Getting Work Done

Sean, Marc, Debbie, Bronson

12/09/2015

Contents

Introduction

We have spent the last 3 courses trying to show you the basic building blocks for command line computing. Now it is time to put all the pieces together and talk about the mechanics of actually getting your work done. To do this we will step through a small RNA-Seq workflow. We will not focus too much on the science behind the workflow at this time. The purpose of this course is to show you how to use the resources we have made available and how to use your new skills as a computational biologist. It is hoped that by going through this workflow, you will get some experience with how to organize your data flow and how to execute a series of commands that you can now generalize to other applications beyond RNA-Seq.

In order to save time and to help everyone keep up, we will be providing a text document that has all of the commands already populated. You might need to adjust one or two things, but otherwise you can just copy and paste. Here are a few tips and tricks regarding copying and pasting into a Linux terminal:

- To **copy** from a Linux terminal, all you need to do is highlight. The highlighted text is automatically copied to the clipboard. No other key strokes necessary.
- To **paste** into a Linux terminal, you cannot use the familiar `CTRL-V` keystroke. If using MobaXTerm, you can right click and paste using the menu. In all Linux systems though, you can usually paste using the middle button of the mouse (or by pressing the scroll wheel).
- You don't have to paste commands one at a time. You can paste in as many as you want, and the terminal will execute them in order. Sometimes though, you have to hit `Enter` once or twice more to make sure that all the commands are sent.

A (very) brief intro to RNA-Seq

RNA-Seq workflows are typically done with the aim of performing differential expression analysis. The idea is that instead of sequencing genomic DNA, instead you will sequence mRNA. By mapping the sequenced mRNA transcripts back to the reference genome, you can theoretically deduce A) what genes are being expressed based on the mapping and B) the relative expression levels of those genes based on how many reads map to a particular gene region. The basic, generic workflow for differential expression analysis goes like this:

1. Extract mRNA and convert to cDNA
2. Library preparation: fragment cDNA, size select, add adapters
3. Sequence ends (usually paired end sequencing)
4. Align reads against a reference genome
5. Perform counting of reads against a feature model (e.g. gene model, transcript model, exon model, junction model, etc)
6. Normalization of reads
7. Statistical comparisons of counts between groups (Hypothesis: There is no difference in relative expression levels of gene X between condition 1 and condition 2)

Other analysis goals of RNA-Seq include things such as alternative expression analysis, transcript discovery and annotation, allele specific expression, mutation discovery, fusion detection, and RNA editing. For this class, we will stick to differential expression analysis.

Using a text editor

Having a good text editor handy is very important to developing and working with scripting languages. Some important features in a good text editor include

- Saves file as plain text (therefore WordPad and other word processors are not appropriate)
- Can handle windows and Unix style line endings
- Can support search and replace with regular expressions
- Can provide line numbers
- Can optionally format keywords and tabs using the conventions and syntax of popular scripting languages (shell, python, perl, R, etc)

MobaXTerm has a decent editor built into it which you can access under the 'Tools' menu. Alternatively, you can use Notepad++ which we have included along with the MobaXTerm application in the ISInformaticsTransfer department share location: `\\childrens\research\ISInformaticsTransfer\CourseDownloads`

Exercise L0:

1. From the above location, copy the folder 'Course4Exercises' to your O drive.
2. Open the text editor of your choice. I will be using Notepad++.
3. From within the text editor, open the file `ExerciseL1-L3.sh` from the CourseExercises folder that you obtained earlier. If you cannot find it, you can get it again from the following Departmental Share location: `\\childrens\research\ISInformaticsTransfer\CourseDownloads\Course4Exercises`

Getting setup

Computational biology involves a fair amount of juggling of various kinds of input files, output files, and intermediate files. It can be easy to be overrun with a swamp of files unless you can come up with a good way to organize them. It's best if you take time at the very beginning to think through what inputs you will need, the outputs you will generate and where you will put it all.

Tools used

- `bowtie2`

Concepts used

- make directories (`mkdir`)
- list contents (`ls`)
- setting and using environment variables (`export` and `echo`)
- exploring file contents (`less`, `zcat`, `head`)

Exercise L1: First, let's make a space to do our work. The best place to do this is on your share drive. For uniformity's sake and demonstration purposes, we will do this in the next best place, which is `/data`.

1. Let's make a working directory for ourselves. Note how we use the command substitution operator to ensure that everyone creates a valid, but unique directory/

```
## Make a subdirectory in /data using your userid
cd /data
mkdir -p $(whoami)/rnaseq

cd $(whoami)/rnaseq
```

2. To save ourselves some trouble typing, let's create an environment variable

```
export RNASEQ=/data/$(whoami)/rnaseq
echo $RNASEQ
```

```
## /data/staylo/rnaseq
```

3. Now lets make a spot for our raw sequence data and copy it over.

```
cd $RNASEQ
mkdir data
cd data
cp /tools/sampledata/rnaseq/data/* .
```

Let's quickly take a look at our raw data

```
ls -1h
```

Here we have 16 sequencing files. The file extension on these is `.fastq.gz`, meaning they are compressed fastq files. These files are from an experiment with Normal and Tumor samples. There are 2 replicates for each (cDNA-1 and cDNA-2). Also, there were two different library preparation methods (lib1 and and lib2). Paired-end sequencing was performed on each sample, so that also gives us two sequencing reads (10pc_1 and 10pc_2). 2 conditions x 2 replicates x 2 libraries x 2 reads = 16 files. If you want, you can take a peek at the output of one these in order to see how a fastq file is structured. A full discussion of this is beyond the scope for today, but suffice it to say that each read consists of four lines:

- A unique sequence identifier starting with `@`
- The read sequence
- `+`
- The per base quality score (ASCII encoded)

So if we look at the first 8 lines of the fastq file, we will see the records from two sequencing reads. (Note, I truncated the sequence to show only 60 bases for the sake of formatting for the pdf)

```
zcat H_KH-540077-Normal-cDNA-1-lib1_ds_10pc_1.fastq.gz | head -n 8 | cut -c -60
```

```
## @HWI-ST664:122871847:D10AJACXX:5:2308:5958:59022/1
## CCTGAACTATGAGAGTAACTACTGACATGATGCTCCATCATTCCTGAATGTTATGCTGCT
## +
## ==?D;BBDDB8,:+<2A:<CFACDFD<C>EE>@EEEE*?BDDD>9?*:???@D<?*B*?9
## @HWI-ST664:122871847:D10AJACXX:5:2302:16305:79289/1
## CTGTGTTATCTTGAATTTCTTTGAGTTTCCTCAACGCAGCTATTTTGAATTCTCCATGTG
## +
## @@@BDEFFDHFDDHJIBHIJGHJJGACFHGIIGHIGGEEGGHFHHIEIGGICGHIIIJDF
```

