options(error = recover)`
- Do "post-mortem" debugging (see `?debugger`)
- At start of script (myCode.R), put

```
options(error=quote({dump.frames(to.file=TRUE); q()}))
```

- Then, after the error can explore in new R session

```
load("last.dump.rda")
debugger(last.dump)
```


Debugging

- This debugger is basically the same as the one used in interactive use
  e.g. with options(error = recover)
- Jump to different calls, find out what variables are available, print values, do computations.
- Debugging is an art. Get experience.
- Think about probable causes and then try to construct experiments to verify that is the reason.

15

What is Garbage Collection?

- Notice that in R, when you create data you don't have to explicitly declare or allocate it.
- And you don't have to release it.
- e.g.
  x = 2*x + 10 + rnorm(length(x))
  the rnorm()s are created added to the other components and then discarded. Same for original x.
- Garbage collection is the process of reclaiming the memory that is associated with objects and computations that are no longer being used - garbage.

16

- When R needs memory to do a computation, it asks its memory manager for space.
- The memory manager has already allocated a lot of space that it doles out, and so it can provide space for such requests.
- If the memory manager doesn't have enough space for the request, then it tries to cleanup - garbage collect.
- It runs through all the spaces that it has given out in earlier requests and reclaims it if it is no longer being used.
- If the Mem. Mgr. still needs more space, it can grow its pool.

Preallocate Space for the Result

- Reiterating what Deb covered last time.
- Consider the following code

```
ans = numeric()
for(i in 1:n)
   ans = cbind(ans, foo(i))
```

- In each step, we combine the new result with the previous ones via cbind.

- Consider the last iteration, i.e. i == n
- The result from the previous iteration is a matrix with n-1 columns.
- We then create a new result with n columns.
- So before we assign the new result to ans, we have approximately 2 copies of the results!
- And we have to copy all the data from the original to the new result.
- This is bad news.
  Some computations will not be feasible.

Alternative

- We know the result is a matrix of size m x n,
  so allocate it first and then assign each iteration's result into the corresponding column.
-

```
ans = matrix(NA, m, n)
for(i in 1:n)
  ans[, i] = foo(i)
```

- This does the allocation (for the result) just once and doesn't create new objects, just modifies the existing one.
- The key thing is that ans[, i] doesn't create a new copy of ans, but writes the values into the appropriate subset.

Time Comparisons

- system.time({ans = numeric() ; for(i in 1:10000) ans = cbind(ans, rnorm(10))})
  [1] 14.57 4.72 19.62 0.00 0.00
- system.time({ans = matrix(NA, 10, 10000) ; for(i in 1:10000) ans[,i] = rnorm(10)})
  [1] 0.32 0.01 0.34 0.00 0.00
- Of course, need to have multiple measurements.
- And the characteristics of the machine, etc. matter, but still can compare the two meaningfully.

- We could use apply() to make this read more easily and be more efficient
  sapply(1:n, function(i) rnorm(10))
- The apply functions allocate the result space for us.
- Note that we can define an "anonymous" function in the call to sapply().
  functions are first class objects in R.
- But when we can't use an apply function, making space and writing into that existing space is much faster.

Why do we need to know this?

- Because, when you run simulations as for your current project, you may run into memory problems.
  It then helps to be able to reason about them.
- It is good to be able to determine approximately how much memory you will need in a computation. Then you can determine if it is feasible or not.
- And it can also allow you to specify hints to R for how much space it will need and can reserve.

- Before we discuss how to control garbage collection, let's just see when it happens.
- gcinfo(TRUE) tells R to print something on the screen when it performs the garbage collection.
- > gcinfo(TRUE)
- Then allocate a big object several times
  > for(i in 1:10) m = rnorm(100000)

- Garbage collection 5 = 4+0+1 (level 0) ...
88686 cons cells free (25%)
2.8 Mbytes of heap free (47%)
Garbage collection 6 = 5+0+1 (level 0) ...
88672 cons cells free (25%)
2.1 Mbytes of heap free (34%)
Garbage collection 7 = 6+0+1 (level 0) ...
88658 cons cells free (25%)
1.3 Mbytes of heap free (21%)
- So 3 calls to the garbage collector when it needed more space and was done with the earlier versions of m.

Size of an object

- Find out how big an object is.
- Back of the envelope calculations, or
- object.size(x)
e.g. object.size(matrix(pi, 100, 100))
[1] 80120
- 100 x 100 = 10,000 elements
each element 8 bytes for a number
extra bytes (120) associated with R information
(dimensions, class, etc.)
- object.size(letters)
[1] 1068

- > p = cbind(runif(75), runif(75))
> D = as.matrix(dist(p))
> object.size(D)
[1] 51276
- This is the number of bytes the distance matrix occupies.

