

# Unix command line; editors

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

`kbroman.org`

`github.com/kbroman`

`@kwbroman`

Course web: [kbroman.org/AdvData](https://kbroman.org/AdvData)

My goal in this lecture is to convince you that

- (a) command-line-based tools are the things to focus on,
- (b) you need to choose a powerful, universal text editor (you'll use it a lot),
- (c) you want to be comfortable and skilled with each.

For your work to be reproducible, it needs to be code-based; don't touch that mouse!

# Windows vs. Mac OSX vs. Linux

## Remote vs. Not

2

The Windows operating system is not very programmer-friendly.

Mac OSX isn't either, but under the hood, it's just unix.

Don't touch the mouse! Open a terminal window and start typing.

I do most of my work directly on my desktop or laptop. You might prefer to work remotely on a server, instead. But I can't stand having any lag in looking at graphics.

If you use Windows...

Consider **Git Bash** (or **Cygwin**)  
or turn on the **Windows subsystem for linux**

3

Cygwin is an effort to get Unix command-line tools in Windows.

Git Bash combines git (for version control) and bash (the unix shell); it's simpler to deal with than Cygwin.

Linux is now accessible in Windows 10, but you have to enable it.

## If you use a Mac...

Consider **Homebrew** and **iTerm2**  
Also the **XCode** command line tools

4

Homebrew is a packaging system; iTerm2 is a Terminal replacement.

The XCode command line tools are a must for most unixy things on a Mac.

I do all of my work on a Mac (except really big computational jobs), and there are a lot of different tools that I like and would recommend:

divvy, <http://mizage.com/divvy>

caffeine, <http://lighthouse.com/caffeine>

bartender, <http://www.macbartender.com>

hazel, <http://www.noodlesoft.com/hazel.php>

launchbar, <http://www.obdev.at/products/launchbar/index.html>

simplenote, <http://simplenote.com>

jumpcut, <http://jumpcut.sourceforge.net>

color oracle, <http://colororacle.org>

textexpander, <http://smilesoftware.com/TextExpander>

## The command line is your friend

- ▶ Don't touch that mouse!
- ▶ Scriptable
- ▶ Flexible

5

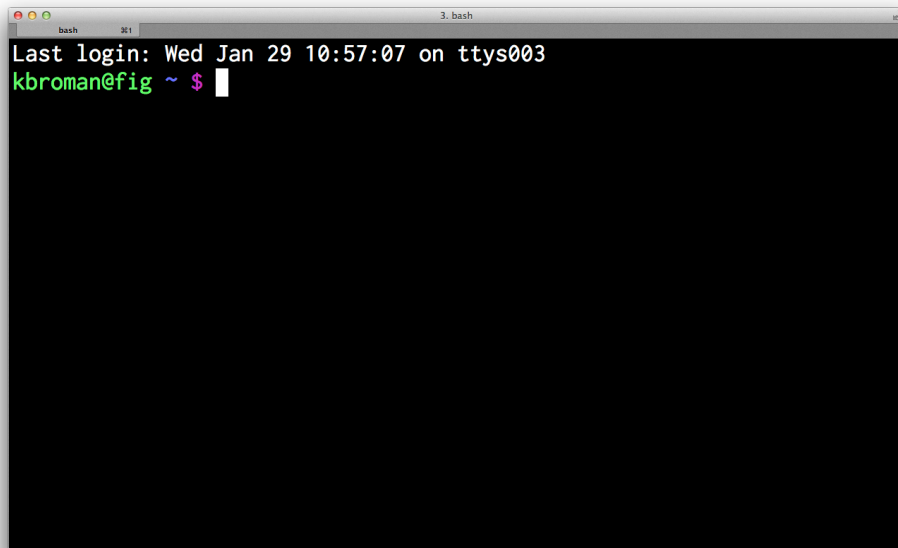
In the long run, you'll be happier, having conquered the command line.

Pointing-and-clicking is not reproducible, and every time you take your hands off the keyboard, there's a loss of efficiency.

