Basics of UNIX

August 23, 2012

By UNIX, I mean any UNIX-like operating system, including Linux and Mac OS X. On the Mac you can access a UNIX terminal window with the Terminal application (under Applications/Utilities). Most modern scientific computing is done on UNIX-based machines, often by remotely logging in to a UNIX-based server.

1 Connecting to a UNIX machine from {UNIX, Mac, Windows}

See the file on bspace on connecting remotely to SCF. In addition, this SCF help page has information on logging in to remote machines via ssh without having to type your password every time. This can save a lot of time.

2 Getting help from SCF

More generally, the department computing FAQs is the place to go for answers to questions about SCF.

For questions not answered there, the SCF requests: "please report any problems regarding equipment or system software to the SCF staff by sending mail to 'trouble' or by reporting the problem directly to room 498/499. For information/questions on the use of application packages (e.g., R, SAS, Matlab), programming languages and libraries send mail to 'consult'. Questions/problems regarding accounts should be sent to 'manager'."

Note that for the purpose of this class, questions about application packages, languages, libraries, etc. can be directed to me.

3 Files and directories

1. Files are stored in directories (aka folders) that are in a (inverted) directory tree, with "/" as the *root* of the tree
2. Where am I?

   ```
   > pwd
   ```

3. What's in a directory?

   ```
   > ls
   > ls -a
   > ls -al
   ```

4. Moving around

   ```
   > cd /home/paciorek/teaching/243
   > cd ~paciorek/teaching/243
   > cd ~/teaching/243
   > cd 243 # provided I am in 'teaching'
   > cd ..
   > cd -
   ```

5. Copying and removing

   ```
   > cp
   > cp -r
   > cp -rp # preserves timestamps and other metainfo (VERY handy
   for tracing your workflows if you move files between machines)
   > mkdir
   > rm
   > rm -r
   > rm -rf # CAREFUL!
   ```

   To copy between machines, we can use *scp*, which has similar options to *cp*:

   ```
   > scp file.txt paciorek@bilbo.berkeley.edu:~/research/.
   > scp paciorek@bilbo.berkeley.edu:/data/file.txt
   ~/research/renamed.txt
   ```

6. File permissions: **`ls -al`** will show the permissions for the '*user*', '*group*', '*other*'

   - to allow a file to be executed as a program:

     ```
     > chmod ugo+x myProg # myProg should be compiled code or a shell script
     ```

   - to allow read and write access to all:

     ```
     > chmod ugo+rw code.q
     ```

   - to prevent write access:

     ```
     > chmod go-w myThesisCode.q
     ```

7. Compressing files

   - the *zip* utility compresses in a format compatible with zip files for Windows:

     ```
     > zip files.zip a.txt b.txt c.txt
     ```

   - *gzip* is the standard in UNIX:

     ```
     > gzip a.txt b.txt c.txt # will create a.txt.gz, b.txt.gz, c.txt.gz
     ```

   - *tar* will nicely wrap up entire directories:

     ```
     > tar -cvf files.tar myDirectory
     > tar -cvzf files.tgz myDirectory
     ```

   - To unwrap a tarball

     ```
     > tar -xvf files.tar
     > tar -xvzf files.tgz
     ```

   - To peek into a zipped (text) file:

     ```
     > gzip -cd file.gz | less
     > zcat file.zip | less
     ```

4 A variety of UNIX tools/capabilities

Many UNIX programs are small programs (tools, utilities) that can be combined to do complicated things.

1. For help on a UNIX program, including command-line utilities like *ls*, *cp*, etc.

   ```
   > man cp
   ```

2. What's the path of an executable?

   ```
   > which R
   ```

3. Tools for remotely mounting the filesystem of a remote UNIX machine/filesystem as a 'local' directory on your machine:

   - *Samba* protocol - see "How can I mount my home directory" on SCF Help Desk FAQs
   - *MacFUSE*
   - Linux:

     ```
     > cd; mkdir nersc # create a directory as a mountpoint
     > sshfs carver.nersc.gov:  /Users/paciorek/nersc # mount the remote filesystem
     > fusermount -u ~/scf # to unmount
     ```

4. Cloud storage: Dropbox and other services will mirror directories on multiple machines and on their servers

5. To do something at the UNIX command line from within R, use the `system()` function in R:

   ```
   > system("ls -al")
   ```

6. Editors

   For statistical computing, we need an editor, not a word processor, because we're going to be operating on files of code and data files, for which word processing formatting gets in the way.

   - traditional UNIX: *emacs*, *vi*
   - Windows: *WinEdt*
   - Mac: *Aquamacs Emacs, TextMate, TextEdit*
   - Be careful in Windows - file suffixes are often hidden

7. Basic emacs:

   - *emacs* has special modes for different types of files: R code files, C code files, Latex files – it's worth your time to figure out how to set this up on your machine for the kinds of files you often work on

Table 1. Helpful emacs *control sequences.*

Sequence	Result
C-x,C-c	Close the file
C-x,C-s	Save the file
C-x,C-w	Save with a new name
C-s	Search
ESC	Get out of command buffer at bottom of screen
C-a	Go to beginning of line
C-e	Go to end of line
C-k	Delete the rest of the line from cursor forward
C-space, then move to end of block	Highlight a block of text
C-w	Remove the highlighted block, putting it in the kill buffer
C-y *(*after using **C-k** or **C-w***)*	Paste from kill buffer ('y' is for 'yank')

  - For working with R, ESS (emacs speaks statistics) mode is helpful. This is built in to Aquamacs emacs. Alternatively, the Windows and Mac versions of R, as well as RStudio (available for all platforms) provide a GUI with a built-in editor.

- To open emacs in the terminal window rather than as a new window, which is handy when it's too slow (or impossible) to tunnel the graphical emacs window through ssh:

```
> emacs -nw file.txt
```

8. Files that provide info about a UNIX machine:

   - */proc/meminfo*
   - */proc/cpuinfo*
   - */etc/issue*
   - Example: how do I find out how many processors a machine has:

```
> grep processor /proc/cpuinfo
```

9. There are (free) tools in UNIX to convert files between lots of formats (pdf, ps, html, latex, jpg). This is particularly handy when preparing figures for a publication. My computing tips page lists a number of these.

The bash shell and UNIX utilities

August 26, 2012

Sections with more advanced material that are not critical for those of you just getting started with UNIX are denoted (***). However, I will ask you to write a shell function on the first problem set.

Note that it can be difficult to distinguish what is shell-specific and what is just part of UNIX. Some of the material here is not bash-specific but general to UNIX.

Reference: Newham and Rosenblatt, Learning the bash Shell, 2nd ed.

1 Shell basics

The shell is the interface between you and the UNIX operating system. When you are working in a terminal window (i.e., a window with the command line interface), you're interacting with a shell.

There are multiple shells (*sh*, *bash*, *csh*, *tcsh*, *ksh*). We'll assume usage of *bash*, as this is the default for Mac OSX and on the SCF machines and is very common for Linux.

1. What shell am I using?
   ```
   > echo $SHELL
   ```
2. To change to bash on a one-time basis:
   ```
   > bash
   ```
3. To make it your default:
   ```
   > chsh /bin/bash
   ```
   /bin/bash should be whatever the path to the bash shell is, which you can figure out using **`which bash`**

Shell commands can be saved in a file (with extension *.sh*) and this file can be executed as if it were a program. To run a shell script called *file.sh*, you would type **`./file.sh`**. Note that if you just typed **`file.sh`**, the shell will generally have trouble finding the script and recognizing that it is executable.

2 Tab completion

When working in the shell, it is often unnecessary to type out an entire command or file name, because of a feature known as tab completion. When you are entering a command or filename in the shell, you can, at any time, hit the tab key, and the shell will try to figure out how to complete the name of the command or filename you are typing. If there is only one command in the search path and you're using tab completion with the first token of a line, then the shell will display its value and the cursor will be one space past the completed name. If there are multiple commands that match the partial name, the shell will display as much as it can. In this case, hitting tab twice will display a list of choices, and redisplay the partial command line for further editing. Similar behavior with regard to filenames occurs when tab completion is used on anything other than the first token of a command.

Note that R does tab completion for objects (including functions) and filenames.

3 Command history

By using the up and down arrows, you can scroll through commands that you have entered previously. So if you want to rerun the same command, or fix a typo in a command you entered, just scroll up to it and hit enter to run it or edit the line and then hit enter.

Note that you can use emacs-like control sequences (**`C-a`**, **`C-e`**, **`C-k`**) to navigate and delete characters, just as you can at the prompt in the shell usually.

You can also rerun previous commands as follows:

```
> !-n # runs the nth previous command
> !xt # runs the last command that started with 'xt'
```

If you're not sure what command you're going to recall, you can append **`:p`** at the end of the text you type to do the recall, and the result will be printed, but not executed. For example:

```
> !xt:p
```

You can then use the up arrow key to bring back that statement for editing or execution.

You can also search for commands by doing **`C-r`** and typing a string of characters to search for in the search history. You can hit return to submit, **`C-c`** to get out, or **`ESC`** to put the result on the regular command line for editing.

4 Basic UNIX utilities

Table 1 shows some basic UNIX programs, which are sometimes referred to as filters. The general syntax for a UNIX program is

Table 1. UNIX utilities.

Name	What it does
tail	shows last few lines of a file
less	shows a file one screen at a time
cat	writes file to screen
wc	counts words and lines in a file
grep	finds patterns in files
wget	downloads files from the web
sort	sorts a file by line
nl	numbers lines in a file
diff	compares two files
uniq	removes repeated (sequential) rows
cut	extracts fields (columns) from a file

```
> command -options argument1 argument2 ...
```

For example, **`> grep -i graphics file.txt`** looks for *graphics* (argument 1) in *file.txt* (argument2) with the option *-i*, which says to ignore the case of the letters. **`> less file.txt`** simply pages through a text file (you can navigate up and down) so you can get a feel for what's in it.

UNIX programs often take options that are identified with a minus followed by a letter, followed by the specific option (adding a space before the specific option is fine). Options may also involve two dashes, e.g., **`R --no-save`**. Here's another example that tells *tail* to keep refreshing as the file changes:

```
tail -f dat.txt
```

A few more tidbits about *grep*:

```
grep ^read code.r # returns lines that start with 'read'
grep dat$ code.r # returns lines that end with 'dat'
grep 7.7 dat.txt # returns lines with two sevens separated by a single character
grep 7.*7 dat.txt # returns lines with two sevens separated by any number of characters
```

If you have a big data file and need to subset it by line (e.g., with *grep*) or by field (e.g., with *cut*), then you can do it really fast from the UNIX command line, rather than reading it with R, SAS, Perl, etc.

Much of the power of these utilities comes in piping between them (see Section 5) and using wildcards (see Section 6) to operate on groups of files. The utilities can also be used in shell scripts to do more complicated things.

Table 2. Redirection.

Syntax	What it does
`cmd > file`	sends stdout from *cmd* into *file*, overwriting *file*
`cmd >> file`	appends stdout from *cmd* to *file*
`cmd >& file`	sends stdout and stderr from *cmd* to *file*
`cmd < file`	execute *cmd* reading stdin from *file*
`cmd <infile >outfile 2>errors`	reads from *infile*, sending stdout to *file* and stderr to *errors*
`cmd1 | cmd2`	sends stdout from *cmd1* as stdin to *cmd2* (a pipe)

Note that *cmd* may include options and arguments as seen in the previous section.

5 Redirection

UNIX programs that involve input and/or output often operate by reading input from a stream known as standard input (*stdin*), and writing their results to a stream known as standard output (*stdout*). In addition, a third stream known as standard error (*stderr*) receives error messages, and other information that's not part of the program's results. In the usual interactive session, standard output and standard error default to your screen, and standard input defaults to your keyboard. You can change the place from which programs read and write through redirection. The shell provides this service, not the individual programs, so redirection will work for all programs. Table 2 shows some examples of redirection.

Operations where output from one command is used as input to another command (via the | operator) are known as pipes; they are made especially useful by the convention that many UNIX commands will accept their input through the standard input stream when no file name is provided to them.

Here's an example of finding out how many unique entries there are in the 2rd column of a data file whose fields are separated by commas:

```
cut -d',' -f2 mileage2009.csv | sort | uniq | wc
```

To see if there are any "M" values in certain fields (fixed width) of a set of files (note I did this on 22,000 files (5 Gb or so) in about 15 minutes on my old desktop; it would have taken hours to read the data into R):

```
> cut -b1,2,3,4,5,6,7,8,9,10,11,29,37,45,53,61,69,77,85,93,101,109
,117,125,133,141,149,157,165,173,181,189,197,205,213,221,229,237,
245,253,261,269 USC*.dly | grep "M" | less
```

A closely related, but subtly different, capability is offered by the use of backticks (‘). When the shell encounters a command surrounded by backticks, it runs the command and replaces the backticked expression with the output from the command; this allows something similar to a pipe, but is appropriate when a command reads its arguments directly from the command line instead of through standard input. For example, suppose we are interested in searching for the text *pdf* in the

Table 3. Wildcards.

Syntax	What it matches
?	any single character
***	zero or more characters
[$c_1c_2\ldots$]	any character in the set
[!$c_1c_2\ldots$]	anything not in the set
[$c_1 - c_2$]	anything in the range from c_1 to c_2
{string1,string2,...}	anything in the set of strings

last 4 R code files (those with suffix *.q*) that were modified in the current directory. We can find the names of the last 4 files ending in ".R" or ".r" which were modified using

```
> ls -t *.{R,r} | head -4
```

and we can search for the required pattern using *grep*. Putting these together with the backtick operator we can solve the problem using

```
> grep pdf `ls -t *.{R,r} | head -4`
```

Note that piping the output of the *ls* command into *grep* would not achieve the desired goal, since *grep* reads its filenames from the command line, not standard input.

You can also redirect output as the arguments to another program using the *xargs* utility. Here's an example:

```
> which bash | xargs chsh
```

And you can redirect output into a shell variable (see section 9) by putting the command that produces the output in parentheses and preceding with a *$*. Here's an example:

```
> files=$(ls) # NOTE - don't put any spaces around the '='
> echo $files
```

6 Wildcards in filenames

The shell will expand certain special characters to match patterns of file names, before passing those filenames on to a program. Note that the programs themselves don't know anything about wildcards; it is the shell that does the expansion, so that programs don't see the wildcards. Table 3 shows some of the special characters that the shell uses for expansion:

Here are some examples of using wildcards:

- List all files ending with a digit:

  ```
  > ls *[0-9]
  ```

- Make a copy of *filename* as *filename.old*

```
> cp filename{,.old}
```

- Remove all files beginning with *a* or *z*:

```
> rm [az]*
```

- List all the R code files with a variety of suffixes:

```
> ls *.{r,q,R}
```

The *echo* command can be used to verify that a wildcard expansion will do what you think it will:

```
> echo cp filename{,.old} # returns cp filename filename.old
```

If you want to suppress the special meaning of a wildcard in a shell command, precede it with a backslash (\). Note that this is a general rule of thumb in many similar situations when a character has a special meaning but you just want to treat it as a character.

7 Job Control

Starting a job When you run a command in a shell by simply typing its name, you are said to be running in the foreground. When a job is running in the foreground, you can't type additional commands into that shell, but there are two signals that can be sent to the running job through the keyboard. To interrupt a program running in the foreground, use **C-c**; to quit a program, use **C-**. While modern windowed systems have lessened the inconvenience of tying up a shell with foreground processes, there are some situations where running in the foreground is not adequate.

The primary need for an alternative to foreground processing arises when you wish to have jobs continue to run after you log off the computer. In cases like this you can run a program in the background by simply terminating the command with an ampersand (*&*). However, before putting a job in the background, you should consider how you will access its results, since stdout is not preserved when you log off from the computer. Thus, redirection (including redirection of *stderr*) is essential when running jobs in the background. As a simple example, suppose that you wish to run an R script, and you don't want it to terminate when you log off. (Note that this can also be done using **R CMD BATCH**, so this is primarily an illustration.)

```
> R --no-save < code.q >& code.Rout &
```

If you forget to put a job in the background when you first execute it, you can do it while it's running in the foreground in two steps. First, suspend the job using the **C-z** signal. After receiving the signal, the program will interrupt execution, but will still have access to all files and other resources. Next, issue the *bg* command, which will put the stopped job in the background.

Listing and killing jobs Since only foreground jobs will accept signals through the keyboard, if you want to terminate a background job you must first determine the unique process id (PID) for the process you wish to terminate through the use of the *ps* command. For example, to see all the R jobs running on a particular computer, you could use a command like:

```
> ps -aux | grep R # sometimes this needs to be "ps aux | grep R"
```

Among the output after the header (shown here) might appear a line that looks like this:

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
paciorek 11998 97.0 39.1 1416644 1204824 pts/16 R+ Jul27 1330:01
/usr/lib64/R/bin/exec/R
```

In this example, the *ps* output tells us that this R job has a PID of *11998*, that it has been running for 1330 minutes (!), is using 97% of CPU and 39% of memory, and that it started on July 27. You could then issue the command:

```
> kill 11998
```

or, if that doesn't work

```
> kill -9 11998
```

to terminate the job. Another useful command in this regard is *killall*, which accepts a program name instead of a process id, and will kill all instances of the named program. E.g.,

```
> killall R
```

Of course, it will only kill the jobs that belong to you, so it will not affect the jobs of other users. Note that the *ps* and *kill* commands only apply to the particular computer on which they are executed, not to the entire computer network. Thus, if you start a job on one machine, you must log back into that same machine in order to manage your job.

Monitoring jobs and memory use The *top* command also allows you to monitor the jobs on the system and in real-time. In particular, it's useful for seeing how much of the CPU and how much memory is being used, as well as figuring out a PID as an alternative to *ps*. You can also renice jobs (see below) and kill jobs from within top: just type *r* or *k*, respectively, and proceed from there.

One of the main things to watch out for is a job that is using close to 100% of memory and much less than 100% of CPU. What is generally happening is that your program has run out of memory and is using virtual memory on disk, spending most of its time writing to/from disk, sometimes called *paging* or *swapping*. If this happens, it can be a very long time, if ever, before your job finishes.

Nicing a job (IMPORTANT) The most important thing to remember when starting a job on a machine that is not your personal machine is how to be a good citizen. This often involves 'nicing' your jobs, and this is REQUIRED on the SCF machines (although the compute servers

automatically nice your jobs). Nicing a job puts it at a lower priority so that a user working at the keyboard has higher priority in using the CPU. Here's how to do it, giving the job a low priority of 19, as required by SCF:

```
> nice -19 R CMD BATCH --no-save in.R out.Rout
```

If you forget and just submit the job without nicing, you can reduce the priority by doing:

```
> renice +19 11998
```

where *11998* is the PID of your job.

On many larger UNIX cluster computers, all jobs are submitted via a job scheduler and enter a queue, which handles the issue of prioritization and jobs conflicting. Syntax varies by system and queueing software, but may look something like this for submitting an R job:

```
> bsub -q long R CMD BATCH --no-save in.R out.Rout # just an example;
this will not work on the SCF network
```

8 Aliases (***)

Aliases allow you to use an abbreviation for a command, to create new functionality or to insure that certain options are always used when you call an existing command. For example, I'm lazy and would rather type **`q`** instead of **`exit`** to terminate a shell window. You could create the alias as follow

```
> alias q="exit"
```

As another example, suppose you find the *-F* option of *ls* (which displays **/** after directories, ***** after executable files and @ after links) to be very useful. The command

```
> alias ls="ls -F"
```

will insure that the *-F* option will be used whenever you use *ls*. If you need to use the unaliased version of something for which you've created an alias, precede the name with a backslash (\). For example, to use the normal version of *ls* after you've created the alias described above, just type

```
> \ls
```

The real power of aliases is only achieved when they are automatically set up whenever you log in to the computer or open a new shell window. To achieve that goal with aliases (or any other bash shell commands), simply insert the commands in the file *.bashrc* in your home directory. See the *example.bashrc* file on bspace for some of what's in my *.bashrc* file.

9 Shell Variables (***)

We can define shell variables that will help us when writing shell scripts. Here's an example of defining a variable:

```
> name="chris"
```

The shell may not like it if you leave any spaces around the = sign. To see the value of a variable we need to precede it by *$*:

```
> echo $chris
```

You can also enclose the variable name in curly brackets, which comes in handy when we're embedding a variable within a line of code to make sure the shell knows where the variable name ends:

```
> echo ${chris}
```

There are also special shell variables called environment variables that help to control the shell's behavior. These are generally named in all caps. Type **`env`** to see them. You can create your own environment variable as follows:

```
> export NAME="chris"
```

The *export* command ensures that other shells created by the current shell (for example, to run a program) will inherit the variable. Without the export command, any shell variables that are set will only be modified within the current shell. More generally, if one wants a variable to always be accessible, one would include the definition of a variable with an export command in your *.bashrc* file.

Here's an example of modifying an environment variable:

```
> export CDPATH=.:~/research:~/teaching
```

Now if you have a subdirectory *bootstrap* in your *research* directory, you can type **`cd bootstrap`** no matter what your pwd is and it will move you to *~/research/bootstrap*. Similarly for any subdirectory within the *teaching* directory.

Here's another example of an environment variable that puts the username, hostname, and pwd in your prompt. This is handy so you know what machine you're on and where in the filesystem you are.

