Genome Analysis Workshop MOLB 7621

version 0.2

Jay Hesselberth

February 08, 2015

Contents

Genome Analysis Workshop

Instructor: Jay Hesselberth <jay.hesselberth@gmail.com>

Website: http://hesselberthlab.github.io/workshop

Next offered: Spring 2015

Course Number: MOLB 7621 (cross-listed with STBB 7621)

Last updated: January 23, 2015

Announcements

- **Class size is limited to 20. We anticipate being full, so register early.**
- Tuition waivers are available for Postdocs and PRAs. You also need to fill out a non-degree application.

Course Description

The Genome Analysis Workshop is a hands-on tutorial of skills needed to process large genomics data sets and visualize their results. The class is taught from the standpoint of a biologist with practical goals (e.g. to interpret the results of a sequencing-based experiment and gain biologically meaningful insight).

We focus on working in the Linux environment, with emphasis on command-line tools, Python programming and the R statistical computing environment. We use publicly available next-generation DNA sequencing data from the ENCODE project to illustrate standard approaches for manipulating sequencing data, aligning sequences to a reference genome, generating coverage plots and displaying them in the UCSC Genome Browser. We will cover specific analyses used in ENCODE project including ChIP-seq, DNase I footprinting, mRNA-seq and genome sequencing to identify single nucleotide and structural variants.

Contents

Syllabus

General Information

Title: Genome Analysis Workshop

Course Number: MOLB 7621

Semester: Spring 2015

Homepage: http://hesselberthlab.github.io/workshop

Instructor: Jay Hesselberth

Organization: University of Colorado School of Medicine

Address: Department of Biochemistry and Molecular Genetics

Last updated: February 08, 2015

PDF Content

The course content is available as a combined PDF: **PDF Download**

Instructor Information

Instructor	Email
Jay Hesselberth	jay.hesselberth@gmail.com
Sally Peach	sally.peach@gmail.com
Charlotte Siska	charlotte.siska@ucdenver.edu
Kyle Smith	kyle.s.smith@ucdenver.edu

Schedule

Tue & Thurs, 1:00 - 3:00 PM. TA office hours from 3:30 PM *See specific dates.*

Location

Health Sciences Library, Computer Teaching Lab 2

Course Description

The Genome Analysis Workshop is a hands-on tutorial of skills needed to process large genomics data sets and visualize their results. The class is taught from the standpoint of biologist with practical goals (e.g. to interpret the results of a sequencing-based experiment and gain biologically meaningful insight).

We focus on working in the Linux environment, with emphasis on Linux command-line tools, Python programming and the R statistical computing environment. We use publicly available next-generation DNA sequencing data from the ENCODE project to illustrate standard approaches for manipulating sequencing data, aligning sequences to a reference genome, generating coverage plots and displaying them in the UCSC Genome Browser. We will cover specific analyses used in ENCODE project including ChIP-seq, DNase I footprinting, mRNA-seq and genome sequencing to identify single nucleotide and structural variants.

Course Credits

This is a 3 credit course.

Texts and Reading Materials

1. **Required**: A Quick Guide to Organizing Computational Biology Projects http://www.ploscompbiol.org/article/metrics/info%3Adoi%2F10.1371%2Fjournal.pcbi.1000424
2. **Required**: Command Line Crash Course http://cli.learncodethehardway.org/book/
3. **Required**: ggplot2: Elegant Graphics for Data Analysis http://ggplot2.org/book/
4. **Required**: Learn Python the Hard Way, http://learnpythonthehardway.org/book/

Course Objectives

- Learn to manipulate large sequencing data sets with Linux command line tools and Python programming.
- Learn to manipulate and visualize data with the R statistical computing environment.
- Learn workflows for ENCODE experiments including ChIP-seq, DNaseI footprinting, mRNA-seq and variant detection.
- Learn to visualize data in the UCSC Genome Browser

Canvas

The course has a Canvas page [1] where announcements are made and problem sets are uploaded. You need to login to see Announcements and Problem Sets.

Assessment

Progress of individual students will be assessed during the daily exercise session, weekly problem sets, as well as a final project.

Grading Criteria

- 50% participation
- 40% problem sets (10 sets, 4% each)
- 10% final project

Specific Dates / Material to be Covered

Class number	Date	Topic	Problem Set
Class 1	T Jan 27	Introduction to VM, Linux and the shell	
Class 2	Th Jan 29	Linux / Utilities	**PS1 due** (Mon Feb 2 5:00 PM)
Class 3	T Feb 3	Linux / Utilities	
Class 4	Th Feb 5	Linux / Utilities	**PS2 due** (Mon Feb 9 5:00 PM)
Class 5	T Feb 10	Cluster Usage / Review	
Class 6	Th Feb 12	Cluster Usage / Review	**PS3 due** (Mon Feb 16 5:00 PM)
Class 7	T Feb 17	Python	
Class 8	Th Feb 19	Python	**PS4 due** (Mon Feb 23 5:00 PM)
Class 9	T Feb 24	Python	
Class 10	Th Feb 26	Python	**PS5 due** (Mon Mar 2 5:00 PM)
Class 11	T Mar 3	Python	
Class 12	Th Mar 5	ENCODE Overview	**PS6 due** (Mon Mar 9 5:00 PM)
Class 13	T Mar 10	BEDtools	

Class 14	Th Mar 12	ChIP-seq (coverage / peaks / motifs)	**PS7 due** (Mon Mar 16 5:00 PM)
	** No Class Mar 16-20 (Campus Spring Break) **		
Class 15	T Mar 24	ChIP-seq / DNaseI-seq (UCSC)	
Class 16	Th Mar 26	Genomic analysis vignettes	
Class 17	T Mar 31	Genomic analysis vignettes	
Class 18	Th Apr 2	R data & plotting	**PS8 due** (Mon Apr 6 5:00 PM)
Class 19	T Apr 7	R data & plotting	
Class 20	Th Apr 9	R data & plotting	**PS9 due** (Mon Apr 13 5:00 PM)
Class 21	T Apr 14	R data & plotting	
Class 22	Th Apr 16	R data & plotting	**PS10 due** (Mon Apr 20 5:00 PM)
Class 23	T Apr 21	mRNA-seq (FPKM / diff exp)	
Class 24	Th Apr 23	mRNA-seq (FPKM / diff exp)	**PS11 due** (Mon Apr 27 5:00 PM)
Class 25	T Apr 28	Exome Alignment	
Class 26	Th Apr 30	Exome Variant Calling	

1 https://ucdenver.instructure.com/courses/325063

Class list

Class 1 : Class Introduction

Class date: Tues 2015 Jan 27

Goals

1. Class overview
2. Get the VM running
3. Overall goals for the Classes

Class Overview

Each class is 2 hours. We intend to spend the first 60 min going through exercises that demonstrate how specific tools are useful in bioinformatics. During the remaining hour, we expect you to work through exercises, asking for help when you get stuck.

We will record the first 60 minutes using Panopto Screen Capture, and these recordings will be available in Canvas. We have found that simply watching someone work in a terminal (move around, open up text editors, write and execute simple programs) can be a very effective way to get started with programming. Hopefully these movies will serve the same purpose.

