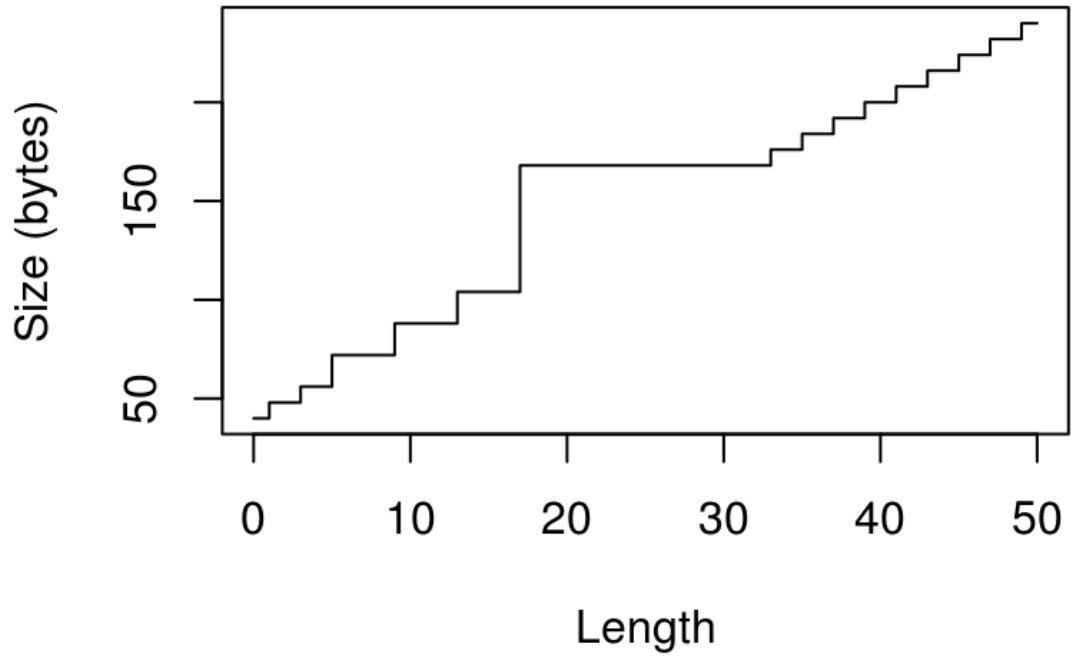
A survival guide to large-scale data analysis in R

Peter Carbonetto

Dept. of Human Genetics and the Research Computing Center
University of Chicago



Tutorial aims

1. Develop skills for large-scale data analysis in R, and apply these skills to a large-ish data set.
2. Learn how to use R **non-interactively** within a high-performance computing environment.
3. Make more effective use of the most precious commodity in computing—**memory**—and appreciate that R often makes poor use of it.
4. Learn through **live coding**—this includes learning from mistakes!

Outline of tutorial

- Brief introduction.
- Initial setup steps.
- Vignettes:
  1. How to measure memory in R, and use it effectively.
  2. How to automate an R analysis on the RCC cluster.
  3. Speeding up R with multithreaded matrix operations.*
  4. Multithreaded R computation with parLapply.*
  5. Distributing your computation on the RCC cluster using the SLURM engine.*

* Incomplete.

Elements of my hands-on tutorial approach

- Developing good tutorials is *hard*; developing good hands-on tutorials is *harder*.
- Don't have all the setup upfront.
- Follow Software Carpentry (and Stefano Allesina's) "live coding" approach, but not strictly.
- Don't make participants write their own code.
- Don't make participants use the RCC cluster.
- Don't provide a handout—all code should be in the slides.
- Flip between typing (running the examples) and talking about it (what happened?).
- Reduce dependencies between examples.
- Create slides using R Markdown:

```
cd docs
make slides
make test
```

It's your choice

Your may choose to...

- Work through the examples on the RCC cluster.
- Work through the examples on your laptop.
- Pair up with your neighbour.
- Follow what I do on the projector.

However,

1. A few of the examples will only run on the RCC cluster.
2. The examples may not produce the exact same result on your laptop.

Software requirements

1. R (ideally, version 3.4.0 or greater)
2. python 3.x
3. SLURM (this is the job scheduler on the RCC cluster)

Outline of tutorial

- Brief introduction.
- **Initial setup steps.**
- Vignettes.