```
> export PS1="\u@\h:\w> "
```

For me, this is one of the most important things to put in my *.bashrc* file. The **** syntax tells bash what to put in the prompt string: *u* for username, *h* for hostname, and *w* for working directory.

10 Functions (***)

You can define your own utilities by creating a shell function. This allows you to automate things that are more complicated than you can do with an alias. Here's an example from my *.bashrc* that uses ssh to locally mount (on my desktop machine) remote filesystems for other systems I have access to:

```
function mounts(){
```

```
    sshfs carver.nersc.gov:  /accounts/gen/vis/paciorek/nersc
    sshfs hpcc.sph.harvard.edu:  /accounts/gen/vis/paciorek/hpcc

}
```

Now I just type **`> mounts`** and the shell will execute the code in the function. (Note: this could be done with an alias by just separating the two items with semicolons.)

One nice thing is that the shell automatically takes care of function arguments for you. It places the arguments given by the user into local variables in the function called (in order): *$1 $2 $3* etc. It also fills *$#* with the number of arguments given by the user. Here's an example of using arguments in a function that saves me some typing when I want to copy a file to the SCF filesystem:

```
function putscf() {
    scp $1 paciorek@bilbo.berkeley.edu:~/$2
}
```

To use this function, I just do the following to copy *unit1.pdf* from the current directory on whatever non-SCF machine I'm on to the directory *~/teaching/243* on SCF:

```
> putscf unit1.pdf teaching/243/.
```

Of course you'd want to put such functions in your *.bashrc* file.

11 If/then/else (***)

We can use if-then-else type syntax to control the flow of a shell script. For an example, see *niceR()* in the demo code file for this unit.

For more details, look in Newham&Rosenblatt or search online.

12 For loops (***)

For loops in shell scripting are primarily designed for iterating through a set of files or directories. Here's an example:

```
for file in $(ls *txt)
do
    mv $file ${file/.txt/.q}
    # this syntax replaces .txt with .q in $file
done
```

Another use of *for* loops is automating file downloads: see the demo code file. And, in my experience, *for* loops are very useful for starting a series of jobs: see the demo code file.

13 How much shell scripting should I learn?

You can do a fair amount of what you need from within R using the *system()* function. This will enable you to avoid dealing with the shell programming syntax (but you'll still need to know how to use UNIX utilities, wildcards, and pipes to be effective). Example: a fellow student in grad school programmed a tool in R to extract concert information from the web for bands appearing in her iTunes library. Not the most elegant solution, but it got the job done.

For more extensive shell programming, it's probably worth learning Python or Perl and doing it there rather than using a shell script.

Using R

October 5, 2012

References:

- Adler
- Chambers
- R intro manual on CRAN (R-intro).
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies.
- R Data Import/Export manual on CRAN (R-data).

I'm going to try to refer to R syntax as statements, where a statement is any code that is a valid, complete R expression. I'll try not to use the term *expression*, as this actually means a specific type of object within the R language.

One of my goals in our coverage of R is for us to think about why things are the way they are in R. I.e., what principles were used in creating the language.

1 Some basic functionality

I'll assume everyone knows about the following functions/functionality in R:

getwd(), setwd(), source(), pdf(), save(), save.image(), load()

- To run UNIX commands from within R, use *system()*, e.g.,

```
system("ls -al")  # knitr/Sweave doesn't seem to show the output of system()
files <- system("ls", intern = TRUE)
files[1:5]

## [1] "badCode.R"  "badCode.R~" "bash.lyx"   "bash.lyx~"  "bash.pdf"
```

- There are also a bunch of functions that will do specific queries of the filesystem, including

```
file.exists("file.txt")
list.files("~/research")
```

- To get some info on the system you're running on:

```
Sys.info()
```

- To see some of the options that control how R behaves, try the *options()* function. The *width* option changes the number of characters of width printed to the screen, while the *max.print* option prevents too much of a large object from being printed to the screen. The *digits* option changes the number of digits of numbers printed to the screen (but be careful as this can be deceptive if you then try to compare two numbers based on what you see on the screen).

```
# options() # this would print out a long list of options
options()[1:5]

## $add.smooth
## [1] TRUE
##
## $bitmapType
## [1] "cairo"
##
## $browser
## [1] "xdg-open"
##
## $browserNLdisabled
## [1] FALSE
##
## $check.bounds
## [1] FALSE

options()[c("width", "digits")]
```

```
## $width
## [1] 75
##
## $digits
## [1] 4

# options(width = 120) # often nice to have more characters on screen
options(width = 55)  # for purpose of making the pdf of this document
options(max.print = 5000)
options(digits = 3)
a <- 0.123456
b <- 0.1234561
a

## [1] 0.123

b

## [1] 0.123

a == b

## [1] FALSE
```

- Use **C-c** to interrupt execution. This will generally back out gracefully, returning you to a state as if the command had not been started. Note that if R is exceeding memory availability, there can be a long delay. This can be frustrating, particularly since a primary reason you would want to interrupt is when R runs out of memory.
- The R mailing list archives are very helpful for getting help - always search the archive before posting a question.
  - *sessionInfo()* gives information on the current R session - it's a good idea to include this information (and information on the operating system such as from *Sys.info()*) when you ask for help on a mailing list
- Any code that you wanted executed automatically when starting R can be placed in *~/.Rprofile* (or in individual *.Rprofile* files in specific directories). This could include loading pack-

ages (see below), sourcing files with user-defined functions (you can also put the function code itself in *.Rprofile*), assigning variables, and specifying options via *options()*.

2 Packages

One of the killer apps of R is the extensive collection of add-on packages on CRAN (www.cran.r-project.org) that provide much of R's functionality. To make use of a package it needs to be installed on your system (using *install.packages()* once only) and loaded into R (using *library()* every time you start R).

Some packages are *installed* by default with R and of these, some are *loaded* by default, while others require a call to *library()*. For packages I use a lot, I install them once and then load them automatically every time I start R using my *~/.Rprofile* file.

Loading packages To make a package available (loading it) and to see all the installed packages that you could load you can use *library()*.

```
library(fields)
library(help = fields)
# library() # I don't want to run this on SCF because
# so many are installed
```

Notice that some of the packages are in a system directory and some are in my home directory. Packages often depend on other packages. In general, if one package depends on another, R will load the dependency, but if the dependency is installed locally (see below), R may not find it automatically and you may have to use *library()* to load the dependency first. *.libPaths()* shows where R looks for packages on your system.

Installing packages If a package is on CRAN but not on your system, you can install it easily (usually). You don't need root permission on a machine to install a package (though sometimes you run into hassles if you are installing it just as a user, so if you have administrative privileges it may help to use them).

```
install.packages("fields", lib = "~/Rlibs")  # ~/Rlibs needs to exist!
```

You can also download the zipped source file from CRAN and install from the file; see the help page for *install.packages()*.

Accessing objects from packages The objects in a package (primarily functions, but also data) are in their own workspaces, and are accessible after you load the package using *library()*, but are not directly visible when you use *ls()*. We'll talk more about this when we talk about scope and environments. If we want to see the objects in one of the other workspaces, we can do the following:

```
search()

##  [1] ".GlobalEnv"        "package:knitr"
##  [3] "package:stats"     "package:graphics"
##  [5] "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:fields"
##  [9] "package:spam"      "package:methods"
## [11] "package:SCF"       "Autoloads"
## [13] "package:base"

# ls(pos = 7) # for the graphics package
ls(pos = 7)[1:5]  # just show the first few

## [1] "ability.cov"   "airmiles"      "AirPassengers"
## [4] "airquality"    "anscombe"
```

3 Objects

3.1 Classes of objects

Everything in R is stored as an object, each of which has a class that describes what the object contains and with standard functions that operate on objects in the class. Much of R is object-oriented, though we can write code in R that is not explicitly object-oriented. The basic classes are

- *character* vectors: these vectors of strings. Examples of individual strings are: *"Sam"*, *"0923"*, *"Sam9"*, *"Sam is"*, *"Sam\t9"*, *"Sam's the man.\nNo doubt.\n"*. Each of these is a character vector of length 1.
- *numeric* vectors (i.e., double precision real numbers)
- *integer* vectors

- *logical* (TRUE/FALSE) vectors
- *complex* vectors
- *lists*: vectors of arbitrary objects
- *factors*: vector-like objects of categorical variables with a pre-defined set of labels

  Notes:

  - Scalars are actually vectors of length 1 in R.
  - Unlike in compiled languages, objects can be created without explicitly initializing them and allocating memory.
  - Factors can be ordered - see *ordered()*.

  More complicated objects:

  - Data frames are a list of vectors of the same length, where the vectors may have different types.
  - Matrices are different from data frames in that all the elements are of a single type. Furthermore, matrices and arrays (which allow dimensions of 1, 2, 3, ...) can be thought of as vectors with information on dimensions (layout). If you pass a matrix into a function expecting a vector, it will just treat it as a vector (of concatenated columns).

  We can check on the class of an object using *class()* or with specific boolean queries: *is.numeric()*, *is.factor()*, *is.vector()*, etc.

3.2 Assignment and coercion

We assign into an object using either '=' or '<-'. A rule of thumb is that for basic assignments where you have an object name, then the assignment operator, and then some code, '=' is fine, but otherwise use '<-'.

```
out = mean(rnorm(7))   # OK
system.time(out = rnorm(10000))   # NOT OK, system.time expects its argument

## Error:  unused argument(s) (out = rnorm(10000))

system.time(out <- rnorm(10000))

##    user  system elapsed
##       0       0       0
```

Let's look at these examples to understand the distinction between '=' and '<-' when passing arguments to a function.

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x476d568>
## <environment: namespace:base>

x <- 0
y <- 0
out <- mean(x = c(3, 7))  # the usual way to pass an argument to a function
out <- mean(x <- c(3, 7))  # this is allowable, though perhaps not useful; w
out <- mean(y = c(3, 7))

## Error:  argument "x" is missing, with no default

out <- mean(y <- c(3, 7))
```

One situation in which you want to use '<-' is if it is being used as part of an argument to a function, so that R realizes you're not indicating one of the function arguments, e.g.:

```
mat <- matrix(c(1, NA, 2, 3), nrow = 2, ncol = 2)
apply(mat, 1, sum.isna <- function(vec) {
    return(sum(is.na(vec)))
})

## [1] 0 1

apply(mat, 1, sum.isna = function(vec) {
    return(sum(is.na(vec)))
})  # NOPE

## Error:  argument "FUN" is missing, with no default
```

R often treats integers as numerics, but we can force R to store values as integers:

```
vals <- c(1, 2, 3)
class(vals)

## [1] "numeric"

vals <- 1:3
class(vals)

## [1] "integer"

vals <- c(1L, 2L, 3L)
vals

## [1] 1 2 3

class(vals)

## [1] "integer"
```

We convert between classes using variants on *as()*: e.g.,

```
as.character(c(1, 2, 3))

## [1] "1" "2" "3"

as.numeric(c("1", "2.73"))

## [1] 1.00 2.73

as.factor(c("a", "b", "c"))

## [1] a b c
## Levels: a b c
```

Some common conversions are converting numbers that are being interpreted as characters into actual numbers, converting between factors and characters, and converting between logical TRUE/FALSE vectors and numeric 1/0 vectors. In some cases R will automatically do conversions behind the scenes in a smart way (or occasionally not so smart way). We'll see implicit conversion (also called coercion) when we read in characters into R using *read.table()* - strings are often automatically coerced to factors. Consider these examples of implicit coercion:

```
x <- rnorm(5)
x[3] <- "hat"  # What do you think is going to happen?
indices = c(1, 2.73)
myVec = 1:10
myVec[indices]

## [1] 1 2
```

In other languages, converting between different classes is sometimes called *casting* a variable.

3.3 Type vs. class

We've discussed vectors as the basic data structure in R, with character, integer, numeric, etc. classes. Objects also have a type, which relates to what kind of values are in the objects and how objects are stored internally in R (i.e., in C).

Let's look at Adler's Table 7.1 to see some other types.

```
a <- data.frame(x = 1:2)
class(a)

## [1] "data.frame"

typeof(a)

## [1] "list"

m <- matrix(1:4, nrow = 2)
class(m)

## [1] "matrix"

typeof(m)

## [1] "integer"
```

We've said that everything in R is an object and all objects have a class. For simple objects class and type are often closely related, but this is not the case for more complicated objects. The class describes what the object contains and standard functions associated with it. In general, you mainly need to know what class an object is rather than its type. Classes can *inherit* from other

classes; for example, the *glm* class inherits characteristics from the *lm* class. We'll see more on the details of object-oriented programming in the R programming unit.

We can create objects with our own defined class.

```
me <- list(firstname = "Chris", surname = "Paciorek")
class(me) <- "personClass"  # it turns out R already has a 'person' class de
class(me)

## [1] "personClass"

is.list(me)

## [1] TRUE

typeof(me)

## [1] "list"

typeof(me$firstname)

## [1] "character"
```

3.4 Information about objects

Some functions that give information about objects are:

```
is(me, "personClass")

## [1] TRUE

str(me)

## List of 2
##  $ firstname: chr "Chris"
##  $ surname  : chr "Paciorek"
##  - attr(*, "class")= chr "personClass"

attributes(me)
```

```
## $names
## [1] "firstname" "surname"
##
## $class
## [1] "personClass"

mat <- matrix(1:4, 2)
class(mat)

## [1] "matrix"

typeof(mat)

## [1] "integer"

length(mat)  # recall that a matrix can be thought of as a vector with dimen

## [1] 4

attributes(mat)

## $dim
## [1] 2 2

dim(mat)

## [1] 2 2
```

Attributes are information about an object attached to an object as something that looks like a named list. Attributes are often copied when operating on an object. This can lead to some weird-looking formatting:

```
x <- rnorm(10 * 365)
qs <- quantile(x, c(0.025, 0.975))
qs

##  2.5% 97.5%
## -2.00  1.88

qs[1] + 3
```

```
## 2.5%
##    1
```

Thus in an subsequent operations with *qs*, the *names* attribute will often get carried along. We can get rid of it:

```
names(qs) <- NULL
qs

## [1] -2.00  1.88
```

A common use of attributes is that rows and columns may be named in matrices and data frames, and elements in vectors:

```
row.names(mtcars)[1:6]

## [1] "Mazda RX4"         "Mazda RX4 Wag"
## [3] "Datsun 710"        "Hornet 4 Drive"
## [5] "Hornet Sportabout" "Valiant"

names(mtcars)

##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec"
##  [8] "vs"   "am"   "gear" "carb"

mat <- data.frame(x = 1:2, y = 3:4)
row.names(mat) <- c("first", "second")
mat

##        x y
## first  1 3
## second 2 4

vec <- c(first = 7, second = 1, third = 5)
vec["first"]

## first
##     7
```

3.5 The workspace

Objects exist in a workspace, which in R is called an environment.

```
# objects() # what objects are in my workspace
identical(ls(), objects())  # synonymous

## [1] TRUE

dat <- 7
dat2 <- 9
subdat <- 3
obj <- 5
objects(pattern = "^dat")

## [1] "dat"  "dat2"

rm(dat2, subdat)
rm(list = c("dat2", "subdat"))  # a bit confusing - the 'list' argument shou

## Warning:  object 'dat2' not found
## Warning:  object 'subdat' not found

rm(list = ls(pattern = "^dat"))
exists("dat")  # can be helpful when programming

## [1] FALSE

dat <- rnorm(5e+05)
object.size(dat)

## 4000040 bytes

print(object.size(dat), units = "Mb")  # this seems pretty clunky!

## 3.8 Mb

rm(list = ls())  # what does this do?
```

3.6 Missing values

The basic missing value token in R is *NA*. Most functions handle NAs as input gracefully (as should any software you write). The user can generally provide input on how to handle NAs, with omitting them being a common desire (e.g., `mean(x, na.rm = TRUE)`) One checks whether a value is an NA using *is.na()*. *NaN* indicates an operation has returned something that is not a number, such as 0/0, while *Inf* and *-Inf* indicate infinity and can be used in calculations.

```
3/Inf

## [1] 0

3 + Inf

## [1] Inf

0/0

## [1] NaN
```

To check for NA or NaN, use *is.na()* and *is.nan()*. == won't work.

```
x <- c(3, NA, 5, NaN)
is.na(x)

## [1] FALSE  TRUE FALSE  TRUE

x == NA

## [1] NA NA NA NA

x[is.na(x)] <- 0
x

## [1] 3 0 5 0
```

3.7 Some other details

Special objects There are also some special objects, which often begin with a period, like hidden files in UNIX. One is *.Last.value*, which stores the last result.

```
rnorm(10)

##  [1] -0.951 -1.468  0.702 -0.258  1.555  0.693  1.044
##  [8]  0.153 -0.515 -1.467

# .Last.value # this should return the 10 random
# normals but knitr is messing things up, commented
# out here
```

Scientific notation R uses the syntax *"xep"* to mean $x * 10^p$.

```
x <- 1e+05
log10(x)
y <- 1e+05
x <- 1e-08
```

Information about functions To get help on functions (I'm having trouble evaluating these with knitr, so just putting these in as text here):

```
?lm # or help(lm)
help.search('lm')
apropos('lm')
help('[[') # operators are functions too
args(lm)
```

Strings and quotation Working with strings and quotes (see **`?Quotes`**). Generally one uses double quotes to denote text. If we want a quotation symbol in text, we can do something like the following, either combining single and double quotes or escaping the quotation:

```
ch1 <- "Chris's\n"
ch2 <- 'He said, "hello."\n'
ch3 <- "He said, \"hello.\"\n"
```

Be careful when cutting and pasting from documents that are not text files as you may paste in something that looks like a single or double quote, but which R cannot interpret as a quote because it's some other ASCII quote character.