4. Now let's make a spot for our reference data set. To speed things up for demonstration purposes, our data has been pre-filtered to show only reads on Chr 22. So we will only align to chr22 rather than the full genome. Remember that reference sequence fasta files for many model organisms have already been downloaded and are available in the `/tools/references` directory. This includes full genome fasta files as well as for individual chromosomes. For reasons that will become apparent later, we still need to make some space in our working directory for the reference. However, we don't really need to copy it over, but instead we will just link to it.

```
cd $RNASEQ
mkdir ref
cd ref
ln -s /tools/references/Homo_sapiens/UCSC/hg19/Sequence/Chromosomes/chr22.fa chr22.fa
```

Notice the notation for links when you run

```
ls -l
```

This indicates that you have created a link to a file that actually lives in another location. However, you can still interact with it as if it were a real copy. For instance, you can try taking a peak inside using

```
head chr22.fa
```

```
## >chr22
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
## NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

5. Before you can do a sequence alignment, you need to 'index' the genome. This process allows the alignment tool to quickly scan through the reference genome to find potential matches. Because different alignment tools use different algorithms they also require different index formats. So a reference must be indexed the first time it is used with a particular alignment tool. This process only has to be done once and then the index files can be reused. *If we were aligning to the whole genome, this process has already been done with a number of alignment tools such as `bowtie`, `bowtie2`, `bwa`, and `blast`.* You could see those if you ran the following code:

```
ls -lh /tools/references/Homo_sapiens/UCSC/hg19/Sequence/
```

Because we are only aligning to chr22, we first need to create an index for it. We will be using the `tophat2` alignment tool, which uses `bowtie2`, so that is what we will use to index

```
bowtie2-build chr22.fa chr22
ls -1h
```

```
## chr22.1.bt2
## chr22.2.bt2
## chr22.3.bt2
## chr22.4.bt2
## chr22.fa
## chr22.rev.1.bt2
## chr22.rev.2.bt2
```

6. You will also need a gene annotation file. A gene annotation file contains all the important information for your gene model, such as gene names, start and stop locations of each feature (gene, exon, CDS), and other important identifying information (Ensembl gene id, transcript id, etc). This information is usually encoded in one of several flexible formats : `.gtf`, `.gff`, or `.gff3`. As with the reference sequences, gene annotation files for many model organisms have been downloaded for you and are available in `/tools/references`. Because we are only aligning to chr22, we will filter the gtf file to only include genes from that chromosome:

```
grep "chr22" /tools/references/Homo_sapiens/UCSC/hg19/Annotation/Genes/genes.gtf >
    genes_chr22.gtf
```

Check your file to make sure it looks the way you expect it to (just showing the first 8 fields)

```
head genes_chr22.gtf | cut -f 1-8
```

```
## chr22    unknown exon    16162066    16162388    .   +   .
## chr22    unknown exon    16162066    16162388    .   +   .
## chr22    unknown exon    16164482    16164569    .   +   .
## chr22    unknown exon    16164482    16164569    .   +   .
## chr22    unknown exon    16171952    16172265    .   +   .
## chr22    unknown exon    16171952    16172265    .   +   .
## chr22    unknown exon    16256332    16256677    .   -   .
## chr22    unknown exon    16258185    16258303    .   -   .
## chr22    unknown stop_codon  16258186    16258188    .   -   .
## chr22    unknown CDS 16258189    16258303    .   -   1
```

Sequence Alignment

Now that we have set up our working environment, we are ready to begin the alignment. We will use Bowtie2/Tophat2 to align all pairs of read files to the genome. The output of this step will be a BAM file for each data set. *Reminder, we will be using some basic defaults. A detailed discussion of the science will be reserved for later. Refer to TopHat manual and tutorial for a more detailed explanation if desired: https://ccb.jhu.edu/software/tophat/manual.shtml*

Tools used

- `tophat2`

Concepts used

- word count (`wc`)

TopHat basic usage `tophat [options] <bowtie_index> <lane1_reads1[,lane2_reads1,...]> <lane1_reads2[,lane2_reads2,...]>`

Extra options specified below:

- `-p 1` tells TopHat to use 1 CPUs for bowtie alignments. On bigger systems you can increase this and get some performance increases.
- `-r 100` tells TopHat the expected inner distance between the reads of a pair. [fragment size - (2*read length)]. 300 - (2*100) = 100
- `-o` tells TopHat to write the output to a particular directory (one per sample). If the directory does not exist, it is created.
- `-G <known transcripts file>` supplies a list of known transcript models. These will be used to help TopHat measure known exon-exon connections (novel connections will still be predicted)
- `--transcriptome-index` TopHat will align to both the transcriptome and genome and figure out the 'best' alignments for you.
  - In order to perform alignments to the transcriptome, an index must be created as we did for the genome.
  - This parameter tells TopHat where to store it and allows it to be reused in multiple TopHat runs.

Exercise L2:

1. Make an alignment directory as well as a sub-directory to hold the transcriptome index

```
cd $RNASEQ
mkdir -p alignments/trans_idx
cd alignments
```

2. Perform the alignment. We will just do alignments for library 1. There are 2 Normal and 2 Tumor samples, so that is 4 alignment calls. Each alignment call gets 2 fastq files (Read 1 and Read 2). In my tests, each sample took ~3 minutes to align