Each week, we will have 1 take home quiz, due the following Tuesday at 5 PM.

Linux installations

The PCs in the library have Virtual Box installed with a minimal Linux installation. If you have your own PC laptop, you can install Virtual Box and any standard Linux distribution (Ubuntu or Mint). If you have a Mac laptop, you can do the same, or just use the native terminal.

In any case, we will create logins on our compute cluster (amc-tesla) and all your work will be done through that.

Terminal and text editors

When you open a Terminal, you also launch a shell process, typically a bash process. At the prompt, you can type things that bash understands, and it will do them. The shell has its own language, which you will learn over the course of the class. It also runs executable files that it can find on the PATH (i.e. the set of directories that contain exectuables).

You can find what is on your PATH by typing:

```
$ echo $PATH
```

The PATH is one of several environment variables that are created when you login. You can see all of these with:

```
$ env
```

One important program in the PATH is *vim*. You will use this program to keep notes and write small programs.

You can run `vim` from the terminal prompt:

```
$ vim filename.txt
```

You should notice that the prompt will disappear and you will be in a *vim* session.

Now press the *i* key to enter *insert* mode, and start typing. Press *ESC* to exit *insert* mode.

To quit a vim session, you need to:

1. enter *command mode* with the colon key

2. write the file
3. quit the program

This can be accomplished by typing:

```
:wq <enter>
```

Practice using *vim* with this tutorial [2].

Shell and Python Programming

It is important that you learn a few new computer languages. Others have developed very good guides to teach you these languages, and we are going to use those in the class. We expect you to begin taking these classes immediately.

You will spend a lot of time going through these online classes, both in scheduled class time, and outside of class time. Instead of focusing on teaching you these languages, we will focus on helping you get through all of the frustating problems that come up when you're learning the languages.

We will spend the first ~2 weeks learning shell [3] and all the things you have access to within the shell.

After learning the shell, we will begin learning R and several packages within R.

Finally, we will begin learning Python [4]. The Python language allows you to do more sophisticated things that would be possible in shell or R, but would be considerably more clunky.

First Quiz : Reading

Computational biology projects inevitably accrue a lot of files. For the first quiz, you'll need to read a paper [5] and be able to put a set of files in the correct places. We highly recommend adopting this scheme for all of your projects in and out of the class.

---

2 OpenVim http://www.openvim.com/
3 The Command Line Crash Course http://cli.learncodethehardway.org/book/
4 Learn Python the Hard Way http://learnpythonthehardway.org/book/
5 A Quick Guide to Organizing Computational Biology Projects (2009) PLoS Comput. Biol. William S. Noble http://dx.plos.org/10.1371/journal.pcbi.1000424

Class 2 : The command-line

Class date: Thurs 29 January 2015

Things to fix from last class

Do this on tesla:

```
$ cp ~jhessel/.vimrc ~
```

This will give line numbers (and a bunch of other stuff) when you run `vim`.

Goals

1. The `bash` shell
2. continue learning to navigate within the terminal
3. understand the linux philosophy (small tools that do one thing well)
4. understand how to apply some common linux utilities to files
5. `vim` to edit files

Unix Philosophy

The Unix philosophy [6] Works well for bioinformatics:

- Make each program do one thing well.
- Make every program a filter.
- Choose portability over efficiency.
- Store data in flat text files.
- Small is beautiful.
- Use software leverage to your advantage.
- Use shell scripts to increase leverage and portability.

Navigating In the Terminal

When you start the terminal, you will be in your home directory.

In Linux home is represented as `~` and also as `$HOME`.

We will often show commands preceded with a '`$`' as you see in your terminal.

Try this in the terminal:

```
$ pwd
```

pwd stands for "print working directory"

Change to another directory:

```
$ cd /tmp/
```

See what's in that directory:

```
$ ls
```

Show more information:

```
$ ls -lh
```

The `-lh` part are flags for the `ls` command.

---

6 http://en.wikipedia.org/wiki/Unix_philosophy

These can also be separated like:

```
$ ls -l -h
```

Getting Help In The Terminal

How can you find out the arguments that `ls` accepts (or expects):

```
$ man ls
```

and use spacebar to go through the pages. `man` is short for "manual" and can be used on most of the commands that we will learn.

In other linux software, it is common to get help by using:

```
$ <program> -h
```

or:

```
$ <program> --help
```

Which of these works for `ls`?

Note

If you see an error message, read it carefully. It may seem cryptic, but it is designed to inform you what went wrong.

Getting Help : Exercises

- use `man` to find out how to list files so that the most recently modified files are listed last.

(This is common when you're working on something and only care about the most recently modified files)

- use google to find the same thing. how else can you sort the output of `ls`?

Getting Help Outside The Terminal

Use google. Useful sites include:

- stackexchange.com
- biostars.org
- seqanswers.com

In many cases, if you receive and error, you can copy-paste it into google and find some info.

Other Commands In The Terminal

Use the `man` command to determine what `head` does.

Use `head` on the file /vol1/opt/data/lamina.bed

Use `tail` to see the end of the file.

Questions

- By default, head and tail show 10 lines. How can you see 13 lines?

Other Commands In The Terminal (Answers)

```
$ man head
$ head /vol1/opt/data/lamina.bed
$ tail /vol1/opt/data/lamina.bed
$ head -n 13 /vol1/opt/data/lamina.bed
```

Word Counts (wc)

Exercise:

- use **wc** to determine how many **lines** are in /vol1/opt/data/lamina.bed
- use **wc** to determine how many **words** are in /vol1/opt/data/lamina.bed

Less (is More)

To view a large file, use less:

```
less /vol1/opt/data/lamina.bed
```

You can forward-search in the file using "/"

You can backward-search in the file using "?"

You can see info about the file (including number of lines) using "ctrl+g"

You can exit **less** using "q"

Terminal History

Press the up arrow in the terminal.

Up and down arrows will allow you to scroll through your previous commands.

This is useful when running similar commands or when remembering what you have done previously.

You can type the start of a command and then up-arrow and it will cycle through commands that start with that prefix.

Tab-Completion

The bash shell has several built-in utilities for expediting typing.

Type the following where [TAB] means the Tab key on the keyboard:

```
$ cd /opt/bio-w[TAB]
```

Then hit tab. And:

```
$ ls /opt/bio-w[TAB]
```

This will work for any file path and for any programs:

```
$ hea[TAB]
```

What happens if you do:

```
$ he[TAB][TAB]
```

or:

```
$ heaaa[TAB][TAB]
```

Directory Shortcuts

We have already used the cd command to change directories. And we have used the ~ shortcut for home.

```
$ cd ~
$ ls ~
```

We can also move to or see what's in the parent directory with:

```
$ ls ..
$ cd ..
```

Or 3 directories up with:

```
$ ls ../../..
$ cd ../../..
```

To explicitly see the current directory:

```
$ ls ./
```

We can go 2 directories up with:

```
$ cd ../../
```

Here, we can remember that "." is the current directory and .. is one directory up. What does this do:

```
$ ls ./*
```

you can go to the last directory with:

```
$ cd -
```

and switch back and forth by using that repeatedly.

Make and remove directories

```
$ mkdir ~/tmp # OK

$ mkdir ~/tmp/asdf/asdf # ERROR

$ mkdir -p ~/tmp/asdf/asdf # OK
```

What does `-p` do?

