Running R in batch mode on Linux

Jan 15, 2014 • ericminikel

After spending my past year as more or less a solo computational scientist at the Center for Human Genetic Research, I've now started a new job with the research team at the Daniel MacArthur lab. That means much bigger datasets and a need to be much more smart about programming. Gone are the days of one-off scripts to be used for one dataset on one machine and then never used again. Here to stay are the days of the reusable, the modular, and the parallel.

One part of this upgrade is a need to be doing most of my analysis in portable R scripts that can be run in parallel with command line arguments. So for instance, instead of the first line of the file being `file = read.table("myfile.txt")` the script needs to able to be run with `myscript.r --file myfile.txt`. This way another wrapper script can fire off hundreds of instances of the same script running on different input files.

It turns out there are several things that need to be in place in order to start running R scripts in batch mode, and there are a few other people in my department who are just figuring out how to do this too, so I figure it's worth sharing in a post. This way, the people who know what they're doing and know what I'm doing wrong can comment and improve too.

FYI, I'm running Linux and using R 3.0.2:

```
$ uname -a
Linux node1376 2.6.18-194.8.1.el5 #1 SMP Thu Jul 1 19:04:48 EDT 2010 x86_64 x86_64 x86
$ which R
/broad/software/free/Linux/redhat_5_x86_64/pkgs/r_3.0.2/bin/R
$ which Rscript
/broad/software/free/Linux/redhat_5_x86_64/pkgs/r_3.0.2/bin/Rscript
```

1. use Rscript

First things first: the best program to run R scripts in batch mode is `Rscript`, which comes with R. There is also `R CMD BATCH`, which I used to use, but apparently this is no longer preferred. To give you an idea of the difference, let's try the simplest possible example script:

```
print("hello world")
```

Save this script as helloworld.r and then try running it with either command:

```
$ Rscript helloworld.r
[1] "hello world"
$ R CMD BATCH helloworld.r
$
```

Rscript prints its output to `stdout`, while it appears at first glance that `R CMD BATCH` has done nothing at all. In fact, `R CMD BATCH` has written its output to a file called `helloworld.r.Rout`, and that output includes both the commands and output, just like in interactive mode, along with some runtime stats:

```
> print("hello world")
[1] "hello world"
>
> proc.time()
   user  system elapsed
  0.479   0.092   0.534
```

There might be someway to change this default output mode, but I'm not sure; either way, let's use `Rscript` instead.

2. run Rscript with a shebang

If you tried following along with the above example and it didn't work (despite having R installed), it might be because you don't have R loaded, which can be done with `use`:

```
use R-3.0
```

I have the above line of code in my `~/.my.bashrc` file which is executed each time I log into the cluster.

Instead of typing `Rscript helloworld.r`, though, you can run the script itself as an executable by having the first line start with a shebang, which is `#!`, followed by the path to the place where Rscript is installed, so in my case:

```
#!/broad/software/free/Linux/redhat_5_x86_64/pkgs/r_3.0.2/bin/Rscript
print("hello world")
```

But if you just try and run this, you'll get a message like:

```
$ helloworld.r
bash: helloworld.r: command not found
```

There are three (3) problems you need to fix to be able to do this trick:

1. The path to Rscript listed in your shebang line needs to be in your `PATH` variable, which I accomplish by adding one line to my `~/.my.bashrc` script:
   ```
   export PATH=$PATH:/broad/software/free/Linux/redhat_5_x86_64/pkgs/r_3.0.2/bin/
   ```
2. Your program, `helloworld.r`, needs to be executable in terms of its permissions. `chmod +x helloworld.r` will set just the executable permission bit, or you can set some combination of bits that works for your needs, like `chmod 755 helloworld.r`.
3. The operating system needs to be able to find this executable file, so you either need to add `.` to your PATH as well (not recommended), or just call the script specifying the current directory, so `./helloworld.r` (recommended).

With that fixed, let's try again:

```
$ chmod +x helloworld.r
$ ./helloworld.r
[1] "hello world"
```

If you're like me, you'll find that 90% of the time you forget to type the `./`, so it's worth a reminder that ctrl-a returns you to the beginning of the line so that you can fix this sort of thing without just pressing the back arrow for half your life.