```
tophat2 -p 1 -r 100 -o Normal-cDNA-1-lib1 -G $RNASEQ/ref/genes_chr22.gtf
    --transcriptome-index $RNASEQ/alignments/trans_idx
    $RNASEQ/ref/chr22
    $RNASEQ/data/H_KH-540077-Normal-cDNA-1-lib1_ds_10pc_1.fastq.gz
    $RNASEQ/data/H_KH-540077-Normal-cDNA-1-lib1_ds_10pc_2.fastq.gz

tophat2 -p 1 -r 100 -o Normal-cDNA-2-lib1 -G $RNASEQ/ref/genes_chr22.gtf
    --transcriptome-index $RNASEQ/alignments/trans_idx
    $RNASEQ/ref/chr22
    $RNASEQ/data/H_KH-540077-Normal-cDNA-2-lib1_ds_10pc_1.fastq.gz
    $RNASEQ/data/H_KH-540077-Normal-cDNA-2-lib1_ds_10pc_2.fastq.gz

tophat2 -p 1 -r 100 -o Tumor-cDNA-1-lib1 -G $RNASEQ/ref/genes_chr22.gtf
    --transcriptome-index $RNASEQ/alignments/trans_idx
    $RNASEQ/ref/chr22
    $RNASEQ/data/H_KH-540077-Tumor-cDNA-1-lib1_ds_10pc_1.fastq.gz
    $RNASEQ/data/H_KH-540077-Tumor-cDNA-1-lib1_ds_10pc_2.fastq.gz

tophat2 -p 1 -r 100 -o Tumor-cDNA-2-lib1 -G $RNASEQ/ref/genes_chr22.gtf
    --transcriptome-index $RNASEQ/alignments/trans_idx
    $RNASEQ/ref/chr22
    $RNASEQ/data/H_KH-540077-Tumor-cDNA-2-lib1_ds_10pc_1.fastq.gz
    $RNASEQ/data/H_KH-540077-Tumor-cDNA-2-lib1_ds_10pc_2.fastq.gz
```

Let's count the alignment BAM files to make sure all were created successfully (you should have 4 total)

```
ls -l */accepted_hits.bam | wc -l
ls -l */accepted_hits.bam
```

```
## 4
## Normal-cDNA-1-lib1/accepted_hits.bam
## Normal-cDNA-2-lib1/accepted_hits.bam
## Tumor-cDNA-1-lib1/accepted_hits.bam
## Tumor-cDNA-2-lib1/accepted_hits.bam
```

Perform feature counting

The next step is count how many reads align to each feature in our gene model. There are many methods to do this, but we will pick `htseq-count`. We choose this because it is fairly straightforward and quick to run, which makes it ideal for demonstration purposes. Also it feeds nicely into a powerful package from Bioconductor called `DESeq2` which does differential expression testing.

Tools used

- `samtools`
- `htseq-count`

Concepts used

- piping commands (`|`)
- output redirect (`>`)

samtools basic usage `samtools <command> [options]`

Samtools is actually a collection of different commands, each with their own set of options. We will be using two commands in this exercise

- `sort` is used to sort an alignment file
  - `-n` sorts on the chromosome name
- `view` converts the alignment file from binary (bam) to human readable form (sam)
  - `-h` includes the header information

htseq-count basic usage `htseq-count [options] <sam_file> <gff_file>`

Extra options specified below

- `--mode` determines how to deal with reads that overlap more than one feature. We believe the 'intersection-strict' mode is best.
- `--stranded` specifies whether data is stranded or not.
- `--minaqual` will skip all reads with alignment quality lower than the given minimum value
- `--type` specifies the feature type (3rd column in GFF file) to be used. (default, suitable for RNA-Seq and Ensembl GTF files: exon)
- `--idattr` The feature ID used to identity the counts in the output table. The default, suitable for RNA-Seq and Ensembl GTF files, is gene_id.
- NOTE: Instead of supplying the SAM file to htseq-count, we specify "-" to tell it to accept a stream from stdout

---

Exercise L3:

1. Before we can count the aligned reads, the alignment file must first be sorted by chromosome name.

```
cd $RNASEQ/alignments
samtools sort -n Normal-cDNA-1-lib1/accepted_hits.bam
    Normal-cDNA-1-lib1/accepted_hits_namesorted

samtools sort -n Normal-cDNA-2-lib1/accepted_hits.bam
    Normal-cDNA-2-lib1/accepted_hits_namesorted

samtools sort -n Tumor-cDNA-1-lib1/accepted_hits.bam
    Tumor-cDNA-1-lib1/accepted_hits_namesorted

samtools sort -n Tumor-cDNA-2-lib1/accepted_hits.bam
    Tumor-cDNA-2-lib1/accepted_hits_namesorted
```

2. The counting is done by the `htseq-count` program. This program expects alignments in the human readable sam format. Ours are currently in bam format. So, we will first convert our alignments from bam to sam using the `samtools view` command, and pipe the results into `htseq-count`. Also, `htseq-count` rather unhelpfully returns it's results as standard output (aka dumps it on the screen). Therefore we will also use the redirect operator `>` to capture this output and write it to a tab-separated text file.

```
cd $RNASEQ
mkdir expression
cd expression
mkdir Normal-cDNA-1-lib1  Normal-cDNA-2-lib1  Tumor-cDNA-1-lib1  Tumor-cDNA-2-lib1

samtools view -h $RNASEQ/alignments/Normal-cDNA-1-lib1/accepted_hits_namesorted.bam |
    htseq-count --mode=intersection-strict --stranded=no --minaqual=1 --type=exon
      --idattr=gene_id - $RNASEQ/ref/genes_chr22.gtf >
    Normal-cDNA-1-lib1/gene_read_counts.tsv

samtools view -h $RNASEQ/alignments/Normal-cDNA-2-lib1/accepted_hits_namesorted.bam |
    htseq-count --mode=intersection-strict --stranded=no --minaqual=1 --type=exon
      --idattr=gene_id - $RNASEQ/ref/genes_chr22.gtf >
    Normal-cDNA-2-lib1/gene_read_counts.tsv

samtools view -h $RNASEQ/alignments/Tumor-cDNA-1-lib1/accepted_hits_namesorted.bam |
    htseq-count --mode=intersection-strict --stranded=no --minaqual=1 --type=exon
      --idattr=gene_id - $RNASEQ/ref/genes_chr22.gtf >
    Tumor-cDNA-1-lib1/gene_read_counts.tsv

samtools view -h $RNASEQ/alignments/Tumor-cDNA-2-lib1/accepted_hits_namesorted.bam |
    htseq-count --mode=intersection-strict --stranded=no --minaqual=1 --type=exon
      --idattr=gene_id - $RNASEQ/ref/genes_chr22.gtf >
    Tumor-cDNA-2-lib1/gene_read_counts.tsv
```

3. Take a moment if you wish to explore the contents of this output.

Perform differential expression analysis

There are several packages available for differential expression analysis. We will use the DESeq2 package which is part of the Bioconductor suite of packages for R. We will now leave the command line environment and move into RStudio.

Tools used

- `DESeq2`

Concepts used

- RStudio
- Bioconductor

Exercise L4:

1. Open RStudio by pointing your Firefox browser at https://sideswiper:8787. From the Files pane (usually the lower right pane), browse to `/tools/sampledata/course4Exercises/` and open the file `ExerciseL4-L5.R`)
2. Set up your work space by loading all the appropriate libraries you will need.