Remove directories:

```
$ rm ~/tmp/asdf # ERROR

$ rm -r ~/tmp/asdf/asdf # OK
```

What does the -r flag do?

> **Warning**
>
> Be careful with *rm -r* and *rm -rf*. You can accidentially remove entire directories that you didn't intend to.

Moving and copying files

mv [source] [dest]:

```
$ touch /tmp/asdf
$ mv /tmp/asdf ~
$ ls -lhtr ~/
```

In-class excercise:

1. make a directory called `/tmp/moveable`
2. move that directory to ~
3. copy that directory to `/tmp/subdir/`

echo

echo means "print":

```
$ echo "hello world"
```

and you can use it to see **bash** variables:

```
$ echo $HOME

$ echo $HISTFILE
```

Variables

We will start covering programming in the next classes, but variables are a key component of programming.

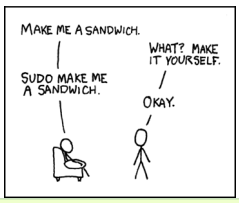
You can do:

```
# define a variable called "important"
$ important=/vol1/opt/data/lamina.bed

# "dereference" (refer to) the variable with a dollah-bill sign
$ ls -lh $important
```

sudo

sudo lets you run commands as root (the super-user). You won't be able to do this on tesla, but you should be able to run sudo on the virtual machine.




```
$ apt-get install cowsay
$ sudo apt-get install cowsay
```

other commands

excercise:

use man to determine the function of:

- wget
- uniq

How many records are present for each chromosome in /vol1/opt/data/lamina.bed (assume it is sorted by chromosome)?

Scripts

A script is simply a series of commands that you save in a file. You will need to write scripts to complete the homework.

Put this text:

```
$ ls /vol1/opt/
```

Into the file `*run.sh*` by opening vim pasting that text then saving the file.

You can then run it as:

```
$ bash run.sh
```

And you should see the same output as if you ran ls /vol1/opt directly.

Scripts will be more useful when you have a series of commands you want to run in series.

For example, a pipeline where you:

1. run quality control on some ChIP-seq reads
2. align reads to a reference genome
3. find peaks (binding sites)
4. annotate the binding sites.

a script will provide a record of what you have done.

Scripts : Commenting

For the homework, it if helpful to us if you comment your scripts.

Comments are not read by the shell, but they tell us (and you) what you were trying to do. You can comment your code using the "#" symbol.

```
# list all files in the /tmp/ directory ordered so that most recently
# changed appear last
$ ls -lhtr /tmp/
```

Pipes

Since linux is made of small utilities, we often want to chain them together. We will cover this in detail next class, but the idea is that each program takes data, modifies it, and sends it to the next.

We can see lines 5-10 of a file with:

```
$ head /vol1/opt/data/lamina.bed | tail -n 5
```

Class 3 : The command-line (part 2)

Class date: Tues 3 Feb 2015

Goals

1. learn additional linux utilities (cut, sort, uniq, less, wget)
2. understand redirects (>, <) and how to combine tools with pipes (|)

wget

fetch a file from the web with `wget`:

```
# N.B.: All common data will be on tesla in /vol1/opt/data

$ cd # go $HOME
$ wget http://hesselberthlab.github.io/workshop/_downloads/states.tab
$ wget http://hesselberthlab.github.io/workshop/_downloads/lamina.bed
$ wget http://hesselberthlab.github.io/workshop/_downloads/t_R1.fastq.gz
```

cut

The `cut` command allows you to extract certain columns of a file:

```
# cut columns 1-4
$ cut -f 1,2,3,4

# cut columns 1-4 and 7-10
$ cut -f 1-4,7-10 /vol1/opt/data/states.tab

# cut first column 1
$ cut -f 1 /vol1/opt/data/lamina.bed

# output all columns after the 1st
$ cut -f 2- /vol1/opt/data/lamina.bed
```

Sort

You will often want to `sort` your data.

Have a look at

```
$ man sort
```

The main flag is *-k* to indicate which column to sort on.

You will also sometimes use *-u* to get unique entries.

Sort Questions

How do you:

1. sort by a particular column? (-k 4)
2. sort as a number (-k4n)
3. sort as a general number (1e-3 < 0.05) (-k4g)
4. change the default delimiter (-t ".")
5. sort by 2 columns (-k1,1 -k2,2n)
6. sort in reverse as a number (-k1rn)
7. get unique entries (-u)

If you know all these, you'll know 99% of what you'll use `sort` for.

Sort Example

BED files have columns *chrom* [tab] *start* [tab] *end* [tab] ...

Sort by chrom, then by start (a lot of tools will require this)

```
$ sort -k1,1 -k2,2n /vol1/opt/data/lamina.bed > /tmp/sorted.bed
```

This tells it to sort the chromosome [column 1] as a character and the start [column 2] as a number.

Question:

What happens if you omit the *n* ?

Sort Example (part 2)

What if we want to sort by Income **descending** in the 3rd column?

```
$ sort -t$'\t' -k3,3rg /vol1/opt/data/states.tab > /tmp/sorted.out
$ head /tmp/sorted.out
```

Sort Exercise

Print out the 10 states (1st column, contains spaces) with the highest income (3rd column) from states.tab using `sort` and piping to `cut`.

Or, use `cut` and pipe to `sort` to do the same.

uniq

The `uniq` command allows you to get and count unique entries

```
# remove duplicate lines
$ cut -f 1 /vol1/opt/data/lamina.bed | uniq

# show duplicate lines
$ cut -f 1 /vol1/opt/data/lamina.bed | uniq -d

# count unique entries:
$ cut -f 1 /vol1/opt/data/lamina.bed | uniq -c
```

Important

uniq assumes that file is sorted by the column of interest.

Use *sort* to sort the data before *uniq*-ing it.

Redirection of output

To send the output of a command (or a file) to another file, use the > operator:

```
$ cut -f 1 /vol1/opt/data/lamina.bed | uniq -c > output.txt
$ head output.txt
```

To **append** the output of a command (or a file) to another file, use the >> operator:

```
$ echo "last line" >> output.txt
$ tail output.txt
```

Compressed Files

The most common way to uncompress single files is `gunzip`:

```
$ gunzip /vol1/opt/data/t_R1.fastq.gz
```

And re-zip the file with `gzip`:

```
$ gzip /vol1/opt/data/t_R1.fastq
```

But if we just want to stream the uncompressed data without changing the file

```
$ zless /vol1/opt/data/t_R1.fastq.gz
```

Pipes

We probably want to do something with the file as we uncompress it

```
$ zless /vol1/opt/data/t_R1.fastq.gz | head
```

We already know the head command prints the first `-n` lines.

Try piping the output to some other commands (tail|echo|cowsay).

Application 1

Use pipes (|) chained together to look see which transcription factor binding sites are the most common in a set of putative sites from ENCODE.