Initial setup (part 1)

- WiFi
- Power outlets
- YubiKeys
- Pace, questions (e.g., keyboard shortcuts).

Initial setup (part 2)

Download the workshop packet to your laptop.

- URL: `https://github.com/pcarbo/R-survival-large-scale`
- Open **slides.pdf** from the **docs** folder. This is useful for viewing and copying the code from the slides.
- We may also take a look at some of the code in the **code** folder. The source code can be viewed with your favourite text editor, or browsed on GitHub.

Initial setup (part 3)

If you are using the RCC cluster, set up your cluster computing environment:

- Connect to midway2.
  - ▷ See `https://rcc.uchicago.edu/docs/connecting`
- Request a midway2 (“Broadwell”) compute node with 8 CPUs and 15 GB of memory:

```
screen -S r-tutorial  # Optional.
sinteractive --partition=broadwl --mem=15G \
  --cpus-per-task=8 --time=3:00:00
```

- Make note of the compute node you have connected to:

```
echo $HOSTNAME
```

Initial setup (part 4)

If you are using the RCC cluster, run these commands to download the workshop packet to your home directory (no spaces in URL):

```
cd $HOME
git clone https://github.com/pcarbo/
  R-survival-large-scale.git
```

Initial setup (part 5)

Next, we set up **htop** for monitoring processes.
Open up a separate Terminal window and connect to midway2.
Connect to the same compute node:

```
ssh $HOSTNAME
```

Copy my htop configuration:

```
mkdir -p ~/.config/htop
cp extras/htoprc ~/.config/htop/
```

Start htop:

```
htop --user=<username>
```

- Some useful htop keyboard shortcuts:
  - q to quit
  - M to sort processes by memory
  - p to toggle program paths
  - H to toggle threads (for multithreaded computing)

Initial setup (part 6)

Move to the **code** folder in the git repository. And on the RCC cluster, load Python3 and R:

```
cd code
module load python/3.5.2
module load R
```

Start up an interactive R environment:

```
R --no-save
```

Check which version of R you are running (hopefully version 3.4.0 or greater):

```
version$version.string
```

Also, before continuing, check your working directory:

```
getwd()   # Should be .../code
```

What's in the workshop packet

```
R-survival-large-scale
  /code     # Source code used in demos
            # (and R Markdown for slides).
  /data     # "Raw" and processed data.
  /docs     # Slides and other materials.
  /extras   # Additional useful files.
  /output   # All results are stored here.
```

Outline of tutorial

- Brief introduction.
- Initial setup steps.
- Vignettes:
  1. **How to measure memory in R, and use it effectively.**
  2. How to automate an R analysis on the RCC cluster.
  3. Speeding up R with multithreaded matrix operations.*
  4. Multithreaded R computation with parLapply.*
  5. Distributing your computation on the RCC cluster using the SLURM engine.*

* Incomplete.

Vignette #1: How to measure memory in R, and use it effectively

We have two tasks:

1. Compute the SNP minor allele frequencies (MAFs) from the a large genotype data matrix.
2. Center and scale the columns of the genotype matrix.

We will use R for both tasks.

- We will monitor memory allocation in R, and find ways to reduce it.
- To monitor memory allocation, we will use:
    1. **htop**
    2. **monitor_memory.py**, a Python script.

Vignette #1: How to measure memory in R, and use it effectively

Outline of the vignette:

1. Simulate a genotype data matrix, and save the data.
2. Monitor memory needed to compute MAFs.
3. Monitor memory needed to scale genotypes.
4. Attempt to reduce memory used in scaling genotypes.

Generate genotype data

We generate an $n \times p$ genotype matrix with $n = 1000$ samples, $p = 300{,}000$ SNPs and 1% missing genotypes.

```
source("sim.geno.R")
set.seed(1)
geno <- sim.geno(n = 1000,p = 3e5,na.rate = 0.01)
save(list = "geno",file = "../data/geno.RData")
```

Compute and summarize MAFs

We will run this code non-interactively to compute the MAFs:

```
# Compute MAFs.
maf <- colMeans(geno,na.rm = TRUE)/2
maf <- pmin(maf,1 - maf)
# Summarize distribution of MAFs.
print(quantile(maf,seq(0,1,0.1)),digits = 2)
```

- Quit R.
- Run these commands in the shell:

```
pwd # Should be .../code
Rscript summarize.maf.R
```

Monitoring memory allocation

While the R script is running, use **htop** in a separate Terminal session to profile memory allocation (look at the **RES** column).