4 Working with data structures

4.1 Lists and dataframes

Extraction You extract from lists with "*[[*" or with "*[*"

```
x <- list(a = 1:2, b = 3:4, sam = rnorm(4))
x[[2]]  # extracts the indicated component, which can be anything, in this c

## [1] 3 4

x[c(1, 3)]  # extracts subvectors, which since it is a list, will also be a

## $a
## [1] 1 2
##
## $sam
## [1] -1.805 -1.803  0.488 -0.667
```

When working with lists, it's handy to be able to use the same function on each element of the list:

```
lapply(x, length)

## $a
## [1] 2
##
## $b
## [1] 2
##
## $sam
## [1] 4

sapply(x, length)  # returns things in a user-friendly way

##   a   b sam
##   2   2   4
```

Note that to operate on a data frame, which is a list, we'll generally want to use *lapply()* or *sapply()*, as *apply()* is really designed for working with elements that are all of the same type:

```
apply(CO2, 2, class)  # hmmm

##       Plant        Type   Treatment        conc
## "character" "character" "character" "character"
##      uptake
## "character"

sapply(CO2, class)

## $Plant
## [1] "ordered" "factor"
##
## $Type
## [1] "factor"
##
## $Treatment
## [1] "factor"
##
## $conc
## [1] "numeric"
##
## $uptake
## [1] "numeric"
```

Here's a nice trick to pull out a specific component from each element of a list. (Note the use of the additional argument(s) to *sapply()* - this can also be done in the other *apply()* variants.)

```
params <- list(a = list(mn = 7, sd = 3), b = list(mn = 6,
    sd = 1), c = list(mn = 2, sd = 1))
sapply(params, "[[", 1)

## a b c
## 7 6 2
```

Finally, we can flatten a list with *unlist()*.

```
unlist(x)

##     a1     a2     b1     b2   sam1   sam2   sam3
##  1.000  2.000  3.000  4.000 -1.805 -1.803  0.488
##   sam4
## -0.667
```

Calculations in the context of stratification We can also use an *apply()* variant to do calculations on subgroups, defined based on a factor or factors.

```
tapply(mtcars$mpg, mtcars$cyl, mean)

##    4    6    8
## 26.7 19.7 15.1

tapply(mtcars$mpg, list(mtcars$cyl, mtcars$gear), mean)

##      3    4    5
## 4 21.5 26.9 28.2
## 6 19.8 19.8 19.7
## 8 15.1   NA 15.4
```

Check out *aggregate()* and *by()* for nice wrappers to *tapply()* when working with data frames. *aggregate()* returns a data frame and works when the output of the function is univariate, while *by()* returns a list, so can return multivariate output:

```
aggregate(mtcars, list(cyl = mtcars$cyl), mean)  # this uses the function or

##   cyl  mpg cyl disp    hp drat   wt qsec    vs    am
## 1   4 26.7   4  105  82.6 4.07 2.29 19.1 0.909 0.727
## 2   6 19.7   6  183 122.3 3.59 3.12 18.0 0.571 0.429
## 3   8 15.1   8  353 209.2 3.23 4.00 16.8 0.000 0.143
##   gear carb
## 1 4.09 1.55
## 2 3.86 3.43
## 3 3.29 3.50
```

```
by(warpbreaks, warpbreaks$tension, function(x) {
    lm(breaks ~ wool, data = x)
})

## warpbreaks$tension: L
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##        44.6        -16.3
##
## ------------------------------------------
## warpbreaks$tension: M
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##       24.00         4.78
##
## ------------------------------------------
## warpbreaks$tension: H
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##       24.56        -5.78
```

In some cases you may actually need an object containing the subsets of the data, for which you can use *split()*:

```
split(mtcars, mtcars$cyl)
```

The *do.call()* function will apply a function to the elements of a list. For example, we can *rbind()* together (if compatible) the elements of a list of vectors instead of having to loop over the elements or manually type them in:

```
myList <- list(a = 1:3, b = 11:13, c = 21:23)
rbind(myList$a, myList$b, myList$c)

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
## [3,]   21   22   23

rbind(myList)

##        a         b         c
## myList Integer,3 Integer,3 Integer,3

do.call(rbind, myList)

##   [,1] [,2] [,3]
## a    1    2    3
## b   11   12   13
## c   21   22   23
```

Why couldn't we just use *rbind()* directly? Basically we're using *do.call()* to use functions that take "..." as input (i.e., functions accepting an arbitrary number of arguments) and to use the list as the input instead (i.e., to use the list elements).

4.2 Vectors and matrices

Column-major vs. row-major matrix storage Matrices in R are column-major ordered, which means they are stored by column as a vector of concatenated columns.

```
mat <- matrix(rnorm(500), nr = 50)
identical(mat[1:50], mat[, 1])

## [1] TRUE
```

```
identical(mat[1:10], mat[1, ])

## [1] FALSE

vec <- c(mat)
mat2 <- matrix(vec, nr = 50)
identical(mat, mat2)

## [1] TRUE
```

If you want to fill a matrix row-wise:

```
matrix(1:4, 2, byrow = TRUE)

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Column-major ordering is also used in Matlab and Fortran, while row-major ordering is used in C.

Recycling R will recycle the right-hand side of an assignment if the left-hand side has more elements, and will give an error or warning if there is partial recycling. Recycling is powerful, but dangerous.

```
mat <- matrix(1:8, 2)
mat[1, ] <- c(1, 2)
mat

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    1    2
## [2,]    2    4    6    8

mat[1, ] <- 1:3

## Error:  number of items to replace is not a multiple of replacement
length
```

Sequences The *rep()* and *seq()* functions are helpful for creating sequences with particular structure. Here are some examples of combining them:

```
rep(seq(1, 9, by = 2), each = 2)

##  [1] 1 1 3 3 5 5 7 7 9 9

rep(seq(1, 9, by = 2), times = 2)  # repeats the whole vector 'times' times

##  [1] 1 3 5 7 9 1 3 5 7 9

rep(seq(1, 9, by = 2), times = 1:5)  # repeats each element based on 'times

##  [1] 1 3 3 5 5 5 7 7 7 7 9 9 9 9 9
```

Sidenote: sometimes you can take a sequence generated with *rep()* and *seq()*, put it into a matrix, do some manipulations (such as transposing the matrix), and pull it out as a vector to get sequences in the order you want. For example, how could I produce *"1 3 5 7 9 3 5 7 9 5 7 9 7 9 9"* without using a loop? You can also get sequences using mathematical expressions and *round()*, *floor()*, *ceiling()*, and modulo operations. We'll play around with this more when we talk about efficiency.

You can create regular combinations of the values in two (or more) vectors as follows:

```
expand.grid(x = 1:2, y = -1:1, theta = c("lo", "hi"))

##    x  y theta
## 1  1 -1    lo
## 2  2 -1    lo
## 3  1  0    lo
## 4  2  0    lo
## 5  1  1    lo
## 6  2  1    lo
## 7  1 -1    hi
## 8  2 -1    hi
## 9  1  0    hi
## 10 2  0    hi
## 11 1  1    hi
## 12 2  1    hi
```

This is useful for setting up simulations and for creating a grid of values for image and surface plots.

Identifying elements by index You can figure out the indices of elements having a given characteristic using *which()*:

```
x <- c(1, 10, 2, 9, 3, 8)
which(x < 3)

## [1] 1 3
```

which.max() and *which.min()* have similar sort of functionality.

Vectorized subsetting We can subset vectors, matrices, and rows of data frames by index or by logical vectors.

```
set.seed(0)
vec <- rnorm(8)
mat <- matrix(rnorm(9), 3)
vec

## [1]  1.263 -0.326  1.330  1.272  0.415 -1.540 -0.929
## [8] -0.295

mat

##          [,1]   [,2]   [,3]
## [1,] -0.00577 -0.799 -0.299
## [2,]  2.40465 -1.148 -0.412
## [3,]  0.76359 -0.289  0.252

vec[vec < 0]

## [1] -0.326 -1.540 -0.929 -0.295

vec[vec < 0] <- 0
vec

## [1] 1.263 0.000 1.330 1.272 0.415 0.000 0.000 0.000
```

```
mat[mat[, 1] < 0, ]  # similarly for data frames
```

```
## [1] -0.00577 -0.79901 -0.29922
```

```
mat[mat[, 1] < 0, 2:3]  # similarly for data frames
```

```
## [1] -0.799 -0.299
```

```
mat[, mat[1, ] < 0]
```

```
##          [,1]   [,2]   [,3]
## [1,] -0.00577 -0.799 -0.299
## [2,]  2.40465 -1.148 -0.412
## [3,]  0.76359 -0.289  0.252
```

```
mat[mat[, 1] < 0, 2:3] <- 0
set.seed(0)  # so we get the same vec as we had before
vec <- rnorm(8)
wh <- which(vec < 0)
logicals <- vec < 0
logicals
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
wh
```

```
## [1] 2 6 7 8
```

```
identical(vec[wh], vec[logicals])
```

```
## [1] TRUE
```

```
vec <- c(1L, 2L, 1L)
is.integer(vec)
```

```
## [1] TRUE
```

```
vec[vec == 1L]  # in general, not safe with numeric vectors
```

```
## [1] 1 1
```

```
vec[vec != 3L]  # nor this
```

```
## [1] 1 2 1
```

Finally, we can also subset a matrix with a two-column matrix of {row,column} indices.

```
mat <- matrix(rnorm(25), 5)
rowInd <- c(1, 3, 5)
colInd <- c(1, 1, 4)
mat[cbind(rowInd, colInd)]

## [1] -0.00577  0.76359 -0.69095
```

Random sampling We can draw random samples with or without replacement using *sample()*.

Apply() The *apply()* function will apply a given function to either the rows or columns of a matrix or a set of dimensions of an array:

```
x <- matrix(1:6, nr = 2)
x

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

apply(x, 1, min)  # by row

## [1] 1 2

apply(x, 2, min)  # by column

## [1] 1 3 5

x <- array(1:24, c(2, 3, 4))
apply(x, 2, min)  # for(j in 1:3) print(min(x[ , j, ]))

## [1] 1 3 5

apply(x, c(2, 3), min)  # for(j in 1:3) {for(k in 1:4) print(min(x[ , j, k]

##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23
```

This can get confusing, but can be very powerful. Basically, I'm calculating the min for each element of the second dimension in the second-to-last example and for each pair of elements in the first and third dimensions in the final example. Caution: if the result of each subcalculation is a vector, these will be *cbind()*'ed together (recall column-major order), so if you used *apply()* on the row margin, you'll need to transpose the result.

Why use apply(), lapply(), etc.? The various *apply()* functions (apply, lapply, sapply, tapply, etc.) may be faster than a loop, but if the dominant part of the calculation lies in the time required by the function on each of the elements, then the main reason for using an *apply()* variant is code clarity.

```
n <- 5e+05
nr <- 1000
nCalcs <- n/nr
mat <- matrix(rnorm(n), nr = nr)
times <- 1:nCalcs
system.time(out1 <- apply(mat, 1, function(vec) {
    mod <- lm(vec ~ times)
    return(mod$coef[2])
}))

##    user  system elapsed
##    3.44   17.78    3.36

system.time({
    out2 <- rep(NA, nCalcs)
    for (i in 1:nCalcs) {
        out2[i] <- lm(mat[i, ] ~ times)$coef[2]
    }
})

##    user  system elapsed
##    1.80    8.25    1.46
```

4.3 Linear algebra

We'll focus on matrices here. A few helpful functions are *nrow()* and *ncol()*, which tell the dimensions of the matrix. The *row()* and *col()* functions will return matrices of the same size as the original, but filled with the row or column number of each element. So to get the upper triangle of a matrix, *X*, we can do:

```
X <- matrix(rnorm(9), 3)
X

##         [,1]   [,2]   [,3]
## [1,] -1.525 -0.514 -0.225
## [2,]  1.600 -0.953  1.804
## [3,] -0.685 -0.338  0.354

X[col(X) >= row(X)]

## [1] -1.525 -0.514 -0.953 -0.225  1.804  0.354
```

See also the *upper.tri()* and *lower.tri()* functions, as well as the *diag()* function. *diag()* is quite handy - you can extract the diagonals, assign into the diagonals, or create a diagonal matrix:

```
diag(X)

## [1] -1.525 -0.953  0.354

diag(X) <- 1
X

##         [,1]   [,2]   [,3]
## [1,]  1.000 -0.514 -0.225
## [2,]  1.600  1.000  1.804
## [3,] -0.685 -0.338  1.000

d <- diag(c(rep(1, 2), rep(2, 2)))
d

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    2    0
## [4,]    0    0    0    2
```

To transpose a matrix, use *t()*, e.g., `t(X)` .

Basic operations The basic matrix-vector operations are:

```
X %*% Y  # matrix multiplication
X * Y  # direct product
x %o% y  # outer product of vectors x, y: x times t(y)
outer(x, y)  # same thing
outer(x, y, function(x, y) cos(y)/(1 + x^2))  # evaluation of f(x,y) for al
crossprod(X, Y)  # same as but faster than t(X) %*% Y!
```

For inverses (X^{-1}) and solutions of systems of linear equations ($X^{-1}y$):

```
solve(X)  # inverse of X
solve(X, y)  # (inverse of X) %*% y
```

Note that if all you need to do is solve the linear system, you should never explicitly find the inverse, UNLESS you need the actual matrix, e.g., to get a covariance matrix of parameters.

Otherwise, to find many solutions, all with the same matrix, X, you can use *solve()* with the second argument being a matrix with each column a different 'y' vector for which you want the solution.

solve() is an example of using a matrix decomposition to solve a system of equations (in particular the LU decomposition). We'll defer matrix decompositions (LU, Cholesky, eigendecomposition, SVD, QR) until the numerical linear algebra unit.

5 Functions

Functions are at the heart of R. In general, you should try to have functions be self-contained - operating only on arguments provided to them, and producing no side effects, though in some cases there are good reasons for making an exception.

Functions that are not implemented internally in R (i.e., user-defined functions) are also referred to offically as *closures* (this is their type) - this terminology sometimes comes up in error messages.

5.1 Inputs

Arguments can be specifed in the correct order, or given out of order by specifying *name* = *value*. In general the more important arguments are specified first. You can see the arguments and defaults

for a function using *args()*:

```
args(lm)

## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

Functions may have unspecified arguments, which is designated using '...'. Unspecified arguments occurring at the beginning of the argument list are generally a collection of like objects that will be manipulated (consider *paste()*, *c()*, and *rbind()*), while unspecified arguments occurring at the end are often optional arguments (consider *plot()*). These optional arguments are sometimes passed along to a function within the function. For example, here's my own wrapper for plotting, where any additional arguments specified by the user will get passed along to plot:

```
pplot <- function(x, y, pch = 16, cex = 0.4, ...) {
    plot(x, y, pch = pch, cex = cex, ...)
}
```

If you want to manipulate what the user passed in as the ... args, rather than just passing them along, you can extract them (the following code would be used within a function to which '...' is an argument:

```
myFun <- function(...) {
    print(..2)
    args <- list(...)
    print(args[[2]])
}
myFun(1, 3, 5, 7)

## [1] 3
## [1] 3
```

You can check if an argument is missing with *missing()*. Arguments can also have default values, which may be *NULL*. If you are writing a function and designate the default as *argname = NULL*, you can check whether the user provided anything using `is.null(argname)`. The default values can also relate to other arguments. As an example, consider *dgamma()*:

```
args(dgamma)

## function (x, shape, rate = 1, scale = 1/rate, log = FALSE)
## NULL
```

Functions can be passed in as arguments (e.g., see the variants of *apply()*). Note that one does not need to pass in a named function - you can create the function on the spot - this is called an *anonymous function*:

```
mat <- matrix(1:9, 3)
apply(mat, 2, function(vec) vec - vec[1])

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    1    1    1
## [3,]    2    2    2

apply(mat, 1, function(vec) vec - vec[1])  # explain why the result is trans

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    3    3    3
## [3,]    6    6    6
```

We can see the arguments using *args()* and extract the arguments using *formals()*. *formals()* can be helpful if you need to manipulate the arguments.

```
f <- function(x, y = 2, z = 3/y) {
    x + y + z
}
args(f)

## function (x, y = 2, z = 3/y)
## NULL

formals(f)

## $x
##
```

```
##
## $y
## [1] 2
##
## $z
## 3/y

class(formals(f))

## [1] "pairlist"
```

match.call() will show the user-suppled arguments explicitly matched to named arguments.

```
match.call(mean, quote(mean(y, na.rm = TRUE)))  # what do you think quote do

## mean(x = y, na.rm = TRUE)
```

Pass by value vs. pass by reference Note that R makes a copy of all objects that are arguments to a function, with the copy residing in the frame (the environment) of the function (we'll see more about frames just below). This is a case of *pass by value*. In other languages it is also possible to *pass by reference*, in which case, changes to the object made within the function affect the value of the argument in the calling environment. R's designers chose not to allow pass by reference because they didn't like the idea that a function could have the side effect of changing an object. However, passing by reference can sometimes be very helpful, and we'll see ways of passing by reference in the R programming unit.

An important exception is *par()*. If you change graphics parameters by calling *par()* in a user-defined function, they are changed permanently. One trick is as follows:

```
f <- function() {
    oldpar <- par()
    par(cex = 2)
    # body of code
    par() <- oldpar
}
```

Note that changing graphics parameters within a specific plotting function - e.g., `plot(x, y, pch = '+')`, doesn't change things except for that particular plot.