3. use optparse to read command line arguments

R has native capability to accept command line arguments using `commandArgs` but if you want to do anything flexible (i.e. optional arguments), then you'll find yourself reinventing a lot of wheels. As helpfully explained here, there are three helpful R packages to choose from: `getopt`, `optparse` and `argparse`. I chose to use optparse on the basis that (1) its dependencies are simple – doesn't require Python; but (2) it works similarly to optparse in Python, so I figure I'll be learning how to do this in two languages at once.

The vignette for `optparse` has an example script, but even after looking at that example, it took me a long time to figure out how to really use the package. I'm not sure if it's a difference of R version or operating system, but the example in the vignette does things which just plum *don't work* on my system. In a moment I'll walk through a simple example I cooked up that *does* work on my system. But first:

4. use `cat()` to write output

If you've only run R in interactive mode before, you now need to figure out how to write R's output to a place where you can look at it after your script finishes running. As far as I can tell, there are three options for this, none of them (IMHO) perfect: `cat`, `write`, and `print`.

What sucks about `print` is that there is apparently no way to have it *not* print numbers such as `[1]` in front of your output, e.g.:

```
> print("hello world")
[1] "hello world"
```

For what it's worth, you can turn off the quotes with `quote=FALSE`, and you can choose where the output goes using `file=stdout()` or `file=stderr()`:

```
> print("hello world",quote=FALSE,file=stderr())
[1] hello world
```

Still, not being able to turn numbers off is a dealbreaker for me.

`write` is just a wrapper that adds line breaks (and not much else?) to `cat`. What sucks about both of them is that they can't write any objects that are too complicated. To wit:

```
> df = data.frame(a=c(1,2,3),b=c(4,5,6))
> write(df)
```

```
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

There are various workarounds for stuff like this, but what a pain. Still, these seem to be the “correct” answer, and both `write` and `cat` are used in the `optparse` vignette. I guess the key is, only use them to output simple strings to the user. If you have a bunch of data to output, hopefully you’re using `write.table` anyway.

example

Here’s a really simple example script demonstrating all of the above tricks:

```
#!/broad/software/free/Linux/redhat_5_x86_64/pkgs/r_3.0.2/bin/Rscript

# Eric Vallabh Minikel
# CureFFI.org
# 2014-01-14
# example of how to use optparse in R scripts

# usage: ./exampleRScript1.r -a thisisa -b hiagain
#        ./exampleRScript1.r --avar thisisa --bvar hiagain

suppressPackageStartupMessages(require(optparse)) # don't say "Loading re
# manual: http://cran.r-project.org/web/packages/optparse/optparse.pdf
# vignette: http://www.icesi.edu.co/CRAN/web/packages/optparse/vignettes/

option_list = list(
  make_option(c("-a", "--avar"), action="store", default=NA, type='charac
              help="just a variable named a"),
  make_option(c("-b", "--bvar"), action="store", default=NA, type='charac
              help="just a variable named b"),
  make_option(c("-v", "--verbose"), action="store_true", default=TRUE,
              help="Should the program print extra stuff out? [default %d
  make_option(c("-q", "--quiet"), action="store_false", dest="verbose",
              help="Make the program not be verbose."),
  make_option(c("-c", "--cvar"), action="store", default="this is c",
              help="a variable named c, with a default [default %default]
)
opt = parse_args(OptionParser(option_list=option_list))

if (opt$v) {
    # you can use either the long or short name
    # so opt$a and opt$avar are the same.
```

```
    cat("avar:\n")
    cat(opt$avar)
    cat("\n\na:\n")
    cat(opt$a)

    # show the user what b and c are
    cat("\n\nb:\n")
    cat(opt$b)
    cat("\n\nc:\n")
    cat(opt$c)
    cat("\n\n")

    # show the user the difference between cat, write and print
    cat("cat(opt$c): \n")
    cat(opt$c) # does NOT produce its own \n
    cat("\n\nwrite(opt$c,file=stdout()): \n")
    write(opt$c,file=stdout()) # does produce its own \n
    cat("\n\nprint(opt$c,quote=FALSE): \n")
    print(opt$c,quote=FALSE) # no way to remove [1] from in front of line
    cat("\n\n")
}