```
# Libraries to load
library(DESeq2)
library(plyr)
```

3. The DESeq2 package takes a table of counts. The column names are the individual sample names. The row names are the gene names. So our first task is to load all the individual counts tables, merge them into a single counts table, and then make sure it is formatted properly. **Before you copy this code, make sure you change the path name to your directory!!!**

```
# Create vector of file paths for all gene_read_counts.tsv files
workingpath <- "/data/staylo/rnaseq/expression"
countsfiles <- list.files(path = workingpath, pattern = "gene_read_counts.tsv",
    recursive = TRUE, full.names = TRUE)

# read all tables in and store them in a list. Notice the use of regular
# expressions to automatically capture the sample names.
counts <- lapply(countsfiles, function(x) {
    sample <- gsub(".+expression/(.+)/gene_read_counts.tsv", "\\1", x)
    df <- read.table(x, header = FALSE, sep = "\t", as.is = TRUE)
    colnames(df) <- c("gene_id", sample)
    return(df)
})

# Take a peak at what you just loaded in
lapply(counts, head)
```

```
## [[1]]
##       gene_id Normal-cDNA-1-lib1
## 1      A4GALT                  1
## 2        ACO2                 59
## 3         ACR                  2
## 4        ADM2                  6
## 5     ADORA2A                  0
## 6 ADORA2A-AS1                  1
##
## [[2]]
##       gene_id Normal-cDNA-2-lib1
## 1      A4GALT                  6
## 2        ACO2                159
## 3         ACR                  0
## 4        ADM2                 24
## 5     ADORA2A                  0
## 6 ADORA2A-AS1                  0
##
## [[3]]
##       gene_id Tumor-cDNA-1-lib1
## 1      A4GALT                 2
## 2        ACO2                62
## 3         ACR                 0
## 4        ADM2                 9
## 5     ADORA2A                 0
## 6 ADORA2A-AS1                 0
##
## [[4]]
##       gene_id Tumor-cDNA-2-lib1
## 1      A4GALT                 0
## 2        ACO2               101
## 3         ACR                 1
## 4        ADM2                13
## 5     ADORA2A                 0
## 6 ADORA2A-AS1                 0
```

```r
# Merge the data frames into a single dataframe
counts <- Reduce(function(...) merge(..., all = TRUE), counts)
head(counts)
```

```
##                  gene_id Normal-cDNA-1-lib1 Normal-cDNA-2-lib1
## 1 __alignment_not_unique               9825              13720
## 2           __ambiguous                420                633
## 3         __no_feature              49529              67641
## 4        __not_aligned                  0                  0
## 5       __too_low_aQual                 0                  0
## 6                A4GALT                 1                  6
##   Tumor-cDNA-1-lib1 Tumor-cDNA-2-lib1
## 1             11162              8462
## 2               532               439
## 3             65273             70180
## 4                 0                 0
## 5                 0                 0
## 6                 2                 0
```

```
# Notice that in addition to gene counts, the first 5 rows detail various
# categories of reads that did not map to any gene. We will want to remove
# these from our table. Also note that the first column of the data frame
# contains the gene id. We want to turn this column into the row names. This
# sounds like a lot, but is actually acheived quite simply in two lines of
# code:
row.names(counts) <- counts$gene_id
counts <- counts[-c(1:5), -1]
head(counts)
```

```
##             Normal-cDNA-1-lib1 Normal-cDNA-2-lib1 Tumor-cDNA-1-lib1
## A4GALT                       1                  6                 2
## ACO2                        59                159                62
## ACR                          2                  0                 0
## ADM2                         6                 24                 9
## ADORA2A                      0                  0                 0
## ADORA2A-AS1                  1                  0                 0
##             Tumor-cDNA-2-lib1
## A4GALT                      0
## ACO2                      101
## ACR                         1
## ADM2                       13
## ADORA2A                     0
## ADORA2A-AS1                 0
```

4. The next input for DESeq2 is the experimental design. This is just a simple table that describes the different variables that will be compared. In this case, we have just one variable that describes the type of sample as being either 'normal' or 'tumor'. The row names in this table should correspond to the columns in the counts table.

```
# Use regular expressions to extract the sample type from the sample names
type<-gsub('(.+)-cDNA.+', '\\1', colnames(counts))

# create the data frame and format the row names correctly
design<-data.frame(type=type)
row.names(design)<- colnames(counts)
design
```

```
##                       type
## Normal-cDNA-1-lib1 Normal
## Normal-cDNA-2-lib1 Normal
## Tumor-cDNA-1-lib1   Tumor
## Tumor-cDNA-2-lib1   Tumor
```

5. Now we take the counts table and the design table and enter them into a data set object. The `DESeq` command then performs the statistical analysis on the data set in one easy step. For details on the science and methods behind the analysis, consult the DESeq2 vignette.

```
# create the data set and make sure the comparison is by type, with tumor vs
# normal.
dataset <- DESeqDataSetFromMatrix(countData = counts, colData = design, design = ~type)
dataset$type <- relevel(dataset$type, "Normal")
```

```r
# run the analysis and extract the results
deseq <- DESeq(dataset)
results <- results(deseq)
results
```

```
## log2 fold change (MAP): type Tumor vs Normal
## Wald test p-value: type Tumor vs Normal
## DataFrame with 579 rows and 6 columns
##             baseMean log2FoldChange     lfcSE       stat    pvalue
##            <numeric>      <numeric> <numeric>  <numeric> <numeric>
## A4GALT     2.0193535     -0.2052552 0.3129377 -0.6558979 0.5118898
## ACO2      89.4591828     -0.3525501 0.3725655 -0.9462768 0.3440074
## ACR        0.8895901     -0.0800388 0.2182333 -0.3667579 0.7137996
## ADM2      11.9432894     -0.1961521 0.4640342 -0.4227105 0.6725065
## ADORA2A    0.0000000             NA        NA         NA        NA
## ...              ...            ...       ...        ...       ...
## ZNF280B   28.646553     -0.1644359 0.4446553 -0.3698053 0.7115276
## ZNF70     28.688723     -0.1960302 0.4390021 -0.4465358 0.6552103
## ZNF74      7.969444      0.2181280 0.4485320  0.4863153 0.6267437
## ZNRF3     39.728702      0.1797905 0.4159538  0.4322366 0.6655694
## ZNRF3-AS1  7.129656      0.3861048 0.4417910  0.8739535 0.3821436
##                padj
##           <numeric>
## A4GALT    0.9954314
## ACO2      0.9954314
## ACR       0.9954314
## ADM2      0.9954314
## ADORA2A          NA
## ...             ...
## ZNF280B   0.9954314
## ZNF70     0.9954314
## ZNF74     0.9954314
## ZNRF3     0.9954314
## ZNRF3-AS1 0.9954314
```