5.2 Outputs

`return(x)` will specify x as the output of the function. By default, if *return()* is not specified, the output is the result of the last evaluated statement. *return()* can occur anywhere in the function, and allows the function to exit as soon as it is done. *invisible(x)* will return x and the result can be assigned in the calling environment but it will not be printed if not assigned:

```
f <- function(x) {
    invisible(x^2)
}
f(3)
a <- f(3)
a

## [1] 9
```

A function can only return a single object (unlike Matlab, e.g.), but of course we can tack things together as a list and return that, as with *lm()* and many other functions.

```
mod <- lm(mpg ~ cyl, data = mtcars)
class(mod)

## [1] "lm"

is.list(mod)

## [1] TRUE
```

5.3 Variable scope

To consider variable scope, we need to define the terms *environment* and *frame*.

Environments and frames are closely related. An environment is a collection of named objects (a frame), with a pointer to the 'enclosing environment', i.e., the next environment to look for something in, also called the parent.

Variables in the parent environment (the environment in which a function is defined, also called the enclosing environment) are available within a function. This is the analog of *global variables* in other languages. Note that the parent/enclosing environment is NOT the environment from which the function was called.

Be careful when using variables from the parent environment as the value of that variable in the parent environment may well not be what you expect it to be. In general it's bad practice to use variables that taken from environments outside that of a function, but it some cases it can be useful.

```
x <- 3
f <- function() {
    x <- x^2
    print(x)
}
f(x)
x  # what do you expect?
f <- function() {
    assign("x", x^2, env = .GlobalEnv)
}  # careful, this could be dangerous as a variable is changed as a side ef
```

Here are some examples to illustrate scope:

```
x <- 3
f <- function() {
    f2 <- function() {
        print(x)
    }
    f2()
}
f()  # what will happen?
f <- function() {
    f2 <- function() {
        print(x)
    }
    x <- 7
    f2()
}
f()  # what will happen?
f2 <- function() print(x)
f <- function() {
    x <- 7
```

```
    f2()
}
f()  # what will happen?
```

Here's a somewhat tricky example:

```
y <- 100
f <- function() {
    y <- 10
    g <- function(x) x + y
    return(g)
}
h <- f()
h(3)

## [1] 13
```

Let's work through this:

1. What is the enclosing environment of the function *g()*?
2. What does *g()* use for *y*?
3. When *f()* finishes, does its environment disappear? What would happen if it did?
4. What is the enclosing environment of *h()*?

Where are arguments evaluated? User-supplied arguments are evaluated in the calling frame, while default arguments are evaluated in the frame of the function:

```
z <- 3
x <- 100
f <- function(x, y = x * 3) {
    x + y
}
f(z * 5)

## [1] 60
```

Here, when *f()* is called, *z* is evaluated in the calling frame and assigned to *x* in the frame of the function, while *y = x*3* is evaluated in the frame of the function.

6 Environments and frames

6.1 Frames and the call stack

R keeps track of the call stack, which is the set of nested calls to functions. The stack operates like a stack of cafeteria trays - when a function is called, it is added to the stack (pushed) and when it finishes, it is removed (popped). There are a bunch of functions that let us query what frames are on the stack and access objects in particular frames of interest. This gives us the ability to work with objects in the environment(s) from which a function was called.

sys.nframe() returns the number of the current frame and *sys.parent()* the number of the parent. Careful: here, *parent* refers to the parent in terms of the call stack, not the enclosing environment. I won't print the results here because *knitr* messes up the frame counting somehow.

```
sys.nframe()
f <- function() cat("Frame number is ", sys.nframe(), "; parent is ",
    sys.parent(), ".\n", sep = "")
f()
f2 <- function() f()
f2()
```

Now let's look at some code that gets more information about the call stack and the frames involved using *sys.status()*, *sys.calls()*, *sys.parents()* and *sys.frames()*.

```
# exploring functions that give us information the
# frames in the stack
g <- function(y) {
    gg <- function() {
        # this gives us the information from sys.calls(),
        # sys.parents() and sys.frames() as one object
        print(sys.status())
    }
    if (y > 0)
        g(y - 1) else gg()
}
g(3)
```

If you're interested in parsing a somewhat complicated example of frames in action, Adler provides a user-defined timing function that evaluates statements in the calling frame.

6.2 Environments and the search path

When R goes looking for an object (in the form of a symbol), it starts in the current environment (e.g., the frame/environment of a function) and then runs up through the enclosing environments, until it reaches the global environment, which is where R starts when you start it (it actually continues further up; see below). In general, these are not the frames on the stack.

By default objects are created in the global environment, *.GlobalEnv*. As we've seen, the environment within a function call has as its enclosing environment the environment where the function was defined (not the environment from which it was called), and this is next place that is searched if an object can't be found in the frame of the function call. This is called lexical scoping (and differs from *S-plus*). As an example, if an object couldn't be found within the environment of an *lm()* function call, R would first look in the environment (also called the namespace) of the stats package, then in packages imported by the stats package, then the base package, and then the global environment.

If R can't find the object when reaching the global environment, it runs through the search path, which you can see with *search()*. The search path is a set of additional environments. Generally packages are created with namespaces, i.e., each has its own environment, as we see based on *search()*. Data frames or list that you attach using *attach()* generally are placed just after the global environment.

```
search()
```

```
##  [1] ".GlobalEnv"        "package:knitr"
##  [3] "package:stats"     "package:graphics"
##  [5] "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:fields"
##  [9] "package:spam"      "package:methods"
## [11] "package:SCF"       "Autoloads"
## [13] "package:base"
```

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "/server/linux/lib/R/2.15/x86_64/site-library/knitr"
##  [3] "/usr/lib/R/library/stats"
##  [4] "/usr/lib/R/library/graphics"
##  [5] "/usr/lib/R/library/grDevices"
##  [6] "/usr/lib/R/library/utils"
```

```
##  [7] "/usr/lib/R/library/datasets"
##  [8] "/server/linux/lib/R/2.15/x86_64/site-library/fields"
##  [9] "/server/linux/lib/R/2.15/x86_64/site-library/spam"
## [10] "/usr/lib/R/library/methods"
## [11] "/server/linux/lib/R/2.15/x86_64/site-library/SCF"
## [12] "Autoloads"
## [13] "/usr/lib/R/library/base"
```

We can also see the nestedness of environments using the following code, using *environmentName()*, which prints out a nice-looking version of the environment name.

```
x <- .GlobalEnv
parent.env(x)

## <environment: package:knitr>
## attr(,"name")
## [1] "package:knitr"
## attr(,"path")
## [1] "/server/linux/lib/R/2.15/x86_64/site-library/knitr"

while (environmentName(x) != environmentName(emptyenv())) {
    print(environmentName(parent.env(x)))
    x <- parent.env(x)
}

## [1] "package:knitr"
## [1] "package:stats"
## [1] "package:graphics"
## [1] "package:grDevices"
## [1] "package:utils"
## [1] "package:datasets"
## [1] "package:fields"
## [1] "package:spam"
## [1] "package:methods"
## [1] "package:SCF"
## [1] "Autoloads"
## [1] "base"
## [1] "R_EmptyEnv"
```

Note that eventually the global environment and the environments of the packages are nested within the base environment (of the base package) and the empty environment. Note that here *parent* **is** referring to the enclosing environment.

We can look at the objects of an environment as follows:

```
ls(pos = 7)[1:5]  # what does this do?

## [1] "ability.cov"   "airmiles"      "AirPassengers"
## [4] "airquality"    "anscombe"

ls("package:graphics")[1:5]

## [1] "abline"    "arrows"    "assocplot" "axis"
## [5] "Axis"

environment(lm)

## <environment: namespace:stats>
```

We can retrieve and assign objects in a particular environment as follows:

```
lm <- function() {
    return(NULL)
}  # this seems dangerous but isn't
x <- 1:3
y <- rnorm(3)
mod <- lm(y ~ x)

## Error:  unused argument(s) (y ~ x)

mod <- get("lm", pos = "package:stats")(y ~ x)
mod <- stats::lm(y ~ x)  # an alternative
```

Note that our (bogus) *lm()* function masks but does not overwrite the default function. If we remove ours, then the default one is still there.

6.3 with() and within()

with() provides a clean way to use a function (or any R code, specified as R statements enclosed within {}, unless you are evaluating a single expression as in the demo here) within the context of

a data frame (or an environment). within() is similar, evaluating within the context of a data frame or a list, but it allows you to modify the data frame (or list) and returns the result.

```
with(mtcars, cyl * mpg)

##  [1] 126.0 126.0  91.2 128.4 149.6 108.6 114.4  97.6
##  [9]  91.2 115.2 106.8 131.2 138.4 121.6  83.2  83.2
## [17] 117.6 129.6 121.6 135.6  86.0 124.0 121.6 106.4
## [25] 153.6 109.2 104.0 121.6 126.4 118.2 120.0  85.6

new.mtcars <- within(mtcars, crazy <- cyl * mpg)
names(new.mtcars)

##  [1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"
##  [7] "qsec"  "vs"    "am"    "gear"  "carb"  "crazy"
```

7 Flow control and logical operations

7.1 Logical operators

Everyone is probably familiar with the comparison operators, <, <=, >, >=, ==, !=. Logical operators are slightly trickier:

Logical operators for subsetting & and | are the "AND" and "OR" operators used when subsetting - they act in a vectorized way:

```
x <- rnorm(10)
x[x > 1 | x < -1]

## [1] -1.35 -2.07 -1.34 -1.45

x <- 1:10
y <- c(rep(10, 9), NA)
x > 5 | y > 5  # note that TRUE | NA evaluates to TRUE

##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Logical operators and if statements *&&* and || use only the first element of a vector and also proceed from left to right, returning the result as soon as possible and then ignoring the remaining comparisons (this can be handy because in some cases the second condition may give an error if the first condition is not passed). They are used in flow control (i.e., with *if* statements). Let's consider how the single and double operators differ:

```
a <- 7
b <- NULL
a < 8 | b > 3

## logical(0)

a < 8 || b > 3

## [1] TRUE

a <- c(0, 3)
b <- c(4, 2)
if (a < 7 & b < 7) print("this is buggy code")

## Warning:  the condition has length > 1 and only the first element
will be used

## [1] "this is buggy code"

if (a < 7 && b < 7) print("this is buggy code too, but runs w/o warnings")

## [1] "this is buggy code too, but runs w/o warnings"

if (a[1] < 7 && b[1] < 7) print("this code is correct and the condition is

## [1] "this code is correct and the condition is TRUE"
```

You can use ! to indicate negation:

```
a <- 7
b <- 5
!(a < 8 && b < 6)

## [1] FALSE
```

7.2 If statements

If statements are at the core of programming. In R, the syntax is **`if(condition) statement else other_statement`**, e.g.,

```
x <- 5
if (x > 7) {
    x <- x + 3
} else {
    x <- x - 3
}
```

When one of the statements is a single statement, you don't need the curly braces around that statement.

```
if (x > 7) x <- x + 3 else x <- x - 3
```

An extension of *if* looks like:

```
x <- -3
if (x > 7) {
    x <- x + 3
    print(x)
} else if (x > 4) {
    x <- x + 1
    print(x)
} else if (x > 0) {
    x <- x - 3
    print(x)
} else {
    x <- x - 7
    print(x)
}

## [1] -10
```

Finally, be careful that *else* should not start its own line, unless it is preceded by a closing brace on the same line. Why?

```
if(x > 7) {
statement1 } # what happens at this point?
else{ # what happens now?
statement2
}
```

There's also the *ifelse()* function, which operates in a vectorized fashion:

```
x <- rnorm(6)
truncx <- ifelse(x > 0, x, 0)
truncx

## [1] 0.769 1.698 0.000 0.000 0.590 0.000
```

Common bugs in the condition of an if statement include the following:

1. Only the first element of *condition* is evaluated. You should be careful that *condition* is a single logical value and does not evaluate to a vector as this would generally be a bug. [see p. 152 of Chambers]

2. Use *identical()* or *all.equal()* rather than "==" to ensure that you deal properly with vectors and always get a single logical value back. We'll talk more about issues that can arise when comparing decimal numbers on a computer later in the course.

3. If *condition* includes some R code, it can fail and produce something that is neither TRUE nor FALSE. Defensive programming practice is to check the condition for validity.

```
vals <- c(1, 2, NA)
eps <- 1e-09
# now pretend vals comes from some other chunk of code
# that we don't control
if (min(vals) > eps) {
    print(vals)
}  # not good practice

## Error:  missing value where TRUE/FALSE needed

minval <- min(vals)
if (!is.na(minval) && minval > eps) {
    print(vals)
}  # better practice
```

7.3 switch()

switch() is handy for choosing amongst multiple outcomes depending on an input, avoiding a long set of if-else syntax. The first argument is a statement that determines what choice is made and the second is a list of the outcomes, in order or by name:

```
x <- 2; y <- 10
switch(x, log(y), sqrt(y), y)

## [1] 3.16

center <- function(x, type){
switch(type,
mean = mean(x), # make sure to use = and not <-
median = median(x),
trimmed = mean(x, trim = .1))
}
x <- rgamma(100, 1)
center(x, 'median')

## [1] 0.625

center(x, 'mean')

## [1] 0.771
```

7.4 Loops

Loops are at the core of programming in other functional languages, but in R, we often try to avoid them as they're often (but not always) slow. In many cases looping can be avoided by using vectorized calculations, versions of *apply()*, and other tricks. But sometimes they're unavoidable and for quick and dirty coding and small problems, they're fine. And in some cases they may be faster than other alternatives. One case we've already seen is that in working with lists they may be faster than their *lapply*-style counterpart, though often they will not be.

The workhorse loop is the *for* loop, which as the syntax: **`for(var in sequence) statement`**, where, as with the *if* statement, we need curly braces around the body of the loop if it contains more than one valid R statement:

```
nIts <- 500
means <- rep(NA, nIts)
for (it in 1:nIts) {
    means[it] <- mean(rnorm(100))
    if (identical(it%%100, 0))
        cat("Iteration", it, date(), "\n")
}

## Iteration 100 Fri Oct  5 08:08:40 2012
## Iteration 200 Fri Oct  5 08:08:40 2012
## Iteration 300 Fri Oct  5 08:08:40 2012
## Iteration 400 Fri Oct  5 08:08:40 2012
## Iteration 500 Fri Oct  5 08:08:40 2012
```

Challenge: how do I do this much faster?

You can also loop over a non-numeric vector of values.

```
for (state in c("Ohio", "Iowa", "Georgia")) print(state.x77[row.names(state
    state, "Income"])

## [1] 4561
## [1] 4628
## [1] 4091
```

Challenge: how can I do this faster?

Note that to print to the screen in a loop you explicitly need to use *print()* or *cat()*; just writing the name of the object will not work. This is similar to *if* statements and functions.

```
for (i in 1:10) i
```

You can use the commands *break* (to end the looping) and *next* (to go to the next iteration) to control the flow:

```
for (i in 1:10) {
    if (i == 5)
        break
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4

for (i in 1:5) {
    if (i == 2)
        next
    print(i)
}

## [1] 1
## [1] 3
## [1] 4
## [1] 5
```

while loops are used less frequently, but can be handy: **while(condition) statement**, e.g. in optimization. See p. 59 of Venables and Ripley, 4th ed., whose code I've included in the demo code file.

A common cause of bugs in for loops is when the range ends at zero or a missing value:

```
mat <- matrix(1:4, 2)
submat <- mat[mat[1, ] > 5]
for (i in 1:nrow(submat)) print(i)

## Error:  argument of length 0
```

8 Formulas

Formulas were introduced into R to specify linear models, but are now used more generally.

Here are some examples of formulas in R, used to specify a model structure:

Additive model:

```
y ~ x1 + x2 + x3
```

Additive model without the intercept:

```
y ~ x1 + x2 + x3 -1
```

All the other variables in the data frame are used as covariates:

`y ~ .`

All possible interactions:

`y ~ x1 * x2 * x3`

Only specified interactions (in this case *x1* by *x2*) (of course, you'd rarely want to fit this without *x2*):

`y ~ x1 + x3 + x1:x2`

Protecting arithmetic expressions:

`y ~ x1 + I(x1^2) + I(x1^3)`

Using functions of variables

`y ~ x1 + log(x2) + sin(x3)`

In some contexts, such as *lattice* package graphics, the "|" indicates conditioning, so **`y ~ x | z`** would mean to plot *y* on *x* within groups of *z*. In the context of lme-related packages (e.g., *lme4*, *nlme*, etc.), variables after "|" are *grouping* variables (e.g., if you have a random effect for each hospital, hospital would be the grouping variable) and multiple grouping variables are separated by "/".

We can manipulate formulae as objects, allowing automation. Consider how this sort of thing could be used to write code for automated model selection.

```
resp <- "y ~"
covTerms <- "x1"
for (i in 2:5) {
    covTerms <- paste(covTerms, "+ x", i, sep = "")
}
form <- as.formula(paste(resp, covTerms, sep = ""))
# lm(form, data = dat)
form

## y ~ x1 + x2 + x3 + x4 + x5

class(form)

## [1] "formula"
```

The for loop is a bit clunky/inefficient - let's do better:

```
resp <- "y ~"
covTerms <- paste("x", 1:5, sep = "", collapse = " + ")
form <- as.formula(paste(resp, covTerms))
```

```
form

## y ~ x1 + x2 + x3 + x4 + x5

# lm(form, data = dat)
```

Standard arguments in model fitting functions, in addition to the formula are *weights*, *data* (indicating the data frame in which to interpret the variable names), *subset* (for using a subset of data), and *na.action*. Note that the default *na.action* in R is set in *options()$na.action* and is *na.omit*, so be wary in fitting models in that you are dropping cases with NAs and may not be aware of it.

There is some more specialized syntax given in *R-intro.pdf* on CRAN.

9 Data storage and formats (outside R)

At this point, we're going to turn to reading data into R and manipulating text, including regular expressions. We'll focus on doing these manipulations in R, but the concepts involved in reading in data, database manipulations, and regular expressions are common to other languages, so familarity with these in R should allow you to pick up other tools more easily. The main downside to working with datasets in R is that the entire dataset resides in memory, so R is not so good for dealing with very large datasets. More on alternatives in a bit. Another common frustration is controlling how the variables are interpreted (numeric, character, factor) when reading data into a data frame.

R has the capability to read in a wide variety of file formats. Let's get a feel for some of the common ones.

1. Flat text files (ASCII files): data are often provided as simple text files. Often one has one record or observation per row and each column or field is a different variable or type of information about the record. Such files can either have a fixed number of characters in each field (fixed width format) or a special character (a delimiter) that separates the fields in each row. Common delimiters are tabs, commas, one or more spaces, and the pipe (|). Common file extensions are *.txt* and *.csv*. Metadata (information about the data) is often stored in a separate file. I like CSV files but if you have files where the data contain commas, other delimiters can be good. Text can be put in quotes in CSV files. This was difficult to deal with in the shell in PS1, but *read.table()* in R handles this situation. A couple more intricate details:

    (a) Sometimes if you have a text file created in Windows, the line endings are coded differently than in UNIX (\n\r in Windows vs \n in UNIX). There are UNIX utilities

(*fromdos* in Ubuntu, including the SCF Linux machines and *dos2unix* in other Linux distributions) that can do the necessary conversion. If you see *^M* at the end of the lines in a file, that's the tool you need. Alternatively, if you open a UNIX file in Windows, it may treat all the lines as a single line. You can fix this with *todos* or *unix2dos*.