The command line allows you to piece together multiple tools and so do things that weren't anticipated by the developer of the GUI.

And it's only through scripts that you'll have truly reproducible analyses.

# The shell

A terminal window with a black background and white text. The window title bar shows '3. bash'. The text inside the terminal reads: 'Last login: Wed Jan 29 10:57:07 on ttys003' followed by a green prompt 'kbroman@fig ~ \$' and a white cursor. The rest of the terminal is empty.

```
bash 21 3. bash
Last login: Wed Jan 29 10:57:07 on ttys003
kbroman@fig ~ $
```

Options: `tcsh`, `bash`, `zsh`

The shell is a program – an interface to the operating system.

There are a number to choose from. I use `bash`; I've heard great things about `zsh`.

## Basics

- ▶ Directory structure

Absolute vs. relative paths

```
ls -l ~/Figs ../Rawdata/
```

- ▶ Creating, removing, changing directories

```
mkdir
```

```
rmdir
```

```
cd
```

```
cd -
```

- ▶ Moving, copying, removing files

```
mv
```

```
cp
```

```
rm -i
```

This stuff is too boring to spend much time on.

But I should emphasize the importance of using relative paths (e.g., `../Figs/fig1.pdf`) in a project; reliance on absolute paths (e.g., `~/Projects/Blah/Figs/fig1.pdf`) make life difficult when you move the project to a different system.

## ~/ .bash\_profile

```
export PATH=./usr/local/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/lib

noclobber=1      # prevent overwriting of files
IGNOREEOF=1     # disable Ctrl-D as a way to exit
HISTCONTROL=ignoredups

alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias ls='ls -GF'
alias 'l.='='ls -d .[a-zA-Z]*'
alias ll='ls -lh'
alias md='mkdir'
alias rd='rmdir'

alias Rb='R CMD build --force --resave-data'
alias Ri='R CMD INSTALL --library=/Users/kbroman/Rlibs'
alias Rc='R CMD check --library=/Users/kbroman/Rlibs'
alias Rcc='R CMD check --as-cran --library=/Users/kbroman/Rlibs'
```

8

Use the `.bash_profile` file to define various variables and aliases to make your life easier.

The most important variable is `PATH`: it defines the set of directories where the shell will look for executable programs. If `."` isn't part of your `PATH`, you'll need to type something like `./myscript.py` to execute a script in your working directory. So put `."` in your `PATH`.