```r
# The results may be more helpful if you sort them by significance. Note
# that we are using the adjusted p-value to account for multiple hypothesis
# testing. That is bigger topic for another time, so for now just trust us.
results[order(results$padj), ]
```

```
## log2 fold change (MAP): type Tumor vs Normal
## Wald test p-value: type Tumor vs Normal
## DataFrame with 579 rows and 6 columns
##          baseMean log2FoldChange     lfcSE      stat       pvalue
##         <numeric>      <numeric> <numeric> <numeric>    <numeric>
## PRODH    27.71121       2.682553 0.4591161  5.842865 5.131066e-09
## CELSR1   47.52712       2.253011 0.4211494  5.349673 8.811350e-08
## ISX      91.70663      -1.794859 0.3574552 -5.021213 5.134615e-07
## SLC5A1  152.17983      -1.469027 0.3281146 -4.477177 7.563637e-06
## TST     166.54438      -1.447417 0.3980606 -3.636173 2.767182e-04
## ...           ...            ...       ...       ...          ...
## TSSK2           0             NA        NA        NA           NA
```

```
## UPK3A           0             NA        NA        NA           NA
## VPREB1          0             NA        NA        NA           NA
## XKR3            0             NA        NA        NA           NA
## ZNF280A         0             NA        NA        NA           NA
##                padj
##           <numeric>
## PRODH   2.252538e-06
## CELSR1  1.934091e-05
## ISX     7.513654e-05
## SLC5A1  8.301092e-04
## TST     2.429586e-02
## ...              ...
## TSSK2             NA
## UPK3A             NA
## VPREB1            NA
## XKR3              NA
## ZNF280A           NA
```

6. The results table can be output as a csv if desired:

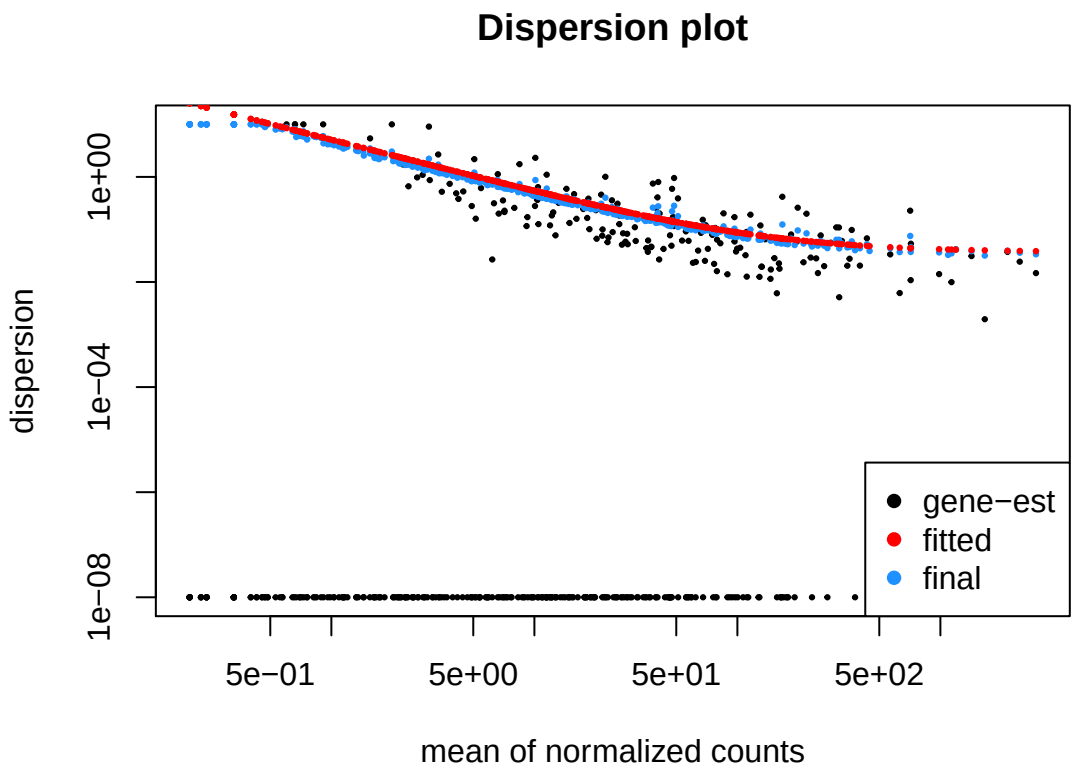
```
write.csv(results[order(results$padj), ], file = paste0(workingpath, "diffex_results.csv",
    quote = FALSE))
```

Visualize the results

In addition to being a power statistical machine, R also has an impressive array of plotting functions that are useful for visualizing the data. We show these here just as a demonstration of what R is capable of doing. A thorough explanation of the plotting functions are beyond the scope of this course. For now, we leave it to you to explore some of these functions on your own. *A note on the science: This RNA-Seq data set was used because it was small and useful to demonstrate how to run the workflow. The results visualized here do not necessarily represent what a "good" experiment should look like.*

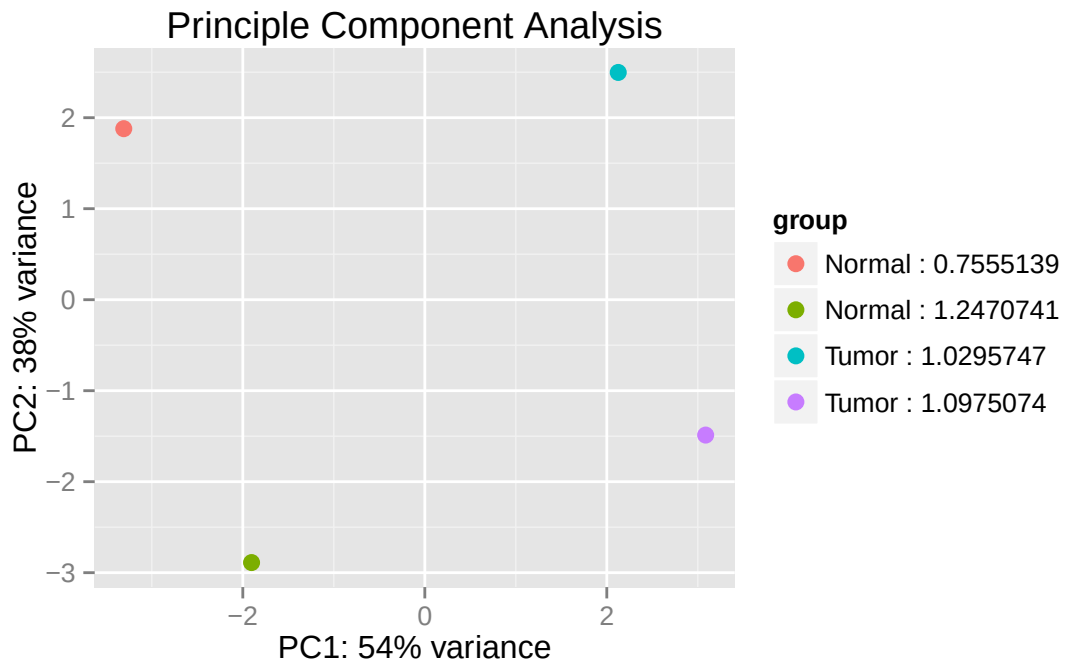
```
require(RColorBrewer)
require(gplots)
require(ggplot2)
```