- data file available from http (wget)
- compressed BED format (zless)
- TF name in 4th column (cut)
- count frequency (uniq -c) after sorting (sort)
- sort resulting frequencies so most common are first (sort -rn)
- show top 10 (head)

Application 2

Note that we are using the variable FILE for the long file name

```
# BED format file of transcription factor binding sites
FILE=http://bit.ly/tfbs-x

wget --quiet -O - $FILE \
    | zless \
    | head -n 7000 \
    | cut -f 4 \
    | sort \
    | uniq -c \
    | sort -k1,1rn \
    | head -n 10
```

Let's go through this line by line ...

In Class Exercises - Class 3

1. To learn about piping (|), use cowsay to:

a. show your current working directory

b. show the number of lines in /vol1/opt/data/lamina.bed

c. show the most recently modified file/dir in $HOME

2. write a bash script that you can run to list only the 2 most recently modified files in a given directory (using what you've learned in this class)

3. make that script executable (use google to learn how to do this).

4. With head, you can see the first line of a file with head -n1. How can you see all of a file *except* the first line. (use google)

5. Without using your history, how few keystrokes can you use to run the following command (must work from any directory)?

```
$ ls /vol1/opt/data/lamina.bed
```

6. How few keystrokes can you do 5. using your history?

Class 4 : awk

Class date: Thurs 5 Feb 2015

Goals

1. Review
2. remember BED format (chr, start, end)
3. learn awk basics to filter and manipulate text

awk

http://en.wikipedia.org/wiki/AWK

AWK is an interpreted **programming language** designed for text processing and typically used as a data extraction and reporting tool. It is a standard feature of most Unix-like operating systems.

Named after authors **A** ho, **W** einberger & **K** ernighan

This is programming

basic principles

1. awk operates on each line of a text file
2. in an awk program, $1 is an alias for the 1st column, $2 for the 2nd, etc.
3. awk can filter lines by a pattern

awk program structure

- **BEGIN** runs before the program starts
- **END** runs after the program runs through all lines in the file
- **PATTERN** and **ACTIONS** check and execute on each line.

```
awk 'BEGIN {} (PATTERN) { ACTIONS } END {}' some.file.txt
```

awk BEGIN

You can use **BEGIN** without a file. We just do one thing then exit:

```
awk 'BEGIN { print 12 * 12 }'
```

Same with **END**:

```
awk 'END { print 12 * 13 }'
# then type ctrl+d so it knows it's not getting more input.
```

filtering

A simple and powerful use of awk is lines that match a pattern or meet set of criteria. Here, we match (and implicitly print) only lines where the first column is chr12:

```
awk '($1 == "chr12")' /vol1/opt/data/lamina.bed
```

We can also filter on start position using '&&' which means 'and':

```
awk '($1 == "chr12" && $2 < 9599990)' /vol1/opt/data/lamina.bed
```

Important

= and == are not the same thing, and are frequently mixed up.

= is the assignment operator == tests for equality != tests for inequality.

program structure

```
awk '($1 == "chr12" && $2 < 9599990)' /vol1/opt/data/lamina.bed
```

Important

- when we are checking as a character ("chr12") we need the quotes.
- when we are checking as a number (9599990) can not use quotes.
- can't use commas (e.g. 9,599,990) in numbers

in-class exercise

we will do the first of these together.

1. how many regions (lines) in lamina.bed have a start less than 1,234,567 on any chromosome?
2. how many regions in lamina.bed have a start less than 1,234,567 on chromosome 8?
3. how many regions (lines) in lamina.bed have a start between 50,000 and 951,000
4. how many regions in lamina.bed overlap the interval **chr12:5,000,000-6,000,000** ?

Important

the last question is not trivial and understanding it will be useful

awk program structure (actions)

print total bases covered on chromosome 13:

```
awk '($1 == "chr13") { coverage = coverage + $3 - $2 }
     END { print coverage }' /vol1/opt/data/lamina.bed
```

Important

1. the entire awk program must be wrapped in quotes. Nearly always best to use single quotes (') on the outside.
2. *coverage* is a variable that stores values; we don't use a $ to access it like we do in bash or like we do for the $1, $2, ... columns

in-class exercise

below is how we find coverage for chr13.

```
awk '($1 == "chr13") { coverage += $3 - $2 }
     END{ print coverage }' /vol1/opt/data/lamina.bed
```

how can we find the total coverage for all chromsomes **except** 13?

awk continued

The `$0` variable contains the entire line.

multiple patterns

```
awk '$3 >= 5000 { print $0"\tGREATER" }
     $3  < 5000   { print $0"\tLESS" }' \
      /vol1/opt/data/states.tab
```

remember we can simply filter to the lines > 5000 with:

```
awk '$3 >= 5000' /vol1/opt/data/states.tab
```

awk special variables

1. we know *$1*, *$2*, ... for the column numbers
2. NR is a special variable that holds the line number
3. NF is a special variable that holds the number of fields in the line
4. FS and OFS are the (F)ield and (O)output (F)ield (S)eparators --meaning the delimiters (default is any space character)

using awk to count lines with NR

```
$ wc -l /vol1/opt/data/lamina.bed

$ awk 'END { print NR }' /vol1/opt/data/lamina.bed
```

using FS and OFS

Let's convert lamina.bed to comma-delimited but only for chr12

remember FS is the input separator and OFS is the output delimiter

```
$ awk 'BEGIN{FS="\t"; OFS=","}
    ($1 == "chr12"){ print $1,$2,$3 }' /vol1/opt/data/lamina.bed
```

regular expressions

we won't cover these in detail, but you can match on *regular expressions*.

The following finds lines containing chr2 (chr2, chr20, chr21) in the first column

```
$ awk '$1 ~ /chr2/' /vol1/opt/data/lamina.bed
```

Often we can get by without *regular expressions* but they are extremeley powerful and available in nearly all programming languages.

advanced awk

You can do a lot more with awk, here are several resources:

- http://www.hcs.harvard.edu/~dholland/computers/awk.html
- http://doc.infosnel.nl/quickawk.html
- http://www.catonmat.net/download/awk.cheat.sheet.pdf

In Class Exercises - Class 4

we will do the first 2. of these together

1. use NR to print each line of *lamina.bed preceded* by it's line number

a. do the above, but only for regions on chromosome 12

2. use NF to see how many columns are in each row of *states.tab*

a. use sort and uniq -c to see uniq column counts.

b. why are there 2 numbers?

c. can you adjust the file separator so that awk thinks all rows have the same number of columns?

review

- $1, $2, $3 (default sep is space)
- adjust sep with: OFS="t"; FS=","
- $0 # entire line

```
BEGIN {}
(match) { coverage += $3 - $2 }
END { print coverage }
```

- NR is line number; NF is number of fields;
- BEGIN {} filter { action } END { }

In Class Exercises - Class 4 (2)

1. are there any regions in *lamina.bed* with start > end?
2. what is the total coverage [sum of (end - start)] of regions on chr13 in *lamina.bed*?
3. what is the mean value (4th column) on chromome 3 of *lamina.bed*
4. print out only the header and the entry for colorado in *states.tab*

#. what is the (single-number) sum of all the incomes for *states.tab* with illiteracy rate: a. less than 0.1? b. greater than 2?

1. use NR to filter out the header from *lamina.bed* (hint: what is NR for the header?)