# main point of program is here, do this whether or not "verbose" is set
if(!is.na(opt$avar) & !is.na(opt$bvar)) {
    cat("here are strings a and b together at last:\n")
    cat(paste(opt$a,opt$b,sep=''))
    cat("\n\n")
} else {
    cat("you didn't specify both variables a and b\n", file=stderr()) # p
}
```

exampleRScript1.r hosted with ♡ by GitHub view raw

And here's one example of how I'd run it:

```
chmod +x exampleRScript1.r
./exampleRScript1.r -a thisisA -b thisisB
```

Now for a step-by-step walkthrough to show all the things that are tricky about this. Indeed, part of what's tricky here is that if you're used to using R in interactive mode, you're used to a really easy debugging process. But when you're learning out how to deal with command line parameters, you can only debug that by calling from the command line, at which point it gets harder because the program

will finish and close before you can interrogate what went wrong. (I suppose you could, in your script, save the workspace to a .Rdata file and then open that in interactive mode later to reconstruct the wreckage. If anyone knows a more ingenious solution let me know). Anyway, this took a while for me to figure out, so here are some tricky points.

First, note the use of `default=NA` in this bit:

```
  make_option(c("-a", "--avar"), action="store", default=NA, type='character',
              help="just a variable named a"),
```

If you don't assign a default, then if the user doesn't specify that flag, the variable simply doesn't exist. This becomes a problem later. For example, you may want to only do X action if variable Y has been set by the user, and while R can do `exists("a")`, it *can't* do `exists("opt$a")`, which is what you'd need since all of the user-set flags come through as elements of `opt` or whatever you want to name your variable in this line:

```
opt = parse_args(OptionParser(option_list=option_list))
```

Second, note the use of `type='character'` in the same section:

```
  make_option(c("-a", "--avar"), action="store", default=NA, type='character',
              help="just a variable named a"),
```

On my system, if I don't specify the `type` of each argument, then R simply doesn't see them *at all*. So for instance, if you take the above script and comment out `type='character'`, and then run it as follows:

```
$ ./exampleRScript1.r -a thisisa -b thisisb
```

Then `a` and `b` will both be `NA`. And *that's* only because I set `default=NA`. If you *don't* do that, then `a` and `b` will both be undefined.

Third, the above problems can be especially difficult to debug if you're also not sure whether you're even calling the script with the correct syntax on the command line. On my system, I found that the only way that works is to use the flag (whether long or short), then a space, then the value of the variable. Just as above:

```
$ ./exampleRScript1.r -a thisisa -b thisisb
```

In the `optparse` vignette, it shows the use of the `=` sign, so `./exampleRScript1.r -a=thisisa`, suggesting that must work on someone's system somewhere; but for me, using an equals sign brings this error upon my head:

```
Error in getopt(spec = spec, opt = args) :
  short flag "a" requires an argument, but has none
Calls: parse_args -> getopt
Execution halted
```

Do note, however, that you can use quotes if your variable value has spaces in it, e.g.:

```
$ ./exampleRScript1.r -a "hello world" -b thisisb
```

Fourth, note that once a command line parameter has been given, you are welcome to refer to it by its short or long name, so these two lines are identical:

```
cat(opt$avar)
cat(opt$a)
```

Fifth, note the use of `dest="verbose"` below:

```
  make_option(c("-v", "--verbose"), action="store_true", default=TRUE,
              help="Should the program print extra stuff out? [default %default]"),
  make_option(c("-q", "--quiet"), action="store_false", dest="verbose",
              help="Make the program not be verbose."),
```

This allows `-q` to set the same variable as `-v` rather than setting a new variable `opt$q`.

And sixth, if for some reason you're using `R CMD BATCH` rather than `Rscript`, I suspect none of this `optparse` business will work for you at all. When I tried `R CMD BATCH exampleRScript1.r -a thisisa -b thisisb`, I found that R interpreted the `-a` as a destination file for `stdout`, thus creating a file named `-a`. In turn, it was hard to figure out how to remove or even view this unwanted file because when I tried to run `rm -a` or `cat -a`, bash interpreted `-a` as an unknown command line parameter rather than as a filename. (The solution to this was `rm ./-a`and `cat ./-a`, as explained helpfully by the `rm` error message).

The other thing I'll mention quickly is that to make your code robust in batch mode, you will probably need to write more `tryCatch` blocks than you're used to in interactive mode. So see this nice introduction to R's `tryCatch`.

That's all – please let me know if this is helpful, and any corrections or suggestions of how to do this even better.