```
# Plot 1: Dispersion This plot models the relationship between the number of
# counts for a gene and the observed variance

plotDispEsts(deseq, main = "Dispersion plot")
```

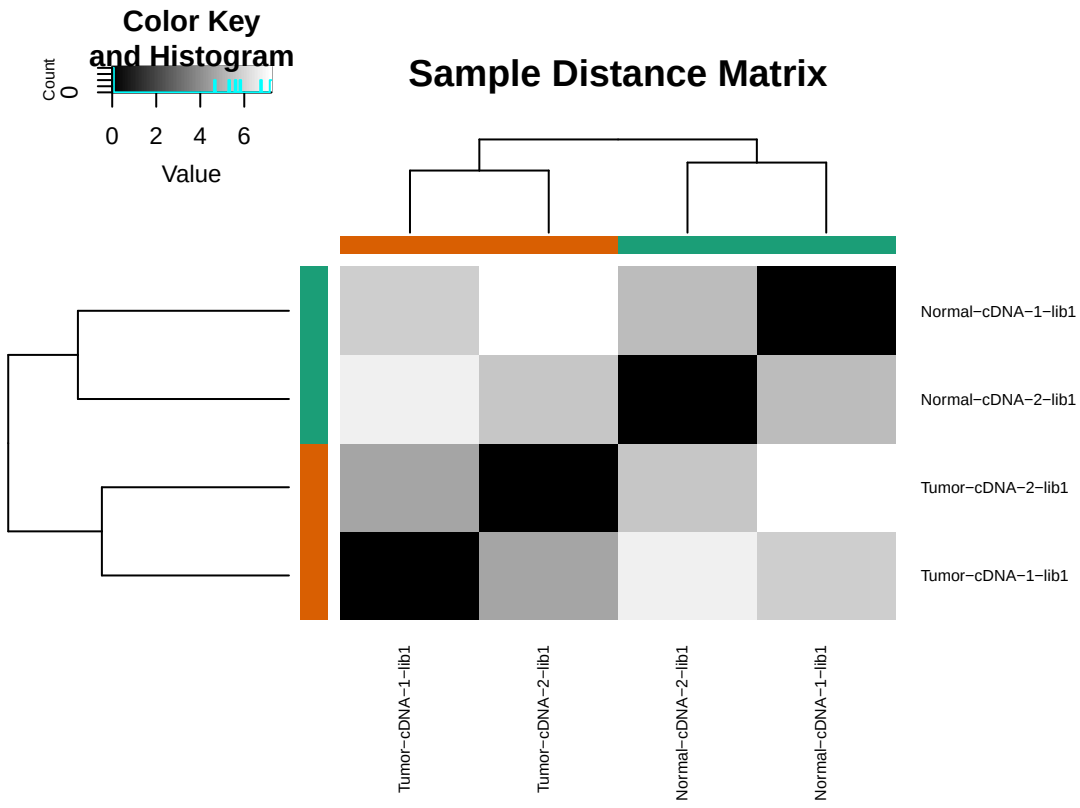


Exercise L5:

```
# Plot 2: Principal components analysis First perform a regularized log
# transformation, then use the plotPCA command. Regularized log transforms
# are an important topic that we will discuss another time. For now, just
# trust us.