Class 5 : Working with genomic data

Class date: Tues 10 Feb 2015

Goals

1. What is the ENCODE project?
2. What kinds of data did the ENCODE project produce?
3. Where can I find these data on the Internet?
4. What can I do with these data sets?

ENCODE

The Human Genome Project was finished, giving us a list of human genes and their locations. Unfortunately, we still had no idea how they were regulated. If only there was an ENCyclopedia Of Dna Elements...

Advantages: massive amounts of information on key cell lines, reproducible experiments, public data access, technology development.

ENCODE Project Cell Lines

Tier 1: GM12878 (EBV-transformed lymphoblast), K562 (CML lymphoblast), H1-hESC

Tier 2: HeLa-S3 (cervical cancer), HepG2 (liver carcinoma), HUVEC (umbilical vein)

Tier 2.5: SKNSH (neuroblastoma), IMR90 (lung fibroblast), A549 (lung carcinoma), MCF7 (breast carcinoma), LHCN (myoblast), CD14+, CD20+

link (this page also has very useful links to cell culture protocols)

Experiments

1. ChIP-seq: Histone marks, transcription factors
2. Chromatin structure: DNaseI-seq, FAIRE, 5C/Hi-C
3. RNA expression: mRNA-seq, GENCODE gene predictions
4. Data Integration: Segway / ChromHMM integration of functional data

Common File Formats

- FASTQ: Raw sequencing data. `[link] <http://maq.sourceforge.net/fastq.shtml>`
- SAM/BAM: Aligned sequence data `[link] <http://samtools.github.io/hts-specs/SAMv1.pdf>`
- Bed/bigBed: List of genomic regions `[link] <http://genome.ucsc.edu/FAQ/FAQformat.html#format1>`
- Bedgraph/Wig/bigWig: Continuous signal `[link] <http://genome.ucsc.edu/goldenPath/help/bedgraph.html>`

Many other formats are described on this page

References

Completion of the entire project, and a ton of papers: Nature, Genome Research, Genome Biology,

How to Access ENCODE Data

The ENCODE project page is the portal to all of the ENCODE data.

Class 6 : BEDtools

Class date: Thurs 14 Feb 2015

```
mkdir ~/learn-bedtools/
cd ~/learn-bedtools
cp ~/src/bedtools2/test/intersect/a.bed .
cp ~/src/bedtools2/test/intersect/b.bed .
wget http://ucd-bioworkshop.github.io/_downloads/cpg.bed.gz
wget http://ucd-bioworkshop.github.io/_downloads/genes.hg19.bed.gz
```

Ask if you have trouble.

Goals

1. Ask Questions!
2. Introduce the BEDTools suite of tools.
3. Understand why using BEDTools is needed.
4. Practice common operations on BED files with BEDTools.

BEDTools Overview

BEDTools will be one of the tools with the best return on investment. For example, to extract out **all genes that overlap a CpG island**:

```
$ bedtools intersect -u -a genes.hg19.bed.gz -b cpg.bed.gz \
                                > genes-in-islands.bed
```

intersect is a bedtools tool. It follows a common pattern in bedtools that the query file is specified after the `-a` flag and the *subject* file after the `-b` flag

BEDTools Utility

Finding all overlaps between a pair of BED files naively in python would look like:

```
for a in parse_bed('a.bed'):
    for b in parse_bed('b.bed'):
        if overlaps(a, b):
            # do stuff
```

If '*a.bed*' has 10K entries and '*b.bed*' has 100K entries, this would involved checking for overlaps **1 billion times**. That will be slow.

BEDTools uses an indexing scheme that reduces the number of tests dramatically.

Note

See the original BEDTools paper for more information: http://bioinformatics.oxfordjournals.org/content/26/6/841.full

BEDTools Utility (2)

- Fast: faster than intersect code you will write
- Terse: syntax is terse, but readable
- Formats: handles BED, VCF and GFF formats (gzip'ed or not)
- Special Cases: handles stranded-ness, 1-base overlaps, abutted intervals, etc. (likely to be bugs if you do code in manually)

BEDTools Commands

To see all available BEDTools commands, type

```
$ bedtools
```

The most commonly used BEDtools are:

- *intersect*
- *genomecov*
- *closest*
- *map*

BEDTools Documentation

The BEDTools documentation is quite good and ever improving.

See the documentation for *intersect* with:

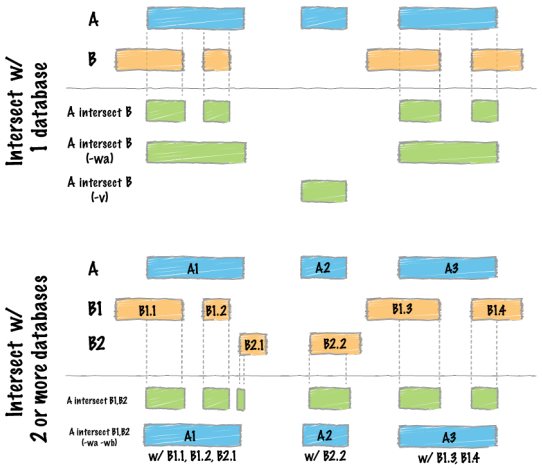
```
$ bedtools intersect
```

The online HTML help is also good and includes pictures:

https://bedtools.readthedocs.org/en/latest/content/tools/intersect.html

BEDTools intersect

Have a browser window open to *BEDTools intersect documentation*. It will likely be the BEDTools function that you use the most. It has a lot of options.



"-v" means (like grep) include all intervals from *-a* that do not overlap intervals in *-b*

Example Files

```
$ cat a.bed
chr1    10  20  a1  1   +
chr1    100 200 a2  2   -

$ cat b.bed
chr1    20  30  b1  1   +
chr1    90  101 b2  2   -
chr1    100 110 b3  3   +
chr1    200 210 b4  4   +
```

What will happen if you intersect those files? For example, the *a.bed* region *chr1:100-200* overlaps:

```
chr1:90-101
chr1:100-110
```

from *b.bed*

intersect

intersect with default arguments means **extract chunks of `-a` that overlap regions in `-b`**

```
$ bedtools intersect -a a.bed -b b.bed
chr1    100 101 a2  2   -
chr1    100 110 a2  2   -
```

Here is the original interval from *a.bed*:

```
chr1        100     200     a2      2       -
```

And the overlapping intervals from *b.bed*:

```
chr1        90      101     b2      2       -
chr1        100     110     b3      3       +
```

intersect -wa

Often, we want the *entire interval from -a if it overlaps any interval in -b*

```
$ bedtools intersect -a a.bed -b b.bed -wa
chr1    100 200 a2  2   -
chr1    100 200 a2  2   -
```

We can get that uniquely with (-u)

intersect -wo

We can see which intervals in *-b* are associated with *-a*

```
$ bedtools intersect -a a.bed -b b.bed -wo
chr1  100  200  a2  2  -  chr1  90  101  b2  2  -  1
chr1  100  200  a2  2  -  chr1  100  110  b3  3  +  10
```

intersect exercise

What happens if you reverse the arguments? E.g. instead of:

```
-a a.bed -b b.bed
```

use:

```
-b a.bed -a b.bed
```

Try that with no extra flags, with -u, -wa, -wo.