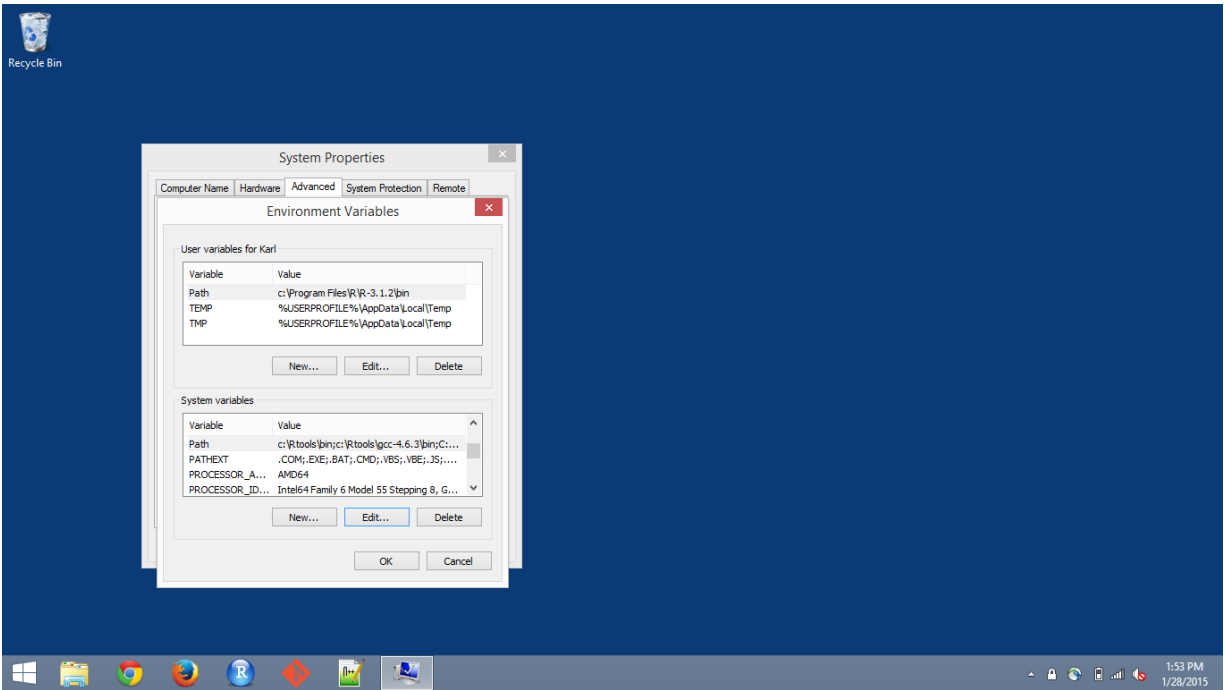
My `.bash_profile` file sources a `.bashrc` file; I don't quite understand when one is used versus the other. Google `".bashrc vs .bash_profile."` There are links to my `.bash_profile` and `.bashrc` files on the resources page at the course web site; some of it might just be total crap.

If you're using Windows and Git Bash, the `.bash_profile` file will be in your Documents folder (I think).

Important note: use of aliases within your code will create reproducibility issues; another user will need those same aliases. Consider testing your code on a more basic account.



# PATH in Windows



With Git Bash, you can have a `~/.bash_profile` file that adds stuff to your `PATH`, just as in Mac OS X and Linux.

But things will also be added to the `PATH` variable via the `Path` system variable and/or a `Path` user variable. You can get to these via the “Control panel,” but it’s a bit cumbersome.

The simplest way to get to the relevant dialog box seems to be to click Win-w (the little windows key and the w key) and searching for “path”.

## Redirection and pipes

```
$ curl -L http://bit.ly/hamlet_txt > hamlet.txt
$ wc -l hamlet.txt
$ grep Ham hamlet.txt > tmp.txt
$ wc -l tmp.txt
$ grep Ham hamlet.txt | wc -l
$ grep Ham hamlet.txt | less
$ cat file1.txt file2.txt > combined.txt
$ cat file3.txt >> combined.txt
```

10

Use `>` to redirect output “stdout” to a file.

Use `>>` to redirect output and append to the file.

Use `<` to have input “stdin” come from a file.

Use `|` to have the output of one command made the input to another.

A key design principle in Unix is the piecing together of small commands using this sort of technique. There are lots of little commands (often with short, cryptic names) that can be combined together with great flexibility.

Important tools mentioned here: `curl` (for downloading web stuff on the command line; `-L` is to follow any re-direction; see also `wget`), `grep` (search for patterns in a file), `less` (look through long files a page at a time), `wc` (count the number of words, lines and/or characters in a file; `-l` is for the number of lines), `cat` (print contents or concatenate text files)

## Wild cards

```
$ grep blah *.txt
$ ls blah.???
$ ls [a-z]*
$ ls /usr/bin/[auz]*
$ ls /usr/bin/[auz]*.*
$ ls -l *.txt | wc -l
$ wc -l *.txt | grep total
```

11

\* stands for anything

? stands for a single character

Use [] to match some specific set or range of characters

## Suspend/foreground/background

```
$ R CMD BATCH input.R output.txt &

$ R CMD BATCH input.R output.txt
[ctrl-Z]
$ bg

$ emacs afile.txt
[ctrl-Z]
$ fg
```

12

Use `&` to run a job in the background.