rld <- rlogTransformation(deseq, blind = TRUE)
plotPCA(rld, intgroup = colnames(colData(deseq))) + ggtitle("Principle Component Analysis")
```

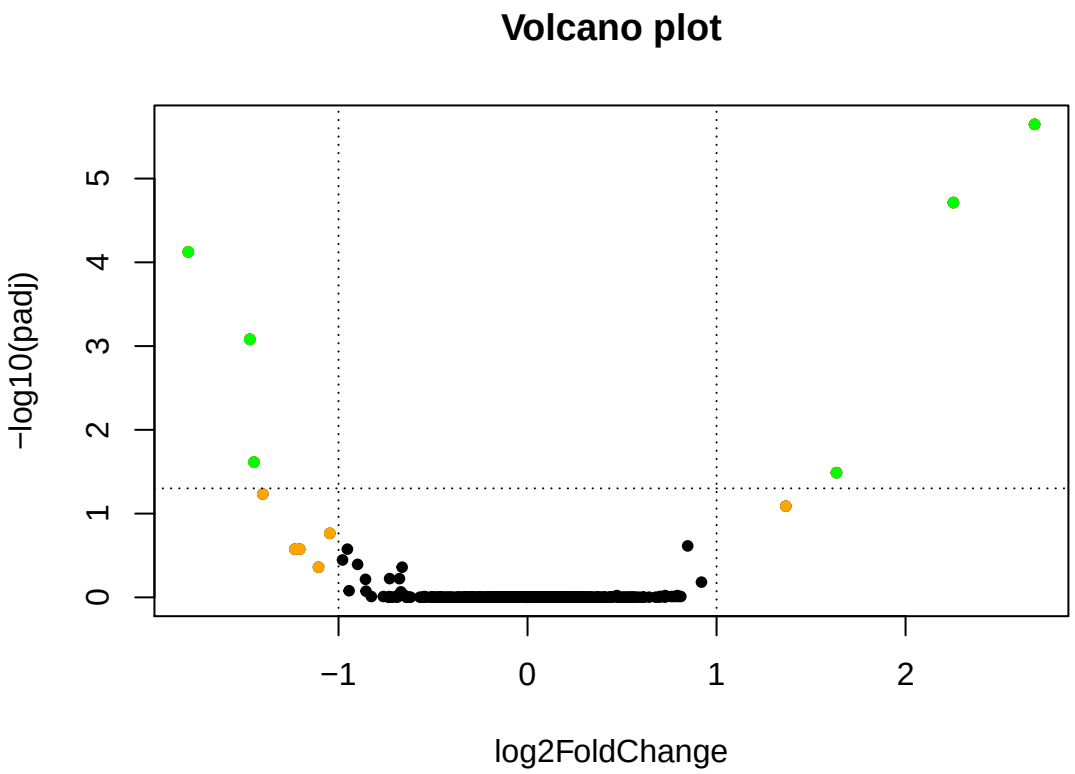



```
# Plot 3: Sample distance heatmap Also uses regularized log transformation
# and takes advantage of nice color pallette from RColorBrewer

mycols <- brewer.pal(8, "Dark2")[1:length(unique(deseq$type))]
sampleDists <- as.matrix(dist(t(assay(rld))))
heatmap.2(as.matrix(sampleDists), key = TRUE, trace = "none", cexRow = 0.8,
    cexCol = 0.8, col = colorpanel(100, "black", "white"), ColSideColors = mycols[deseq$type],
    RowSideColors = mycols[deseq$type], margin = c(10, 10), main = "Sample Distance Matrix")
```




```
# Plot4: MA plot Shows the fold change in expression (y axis) vs mean counts
# (x axis). Red points show genes with adjusted p value less than 0.1. These
# plots illustrate the fact that when the counts are low, a greater fold
# change is required to demonstrate significance.

plotMA(results, main = "MA plot")
```


```
# Plot 5: Volcano plot Graphical way to show genes with differential
# expression that is both significant (y-axis) and large (x-axis)

with(results, plot(log2FoldChange, -log10(padj), pch = 20, main = "Volcano plot"))

# Add colored points: red if padj<0.05, orange of log2FC>1, green if both)

with(subset(results, padj < 0.05), points(log2FoldChange, -log10(padj), pch = 20,
    col = "red"))
with(subset(results, abs(log2FoldChange) > 1), points(log2FoldChange, -log10(padj),
    pch = 20, col = "orange"))
with(subset(results, padj < 0.05 & abs(log2FoldChange) > 1), points(log2FoldChange,
    -log10(padj), pch = 20, col = "green"))

# Add reference lines

abline(v = c(-1, 1), h = -log10(0.05), lty = 3)
```



Cleaning Up

Analyses like these tend to generate a good number of intermediate files. When you are done with your analysis it is important that you clean up after yourself. Otherwise, you will find that you have accumulated a significant number of files that you don't know what to do with and you will soon run out of space on your disk. This is particularly true if you are using a shared disk, such as `/data`. However, "Cleaning up after yourself" can actually present a surprisingly difficult challenge. What do you keep and what do you get rid of? When do you clean up? There are no hard and fast rules per se, but we will attempt to outline some good principles to help guide you.

- **Clean up when you are done**. Often researchers will want to play around with different parameter sets until they find the parameter values that are most appropriate. This involves a lot of iterative reanalyzing of the data. It is not necessarily practical or desirable to throw away everything at the end of each iteration. It is good practice to put final and intermediate outputs of each iteration in different places to keep them separate. When you are done with your iterations and you think you have the best output, don't forget to go back and remove all the other iterations. If you wait too long, you will have a hard time remembering which iteration gave you the best result.
- **Don't leave things in a shared space**. Remember, `/data` is for scratch space only. Do not let your data spend more time here than it needs to. Not only is it wasteful of a shared resource, but it isn't safe. Nothing stored on `/data` or anywhere else on the autobots is backed up. The only safe place to put your data is back on your departmental share drive.

- **Only keep the raw data and the final data**. Sounds simple right? This turns out to be a surprisingly contentious issue, particularly for sequence data. What is the right form of the 'raw' data for your sequence data? Is it the fastq or bam?

  - **Fastq files** are technically the most raw form of the data that is practical to keep. However, the alignment is often the most painstaking, time consuming step in a workflow. So it's nice if you don't have to do that again.
  - **BAM files** are a superset of the fastq file, meaning that everything that is in a fastq file is also in the BAM file. You can always regenerate a fastq file from your BAM. Once you get a good alignment, you may never need to go back and realign unless you want to align to a different or newer reference. The catch is that the main BAM file only keeps the sequences that aligned. All of the sequences that didn't align are stored in a separate BAM file (often with a label like 'unmapped.bam'). So if you choose to keep just the BAM, you will want to think about if you want to keep the unmapped sequences as well.
  - **Intermediate BAMs** like the 'namesorted' BAM that we generated can easily be regenerated provided you kept a good record of the steps you used to make it the first time.
  - **Files with the `.bed` extension** are text files that define the locations of various features. They are useful for visualization programs such as IGV and other downstream analyses. Depending on your workflow these will be more or less useful. The same logic will apply to all intermediate outputs in any workflow.
  - **Log files** are often useful for troubleshooting, so they are handy to have while you are working out the bugs. Once your analysis has completed successfully you may not need these anymore. Sometimes they log the commands and parameters that you used. But if you take care to document this separately elsewhere (see below) that may not be necessary.

- **Keep a record of what you did**. This is one of the most critical things to keep. As long as you have your raw data and a well documented record of what you did, you can always regenerate your analysis. Below we list some minimal things to include in that record. You may discover other pieces of information that are also useful to you.

  - The date the analysis was done and who did it
  - The exact commands and parameters used
  - The version numbers of any software, programs, or packages that you used

Using shell scripts

One the most powerful aspects of working in the command line is the ability to write your own mini pieces of software, called *scripts*. At first, you may think that this is well beyond your ability. However, it is actually very easy. If you have taken the trouble to keep track of all the commands that you have executed, you are already 90% there! In fact, the text document that you have been using to copy and paste from up to this point is already a fully executable piece of custom software.

Making your text file executable The secret is the little line of code at the top of the text file:

```
#! /usr/bin/bash
```

- The first bit `#!` is called the **shebang** (ha **sh** + **bang**, which is what computer nerds call the exclamation mark). When Linux sees this at the beginning of a script, it tells the system that this is a program

- The next bit `/usr/bin/bash` is called the **interpreter directive**, which basically tells the system what language environment to run the script in and where it is located. It turns out that all along, when you see the command prompt and entering commands, you have been using the ___b___orn ___a___gain **sh** ell scripting language, or **bash**. That is why the command prompt is also sometimes called the bash prompt. That is also why scripting in the command line is referred to as shell scripting, as opposed to scripting in another language such as python, perl, java, or R.
- With the exception of the shebang, any line starting with `#` is ignored. These lines are comments, and are used to help document or explain what the code is supposed to do. Good code is well-commented code.
- By convention, shell scripts receive a `.sh` extension although this isn't strictly necessary.
- You will have to change the mode of the script to be executable. This is done with the `chmod` command whose argument should be understood to mean: (ch)ange (mod)e for (u)ser to add(+) e(x)ecutable permissions to ExamplesL1-L3.sh:

```
chmod u+x ExamplesL1-L3.sh
```

- Rather than copying and pasting each line, you could have executed the entire document simply by calling the file with it's full path as if it were a command.

```
./ExamplesL1-L3.sh
```

- Note that this code also contains many exploratory calls (e.g. calls to `less` or `head` or `ls`) where we were just looking around at the environment and the files. You will see the output of all of these scroll quickly across your screen. In real documents, you would probably not have many of these. Take note however, that these do provide a sort of 'progress' bar to see how much of your code you have executed. Thus judicious use of these functions can be quite handy for debugging your code.

Homework L6:

1. Try executing this script on your own. Before you do that, you will want to clear out your working directory so the script has a fresh start. **CAREFUL! Make sure you are in the correct directory!!! Linux will not ask you if you are sure you want to do this!**

```
cd $RNASEQ
rm -r alignments
rm -r data
rm -r expression
rm -r refs
```

2. Copy the script into your working directory. Note that it shouldn't matter what my current working directory is because I have used absolute path names throughout the script. But for uniformity sake, I am copying it into our $RNASEQ directory

```
cd $RNASEQ
cp /tools/sampledata/course4Exercises/ExerciseL1-L3.sh .
```

3. Execute the script by calling it with it's full path

```
$RNASEQ/ExerciseL1-L3.sh

## or alternatively if you are currently in the same directory as the script

./ExerceiseL1-L3.sh
```

Optional Homework L7: Creating the above script can be tedious and/or challenging when you are trying to repeat the same command on different files. This is where a good text editor and the judicious use of regular expressions can save you a lot of work. Learning regular expressions is beyond the scope of this course. However, this bonus exercise should be sufficient to give you a taste of why they are so useful and perhaps motivate you to learn more about them outside of this class.

1. Open a good text editor.
2. Open the file `HomeworkL7.sh` from the Course4Exercises folder that you obtained earlier. If you cannot find it, you can get it again from the following Departmental Share location: `\\childrens\ISInformaticsTransfer\CourseDownloads\Course4Exercises`
3. The document consists of three examples of how you would use a text editor to build a command string from a list of input files.
4. The first section builds the tophat2 alignment commands. Navigate to the indicated directory and paste the specified command into your terminal to generate a list of input files. Copy the output of that command back into the text document.
5. Open the search/replace tool in your text editor. Make sure that the 'Regular expression' option is selected. Copy and paste the indicated search and replace terms from the text document into the appropriate fields.
6. Use the 'Find Next' and 'Replace' buttons to convert your list of input files to complete commands.
7. Repeat for the other two sections.

Using R scripts

It is fairly common to have a workflow where you spend some time at the command line doing some preliminary data wrangling, and then move into R to perform the final statistical analysis. The only potential problem with this is that you end up with your workflow documentation spread out over multiple shell and R scripts. Unless you keep close track of these, it can be confusing to keep track of what you have done.

Wouldn't it be nice if you could just do all of your work from within RStudio? Turns out you can. There is function in R called `system2` which allows you to send system calls from within R back down to the command line to be executed. As with anything there are advantages and disadvantages to this approach. Here is a brief list of a few of each:

Advantages

- Your entire workflow can be contained within a single script
- You can take advantage of useful R features such as `lapply` to run the same bit of code against several different input files.

- You can take advantage of powerful regular expression commands to easily capture sample names and format input and output file names. This is huge as it significantly reduces typo errors.
- You can often use built-in R functions or packages that will do the same job as your command-line script, only easier, faster, or more intuitively.
- You can easily parallelize operations in R, significantly increasing the performance of these machines.
- You can use RStudio as your text editor, which is actually a really nice environment to program in.

Disadvantages

- Once you start making system calls, you are locked into a particular system configuration and your code is no longer portable. It more or less has to be run on the same machine or cluster unless you go back in and edit it.
- Some system calls require slightly different syntax when called from R.
- Complex operations involving piping and redirect can be especially difficult or may need to be redesigned from the way you would normally accomplish the task in bash.
- It is less easy to access the `man` pages for your system commands when you are in R.
- It can sometimes be a little more challenging to troubleshoot your system calls because of the added layer of the R environment.

Homework L8:

1. Within RStudio, use the **Files** pane (usually in the lower right corner) to find and open the following file: `/tools/sampledata/course4Exercises/HomeworkL8.R`
2. Go through the script and execute it line by line. *Hint: use `Ctrl-ENTER` to send a single line of code (or a selected code chunk) to the R console.* As with Homework L6, you will want to clear out your working directory so the script has a fresh start.
3. Take note where the script seems to differ from the syntax used in the bash shell. If you come across a command that you are not familiar with, read more about it from it's help page. *Hint: use `help()` or `?` to access the man page.

Homework L9:

1. All of the above exercises were completed with just half of the available data set, i.e. library 1. Try to repeat the workflow, but this time use library 2. You can execute interactively (by composing each), from a shell script, or from an R script. Use the .sh and .R scripts provided as templates.