How does it compare to the original?

intersect -c

We can count overlaps for each interval in *-a* with those in *-b* with

```
$ bedtools intersect -a a.bed -b b.bed -c
chr1        10      20      a1      1       +       0
chr1        100     200     a2      2       -       2
```

This is our original *a.bed* with an **additional column indicating number of overlaps** with *b.bed*

intersect -v

Extract intervals in *a.bed* that do not overlap any interval in *b.bed*

```
$ bedtools intersect -a a.bed -b b.bed -v
chr1        10      20      a1      1       +
```

Extract intervals in *b.bed* that do not overlap any interval in *a.bed*

```
$ bedtools intersect -a b.bed -b a.bed -v
chr1        20      30      b1      1       +
chr1        200     210     b4      4       +
```

Intersect Summary

- fragments of *a* that overlap *b*: *intersect -a a.bed -b b.bed*
- complete regions of *a* that overlap *b*: *intersect -a a.bed -b b.bed -u*
- intervals of *b* as well as *a*: *intersect -a a.bed -b b.bed -wo*
- number of times each *a* overlaps *b*: *intersect -a a.bed -b b.bed -c*
- intervals of *a* that do not overlap *b*: *intersect -a a.bed -b b.bed -v*

Exercises (Or Other Tools)

1. zless **`cpg.bed.gz`** and **`genes.hg19.bed.gz`**
2. Extract the fragment of CpG Islands that touch any gene [**24611**]
3. Extract CpG's that do not touch any gene [**7012**]
4. Extract (uniquely) all of each CpG Island that touches any gene [**21679**]
5. Extract CpG's that are completely contained within a gene (look at the help for a flag to indicate that you want the fraction of overlap to be 1 (for 100 %). [**10714**]
6. Report genes that overlap any CpG island. [**16908**]
7. Report genes that overlap more than 1 CpG Island (use -c and awk). [**3703**].

> ***Important***
>
> as you are figuring these out, make sure to pipe the output to less or head

Other Reading

- Check out the online documentation.
- A tutorial by the author of BEDTools

Intersect Bam

We have seen that *intersect <bedtools:intersect>* takes *-a* and *-b* arguments. It can also intersect against an alignment BAM file by using *-abam* in place of *-a*

e.g:

```
$ bedtools intersect \
    -abam experiment.bam \
    -b target-regions.bed \
    > on-target.bam
```

Intersect Strand

From the help , one can see that intersect can consider strand. For example if both files have a strand field then

```
$ bedtools intersect -a a.bed -b b.bed -s
```

Will only consider as overlapping those intervals in *a.bed* that have the same strand as *b.bed*.

Closest

with *intersect* we can only get overlapping intervals. *closest* reports the nearest interval even if it's not overlapping.

Example: report the nearest CpG to each gene as long as it is within 5KB.

```
bedtools closest \
    -a genes.hg19.bed.gz \
    -b cpg.bed.gz -d \
    | awk '$NF <= 5000'
```

Map

For each CpG print the sum of the values (4th column) of overlapping intervals from lamina.bed (and filter out those with no overlap using awk)

```
$ bedtools map \
    -a cpg.bed.gz \
    -b /vol1/opt/data/lamina.bed \
    -c 4 -o sum \
    | awk '$5 != "."'
```

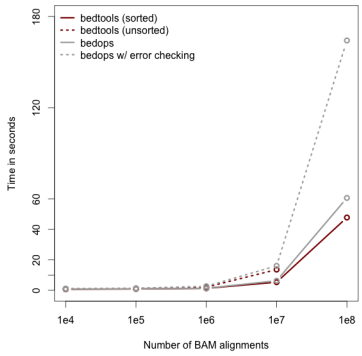
Other *-o* perations include **min**, **max**, **mean**, **median**, **concat**

Sorted

When you start dealing with larger data-files. Look at the *-sorted* flag. For example in *intersect*.

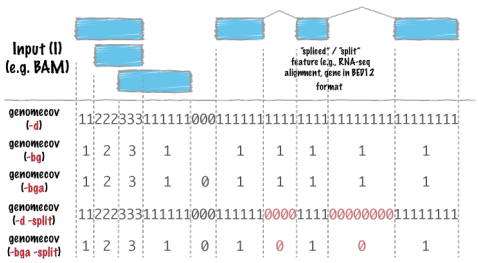
- Uses less memory
- Faster

Takes advantage of sorted chromosome, positions in both files so it doesn't have to create an index.



Genomecov

Get coverage of intervals in BED by BAM



Usually want the last option *-bg -split*

Problem Sets

Problem Sets for the Genome Analysis Workshop class (MOLB 7621):

Problem Set 1

Due date: 2015 Feb 2 at 5 PM MST

Reading

For this problem set, you'll need to read Bill Noble's paper on organizing computational biology projects [7] and be able to put a set of files in the correct places. You should adopt this scheme for all of your projects in and out of class.

Computational biology projects inevitably accrue a lot of files, and this paper is a great way to organize all of this information. As you read in the paper, organization of projects is important for remembering what you did, and reanalyzing data when changes are made.

This quiz will also test your ability to do simple tasks on the command line. You will need to take the tutorial to learn the necessary tools [8].

You will use the command line tools discussed in the tutorial (e.g. mkdir, cd, ls, mv) to create the directory structure, move files into place and check whether everything looks ok.

Problem 1

Make a directory structure as outlined in the paper. The directories should be nested under a common project directory, with directories for data, results and documentation (doc). You should also create dated folders with today's date so you know where to put the dated data and reuslts. Finally, keep a list of the commands that you used to create the directory (**10 points**).

Note

Name your directories with YYYY-MM-DD format (year-month-date, e.g. 2014-01-10). If you do this, the directories will sort chronologically using the directory name.

Avoid MM-DD-YY, they will not sort chronologically by name.

Problem 2

Download the the following data table: **`states.tab`**. Then move the data file to the appropriate *dated* directory.

```
$ wget http://hesselberthlab.github.io/workshop/_downloads/states.tab
```

You need to create a run.sh shell script that runs the following code, and writes the output into a results directory with the current date (**10 points**).

You should copy the following code block into a file using *vim*. You will need to change the '???' characters in the file to correspond to the path you want to write the results in (hint: it should include today's date).

```
 1 #! /usr/bin/env bash
 2 #
 3 # run script for quiz 1
 4
 5 # these are bash flags the print out variables that get set when you
 6 # run the script.
 7 set -o nounset -o pipefail -o errexit -x
 8
 9 # You will need to change the '???' strings below.
10 #
11 # define the project variable here. this should be the full path to
```

```
# your project directory, i.e. the directory at the top of the
# results/data/doc directories.

project=???

# fill in the date here
date=???

# these refer to the data file that you moved into place
data=$project/data/$date
datafile=$data/states.tab

# these refer to the place where you will write the results of the
# "analysis"
results=$project/results/$date
resultsfile=$results/result.tab

# if the directory doesn't exist, make it
if [[ ! -d $results ]]; then
    mkdir -p $results
fi

# Note how we are using redirects here. The first ">" writes a file,
# and overwrites existing data. The following ">>" append data to the
# existing file

echo "here is our starting data ..."
cat $datafile > $resultsfile
echo

echo "here are the states sorted by population size ..."
sort -k2n $datafile >> $resultsfile
echo

echo "here are the states with the highest number of murders ..."
sort -k6n $datafile | head -n 10 >> $resultsfile
```

Then, save the above text in a run.sh script in your results directory. To run the file, use:

```
$ bash run.sh
```

If this ran correctly, you should see a new results.tab file in the results directory you specified in the run.sh script. If you don't see the file, double check the path you specified, and make sure you're looking in the right spot. If it's in a different spot than you intended. remove the results file you wrote, update the program and run it again.