(b) Another difficulty can be dealing with the text encoding (the mapping of individual characters (including tabs, returns, etc.) to a set of numeric codes) when the file is not simply an ASCII file (i.e., contains non-ASCII characters - stuff that's not on a US-style keyboard). There are a variety of different encodings for text files (ASCII is the most basic encoding, and allows for 2^7 different characters; others allow for 2^{16}), with different ones common on different operating systems. The UNIX utility *file*, e.g. `file tmp.txt` can help provide some information. *read.table()* in R takes arguments *fileEncoding* and *encoding* that address this issue. The UNIX utility *iconv* and the R function *iconv()* can help with conversions.

```
text <- "_Melhore sua seguran\xe7a_"
iconv(text, from = "latin1", to = "UTF-8")
iconv(text, from = "latin1", to = "ASCII", sub = "???")
```

2. In some contexts, such as textual data and bioinformatics data, the data may in a text file with one piece of information per row, but without meaningful columns/fields.

3. In scientific contexts, netCDF (*.nc*) (and the related HDF5) are popular format for gridded data that allows for highly-efficient storage and contains the metadata within the file. The basic structure of a netCDF file is that each variable is an array with multiple dimensions (e.g., latitude, longitude, and time), and one can also extract the values of and metadata about each dimension. The *ncdf* package in R nicely handles working with netCDF files. These are examples of a binary format, which is not (easily) human readable but can be more space-efficient and faster to work with (because they can allow random access into the data rather than requiring sequential reading).

4. Data may also be in the form of XML or HTML files. We won't deal with these in 243, except to the extent that they come up in a problem set.

5. Data may already be in a database or in the data storage of another statistical package (*Stata*, *SAS*, *SPSS*, etc.). The *foreign* package in R has excellent capabilities for importing Stata (*read.dta()*), SPSS (*read.spss()*), SAS (*read.ssd()* and, for XPORT files, *read.xport()*), and dbf (a common database format) (*read.dbf()*), among others.

6. For Excel, there are capabilities to read an Excel file (see the *XLConnect* package among others), but you can also just go into Excel and export as a CSV file or the like and then read that into R. In general, it's best not to pass around data files as Excel or other spreadsheet format files because (1) Excel is proprietary, so someone may not have Excel and the format is subject to change, (2) Excel imposes limits on the number of rows, (3) one can easily manipulate text files such as CSV using UNIX tools, but this is not possible with an Excel file, (4) Excel files often have more than one sheet, graphs, macros, etc., so they're not a data storage format per se.

10 Reading data from text files into R

We can use *read.fwf()* to read from a fixed width text file into a data frame. *read.table()* is probably the most commonly-used function for reading in data, it reads in delimited files (*read.csv()* and *read.delim()* are special cases of *read.table()*). The key arguments are the delimiter (the *sep* argument) and whether the file contains a header, a line with the variable names.

The most difficult part of reading in such files can be dealing with how R determines the classes of the fields that are read in. There are a number of arguments to *read.table()* and *read.fwf()* that allow the user to control the classes. One difficulty is that character and numeric fields are sometimes read in as factors. Basically *read.table()* tries to read fields in as numeric and if it finds non-numeric and non-NA values, it reads in as a factor. This can be annoying.

Let's work through a couple examples. First let's look at the arguments to *read.table()*. Note that *sep=""* separates on any amount of white space.

```
setwd("~/Desktop/243/data")
dat <- read.table("RTAData.csv", sep = ",", head = TRUE)
lapply(dat, class)
levels(dat[, 2])
dat2 <- read.table("RTAData.csv", sep = ",", head = TRUE,
    na.strings = c("NA", "x"), stringsAsFactors = FALSE)
unique(dat2[, 2])
# hmmm, what happened to the blank values this time?
which(dat[, 2] == "")
dat2[which(dat[, 2] == "")[1], ]  # deconstruct it!
sequ <- read.table("hivSequ.csv", sep = ",", header = TRUE,
    colClasses = c("integer", "integer", "character", "character",
        "numeric", "integer"))
```

```
# let's make sure the coercion worked - sometimes R is
# obstinant
lapply(sequ, class)   # this makes use of the fact that a data frame is a lis
```

The basic function *scan()* simply reads everything in, ignoring lines, which works well and very quickly if you are reading in a numeric vector or matrix. *scan()* is also useful if your file is free format - i.e., if it's not one line per observation, but just all the data one value after another; in this case you can use *scan()* to read it in and then format the resulting character or numeric vector as a matrix with as many columns as fields in the dataset. Remember that the default is to fill the matrix by column.

If the file is not nicely arranged by field (e.g., if it has ragged lines), we'll need to do some more work. *readLines()* will read in each line into a separate character vector, after which we can process the lines using text manipulation.

```
dat <- readLines("~/Desktop/243/data/precip.txt")
id <- as.factor(substring(dat, 4, 11))
year <- substring(dat, 17, 20)
class(year)

## [1] "character"

year <- as.integer(substring(dat, 18, 21))
month <- as.integer(substring(dat, 22, 23))
nvalues <- as.integer(substring(dat, 28, 30))
```

R allows you to read in not just from a file but from a more general construct called a *connection*. Here are some examples of connections:

```
dat <- readLines(pipe("ls -al"))
dat <- read.table(pipe("unzip dat.zip"))
dat <- read.csv(gzfile("dat.csv.gz"))
dat <- readLines("http://www.stat.berkeley.edu/~paciorek/index.html")
```

If a file is large, we may want to read it in in chunks (of lines), do some computations to reduce the size of things, and iterate. *read.table()*, *read.fwf()* and *readLines()* all have the arguments that let you read in a fixed number of lines. To read-on-the-fly in blocks, we need to first establish the connection and then read from it sequentially.

```
con <- file("~/Desktop/243/data/precip.txt", "r")
class(con)
blockSize <- 1000  # obviously this would be large in any real application
nLines <- 3e+05
for (i in 1:ceiling(nLines/blockSize)) {
    lines <- readLines(con, n = blockSize)
    # manipulate the lines and store the key stuff
}
close(con)
```

One cool trick that can come in handy is to create a *text connection*. This lets you 'read' from an R character vector as if it were a text file and could be handy for processing text. For example, you could then use *read.fwf()* applied to *con*.

```
dat <- readLines("~/Desktop/243/data/precip.txt")
con <- textConnection(dat[1], "r")
read.fwf(con, c(3, 8, 4, 2, 4, 2))

##    V1      V2   V3 V4   V5 V6
## 1 DLY 1000807 PRCP HI 2010  2
```

We can create connections for writing output too. Just make sure to open the connection first.

Be careful with the directory separator in Windows files: you can either do *"C:\\mydir\\file.txt"* or *"C:/mydir/file.txt"*, but not *"C:\mydir\file.txt"*. [I think; I haven't checked this, so a Windows user should correct me if I'm wrong.]

11 Text manipulations and regular expressions

Text manipulations in R have a number of things in common with Perl, Python and UNIX, as many of these evolved from UNIX. When I use the term *string* here, I'll be refering to any sequence of characters that may include numbers, white space, and special characters, rather than to the character class of R objects. The string or strings will generally be stored as R character vectors.

11.1 Basic text manipulation

A few of the basic R functions for manipulating strings are *paste()*, *strsplit()*, and *substring()*. *paste()* and *strsplit()* are basically inverses of each other: *paste()* concatenates together an arbitrary

set of strings (or a vector, if using the *collapse* argument) with a user-specified separator character, while *strsplit()* splits apart based on a delimiter/separator. *substring()* splits apart the elements of a character vector based on fixed widths. Note that all of these operate in a vectorized fashion.

```
out <- paste("My", "name", "is", "Chris", ".", sep = " ")
paste(c("My", "name", "is", "Chris", "."), collapse = " ")

## [1] "My name is Chris ."

strsplit(out, split = " ")

## [[1]]
## [1] "My"    "name"  "is"    "Chris" "."
```

nchar() tells the number of characters in a string.

To identify particular subsequences in strings, there are several related R functions. *grep()* will look for a specified string within an R character vector and report back indices identifying the elements of the vector in which the string was found in (using the *fixed=TRUE* argument ensures that regular expressions are NOT used). *gregexpr()* will indicate the position in each string that the specified string is found (use *regexpr()* if you only want the first occurrence). *gsub()* can be used to replace a specified string with a replacement string (use *sub()* if you only want to replace only the first occurrence).

```
vars <- c("P", "HCA24", "SOH02")
substring(vars, 2, 3)

## [1] ""   "CA" "OH"

vars <- c("date98", "size98", "x98weights98", "sdfsd")
grep("98", vars)

## [1] 1 2 3

gregexpr("98", vars)

## [[1]]
## [1] 5
## attr(,"match.length")
## [1] 2
```

```
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] 5
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1]  2 11
## attr(,"match.length")
## [1] 2 2
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
```

```
gsub("98", "04", vars)
```

```
## [1] "date04"       "size04"       "x04weights04"
## [4] "sdfsd"
```

11.2 Regular expressions (regexp)

Overview and core syntax The *grep()*, *gregexpr()* and *gsub()* functions are more powerful when used with regular expressions. Regular expressions are a domain-specific language for finding patterns and are one of the key functionalities in scripting languages such as Perl and Python. Duncan Temple Lang (UC Davis Stats) has written a nice tutorial that I've posted on bspace (*regexprLang.pdf*) or check out Sections 9.9 and 11 of Murrell. We'll just cover their use in R, but once

you know that, it would be easy to use them elsewhere. What I describe here is the "extended regular expression" syntax (POSIX 1003.2), but with the argument *Perl=TRUE*, you can get Perl-style regular expressions. At the level we'll consider them, the syntax is quite similar.

The basic idea of regular expressions is that they allow us to find matches of strings or patterns in strings, as well as do substitution. Regular expressions are good for tasks such as:

- extracting pieces of text - for example finding all the links in an html document;
- creating variables from information found in text;
- cleaning and transforming text into a uniform format;
- mining text by treating documents as data; and
- scraping the web for data.

Regular expressions are constructed from three things:

Literal characters are matched only by the characters themselves,

Character classes are matched by any single member in the class, and

Modifiers operate on either of the above or combinations of them.

Note that the syntax is very concise, so it's helpful to break down individual regular expressions into the component parts to understand them. As Murrell notes, since regexp are their own language, it's a good idea to build up a regexp in pieces as a way of avoiding errors just as we would with any computer code. *gregexpr()* is particularly useful in seeing **what** was matched to help in understanding and learning regular expression syntax and debugging your regexp.

The special characters (meta-characters) used for defining regular expressions are: * . ^ $ + ? ( ) [ ] { } | \ . To use these characters literally as characters, we have to 'escape' them. In R, we have to use two backslashes insstead of a single backslash because R uses a single backslash to symbolize certain control characters, such as \n for newline. Outside of R, one would only need a single backslash.

Character sets and character classes If we want to search for any one of a set of characters, we use a character set, such as `[13579]` or `[abcd]` or `[0-9]` (where the dash indicates a sequence) or `[0-9a-z]` or `[ \t]`. To indicate any character not in a set, we place a ^ just inside the first bracket: `[^abcd]`. The period stands for any character.

There are a bunch of named character classes so that we don't have write out common sets of characters. The syntax is `[:class:]` where *class* is the name of the class. The classes include the *digit*, *alpha*, *alnum*, *lower*, *upper*, *punct*, *blank*, *space* (see ?regexp in R for formal definitions of all of these, but most are fairly self-explanatory). To make a character set with a character

class, e.g. the digit class: `[[:digit:]]`. Or we can make a combined character set such as `[[:alnum]_]`. E.g. the latter would be useful in looking for email addresses.

```
addresses <- c("john@att.com", "stat243@bspace.berkeley.edu",
    "john_smith@att.com")
grep("[[:digit:]_]", addresses)

## [1] 2 3
```

Some synonyms for the various classes are: `\\w` = `[:alnum:]`, `\\W` = `^[:alnum:]`, `\\d` = `[:digit]`, `\\D` = `^[:digit:]`, `\\s` = `[:space:]`, `\\S` = `^[:space:]`.

Challenge: how would we find a spam-like pattern with digits or non-letters inside a word? E.g., I want to find

"V1agra" or "Fancy repl!c@ted watches".

Location-specific matches To find a pattern at the beginning of the string, we use ^ (note this was also used for negation, but in that case occurs only inside square brackets) and to find it at the end we use $.

```
text <- c("john", "jennifer pierce", "Juan carlos rey")
grep("^[[:upper:]]", text)  # finds text that starts with an upper case lett
grep("[[:digit:]]$", text)  # finds text with a number at the end
```

What does this match: `^[^[:lower:]]$` ?

Here are some more examples, illustrating the use of regexp in *grep()*, *gregexpr()*, and *gsub()*.

```
text <- c("john", "jennifer pierce", "Juan carlos rey")
grep("[ \t]", text)

## [1] 2 3

gregexpr("[ \t]", text)

## [[1]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
```

```
## [1] TRUE
##
## [[2]]
## [1] 9
## attr(,"match.length")
## [1] 1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1]  5 12
## attr(,"match.length")
## [1] 1 1
## attr(,"useBytes")
## [1] TRUE
```

```
gsub("^j", "J", text)
```

```
## [1] "John"            "Jennifer pierce"
## [3] "Juan carlos rey"
```

Repetitions Now suppose I wanted to be able to detect "V1@gra" as well. I need to be able to deal with repetitions of character sets.

I can indicate repetitions as indicated in these examples:

- [[:digit:]]* – any number of digits
- [[:digit:]]+ – at least one digit
- [[:digit:]]? – zero or one digits
- [[:digit:]]{1,3} – at least one and no more than three digits
- [[:digit:]]{2,} – two or more digits

An example is that `\\[.*\\]` is the pattern of any number of characters (.*) separated by square brackets.

So the spam search might becomes:

```
text <- c("hi John", "V1@gra", "here's the problem set")
grep("[[:alpha:]]+[[:digit:][:punct:]]+[[:alpha:]]*", text)  # ok, we need t

## [1] 2 3
```

Grouping and references We often want to be able to look for multi-character patterns and to be able to refer back to the patterns that are found. Both are accomplished with parentheses. We can search for a group of characters as follows by putting the group in parentheses, such as `([[:digit:]]{1,3}\\.)` to find a 1 to 3 digit number followed by a period. Here's an example of searching for an IP number:

```
grep("([[:digit:]]{1,3}\\.){3}[[:digit:]]{1,3}", text)
```

You'll explore this further on PS2.

It's often helpful to be able to save a pattern as a variable and refer back to it. We saw an example of this in the sed syntax I gave you in PS1. Let's translate that syntax here:

```
text <- ("\"H4NY07011\",\"ACKERMAN, GARY L.\",\"H\",\"$13,242\",,,")
gsub("([^\",]),", "\\1", text)

## [1] "\"H4NY07011\",\"ACKERMAN GARY L.\",\"H\",\"$13242\",,,"
```

We can have multiple sets of parentheses, using `\\1`, `\\2`, etc.

We can indicate any one of a set of multi-character sequences as: `(http|ftp)`.

```
gregexpr("(http|ftp):\\/\\/", c("at the site http://www.ibm.com",
    "other text"))

## [[1]]
## [1] 13
## attr(,"match.length")
## [1] 7
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
```

```
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
```

Challenge: How would I extract an email address from an arbitrary text string? Comment: if you want to just return the address using `gsub(pattern, "\\1", text)` will not do the trick. Why not?

Challenge: Suppose a text string has dates in the form "Aug-3", "May-9", etc. and I want them in the form "3 Aug", "9 May", etc. How would I do this search/replace?

Greedy matching It turns out the pattern matching is 'greedy' - it looks for the longest match possible.

Suppose we want to strip out html tags as follows:

```
text <- "Students may participate in an internship <b> in place</b> of <b> c
gsub("<.*>", "", text)

## [1] "Students may participate in an internship  of their courses."
```

One additional bit of syntax is that one can append a ? to the repetition syntax to cause the matching to be non-greedy. Here's an example.

```
gregexpr("[[:space:]]+.+?[[:space:]]+", "the dog jumped over the blue moon")

## [[1]]
## [1]  4 15 24
## attr(,"match.length")
## [1] 5 6 6
## attr(,"useBytes")
## [1] TRUE
```

Challenge: How could we change our regexp to avoid the greedy matching?

Regular expressions in other contexts Regular expression can be used in a variety of places. E.g., to split by any number of white space characters

Table 1. Regular expression syntax.

Syntax	What it matches
^ab	match 'ab' at the beginning of the string
ab$	match 'ab' at the end of the string
[abc]	match a or b or c anywhere (this is a character class)
[\t]	match a space or a tab
`(ab|cd|def)`	match any of the strings in the set
`(ab){2,9}`	match 'ab' repeated at least 2 and no more than 9 times
(ab){2,}	match 'ab' repeated 2 or more times
[0-9a-z]	match a single digit or lower-case alphabetical
[^0-9]	match any single character except a digit
`a.b`	match a and b separated by a single character
*a.*b*	match a and b separated by any number of (or no) characters
a.+b	like a.*b but must have at least one character in between
[[:digit:]]	match *digit* class; other classes are *alpha*, *alnum*, *lower*, *upper*, *punct*, *blank*, *space* (see ?regexp)
\\[	double backslashes are used if we want to search for a meta-character used in regexp syntax

```
line <- "a dog\tjumped\nover \tthe moon."
cat(line)

## a dog jumped
## over  the moon.

strsplit(line, split = "[[:space:]]+")

## [[1]]
## [1] "a"      "dog"    "jumped" "over"   "the"
## [6] "moon."

strsplit(line, split = "[[:blank:]]+")

## [[1]]
## [1] "a"            "dog"          "jumped\nover"
## [4] "the"          "moon."
```

Summary Table 1 summarizes the key syntax in regular expressions.

12 Manipulating dates

One common form of text information is dates and times. These can be a hassle because of the variety of ways of representing them, issues of time zones and leap years, the complicated mapping between dates and day of week, and the irregularity of the number of days in a month. We can use *as.Date()* to convert from text to an object of class *Date*, providing a character string such as *"%Y-%m-%d"* or *"%d/%m/%Y"* as the format for R to expect.

```
date1 <- as.Date("03-01-2011", format = "%m-%d-%Y")
date2 <- as.Date("03/01/11", format = "%m/%d/%y")
date3 <- as.Date("05-May-11", format = "%d-%b-%y")
date1

## [1] "2011-03-01"

date1 + 42

## [1] "2011-04-12"

date3 - date1

## Time difference of 65 days
```

With an object in the *Date* class, you can get information with functions such as *weekdays()*, *months()*, *julian()*. Note that the Julian date is the number of days since a fixed initial day (in R's case this defaults to Jan 1, 1970). One handy feature is to use *as.Date()* to convert from Julian date to a more informative format:

```
julianValues <- c(1, 100, 1000)
as.Date(julianValues, origin = "1990-01-01")

## [1] "1990-01-02" "1990-04-11" "1992-09-27"
```

One can also use the *POSIXlt* and *POSIXct* classes to work with dates and times – see **`?DateTimeClasses`**.