Use `ctrl-Z` to suspend the current job (but this doesn't work in Windows). Then use `bg` to then put it in the background or `fg` to bring it back to the foreground.

I use `ctrl-Z` and `bg` if I had forgotten to use `&`.

I use `ctrl-Z` with `emacs` sometimes, to do some command-line things without opening another shell/terminal; I'll then use `fg` to bring `emacs` back. Or I'll forget about it and muck a bunch of stuff up.

## Moving around the command line

<code>ctrl-f, ctrl-b</code>	move forward and back
<code>ctrl-a, ctrl-e</code>	move to beginning and end of line
<code>ctrl-k, ctrl-u</code>	delete rest of line, or to the start
<code>ctrl-l</code>	clear the screen
<code>ctrl-c</code>	cancel what you've typed
<code>tab</code>	autocomplete command or file
<code>ctrl-p, ctrl-n</code>	forward and backward in history
<code>ctrl-r</code>	search for a previous command

These are mostly emacs-like key “bindings”.

## How to solve computing problems

- ▶ Try stuff!
- ▶ man pages and help files
- ▶ `blah -h` or `blah --help`
- ▶ Google
- ▶ Stackoverflow and other StackExchange sites
- ▶ Google with `site:stackoverflow.com`
- ▶ email lists and google groups
- ▶ friends or colleagues
- ▶ Twitter
- ▶ Buy a book. Buy **all** of the books.

14

You will run into crazy and mysterious errors. Will you give up, or figure them out?

Rule number 1: try stuff. Figure out how something works by trying it out in different ways.

Rule number 2: Google. Google the exact error message, or a part of an error message. You'll often get to a stackexchange site; don't forget to read the comments as well as the answers. Often the best answer is in a comment.

Rule number 3: Ask for help. Talk to your friends. Talk to me. Post a question at a stackexchange site.

I'm a relatively recent convert to Twitter. I focus on just a few things that interest me (mostly academic publishing, reproducible research, and interactive graphics). If you tweet a question, you'll be surprised at how quickly you get an answer.

I do tend to buy all possible books on a topic that is of even passing interest to me. I read at least part of each of them.

## Examples

- ▶ How do you suppress warnings in knitr?
- ▶ What symbol corresponds to the unicode `\u00B1`?
- ▶ What's the difference between `curl` and `wget`?
- ▶ What does "502 Bad Gateway" mean?
- ▶ "To open `gs` you need to install X11"
- ▶ `mclapply` isn't working in Windows
- ▶ How to ping a server in Python?
- ▶ Font shape ``EU1/pplx/m/n'` undefined
- ▶ `except KeyError, k: raise AttributeError, k`

15

These are examples of things you might search for.

If you don't understand an error message, start by pasting it into google.

## Important principle

Learn to code by looking at good code.

Identify programmers that you respect (e.g., Hadley Wickham), and study what they do.



## Choose a good editor

- ▶ Emacs
- ▶ VIM
- ▶ RStudio
- ▶ Textwrangler
- ▶ Notepad++
- ▶ Sublime Text
- ▶ Atom

17

I use emacs; I should probably use vim.

RStudio is increasingly useful, but as a general editor (for things that aren't R), I think it's insufficient.

The choice of editor is very personal.

## A good editor

- ▶ Doesn't require pointing-and-clicking
- ▶ Easy to get code between R and a script
- ▶ Syntax highlighting of code
- ▶ Automatic indentation
- ▶ Close parentheses/brackets/braces
- ▶ Browse code across files
- ▶ Integrated with other tools (e.g., version control)

18

I've not figured out how to explore code across a set of files in emacs; otherwise I'm very happy with it.

## Other useful tools

```
$ find . -name *.py
$ locate article.cls

$ ps ux

$ top

$ df -hk

$ du -h
$ du -hd2

$ ln -s ~/Projects/SomeFriend/Data
$ ln -s ~/Projects/SomeFriend/Data SomeFriend_Data

$ tar xzvf qtl_1.29-2.tar.gz
$ tar czvf blah.tgz Blah/
$ tar tzvf blah.tgz
```

19

`find` and `locate` for finding files.