- If necessary, re-run the code.

Re-run the R script again, this time using the Python program to measure memory usage every 0.01 s:

```
export MEM_CHECK_INTERVAL=0.01
./monitor_memory.py Rscript summarize.maf.R
```

How well does the **htop** estimate agree with **max rss_memory** from the Python script?

Center and scale the genotype matrix

Next, we will run this code non-interactively to center and scale the genotype matrix:

```
source("scale.R")
cat("Loading genotype data.\n")
load("../data/geno.RData")
cat("Centering and scaling genotype matrix.\n")
geno <- scale(geno)
```

Using the same methods as before, monitor memory usage.

```
./monitor_memory.py Rscript scale.geno.R
```

Does the scaling operation require more memory than computing MAFs?

- If so, how much more?
- How do you explain this result?

A better implementation of “scale”

Motivated by out observations in the previous slide, let’s try to reduce the memory usage by writing our own **scale** function.

```
scale_better <- function (X) {
  for (i in 1:ncol(X)) {
    X[,i] <- X[,i] - mean(X[,i],na.rm = TRUE)
    X[,i] <- X[,i] / sd(X[,i],na.rm = TRUE)
  }
  return(X)
}
```

Now run the script that uses our custom scaling function:

```
./monitor_memory.py Rscript scale.geno.better.R
```

Does **scale_better** require less memory than **scale**? Why? Is there more room to reduce memory usage?

Matrix scaling: a second attempt

This next script attempts to reduce memory usage even further by avoiding any function calls that modify `geno` within a local environment.

```
./monitor_memory.py Rscript \
  scale.geno.even.better.R
```

This is the code:

```
load("../data/geno.RData")
for (i in 1:ncol(geno)) {
  x           <- geno[,i]
  x           <- x - mean(x,na.rm = TRUE)
  x           <- x / sd(x,na.rm = TRUE)
  geno[,i] <- x
}
```

Is this an improvement over our previous attempt? Is there still room for improvement?

Matrix scaling: one last attempt using Rcpp

The root issue is that R does not allow for “copying in place”.

- We can circumvent this by implementing a C++ function *that modifies the input matrix directly.*
- See files **scale.cpp** and **scale.geno.rcpp.R** for how this is implemented using the **Rcpp** package.

Now try monitoring memory usage in the script that uses the Rcpp implementation:

```
./monitor_memory.py Rscript scale.geno.rcpp.R
```

Does this implementation have a smaller memory footprint than previous attempts? Is there still room for improvement?

Memory allocation in R: take-home points

- R duplicates objects aggressively ("copy on modify"). This can be an issue with objects are large.
- Rcpp can circumvent some of R's limitations, but requires much more effort!
- An interesting effort is the **purrr** package. But it does not yet handle matrices.
- To accurately assess memory needs, run the code in a fresh R session.
- For more advanced tools for memory profiling, see:
  - `http://adv-r.had.co.nz/memory.html`
  - `https://adv-r.hadley.nz/memory.html`
  - profmem R package

Outline of tutorial

- Brief introduction.
- Initial setup steps.
- Vignettes:
  1. How to measure memory in R, and use it effectively.
  2. **How to automate an R analysis on the RCC cluster.**
  3. Speeding up R with multithreaded matrix operations.*
  4. Multithreaded R computation with parLapply.*
  5. Distributing your computation on the RCC cluster using the SLURM engine.*

* Incomplete.

Vignette #2: How to automate an R analysis on the RCC cluster

Now that we have a good understanding of the memory needs for the matrix scaling task, let's automate the analysis.

- We want this script to work for any genotype data file provided as input, and we want to be able to choose the location where the scaled matrix is saved.

On the RCC cluster, this will involve two steps:

1. Develop an R script that accepts arguments from the command-line shell.
2. Develop a script for automating the job submission with the SLURM scheduler.

Running an R script from the shell

We take these steps to automate the analysis:

- Use the **commandArgs** function to read in command-line arguments.
- This is implemented in **scale.geno.automated.R**.