The *chron* package provides classes for dates and times and for manipulating them. You can use *as.chron()* to convert from a *Date* object. Here's some *chron* syntax:

```
library(chron)
d1 <- chron("12/25/2004", "11:37:59")  # default format of m/d/Y and h:m:s
d2 <- as.chron(date1)
d1

## [1] (12/25/04 11:37:59)

d2

## [1] 03/01/11
```

13 Database manipulations

13.1 Databases

A relational database stores data as a database of tables (or relations), which are rather similar to R data frames, in that they are made up of columns or fields of one type (numeric, character, date, currency, ...) and rows or records containing the observations for one entity. One principle of databases is that if a category is repeated in a given variable, you can more efficiently store information about each level of the category in a separate table; consider information about people living in a state and information about each state - you don't want to include variables that only vary by state in the table containing information about individuals (at least until you're doing the actual analysis that needs the information in a single table. You can also interact with databases from a variety of database systems (DBMS=database management system) (some systems are SQLite, MySQL, Oracle, Access). We'll concentrate on accessing data in a database rather than management of a database.

The *DBI* package provides a front-end for manipulating databases from a variety of DBMS (MySQL, SQLite, Oracle, among others). Basically, you tell the package what DBMS is being used on the backend, link to the actual database, and then you can use the syntax in the package. Here's an example of using *DBI* to interact with a database called *myDatabase* (containing the table *MarsData*) using the SQLite system:

```
drv <- dbDriver("SQLite")
con <- dbConnect(drv, dbname = "myDatabase")  # so we're using a connection
mars <- dbReadTable(con, "MarsData")
dbDisconnect(con)
```

SQL is a common database query language, with variants on it used in the various database systems. It has statements like:

```
SELECT Id, City, State FROM Station WHERE Lat_N > 39.7 AND Lon_W
> 80 ORDER BY Lon_W
```

You can use DBI to send a database query from R rather than reading in the whole table:

```
myQuery <- "SELECT City, State FROM Station WHERE Lat_N > 39.7"
marsSub <- dbGetQuery(con, myQuery)
```

You can also use *dbSendQuery()* combined with *fetch()* to pull in a fixed number of records at a time, if you're working with a big database. Finally you can do database joins (i.e., merges) using SQL, but we won't go into the syntax here.

SAS can also be used for database management and manipulation and is quite fast for working with large datasets, storing them on disk rather than in memory. Some statisticians (including myself) often do database pre-processing in SAS and then output to R for analysis.

13.2 Dataset manipulations in R

Ok, let's assume you've gotten your data into an R data frame. It might be numeric, character, or some combination. R now allows you to do a variety of database manipulations.

Getting information about variable(s) One can get a variety of information about a variable:

```
sum(mtcars$cyl == 6)  # counts the number of TRUE values

## [1] 7

mean(mtcars$cyl == 6)  # proportion of TRUE values

## [1] 0.219

unique(mtcars$cyl)

## [1] 6 4 8

length(unique(mtcars$cyl))

## [1] 3

duplicated(mtcars$cyl)  # tells us which elements are repeated
```

```
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
##  [9]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [17]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
table(mtcars$gear)  # tabulates # of records in each unique value of mtcars$
```

```
##
##  3  4  5
## 15 12  5
```

```
table(mtcars$cyl, mtcars$gear)  # cross-tabulation by two variables
```

```
##
##      3  4  5
##   4  1  8  2
##   6  2  4  1
##   8 12  0  2
```

The output from *table()* can be hard to read, but *ftable()* can help. Compare

```
table(mtcars[c("cyl", "vs", "am", "gear")])
```

```
## , , am = 0, gear = 3
##
##    vs
## cyl  0  1
##   4  0  1
##   6  0  2
##   8 12  0
##
## , , am = 1, gear = 3
##
##    vs
## cyl  0  1
##   4  0  0
##   6  0  0
##   8  0  0
```

```
##
## , , am = 0, gear = 4
##
##    vs
## cyl  0  1
##   4  0  2
##   6  0  2
##   8  0  0
##
## , , am = 1, gear = 4
##
##    vs
## cyl  0  1
##   4  0  6
##   6  2  0
##   8  0  0
##
## , , am = 0, gear = 5
##
##    vs
## cyl  0  1
##   4  0  0
##   6  0  0
##   8  0  0
##
## , , am = 1, gear = 5
##
##    vs
## cyl  0  1
##   4  1  1
##   6  1  0
##   8  2  0
```

```
ftable(mtcars[c("cyl", "vs", "am", "gear")])
```

```
##           gear  3  4  5
## cyl vs am
```

```
## 4   0  0         0  0  0
##        1         0  0  1
##     1  0         1  2  0
##        1         0  6  1
## 6   0  0         0  0  0
##        1         0  2  1
##     1  0         2  2  0
##        1         0  0  0
## 8   0  0        12  0  0
##        1         0  0  2
##     1  0         0  0  0
##        1         0  0  0
```

Sorting Dataset sorting in R is a bit of a hassle, as the *sort()* function only sorts a single vector. Instead we rely on the *order()* function which gives the set of indices required to order a given vector (or multiple vectors, with ties within the first vector broken based on subsequent vectors) in either increasing or decreasing order.

```
mtcarsSorted <- mtcars[order(mtcars$cyl, mtcars$mpg), ]
head(mtcarsSorted)
```

```
##                 mpg cyl disp  hp drat   wt qsec vs am
## Volvo 142E     21.4   4  121 109 4.11 2.78 18.6  1  1
## Toyota Corona  21.5   4  120  97 3.70 2.46 20.0  1  0
## Datsun 710     22.8   4  108  93 3.85 2.32 18.6  1  1
## Merc 230       22.8   4  141  95 3.92 3.15 22.9  1  0
## Merc 240D      24.4   4  147  62 3.69 3.19 20.0  1  0
## Porsche 914-2  26.0   4  120  91 4.43 2.14 16.7  0  1
##                gear carb
## Volvo 142E        4    2
## Toyota Corona     3    1
## Datsun 710        4    1
## Merc 230          4    2
## Merc 240D         4    2
## Porsche 914-2     5    2
```

Merging/joining Database joins are used to combine information from multiple database tables. In R, this is done with the *merge()* function. Let's look at an example.

```
load("~/Desktop/243/data/exampleMerge.RData")
fits2 <- merge(fits, master, by.x = "stn", by.y = "coop",
    all.x = TRUE, sort = FALSE)
identical(fits$stn, fits2$stn)  # hmmm...., ordering has not been preserved

## [1] FALSE

which(fits2$stn != fits$stn)[1:5]

## [1] 3 4 5 6 7
```

The output of *merge()* can have an arbitrary ordering, which may be unappealing. Here's a little function that deals with that.

```
f.merge <- function(x, y, ...) {
    # merges two datasets but preserves the ordering based
    # on the first dataset
    x$ordering <- 1:nrow(x)
    tmp <- merge(x, y, ...)
    tmp <- tmp[order(tmp$ordering), ]  # sort by the indexing (note that sor
    return(tmp[, !(names(tmp) == "ordering")])
}
```

Long and wide formats Finally, we may want to convert between so-called 'long' and 'wide' formats, which are motivated by working with longitudinal data (multiple observations per subject). The wide format has repeated measurements for a subject in separate columns, while the long format has repeated measurements in separate rows, with a column for differentiating the repeated measurements. *stack()* converts from wide to long while *unstack()* does the reverse. *reshape()* is similar but more flexible and it can go in either direction. The wide format is useful for doing separate analyses by group, while the long format is useful for doing a single analysis that makes use of the groups, such as ANOVA or mixed models. Let's use the precipitation data as an example.

```
load("~/Desktop/243/data/prec.RData")
prec <- prec[1:1000, ]  # just to make the example code run faster
precVars <- 5:ncol(prec)
precStacked <- stack(prec, select = precVars)
out <- unstack(precStacked)
# to use reshape, we need a unique id for each row
# since reshape considers each row in the wide format
# as a subject
prec <- cbind(unique = 1:nrow(prec), prec)
precVars <- precVars + 1
precLong <- reshape(prec, varying = names(prec)[precVars],
    idvar = "unique", direction = "long", sep = "")
precLong <- precLong[!is.na(precLong$prec), ]
precWide <- reshape(precLong, v.names = "prec", idvar = "unique",
    direction = "wide", sep = "")
```

Check out *melt()* and *cast()* in the *reshape* package for easier argument formats than *reshape()*.

Working with factors You can create a factor with the cut() function. By default the levels are not ordered, but we can manipulate them, or make sure they're ordered directly from cut().

```
x <- rnorm(100)
f <- cut(x, breaks = c(-Inf, -1, 1, Inf), labels = c("low",
    "medium", "high"))
levels(f)  # note that f is not explicitly ordered

## [1] "low"    "medium" "high"

f <- relevel(f, "high")  # puts high as first level
f <- cut(x, breaks = c(-Inf, -1, 1, Inf), labels = c("low",
    "medium", "high"), ordered_result = TRUE)
```

Changing the order of the levels can be helpful for controlling the order of plotting of levels and for controlling the baseline category in an ANOVA setting.

14 Output from R

14.1 Writing output to files

Functions for text output are generally analogous to those for input. *write.table()*, *write.csv()*, and *writeLines()* are analogs of *read.table()*, *read.csv()*, and *readLines()*. *write()* can be used to write a matrix to a file, specifying the number of columns desired. *cat()* can be used when you want fine control of the format of what is written out and allows for outputting to a connection (e.g., a file).

And of course you can always save to an R data file using *save.image()* (to save all the objects in the workspace or *save()* to save only some objects. Happily this is platform-independent so can be used to transfer R objects between different OS.

14.2 Formatting output

One thing to be aware of when writing out numerical data is how many digits are included. For example, the default with *write()* and *cat()* is the number of digits displayed to the screen, controlled by `options()$digits`. (to change this, do `options(digits = 5)` or specify as an argument to *write()* or *cat()*) If you want finer control, use *sprintf()*, e.g. to print out print out temperatures as reals ("*f*"=floating points) with four decimal places and nine total character positions, followed by a C for Celsius:

```
temps <- c(12.5, 37.234324, 1342434324.79997, 2.3456e-06,
    1e+10)
sprintf("%9.4f C", temps)

## [1] "  12.5000 C"         "  37.2343 C"
## [3] "1342434324.8000 C"   "   0.0000 C"
## [5] "10000000000.0000 C"
```

cat() is a good choice for printing a message to the screen, often better than *print()*, which is an object-oriented method. You generally won't have control over how the output of a *print()* statement is actually printed.

```
val <- 1.5
cat("My value is ", val, ".\n", sep = "")

## My value is 1.5.

print(paste("My value is ", val, ".", sep = ""))

## [1] "My value is 1.5."
```

We can do more to control formatting with *cat()*:

```
# input
x <- 7
n <- 5
# display powers
cat("Powers of", x, "\n")
cat("exponent   result\n\n")
result <- 1
for (i in 1:n) {
    result <- result * x
    cat(format(i, width = 8), format(result, width = 10),
        "\n", sep = "")
}
x <- 7
n <- 5
# display powers
cat("Powers of", x, "\n")
cat("exponent result\n\n")
result <- 1
for (i in 1:n) {
    result <- result * x
    cat(i, "\t", result, "\n", sep = "")
}
```

Good coding practices

September 20, 2012

References:

- Murrell, Introduction to Data Technologies, Ch. 2
- Journal of Statistical Software vol. 42: 19 Ways of Looking at Statistical Software

Some of these tips apply more to software development and some more to analyses done for specific projects; hopefully it will be clear in most cases.

1 Editors

Use an editor that supports the language you are using (e.g., *Emacs*, *WinEdt*, *Tinn-R* on Windows, or the built-in editors in *RStudio* or the Mac R GUI). Some advantages of this can include: (1) helpful color coding of different types of syntax and of strings, (2) automatic indentation and spacing, (3) code can often be run or compiled from within the editor, (4) parenthesis matching, (5) line numbering (good for finding bugs).

2 Coding syntax

- Header information: put metainfo on the code into the first few lines of the file as comments. Include who, when, what, how the code fits within a larger program (if appropriate), possibly the versions of R and key packages that you wrote this for
- Indentation: do this systematically (your editor can help here). This helps you and others to read and understand the code and can help in detecting errors in your code because it can expose lack of symmetry.

- Whitespace: use a lot of it. Some places where it is good to have it are (1) around operators (assignment and arithmetic), (2) between function arguments and list elements, (3) between matrix/array indices, in particular for missing indices.
- Use blank lines to separate blocks of code and comments to say what the block does
- Split long lines at meaningful places.
- Use parentheses for clarity even if not needed for order of operations. For example, `a/y*x` will work but is not easy to read and you can easily induce a bug if you forget the order of ops.
- Documentation - add lots of comments (but don't belabor the obvious). Remember that in a few months, you may not follow your own code any better than a stranger. Some key things to document: (1) summarizing a block of code, (2) explaining a very complicated piece of code - recall our complicated regular expressions, (3) explaining arbitrary constant values.
- For software development, break code into separate files (<2000-3000 lines per file) with meaningful file names and related functions grouped within a file.
- Choose a consistent naming style for objects and functions: e.g. *nIts* vs. *n.its* vs *numberOfIts* vs. *n_its*
  - Adler and Google's R style guide recommend naming objects with lowercase words, separated by periods, while naming functions by capitalizing the name of each word that is joined together, with no periods.
- Try to have the names be informative without being overly long.
- Don't overwrite names of objects/functions that already exist in R. E.g., don't use 'lm'.

```
> exists("lm")
```

- Use active names for functions (e.g., *calc.loglik*, *calcLogLik*)
- Learn from others' code

This semester, someone will be reading your code - me, when I look at your assignments. So to help me in understanding your code and develop good habits, put these ideas into practice in your assignments.

3 Coding style

This is particularly focused on software development, but some of the ideas are useful for data analysis as well.

- Break down tasks into core units
- Write reusable code for core functionality and keep a single copy of the code (w/ backups of course) so you only need to change it once
- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities)
- Write functions that take data as an argument and not lines of code that operate on specific data objects. Why? Functions allow us to reuse blocks of code easily for later use and for recreating an analysis (reproducible research). It's more transparent than sourcing a file of code because the inputs and outputs are specified formally, so you don't have to read through the code to figure out what it does.
- Functions should:
  - be modular (having a single task);
  - have meaningful name; and
  - have a comment describing their purpose, inputs and outputs (see the help file for an R function for how this is done in that context).
- Object orientation is a nice way to go
- Don't hard code numbers - use variables (e.g., number of iterations, parameter values in simulations), even if you don't expect to change the value, as this makes the code more readable:

  ```
  > speedOfLight <- 3e8
  ```

- Use lists to keep disparate parts of related data together
- Practice defensive programming
  - check function inputs and warn users if the code will do something they might not expect or makes particular choices;
  - check inputs to *if* and the ranges in *for* loops;

  - provide reasonable default arguments;
  - document the range of valid inputs;
  - check that the output produced is valid; and
  - stop execution based on checks and give an informative error message.

- Try to avoid system-dependent code that only runs on a specific version of an OS or specific OS
- Learn from others' code
- Consider rewriting your code once you know all the settings and conditions; often analyses and projects meander as we do our work and the initial plan for the code no longer makes sense and the code is no longer designed specifically for the job being done.

4 Tips for avoiding bugs

- Write an initial version of the code in the simplest way, without trying to be efficient (e.g., use *for* loops even if you're coding in R); then make a second version that employs efficiency tricks and check that both produce the same output.
- Make big changes in small steps, sequentially checking to see if the code has broken on test case(s).
- Plan out your code in advance, including all special cases/possibilities.
- Figure out if a functionality already exists in (or can be adapted from) an R package (or potentially in a C/Fortran library/package): code that is part of standard mathematical/numerical packages will probably be more efficient and bug-free than anything you would write.
- When doing software development, write tests for your code early in the process.

5 Dealing with errors

When writing functions, and software more generally, you'll want to warn the user or stop execution when there is an error and exit gracefully, giving the user some idea of what happened. The *warning()* and *stop()* functions allow you to do this; in general they would be called based on an if statement. *stopifnot()* can stop based on checking multiple conditions.

You can control what happens when a warning occurs with *options()$warning*. This can be helpful for debugging - e.g., you can force R to stop if a warning is issued rather than continuing so you can delve into what happened.

Also, sometimes a function you call will fail, but you want to continue execution. For example, suppose you are fitting a bunch of linear models and occasionally the design matrix is singular. You can wrap a function call within the *try()* function (or *tryCatch()*) and then your code won't stop. You can also evaluate whether a given function call executed properly or not. Here's an example of fitting a model for extreme values:

```
library(ismev)
library(methods)
n <- 100
nDays <- 365
x <- matrix(rnorm(nDays * n), nr = nDays)
x <- apply(x, 2, max)
x <- cbind(rep(0, 100), x)
params <- matrix(NA, nr = ncol(x), nc = 3)
for (i in 1:ncol(x)) {
    fit <- try(gev.fit(x[, i], show = FALSE))
    if (!is(fit, "try-error"))
        params[i, ] = fit$mle
}
params

##       [,1]  [,2]    [,3]
## [1,]    NA    NA      NA
## [2,] 2.772 0.337 -0.1401
```

Challenge: figure out how to use *tryCatch()* to deal with the error above. Note that I haven't used it and it seemed somewhat inscrutable on quick look.

6 Running analyses

Save your output at intermediate steps (including the random seed state) so you can restart if an error occurs or a computer fails. Using *save()* and *save.image()* to write to *.RData* files work well for this.

Run your code on a small subset of the problem before setting off a job that runs for hours or days. Make sure that the code works on the small subset and saves what you need properly at the end.

7 Versioning and backup

For data analysis and your own project work:

- Basic version control ideas: whenever you make changes to the structure of your code/program, make it a new version, numbering sequentially (or with more sophisticated version numbering used by software developers).
- Keep a text file that documents what changes from version to version (or do this in the document header). I find this easier than trying to name my code files informatively.
- DO NOT get rid of old code versions (text is cheap to store) and DO NOT get rid of blocks of code that work.
- Keep a running file (or comments in the code) of what you'd like to change in the future.

In addition, for software development you might consider:

- Using version control software: *git, cvs* and *svn* are several tools for this. You can host a *git* repository at *github.com* for free (provided you don't mind having others see it).

8 Documenting an analysis

Provenance is becoming increasingly important in science. It basically means being able to trace the steps of an analysis back to its origins. *Replicability* is a related concept - the idea is that you or someone else could replicate the analysis that you've done. This can be surprisingly hard as time passes even if you're the one attempting the replication. Let's think through what is required for something to be replicable.