Problem 3

Finally you need to create a log of what you did in the root of the results directory to summarize the key points of your analysis (**5 points**). For example:

```
Captain's log, star date 2014-07-16
-----------------------------------
After examining the results.tab file, learned that Alaska has the highest
income per person. Something must be wrong ...
```

---

7 A Quick Guide to Organizing Computational Biology Projects (2009) PLoS Comput. Biol. William S. Noble http://dx.plos.org/10.1371/journal.pcbi.1000424

8 The Command Line Crash Course http://cli.learncodethehardway.org/book/

Problem Set Submission

Submit your problem set as a tar file to Canvas (*Problem Set Submission*).

Problem Set 2

Due date: 2015 Feb 9 at 5 PM MST

Overview

For this quiz you will use the tools we learned about in the last several classes, focusing on manipulating text files with Linux command line tools.

> ### Note
>
> Continue to use the organization scheme that we learned about in *Problem Set 1*. Part of our evaluation will include whether you are developing good organizational habits.

In the solution for all of the problems below, print out the region(s) as they appear in the BED file, with additional columns at the end. e.g.:

```
chr12 <tab> 1234 <tab> 5678 <tab> 0.93 <tab> my-extra-info
```

These should be separated by tabs (a `\t` character), **not spaces**, unless otherwise indicated.

Combine `awk` with the other utilities you have learned. Create a `run.sh` file that executes the commands for each problem and writes out each result in a dated directory.

Problem 1

1. What is the region with the largest start position (2nd column) on any chromosome in *lamina.bed*? (**5 points**)
2. What is the region with the largest end position on chrY in lamina.bed? Report this region in the format: `chr12:1234-5678`" (**5 points**)

Problem 2

1. What is the longest region (end - start) in *lamina.bed*? (**5 points**) Report as:

   ```
   chrom <tab> start <tab> end <tab> value <tab> region_length
   ```

2. What is the longest region (end - start) in *lamina.bed* with a value (4th column) greater than 0.9 on chr13. Report the header (1st line) in lamina.bed as well as the region (**5 points**).

Problem 3

1. What are the regions that overlap this interval in *lamina.bed*: `chr12:5,000,000-6,000,000`? Report regions so that they are ordered by descending value (4th column), and the columns are separated by commas rather than tabs (**5 points**).
2. What is the average value (4th column) of those regions from *lamina.bed* that overlap the region (**5 points**): `chr12:5,000,000-6,000,000`?

Problem Set Submission

Submit your problem set by posting the path to your tar file on the Canvas page. (*Problem Set Submission*).

Problem Set Keys

Past keys at *problem-set-keys*

Problem Set Submission

In general, we want one run.sh file that includes all of the code necessary to run the problem set. This run.sh file should create any new directories (ex. a dated directory in the results folder), perform commands (ex. awk, cut, etc), and output results into a well-named file (ex. > $results/$date/problem1.txt). For each problem set, you should also create a log file summarizing your results.

Once your problem set is complete, you will need to create a tar file. Specify the root of your project directory and create a tar file of the whole directory like this (change STUDENTID to your student ID):

```
$ projectdir=$HOME/project
$ tar -cvf STUDENTID-pset1.tar $projectdir
```

On the specific Problem Set assignment page at the Canvas site [9], click the Submit Assignment button on the top right and paste the full path to the tarfile (/vol3/home/username/.../foo.tar) in the text box. Click the Submit Assignment button below the text box to complete the submission. Auditors can e-mail their path to sallypeach@gmail.com.

9 https://ucdenver.instructure.com/courses/325063/assignments

Miscellaneous

General content not tied to specific classes.

Reference: list of commands

cd

change directories:

```
$ cd /tmp/
$ cd ~ # chage to home directory
$ cd /vol1/opt/
$ cd - # change to previous directory
```

cp

copy files and directories:

```
$ touch /tmp/asdf
$ cp /tmp/asdf ~ # copy to home
```

must use -r for directories:

```
$ mkdir /tmp/adir
$ cp -r /tmp/adir ~/
```

ctrl+c

interrupt a running process:

```
$ head
<ctrl+c>
```

cut

extract columns from a file:

```
$ cut -f 1-3 /vol1/opt/data/lamina.bed
$ cut -f 1,3 /vol1/opt/data/lamina.bed
$ cut -f 1 /vol1/opt/data/lamina.bed

# use comma as delimiter instead of default tab
$ cut -f 1-3 -d, /path/to/some.csv

# keep all columns after the 1st:
$ cut -f 2- /vol1/opt/data/lamina.bed
```

echo

print some text:

```
$ echo "hello world" | cowsay
```

head

show the start of a file:

```
$ head /vol1/opt/data/lamina.bed

# show the first 4 lines
$ head -n 4 /vol1/opt/data/lamina.bed
```

grep

To find any instance of *chr5* in the lamina.bed file

```
# grep [pattern] [filename]
$ grep chr5 /vol1/opt/data/lamina.bed | head
```

To find all lines that start with a number sign:

```
# The caret (^) matches the beginning of the line
# FYI dollar sign ($) matches the end
$ grep '^#' /vol1/opt/data/lamina.bed
```

To find any line that *does not* start with "chr":

```
# the -v flag inverts the match (grep "not" [pattern])
$ grep -v '^chr' /vol1/opt/data/lamina.bed
```

Find exact matches that are split on words with the `-w` flag:

```
$ grep -w chr1 /vol1/opt/data/lamina.bed | cut -f1 | uniq
```

less

page through a file:

```
$ less /vol1/opt/data/lamina.bed
```

use "/", "?" to search forward, backard. 'q' to exit.

use '[space]' to go page by page.

ls

list files and directories:

```
$ ls /tmp/

# show current directory
$ ls

# show current directory (2)
$ ls .

# list files with most recently modified last
$ ls -lhtr

# list files in temp ordered by modification date
$ ls -lhtr /tmp/
```

man

show the manual entry for a command:

```
$ man head
```

mkdir

make a directory:

```
$ mkdir /tmp/aa
```

make sub-directories, too:

```
$ mkdir -p /tmp/aaa/bbb/
```

mv

move a file or directory:

```
$ touch /tmp/aa
$ mv /tmp/aa /tmp/bb
```

rm

remove a file or directory:

```
$ touch /tmp/asdf
$ rm /tmp/asdf

# use -r to remove directory
$ mkdir /tmp/asdf
$ rm -r /tmp/asdf
```

sort

sort a file by selected columns:

```
$ sort -k1n /vol1/opt/data/lamina.bed
```

sort a BED file by chromosome (1st column) as character and then by start (2nd column) as number:

```
$ sort -k1,1 -k2,2n /vol1/opt/data/lamina.bed
```

sort by 4th column as a general number, including scientific notation showing largest numbers first:

```
$ sort -k4,4rg /vol1/opt/data/lamina.bed | head
```

use literal tab ('\t') as the delimiter (default is whitespace):

```
$ sort -t$'\t' -k4,4rg /vol1/opt/data/lamina.bed | head
```