`ps ux` to see what processes are running.

`top` gives an interactive view of what processes are running.

`df -hk` shows disk usage

`du -hd2` shows disk usage in a directory and its subdirectories; the `d2` bit says go no more than 2 levels down through the subdirectories.

`ln -s` makes a “soft link” to a file or directory. It acts like there’s a copy, but it’s not really copied.

`tar` is used to archive a bunch of files within a single file. `x` for extract, `c` for combine, `t` for test, `z` for compress/zip, `v` for verbose, `f` for “file name to follow.”

## Further useful tools

```
$ whereis bash
$ type rm
$ type emacs

$ pwd

$ head afile.txt
$ tail afile.txt
$ head -n20 afile.txt
$ man head

$ kill 8453
$ kill -9 8453

$ history
$ !!
$ !-2
$ !503

$ ping www.google.com

$ ispell afile.txt
```

20

`whereis` for finding a program.

`type` for figuring out the location of a program or the definition of an alias.

`pwd` – print working directory

`head` – print first few lines of a file

`tail` – print the last few lines of a file

`man` – view manual page

`kill` – kill a job

`history` – view command history

`!` – execute past commands

`ping` – see if you can connect to some server

`ispell` – spell checker

## Opening a file from the command line

### Windows:

```
$ start mypaper.pdf  
$ start http://google.com
```

### Mac:

```
$ open mypaper.pdf  
$ open http://google.com
```

I often like to open a file from the command line. If the file extension is known, you can use `start` in Windows or `open` in Mac OS X.

In Linux, you may have `xdg-open` (in the `xdg-utils` package on Ubuntu). You might make an alias (e.g. `open`) for that in your `.bash_profile` file.

# File modes

```
kbroman@fig ~/Teaching/Tools4RR/Lectures (master) $ ll
total 8
drwxr-xr-x 27 kbroman staff 918B Jan 27 11:35 01_Intro/
drwxr-xr-x 30 kbroman staff 1.0K Jan 29 11:38 02_Unix/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:28 03_KnitrMarkdown/
drwxr-xr-x 37 kbroman staff 1.2K Jan 20 23:05 04_Git/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:28 05_Organization/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:28 06_EDA/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:28 07_ClearCode/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:28 08_Rpack/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 09_TestingDebugging/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 10_BigJobs/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 11_KnitrPapers/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 12_KnitrTalks/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 13_KnitrPosters/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 14_Python/
drwxr-xr-x 2 kbroman staff 68B Jan 14 06:29 15_Licenses/
-rwxr-xr-x 1 kbroman staff 488B Oct 23 16:18 createVersionWithNotes.rb*
kbroman@fig ~/Teaching/Tools4RR/Lectures (master) $
```

Note the mode, owner, and group for each file.

mode = read/write/executable for owner/group/everyone

r = readable; w = writable; x = executable (for a directory, enter-able)

## File modes/owner/group

```
sudo chown kbroman .  
chgrp -R staff .  
chmod +x createVersionWithNotes.rb  
chmod 755 02_Unix  
chmod 644 02_Unix/02_unix.tex  
chmod 700 Private_stuff
```

23

You don't usually need to change the owner or group assigned to a file or directory, but it's good to be aware of the possibility.

Groups are useful if you want a file accessible by some set of people but not everyone. You need a system admin to set up the group.

You often want to make scripts executable, or make files/directories unreadable or unwritable.

For example, primary raw data files should not be writable. Large Excel-based data files often contain screwed up cells where someone was typing in some random spot without realizing it. I found myself doing that yesterday!

The octal codes (e.g, 755 and 644) are convenient, once you get the hang of them.

# Don't forget to look at the resources page

[kbroman.org/AdvData/resources](http://kbroman.org/AdvData/resources)

24

If you find other useful resources, let me know.

When we get to git and GitHub, make a pull request!