- Have a directory for each project with meaningful subdirectories: e.g., *code*, *data*, *paper*
- Keep a document describing your running analysis with dates in a text file (i.e., a lab book)
- Note where data were obtained (and when, which can be helpful when publishing) and preprocessing steps in the lab book. Have data version numbers with a file describing the changes and dates (or in lab book).

- Have a file of code for pre-processing, one or more for analysis, and one for figure/table preparation.
  - Have the code file for the figures produce the EXACT manuscript figures, operating on an RData file that contains all the objects necessary to run the figure-producing code; the code producing the RData file should be in your analysis code file (or somewhere else sensible).
  - Alternatively, use *Sweave* or *knitr* for your document preparation.
- Note what code files do what in the lab book.

Debugging and Profiling

September 26, 2012

Sources:

- Chambers
- Roger Peng's notes on debugging in R

1 Common syntax errors and bugs

Tips for avoiding bugs

1. Build up code in pieces, testing along the way.
2. Use core R functionality and algorithms already coded.
3. Remove objects you don't need, to avoid accidentally using values from an old object via the scoping rules.
4. Be careful that the conditions of *if* statements and the sequences of *for* loops are robust when they involve evaluating R code.
5. Write code for clarity and accuracy first; then worry about efficiency.
6. Code in a modular fashion, making good use of functions, so that you don't need to debug the same code multiple times.

Common syntax errors and bugs:

1. Parenthesis mis-matches
2. *[[. . .]]* vs. *[. . .]*
3. == vs. =

4. Comparing real numbers exactly using '==' is dangerous (more in a later Unit). Suppose you generate *x = 0.333333* in some fashion with some code and then check:

   ```
   > x == 1/3 # FALSE is the result
   ```

5. Vectors vs. single values:

   (a) || vs. | and *&&* vs. *&*

   (b) You expect a single value but your code gives you a vector

   (c) You want to compare an entire vector but your code just compares the first value (e.g., in an *if* statement) – consider using *identical()* or *all.equal()*

6. Silent type conversion when you don't want it, or lack of coercion where you're expecting it

7. Using the wrong function or variable name

8. Giving arguments to a function in the wrong order

9. In an if-then-else statement, the *else* cannot be on its own line (unless all the code is enclosed in {}) because R will see the if-then part of the statement, which is a valid R statement, will execute that, and then will encounter the *else* and return an error. We saw this in Unit 3.

10. Forgetting to define a variable in the environment of a function and having the function, via lexical scoping, get that variable as a global variable from one of the enclosing environments. At best the types are not compatible and you get an error; at worst, you use a garbage value and the bug is hard to trace. In some cases your code may work fine when you develop the code (if the variable exists in the enclosing environment), but then may not work when you restart R if the variable no longer exists or is different.

11. R (usually helpfully) drops matrix and array dimensions that are extraneous; which can sometimes confuse later code that expects an object of a certain dimension. The '[' operator takes an additional optional argument that can avoid dropping dimensions.

```
mat <- matrix(1:4, 2, 2)[1, ]
dim(mat)

## [1] 2

print(mat)
```

```
## [1] 1 3

colSums(mat)

## Error:  'x' must be an array of at least two dimensions

mat <- matrix(1:4, 2, 2)[1, , drop = FALSE]
```

2 Debugging Strategies

Debugging is about figuring out what went wrong and where it went wrong.

In compiled languages, one of the difficulties is figuring out what is going on at any given place in the program. This is a lot easier in R by virtue of the fact that R is interpreted and we can step through code line by line at the command line. However, beyond this, there are a variety of helpful tools for debugging R code. In particular these tools can help you step through functions and work inside of functions from packages.

2.1 Basic strategies

Read and think about the error message. Sometimes it's inscrutable, but often it just needs a bit of deciphering. Looking up a given error message in the R mailing list archive can be a good strategy.

Fix errors from the top down - fix the first error that is reported, because later errors are often caused by the initial error. It's common to have a string of many errors, which looks daunting, caused by a single initial error.

Is the bug reproducible - does it always happen in the same way at at the same point? It can help to restart R and see if the bug persists - this can sometimes help in figuring out if there is a scoping issue and we are using a global variable that we did not mean to.

Another basic strategy is to build up code in pieces (or tear it back in pieces to a simpler version). This allows you to isolate where the error is occurring.

To stop code if a condition is not satisfied, you can use *stopifnot()*, e.g.,

```
> x <- 3
> stopifnot(is(x, "matrix"))
```

This allows you to catch errors that can be anticipated.

The *codetools* library has some useful tools for checking code, including a function, *findGlobals()*, that let's you look for the use of global variables

```
options(width = 55)
library(codetools)
findGlobals(lm)[1:25]

##  [1] "<-"             "=="             "-"
##  [4] "!"              "!="             "["
##  [7] "[[<-"           "{"              "$<-"
## [10] "*"              "&&"             "as.name"
## [13] "as.vector"      "attr"           "c"
## [16] "class<-"        "eval"           "gettextf"
## [19] ".getXlevels"    "if"             "is.empty.model"
## [22] "is.matrix"      "is.null"        "is.numeric"
## [25] "length"

f <- function() {
    y <- 3
    print(x + y)
}
findGlobals(f)

## [1] "<-"    "{"     "+"     "print" "x"
```

If you've written your code modularly with lots of functions, you can test individual functions. Often the error will be in what gets passed into and out of each function.

You can have warnings printed as they occurred, rather than saved, using `options(warn = 1)`. This can help figure out where in a loop a warning is being generated. You can also have R convert warnings to error using `options(warn = 2)`.

At the beginning of time (the 1970s?), the standard debugging strategy was to insert print statements in one's code to see the value of a variable and thereby decipher what could be going wrong. We have better tools nowadays.

2.2 Interactive debugging via the browser

The core strategy for interactive debugging is to use *browser()*, which pauses the current execution, and provides an interpreter, allowing you to view the current state of R. You can invoke *browser()* in four ways

- by inserting a call to *browser()* in your code if you suspect where things are going wrong

- by invoking the browser after every step of a function using *debug()*
- by using `options(error = recover)` to invoke the browser when *error()* is called
- by temporarily modifying a function to allow browsing using *trace()*

Once in the browser, you can execute any R commands you want. In particular, using *ls()* to look at the objects residing in the current function environment, looking at the values of objects, and examining the classes of objects is often helpful.

2.3 Using *debug()* to step through code

To step through a function, use `debug(nameOfFunction)`. Then run your code. When the function is executed, R will pause execution just before the first line of the function. You are now using the browser and can examine the state of R and execute R statements.

In addition, you can use "n" or return to step to the next line, "c" to execute the entire current function or current loop, and "Q" to stop debugging. We'll see an example in the demo code.

To unflag the function so that calling it doesn't invoke debug, use `undebug(nameOfFunction)`. In addition to working with functions you write you can use debug with standard R functions and functions from packages. For example you could do `debug(glm)`.

2.4 Tracing errors in the call stack

traceback() and *recover()* allow you to see the call stack (the sequence of nested function calls) at the time of an error. This helps pinpoint where in a series of function calls the error may be occurring.

If you've run the code and gotten an error, you can invoke *traceback()* after things have gone awry. R will show you the stack, which can help pinpoint where an error is occurring.

More helpful is to be able to browse within the call stack. To do this invoke `options(error = recover)` (potentially in your *.Rprofile* if you do a lot of programming). Then when an error occurs, *recover()* gets called, usually from the function in which the error occurred. The call to *recover()* allows you to navigate the stack of active function calls at the time of the error and browse within the desired call. You just enter the number of the call you'd like to enter (or 0 to exit). You can then look around in the frame of a given function, entering an empty line when you want to return to the list of calls again.

You can also combine this with `options(warn = 2)`, which turns warnings into errors to get to the point where a warning was issued.

2.5 Using *trace()* to temporarily insert code

trace() lets you temporarily insert code into a function (including standard R functions and functions in packages!) that can then be easily removed. You can use trace in a few ways - here's how you would do it most simply, where by default the second argument is invoked at the start of the function given as the first argument, but it is also possible to invoke just before exiting a function:

```
trace(lm, recover) # invoke recover() when the function starts
trace(lm, exit = browser) # invoke browser() when the function ends
trace(lm, browser, exit = browser) # invoke browser() at start and
end
```

Then in this example, once the browser activates I can poke around within the *lm()* function and see what is going on.

The most flexible way to use *trace()* is to use the argument *edit = TRUE* and then insert whatever code you want whereever you want. If I want to ensure I use a particular editor, such as *emacs*, I can use the argument *edit = "emacs"* . A standard approach would be to add a line with *browser()* to step through the code or *recover()* (to see the call stack and just look at the current state of objects). Alternatively, you can manually change the code in a function without using *trace()*, but it's very easy to forget to change things back and hard to do this with functions in packages, so *trace()* is a nice way to do things.

You call *untrace()*, e.g., `untrace(lm)` , to remove the temporarily inserted code; otherwise it's removed when the session ends.

Alternatively you can do `trace(warning, recover)` which will insert a call to *recover()* whenever *warning()* is called.

3 Memory management

3.1 Allocating and freeing memory

Unlike compiled languages like C, in R we do not need to explicitly allocate storage for objects. However, we have seen that there are times that we do want to allocate storage in advance, rather than successively concatenating onto a larger object.

R automatically manages memory, releasing memory back to the operating system when it's not needed via garbage collection. Occasionally you will want to remove large objects as soon as they are not needed. *rm()* does not actually free up memory, it just disassociates the name from the memory used to store the object. In general R will clean up such objects without a reference (i.e., a name) but you may need to call *gc()* to force the garbage collection. This uses some computation so it's generally not recommended.

In a language like C in which the user allocates and frees up memory, memory leaks are a major cause of bugs. Basically if you are looping and you allocate memory at each iteration and forget to free it, the memory use builds up inexorably and eventually the machine runs out of memory. In R, with automatic garbage collection, this is generally not an issue, but occasionally memory leaks do occur.

3.2 Monitoring memory use

There are a number of ways to see how much memory is being used. When R is actively executing statements, you can use *top* from the UNIX shell. In R, *gc()* reports memory use and free memory as *Ncells* and *Vcells*. As far as I know, *Ncells* concerns the overhead of running R and *Vcells* relates to objects created by the user, so you'll want to focus on *Vcells*. You can see the number of Mb currently used (the "*used*" column of the output) and the maximum used in the session (the "*max used*" column)"

```
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 152797  8.2     350000 18.7   350000 18.7
## Vcells 237920  1.9     786432  6.0   669149  5.2

x <- rnorm(1e+08)  # should use about 800 Mb
gc()

##             used  (Mb) gc trigger  (Mb)  max used
## Ncells    152824   8.2     350000  18.7    350000
## Vcells 100238272 764.8  110849727 845.8 100240351
##         (Mb)
## Ncells  18.7
## Vcells 764.8

rm(x)
gc()

##          used (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells 152842  8.2     350000  18.7    350000  18.7
## Vcells 238313  1.9   88679781 676.6 100243914 764.9
```

In Windows only, *memory.size()* tells how much memory is being used.

You can check the amount of memory used by individual objects with *object.size()*.

One frustration with memory management is that if your code bumps up against the memory limits of the machine, it can be very slow to respond even when you're trying to cancel the statement with *Ctrl-C*. You can impose memory limits in Linux by starting R (from the UNIX prompt) in a fashion such as this

```
> R --max-vsize=1000M
```

Then if you try to create an object that will push you over that limit or execute code that involves going over the limit, it will simply fail with the message "*Error: vector memory exhausted (limit reached?)*". So this approach may be a nice way to avoid paging by setting the maximum in relation to the physical memory of the machine. It might also help in debugging memory leaks because the program would fail at the point that memory use was increasing. I haven't played around with this much, so offer this with a note of caution.

4 Benchmarking

As we've seen, *system.time()* is very handy for comparing the speed of different implementations.

```
n <- 1000
x <- matrix(rnorm(n^2), n)
system.time({
    mns <- rep(NA, n)
    for (i in 1:n) mns[i] <- mean(x[i, ])
})

##    user  system elapsed
##   0.024   0.004   0.027

system.time(rowMeans(x))

##    user  system elapsed
##   0.004   0.000   0.003
```

The *rbenchmark* package provides a nice wrapper function, *benchmark()*, that automates speed assessments.

```
library(rbenchmark)
# speed of one calculation
```

```
n <- 1000
x <- matrix(rnorm(n^2), n)
benchmark(crossprod(x), replications = 10, columns = c("test",
    "elapsed", "replications"))

##           test elapsed replications
## 1 crossprod(x)   0.605           10

# comparing different approaches to a task
benchmark({
    mns <- rep(NA, n)
    for (i in 1:n) mns[i] <- mean(x[i, ])
}, rowMeans(x), replications = 10, columns = c("test", "elapsed",
    "replications"))

##                                                                    test
## 1 {\n    mns <- rep(NA, n)\n    for (i in 1:n) mns[i] <- mean(x[i, ])\n}
## 2                                                              rowMeans(x)
##   elapsed replications
## 1   0.234           10
## 2   0.029           10
```

5 Profiling

The *Rprof()* function will show you how much time is spent in different functions, which can help you pinpoint bottlenecks in your code.

```
library(fields)
Rprof("makeTS.prof")
out <- makeTS(0.1, 1000)
Rprof(NULL)
summaryRprof("makeTS.prof")
```

Here's the result for the *makeTS()* function from the demo code file:

```
$by.self   self.time self.pct total.time total.pct
".Call"         0.38    48.72       0.38     48.72
```

```
".Fortran"      0.22    28.21       0.22     28.21
"matrix"        0.08    10.26       0.30     38.46
"exp"           0.08    10.26       0.08     10.26
"/"             0.02     2.56       0.02      2.56
$by.total         total.time total.pct self.time self.pct
"makeTS"                0.78    100.00      0.00     0.00
".Call"                 0.38     48.72      0.38    48.72
"chol.default"          0.38     48.72      0.00     0.00
"chol"                  0.38     48.72      0.00     0.00
"standardGeneric"       0.38     48.72      0.00     0.00
"matrix"                0.30     38.46      0.08    10.26
"rdist"                 0.30     38.46      0.00     0.00
".Fortran"              0.22     28.21      0.22    28.21
"exp"                   0.08     10.26      0.08    10.26
"/"                     0.02      2.56      0.02     2.56
$sample.interval [1] 0.02
$sampling.time [1] 0.78
```

Rprof() tells how much time was spent in each function alone (the *by.self* bit) and aggregating the time spent in a function and all of the functions that it calls (the *by.total* bit). Usually the former is going to be more useful, but in some cases we need to decipher what is going on based on the latter.

Let's figure out what is going on here. The self time tells us that *.Call* (a call to C code), *.Fortran* (a call to Fortran code) and *matrix()* take up most of the time. Looking at the total time and seeing in *chol.default()* that *.Call* is used and in *rdist()* that *.Fortran()* and *matrix()* are used we can infer that about 49% of the time is being spent in the Cholesky and 38% in the *rdist()* calculation, with 10% in *exp()*. As we increase the number of time points, the time taken up by the Cholesky would increase since that calculation is order of n^3 while the others are order n^2 (more in the linear algebra unit).

Apparently there is a memory profiler in R, *Rprofmem()*, but it needs to be enabled when R is compiled (i.e., installed on the machine), because it slows R down even when not used. So I've never gotten to the point of playing around with it.

6 Getting help online

Just searching the R mailing list archive often gives you a hint of how to fix things. An example occurred when I was trying to figure out how fix a problem that was reporting a "*multibyte string*"

error in some emails in a dataset of Spam emails. I knew it had something to do with the character encoding and R not interpreting the codes for non-ASCII characters correctly but I wasn't sure how to fix it. So I searched for *"invalid multibyte string*". Around the 8th hit or so there was a comment about using *iconv()* to convert to the UTF-8 encoding, which solved the problem.

If you've searched the archive and haven't found an answer to your problem, you can often get help by posting to the *R-help* mailing list. A few guidelines (generally relevant when posting to mailing lists beyond just the R lists):

1. Search the archives and look through relevant R books or manuals first.
2. Boil your problem down to the essence of the problem, giving an example, including the output and error message
3. Say what version of R, what operating system and what operating system version you're using. Both *sessionInfo()* and *Sys.info()* can be helpful for getting this information.
4. Read the posting guide.

The mailing list is a way to get free advice from the experts, who include some of the world's most knowledgeable R experts - seriously - members of the R core development team contribute frequently. The cost is that you should do your homework and that sometimes the responses you get may be blunt, along the lines of "read the manual". I think it's a pretty good tradeoff - where else do you get the foremost experts in a domain actually helping you?

Note: of course the mailing list archive is also helpful for figuring out how to do things, not just for fixing bugs.

R Programming

October 8, 2012

References:

- Adler
- Chambers
- Murrell, Introduction to Data Technologies.
- R intro manual (R-intro) and R language manual (R-lang), both on CRAN.
- Venables and Ripley, Modern Applied Statistics with S

1 Efficiency

In general, make use of R's built-in functions, as these tend to be implemented internally (i.e., via compiled code in C or Fortran). In particular, if R is linked to optimized BLAS and Lapack code (e.g. Intel's MKL, OpenBLAS [on the SCF Linux servers], and AMD's ACML), you should have good performance (potentially comparable to Matlab and to coding in C). As far as I can tell, Macs have good built-in linear algebra, but I'm not sure what BLAS is being used behind the scenes. Often you can figure out a trick to take your problem and transform it to make use of the built-in functions.

Note that I run a lot of MCMCs so I pay attention to making sure my calculations are fast as they are done repeatedly. Similarly, one would want to pay attention to speed when doing large simulations and bootstrapping, and in some cases for optimization. And if you're distributing code, it's good to have it be efficient. But in other contexts, it may not be worth your time. Also, it's good practice to code it transparently first to reduce bugs and then to use tricks to speed it up and make sure the fast version works correctly.

Results can vary with with your system setup and version of R, so the best thing to do is figure out where the bottlenecks are in your code, and then play around with alternative specifications.

Finally, as you gain more experience, you'll get some intuition for what approaches might improve speed, but even with experience I find myself often surprised by what matters and what doesn't. It's often worth trying out a bunch of different ideas; *system.time()* and *benchmark()* are your workhorse tools in this context.

1.1 Fast initialization

It is very inefficient to iteratively add elements to a vector (using *c()*) or iteratively use *cbind()* or *rbind()* to add rows or columns to matrices or dataframes. Instead, create the full object in advance (this is equivalent to variable initialization in compiled languages) and then fill in the appropriate elements. The reason is that when R appends to an existing object, it creates a new copy and as the object gets big, this gets slow when one does it a lot of times. Here's an illustrative example, but of course we would not fill a vector like this (see the section on vectorization).

```
options(width = 50)
n <- 10000
x <- 1
system.time(for (i in 2:n) x <- c(x, i))

##    user  system elapsed
##   0.112   0.004   0.116

system.time({
    x <- rep(as.numeric(NA), n)
    for (i in 1:n) x[i] <- i
})

##    user  system elapsed
##   0.012   0.004   0.015
```

It's not necessary to use as.numeric() above though it saves a bit of time. **Challenge**: figure out why I have `as.numeric(NA)` and not just `NA`.