Sometimes we want to get uniq entries with sort -u:

```
$ cut -f 1 /vol1/opt/data/lamina.bed | sort -u
```

will print out the uniq chromsomes represent in the BED file.

tail

show the end of a file:

```
$ tail /vol1/opt/data/lamina.bed
# show the last 4 lines
$ tail -n 4 /vol1/opt/data/lamina.bed
```

tar

create or untar a .tar.gz file:

```
# -c create -z compress (.gz) -v verbose -f the name
$ tar -czvf some.tar.gz /tmp/*

# -x untar
$ tar -zxvf some.tar.gz
```

uniq

show or count unique or non-unique entries in a file:

```
# count number of times each chromosome appears.
$ cut -f 1 /vol1/opt/data/lamina.bed | uniq -c

# get non unique entries
$ cut -f 2 /vol1/opt/data/lamina.bed | uniq -d
```

Important

`uniq` assumes that the file is `sort`-ed first! To test this, run `uniq` on an unsorted file. What happens?

wget

get a file from the web:

```
$ wget http://ucd-bioworkshop.github.io/_downloads/states.tab
```

zless

like less, but for compressed files:

```
$ zless /vol1/opt/data/t_R1.fastq.gz
```

Redirection (>> and >)

send output to a file:

```
# start a new file
$ echo "hello" > file.txt

# overwite that file
$ echo "hello!" > file.txt

# append to the file
$ echo "world" >> file.txt
```

Credits

Several people have contributed to the development and execution of this course:

```
Spring 2014
-----------
Brent Pedersen
Sally Peach
Eric Nguyen
```

Sample Data files

We will use several example data files throughout the class.

BED format

Data in BED format contains region information (e.g. single nucleotides or megbase regions) in a simple format [10]:

Download a sample BED file: `lamina.bed`

FASTQ format

FASTQ format contains DNA sequence data with quality scores:

```
@cluster_2:UMI_ATTCCG             # record name; starts with '@'
TTTCCGGGGCACATAATCTTCAGCCGGGCGC   # DNA sequence
+                                 # empty line; starts with '+'
9C;=;=<9@4868>9:67AA<9>65<=>591   # phred-scaled quality scores
```

Download a sample FASTQ file: `SP1.fq`

FASTA format

FASTA format just contains DNA sequence data; no quality scores:

```
>cluster_2:UMI_ATTCCG             # record name; starts with '>'
TTTCCGGGGCACATAATCTTCAGCCGGGCGC   # DNA sequence
```

Download a sample FASTA file: `sample.fa`

ENCODE data

All encode data are available at https://genome.ucsc.edu/ENCODE/downloads.html

For Problem Set 5, you will need these files on the amc-tesla cluster, available in:

```
/vol1/opt/data
```

Experiment	Target	Cell line	Replicate	File Type	File name
ChIP-seq	Histone H3 Lysine 4 trimethyl (H3K4me3)	Hela-S3	1	FASTQ	wgEncodeBroadHistoneHelas3H3k4me3StdRawDataRep1.fastq.gz
ChIP-seq	CTCF	Hela-S3	1	narrowPeak	wgEncodeUwTfbsHelas3CtcfStdPkRep1.narrowPeak.gz
Merged TFBS ChIP-seq	all	all	n/a	BED	wgEncodeRegTfbsClusteredV3.bed.gz

Merged DNase I hypersensitive sites	all	all	n/a	BED	wgEncodeRegDnaseClusteredV2.bed.gz

Getting Started

VPN on Windows computers

Several people have had problems using VPN while on Windows computers with VirtualBox running LinuxMint OS. Here are Erin Baschal's notes on Mike Campbell's solutions:

```
How to fix internet/VPN issue in VirtualBox
Open Network and Sharing Center
     Windows 8: right click on network icon at bottom of screen
Change adapter settings (on left)
Virtual Box Host Adapter (right click)
    Properties
    Uncheck TCP/IPv6
    Double click TCP/IPv4
    Use the following DNS server addresses:
        140.226.189.35
        132.194.70.65
    Advanced, DNS tab
        Append these DNS suffixes:
            ucdenver.pvt
```

Programming Tips & Tricks

Overview

Several things influence how effectively you learn to program, and how well you program once you have mastered the basic ideas. A major issue is your efficiency in actaully *using* a computer, and not programming *per se*.

For example, the longer you spend searching for a particular key to type, or surfing around with your mouse, the less time you spend writing, running and debugging programs. Here are several pointers to help you become more efficient at using computers, independent of learning programming languages.

Learn to type

Hunting and pecking is inefficient, and prevents you from spending your valuable time efficiently. If you're looking at your keyboard, you're *not* looking at the screen, reading and debugging code. Once you have the typing basics down, you should be typing 40-50 words per minute, without ever looking at the keyboard.

There are several good, modern tools [11] to help you master touch-typing.

Learn to type funny characters

In programming you use a variety of characters that you don't always use typing other kinds of documents. Learn the locations of the following by heart:

---

10 BED documentation http://genome.ucsc.edu/FAQ/FAQformat.html#format1

11
- Touch Typing Tutorial : http://www.typingweb.com/
- Typing IO : Language specific typing practice http://typing.io/

- Number sign (for commenting): #
- Dollar sign (for variables): $
- Underscore (for variable naming): _
- Parentheses: ()
- Brackets: []
- Curly brackets: {}
- Tilde (i.e. the squiggle; for going $HOME): ~
- Math symbols ("+", "-", "*", "/", "=")

Learn hot keys for window management

The mouse is your enemy. Yes, it revolutionized the computer–human interaction. But the more time you spend using your mouse, the less time you spend with your hands on the keyboard and doing useful things.

You can do most things with your keyboard. There are several hot keys you should learn that will maximize your productivity on the computer by minimizing your use of the mouse:

- **<Alt>-<Tab>** : Flip through windows quickly and effortlessly without ever touching your mouse.
- **<Ctrl>-<Page Up/Down>** : Switch between Terminal windows on Linux

> ***Tip***
>
> **Launch your most-used apps automatically during login.**
>
> For example, automatically launch four terminal windows and a browser window, without having to click.

Learn to use a terminal text editor

There are two types of nerds in this world:

- `vim` users
- `emacs` users

I'm a `vim`-user. I can't even log out of an `emacs` session.

Learning a terminal text editor like `vim` increases productivity substantially, because it allows you to:

- run the editor within an existing terminal, without opening a new window
- work on multiple documents simultaneously
- syntax highlight your code
- manipulate blocks of text quickly and efficiently

You can run `vim` from the terminal prompt:

```
$ vim filename.txt
```

To quit a vim session, you need to:

1. enter *command mode* with the colon key
2. write the file
3. quit the program

This can be accomplished by typing:

```
:wq <enter>
```

In your copious spare time, and after you have mastered the basics of shell, Python and R programming, you should take a tutorial [12] on using a terminal text editor.

12 OpenVim http://www.openvim.com/