- object.size() can be used to approximately determine how much space you might need for a calculation.
- Things are more complex than just being the sum of all the necessary space as R may reuse some of that space and so need less space.
- And R may use a little more for an object.
- But it gives us a good estimate of the approximate requirements.
- Also, we can compute this for different sizes of inputs and extrapolate.

- Suppose we have a computation that works on an n objects, e.g. 75 nodes in an ad hoc network.
- We write a function to determine if the network is fully connected.
- How many nodes can we realistically run this for?
- Try it for n = 5, 10, 15, 20, …., 100 and compute the total memory and time used during that computation.
- Then extrapolate to get approximate memory and time as a function of n.

- Find out how much space is being used using gc()
  Forces garbage collection, and also reports status.
- ```
  > gc(reset = TRUE)
         used (Mb) gc trigger (Mb) max used (Mb)
  Ncells 171168  4.6     350000  9.4   350000  9.4
  Vcells  62857  0.5     786432  6.0   337583  2.6
  ```
- The Vcells are the interesting ones for us - our computations and data.
- Note that gc() returns a matrix with 2 rows, 6 columns.
- We can extract the max used columns and compare across calls.

- Call gc(reset = TRUE), run computation and then call gc(reset = FALSE)
- ```
  > orig = gc(reset = TRUE)
  > sapply(1:10, simulation)
  > end = gc()
  ```
- ```
  end["Vcells", 6] - orig["Vcells", 6]
  ```


- gcinfo()
- object.size()
- gc(reset = TRUE) followed by gc()
  ```

- Start R with a large workspace
- R --min-vsize=vl --max-vsize=vu --min-nsize=nl --max-nsize=nu --max-ppsize=N
- mem.limits()
- ?Memory

- Start R with default settings
  R
- gc()

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 169451  4.6     350000  9.4   350000  9.4
Vcells  62425  0.5     786432  6.0   337539  2.6
```

- Now, let's create a large matrix - 1000 x 1000
- Before we do, ask R to tell us when it does garbage collection/resizing of the available space.

- gcinfo(TRUE)
- m = matrix(rnorm(1000 * 1000), 1000, 1000)
-
```
Garbage collection 4 = 1+0+3 (level 2) ...
180323 cons cells free (51%)
9.6 Mbytes of heap free (95%)
Garbage collection 5 = 1+0+4 (level 2) ...
180330 cons cells free (51%)
9.6 Mbytes of heap free (54%)
Garbage collection 6 = 1+0+5 (level 2) ...
180333 cons cells free (51%)
9.6 Mbytes of heap free (37%)
```

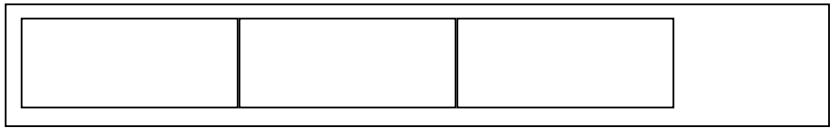
-
```
object.size(m)
[1] 8000120
```

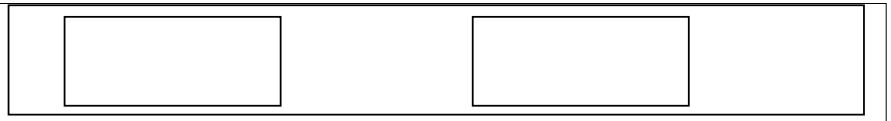

- Now, let's try that again, but this time start R with 1Gb of memory.
  Don't do this unless you know you need it!
- Start R and tell it to use 2Gb of space for data objects
  R --min-vsize=2G

- Again, turn on reporting of garbage collection
  gcinfo(TRUE)
- Now, allocate the same matrix.
  m = matrix(rnorm(1000 * 1000), 1000, 1000)
- Note, there was no garbage collection.

Fragmentation

- Fragmentation happens when we create numerous objects and then remove some and leave holes in the allocated memory.
- x1 = rnorm(10000)
  x2 = rnorm(10000)
  y = 10 * x1 + 20 * x2
  rm(x2)





- When we remove x2, we are left with a big hole.
- If we go to allocate space for say 10001 elements, we cannot use this space.
- We may have lots of little pieces of space which cumulatively total more than the desired amount of new space.
- But since they are not contiguous, we cannot use them and so we cannot satisfy the new request.
- We don't have much control over this in R, but it is good to know about it.

Quick Aside

- Statistical Computing is an important and very under-represented area of research in the statistics field.
- There are many opportunities to do research in this area.
- We are developing a progam in Davis and there are several others emerging.