To illustrate its features, let's use **R CMD BATCH** instead of Rscript to run the script from the command line:

```
R CMD BATCH --no-save --no-restore \
  '--args ../data/geno.RData
   ../output/geno.scaled.RData' \
  scale.geno.automated.R \
  ../output/geno.scaled.automated.out
```

Submitting an R script to SLURM (part 1)

On the RCC cluster, a completely automated analysis should include instructions for allocating resources with SLURM.

- We still want this script to work for any input and output file.

This is the most basic script for accomplishing this:

```
#SBATCH --partition=broadwl
#SBATCH --mem=3G
#SBATCH --time=00:10:00
INPUT=${1}
OUTPUT=${2}
module load R/3.4.3
R CMD BATCH --no-save --no-restore \
  '--args ${INPUT} ${OUTPUT}' \
  scale.geno.automated.R \
  ../output/geno.scaled.automated.Rout
```

See also file **scale.geno.sbatch**.

Submitting an R script to SLURM (part 2)

Run this command to schedule an analysis:

```
sbatch scale.geno.sbatch ../data/geno.RData \
  ../output/geno.scaled.RData
```

While it is running, we can monitor the job's status:

```
squeue --user=<cnetid> | less -S
```

- Optionally, add the lines in **extras/bashrc** to your `~/.bashrc` file.
- What happens when we change "3G" to "2G" in the sbatch script?

Automating an R analysis on the RCC cluster: take-home points

- I *always* first run R code interactively (on smaller data sets) to assess time and memory needs.
- Automation is most useful when a script needs to be run many times, or for a long-running computation.
- For long-running scripts, it is helpful to print progress to the console, e.g., `cat("Centering and scaling genotype matrix.\n")`.
- A side benefit of the sbatch script is that it records the computational setup (compute time, memory, software used).
- Below we will see an example of combining R scripts with sbatch scripts in multiple layers.

Outline of tutorial

- Brief introduction.
- Initial setup steps.
- Vignettes:
  1. How to measure memory in R, and use it effectively.
  2. How to automate an R analysis on the RCC cluster.
  3. **Speeding up R with multithreaded matrix operations.***
  4. Multithreaded R computation with parLapply.*
  5. Distributing your computation on the RCC cluster using the SLURM engine.*

* Incomplete.

Vignette #3: Speeding up R with multithreaded matrix operations

- Many of the problems we solve in our area involve matrix-vector computations, often with large matrices and vectors.
- For these problems, we can get multithreading (parallel computation) for free because multithreaded operations are already implemented in many BLAS and LAPACK library configurations. However, R needs to be set up to take advantage of these configurations.
  - ▷ On midway2, R 3.4.3 is installed from source and linked to OpenBLAS.
- This is a very system-dependent feature of R! May require installation from source.
  - ▷ See also Microsoft R Open.

Vignette #3: Speeding up R with multithreaded matrix operations

First generate the data set by running:

```
Rscript sim.gwas.R
```

This will create a file `gwas.RData` in the data directory.

Vignette #3: Speeding up R with multithreaded matrix operations

Try running the analysis with no multithreaded matrix operations:

```
export OPENBLAS_NUM_THREADS=1
Rscript calc.pve.R
```

Then try running it with 8 threads:

```
export OPENBLAS_NUM_THREADS=$SLURM_CPUS_ON_NODE
Rscript calc.pve.R
```

- Explain how to interpret `system.time`.
- How much of a computation time improvement do we get from making more threads available to OpenBLAS?

Vignette #4: Multithreaded R computation with parLapply

Here we will run the same analysis as the previous vignette, but we will se `parLapply` to speed up the computation when multiple cores (CPUs) are available.
First try with 8 threads for OpenBLAS, and no parLapply multithreading:

```
export OPENBLAS_NUM_THREADS=8
Rscript calc.pve.multicore.R 1
```

Next, try with 8 threads for parLapply, and no multithreaded matrix operations:

```
export OPENBLAS_NUM_THREADS=1
Rscript calc.pve.multicore.R 8
```

Which parallel computing scheme is more efficient—parLapply, or multithreaded matrix operations?

Vignette #5: Distributing computation using SLURM

Add introduction to vignette here.

Some final thoughts

Add final thoughts and recap here.