We can actually speed up the initialization (though in most practical circumstances, the second approach here would be overkill):

```
n <- 1e+06
system.time(x <- rep(as.numeric(NA), n))
```

```
##    user  system elapsed
##   0.068   0.000   0.071

system.time({
    x <- as.numeric(NA)
    length(x) <- n
})

##    user  system elapsed
##   0.004   0.000   0.003
```

For matrices, start with the right length vector and then change the dimensions

```
nr <- nc <- 2000
system.time(x <- matrix(as.numeric(NA), nr, nc))

##    user  system elapsed
##   0.036   0.008   0.048

system.time({
    x <- as.numeric(NA)
    length(x) <- nr * nc
    dim(x) <- c(nr, nc)
})

##    user  system elapsed
##   0.028   0.012   0.041
```

For lists, we can do this

```
myList <- vector("list", length = n)
```

1.2 Vectorized calculations

One key way to write efficient R code is to take advantage of R's vectorized operations.

```
n <- 1e+06
x <- rnorm(n)
system.time(x2 <- x^2)

##    user  system elapsed
##   0.000   0.004   0.002

x2 <- as.numeric(NA)
system.time({
    length(x2) <- n
    for (i in 1:n) {
        x2[i] <- x[i]^2
    }
})  # how many orders of magnitude slower?

##    user  system elapsed
##   2.116   0.000   2.127
```

So what's different in how R handles the calculations that explains the disparity? The vectorized calculation is being done natively in C in a for loop. The for loop above involves executing the for loop in R with repeated calls to C code at each iteration. You can usually get a sense for how quickly an R call will pass things along to C by looking at the body of the relevant function(s) being called and looking for *.Primitive*, *.Internal*, *.C*, *.Call*, or *.Fortran*. Let's take a look at the code for '+', *mean.default()*, and *chol.default()*.

Many R functions allow you to pass in vectors, and operate on those vectors in vectorized fashion. So before writing a for loop, look at the help information on the relevant function(s) to see if they operate in a vectorized fashion.

```
line <- c("Four score and 7 years ago, this nation")
startIndices = seq(1, by = 3, length = nchar(line)/3)
substring(line, startIndices, startIndices + 1)

##  [1] "Fo" "r " "co" "e " "nd" "7 " "ea" "s " "go"
## [10] " t" "is" "na" "io"
```

Challenge: Consider the chi-squared statistic involved in a test of independence in a contingency table:

$$\chi^2 = \sum_i \sum_j \frac{(y_{ij} - e_{ij})^2}{e_{ij}}, \quad e_{ij} = \frac{y_{i\cdot} y_{\cdot j}}{y_{\cdot\cdot}}$$

where $f_{i\cdot} = \sum_j f_{ij}$. Write this in a vectorized way without any loops. Note that 'vectorized' calculations also work with matrices and arrays.

Vectorized operations can also be faster than built-in functions, and clever vectorized calculations even better, though sometimes the code is uglier:

```
x <- rnorm(1e+06)
system.time(truncx <- ifelse(x > 0, x, 0))

##    user  system elapsed
##   0.344   0.008   0.353

system.time({
    truncx <- x
    truncx[x < 0] <- 0
})

##    user  system elapsed
##   0.020   0.000   0.022

system.time(truncx <- x * (x > 0))

##    user  system elapsed
##   0.008   0.000   0.009
```

The demo code has a surprising example where combining vectorized calculations with a *for* loop is actually faster than using *apply()*. The goal is to remove rows of a large matrix that have any NAs in them.

Additional tips:

- If you do need to loop over dimensions of a matrix or array, if possible loop over the smallest dimension and use the vectorized calculation on the larger dimension(s).
- Looping over columns is likely to be faster than looping over rows given column-major ordering.
- You can use direct arithmetic operations to add/subtract/multiply/divide a vector by each column of a matrix, e.g. `A*b`, multiplies each column of A times a vector b. If you need to operate by row, you can do it by transposing the matrix.

Caution: relying on R's recycling rule in the context of vectorized operations, such as is done when direct-multiplying a matrix by a vector to scale the rows, can be dangerous as the code is

not transparent and poses greater dangers of bugs. If it's needed to speed up a calculation, the best approach is to (1) first write the code transparently and then compare the efficient code to make sure the results are the same and (2) comment your code.

Challenge : What do the points above imply about how to choose to store values in a matrix. How would you choose what should be the row dimension and what should be the column dimension?

1.3 Using *apply()* and specialized functions

Another core efficiency strategy is to use the *apply()* functionality. Even better than *apply()* for calculating sums or means of columns or rows (it also can be used for arrays) is *{row,col}{Sums,Means}*:

```
n <- 3000
x <- matrix(rnorm(n * n), nr = n)
system.time(out <- apply(x, 1, mean))

##    user  system elapsed
##   0.220   0.028   0.250

system.time(out <- rowMeans(x))

##    user  system elapsed
##   0.024   0.000   0.025
```

We can '*sweep*' out a summary statistic, such as subtracting off a mean from each column, using *sweep()*

```
system.time(out <- sweep(x, 2, STATS = colMeans(x),
    FUN = "-"))

##    user  system elapsed
##   0.300   0.080   0.385
```

Here's a trick for doing it even faster based on vectorized calculations, remembering that if we subtract a vector from a matrix, it subtracts each element of the vector from all the elements in the corresponding ROW.

```
system.time(out2 <- t(t(x) - colMeans(x)))

##    user  system elapsed
##   0.252   0.056   0.306

identical(out, out2)

## [1] TRUE
```

As we've discussed using versions of *apply()* with lists may or may not be faster than looping but generally produces cleaner code. If you're worried about speed, it's a good idea to benchmark the *apply()* variant against looping.

1.4 Matrix algebra efficiency

Often calculations that are not explictly linear algebra calculations can be done as matrix algebra. The following can be done faster with *rowSums()*, so it's not a great example, but this sort of trick does come in handy in surprising places.

```
mat <- matrix(rnorm(500 * 500), 500)
system.time(apply(mat, 1, sum))

##    user  system elapsed
##   0.008   0.000   0.005

system.time(mat %*% rep(1, ncol(mat)))

##    user  system elapsed
##   0.004   0.000   0.001

system.time(rowSums(mat))

##    user  system elapsed
##   0.004   0.000   0.002
```

On the other hand, big matrix operations can be slow. Suppose you want a new matrix that computes the differences between successive columns of a matrix of arbitrary size. How would you do this as matrix algebra operations? [see demo code] Here it turns out that the *for* loop is much faster than matrix multiplication. However, there is a way to do it faster as matrix direct

subtraction. Comment: the demo code also contains some exploration of different ways of creating patterned matrices. Note that this level of optimization is only worth it if you're doing something over and over again, or writing code that you will distribute.

When doing matrix algebra, the order in which you do operations can be critical for efficiency. How should I order the following calculation?

```
n <- 5000
A <- matrix(rnorm(5000 * 5000), 5000)
B <- matrix(rnorm(5000 * 5000), 5000)
x <- rnorm(5000)
system.time(res <- A %*% B %*% x)

##    user  system elapsed
##   48.44   24.15   10.67
```

We can use the matrix direct product (i.e., `A*B`) to do some manipulations much more quickly than using matrix multiplication. **Challenge**: How can I use the direct product to find the trace of XY?

You can generally get much faster results by being smart when using diagonal matrices. Here are some examples, where we avoid directly doing: $X + D$, DX, XD:

```
n <- 1000
X <- matrix(rnorm(n^2), n)
diagvals <- rnorm(n)
D = diag(diagvals)
diag(X) <- diag(X) + diagvals
tmp <- diagvals * X  # instead of D %*% X
tmp2 <- t(t(X) * diagvals)  # instead of X %*% D
```

More generally, sparse matrices and structured matrices (such as block diagonal matrices) can generally be worked with MUCH more efficiently than treating them as arbitrary matrices. The *spam* (for arbitrary sparse matrices) and *bdsmatrix* (for block-diagonal matrices) packages in R can help, as can specialized code available in other languages, such as C and Fortran packages.

1.5 Fast mapping/lookup tables

Sometimes you need to map between two vectors. E.g., $y_{ij} \sim \mathcal{N}(\mu_j, \sigma^2)$ is a basic ANOVA type structure. Here are some efficient ways to aggregate to the cluster level and disaggregate to the observation level.

Disaggregate: Create a vector, *idVec*, that gives a numeric mapping of the observations to their cluster. Then you can access the μ value relevant for each observation as: `mus[idVec]`.

Aggregate: To find the sample means by cluster: `sapply(split(dat$obs, idVec), mean)`

R provides something akin to a hash table when you have named objects. For example:

```
vals <- rnorm(10)
names(vals) <- letters[1:10]
labs <- c("h", "h", "a", "c")
vals[labs]

##       h       h       a       c
##  1.7634  1.7634  0.4193 -0.2495
```

You can do similar things with dimension names of matrices/arrays, row and column names of dataframes, and named lists.

I haven't looked into this much, but according to Adler, if you need to do something like this where you are looking up from amongst very many items, the fast way to do it is by looking up items within an environment (see Section 5 of this unit) rather than within a named vector or list, because environments are implemented using hash tables.

1.6 Byte compiling

R now allows you to compile R code, which goes by the name of byte compiling. Byte-compiled code is a special representation that can be executed more efficiently because it is in the form of compact codes that encode the results of parsing and semantic analysis of scoping and other complexities of the R source code. This byte code can be executed faster than the original R code because it skips the stage of having to be interpreted by the R interpreter.

The functions in the *base* and *stats* packages are now byte-compiled by default. (If you print out a function that is byte-compiled, you'll see something like <*bytecode: 0x243a368*> at the bottom.

We can byte compile our own functions using *cmpfun()*. Here's an example (silly since we we actually do this calculation using vectorized operations):

```
library(compiler)
library(rbenchmark)
f <- function(x) {
```

```
    for (i in 1:length(x)) x[i] <- x[i] + 1
    return(x)
}
fc <- cmpfun(f)
fc  # notice the indication that the function is byte compiled.

## function(x) {
##     for (i in 1:length(x)) x[i] <- x[i] + 1
##     return(x)
## }
## <bytecode: 0x2b19130>

benchmark(f(x), fc(x), replications = 5)

##    test replications elapsed relative user.self
## 2 fc(x)            5   0.007    1.000     0.008
## 1  f(x)            5   0.046    6.571     0.048
##   sys.self user.child sys.child
## 2        0          0         0
## 1        0          0         0
```

You can compile an entire source file with *cmpfile()*, which produces a *.Rc* file. You then need to use *loadcmp()* to load in the *.Rc* file, which runs the code.

1.7 Challenges

One or more of these challenges may appear on a problem set.

Challenge 1: here's a calculation of the sort needed in mixture component modeling. I have a vector of n observations. I need to find the likelihood of each observation under each of p mixture components (i.e., what's the likelihood if it came from each of the components). So I should produce a matrix of n rows and p columns where the value in the ith row, jth column is the likelihood of the ith observation under the jth mixture component. The idea is that the likelihoods for a given observation are used in assigning observations to clusters. A naive implementation is:

```
> lik <- matrix(NA, nr = n, nc = p)
> for(j in 1:p) lik[ , j] <- dnorm(y, mns[j], sds[j])
```

Note that *dnorm()* can handle matrices and vectors as the observations **and** as the means and sds, so there are multiple ways to do this.

Challenge 2: Suppose you have $y_i \sim \mathcal{N}(\sum_{k=1}^{m_i} w_{i,k}\mu_{ID[i,k]}, \sigma^2)$ for a large number of observations, n. I give you a vector of μ values and a ragged list of weights and a ragged list of IDs identifying the cluster corresponding to each weight (note m_i varies by observation); this is a mixed membership type model. Figure out how to calculate the vector of means, $\sum_k w_{i,k}\mu_{ID[i,k]}$ as fast as possible. Suppose that m_i never gets too big (but μ might have many elements) - could this help you? Part of thinking this through involves thinking about how you want to store the information so that the calculations can be done quickly.

Challenge 3: Write code that simulates a random walk in two dimensions for n steps. First write out a straightforward implementation that involves looping. Then try to speed it up. The *cumsum()* function may be helpful.

2 Advanced topics in working with functions

2.1 Pointers

By way of contrast to R's pass by value system, I want to briefly discuss the idea of a pointer, common in compiled languages such as C.

```
int x = 3;
int* ptr;
ptr = &x;
*ptr * 7; // returns 21
```

Here *ptr* is the address of the integer *x*.

Vectors in C are really pointers to a block of memory:

```
int x[10];
```

In this case *x* will be the address of the first element of the vector. We can access the first element as **`x[0]`** or **`*x`**.

Why have we gone into this? In C, you can pass a pointer as an argument to a function. The result is that only the scalar address is copied and not the entire vector, and inside the function, one can modify the original vector, with the new value persisting on exit from the function. For example:

```
int myCal(int *ptr){
*ptr = *ptr + *ptr;
}
```

When calling C or C++ from R, one (implicitly) passes pointers to the vectors into C. Let's see an example:

```
out <- rep(0, n)
```

```
out <- .C("logLik", out = as.double(out),
                    theta = as.double(theta))$out
```

In C, the function definition looks like this:

```
void logLik(double* out, double* theta)
```

2.2 Alternatives to pass by value in R

There are occasions we do not want to pass by value. The main reason is when we want a function to modify a complicated object without having to return it and re-assign it in the parent environment. There are several work-arounds:

1. We can use Reference Class objects. Reference classes are new in R. We'll discuss these in Section 4.

2. We can access the object in the enclosing environment as a 'global variable', as we've seen when discussing scoping. More generally we can access the object using *get()*, specifying the environment from which we want to obtain the variable. Recall that to specify the location of an object, we can generally specify (1) a position in the search path, (2) an explicit environment, or (3) a location in the call stack by using *sys.frame()*. However we cannot change the value of the object in the parent environment without some additional tools.

   (a) We can use the '<<-' operator to assign into an object in the parent environment (provided an object of that name exists in the parent environment).

   (b) We can also use *assign()*, specifying the environment in which we want the assignment to occur.

3. We can use replacement functions (Section 2.4), which hide the reassignment in the parent environment from the user. Note that a second copy is generally created in this case, but the original copy is quickly removed.

4. We can use a closure. This involves creating functions within a function call and returning the functions as a list. When one executes the enclosing function, the list is created and one can call the functions of that object. Those functions then can access objects in the enclosing environment (the environment of the original function) and can use '<<-' to assign into the enclosing environment, to which all the functions have access. Chambers provides an example of this in Sec. 5.4.

   - A simplified version of using closures appeared in Unit 3:

```
x <- rnorm(10)
f <- function(input) {
    data <- input
    g <- function(param) return(param * data)
}
myFun <- f(x)
rm(x)  # to demonstrate we no longer need x
myFun(3)

##  [1] -3.3518 -2.7875  2.0417 -0.3284 -4.2282
##  [6] -2.3780 -1.3514 -2.4657 -5.1306 -3.9202

x <- rnorm(1e+07)
myFun <- f(x)
object.size(myFun)

## 1560 bytes
```

- A related approach is to wrap data with a function using *with()*. This approach appeared in Problem Set 2.

```
x <- rnorm(10)
myFun2 <- with(list(data = x), function(param) return(param *
    data))
rm(x)
myFun2(3)

##  [1]  1.5227  4.4157  2.9531  0.4543 -2.3201
##  [6]  1.7621 -1.6900 -0.3851  0.5225 -1.6843

x <- rnorm(1e+07)
myFun2 <- with(list(data = x), function(param) return(param *
    data))
object.size(myFun2)

## 1560 bytes
```

2.3 Operators

Operators, such as '*+*', '*[*' are just functions, but their arguments can occur both before and after the function call:

```
a <- 7; b <- 3
# let's think about the following as a mathematical function
#  -- what's the function call?
a + b

## [1] 10

`+`(a, b)

## [1] 10
```

In general, you can use back-ticks to refer to the operators as operators instead of characters. In some cases single or double quotes also work. We can look at the code of an operator as follows using back-ticks to escape out of the standard R parsing, e.g., `` `%*%` ``.

Finally, since an operator is just a function, you can use it as an argument in various places:

```
myList = list(list(a = 1:5, b = "sdf"), list(a = 6:10,
    b = "wer"))
myMat = sapply(myList, `[[`, 1)
# note that the index '1' is the additional
# argument to the [[ function
x <- 1:3
y <- c(100, 200, 300)
outer(x, y, `+`)

##      [,1] [,2] [,3]
## [1,]  101  201  301
## [2,]  102  202  302
## [3,]  103  203  303
```

You can define your own *binary* operator (an operator taking two arguments) using a string inside % symbols:

```
`%2%` <- function(a, b) {
    2 * (a + b)
}
3 %2% 7

## [1] 20
```

Since operators are just functions, there are cases in which there are optional arguments that we might not expect. We've already briefly seen the drop argument to the '[' operator:

```
mat <- matrix(1:4, 2, 2)
mat[, 1]

## [1] 1 2

mat[, 1, drop = FALSE]  # what's the difference?

##      [,1]
## [1,]    1
## [2,]    2
```

2.4 Unexpected functions and replacement functions

All code in R can be viewed as a function call.

What do you think is the functional version of the following code? What are the arguments?

```
if (x > 27) {
    print(x)
} else {
    print("too small")
}
```

Assignments that involve functions or operators on the left-hand side (LHS) are called *replacement expressions* or *replacement functions.* These can be quite handy. Here are a few examples:

```
diag(mat) <- c(3, 2)
is.na(vec) <- 3
names(df) <- c("var1", "var2")
```

Replacement expressions are actually function calls. The R interpreter calls the replacement function (which often creates a new object that includes the replacement) and then assigns the result to the name of the original object.

```
mat <- matrix(rnorm(4), 2, 2)
diag(mat) <- c(3, 2)
mat <- `diag<-`(mat, c(10, 21))
base::`diag<-`

## function (x, value)
## {
##     dx <- dim(x)
##     if (length(dx) != 2L)
##         stop("only matrix diagonals can be replaced")
##     len.i <- min(dx)
##     i <- seq_len(len.i)
##     len.v <- length(value)
##     if (len.v != 1L && len.v != len.i)
##         stop("replacement diagonal has wrong length")
##     if (len.i > 0L)
##         x[cbind(i, i)] <- value
##     x
## }
## <bytecode: 0x17135d0>
## <environment: namespace:base>
```

The old version of *mat* still exists until R's memory management cleans it up, but it's no longer referred to by the symbol '*mat*'. Occasionally this sort of thing might cause memory usage to increase (for example it's possible if you're doing replacements on large objects within a loop), but in general things should be fine.

You can define your own replacement functions like this, with the requirements that the last argument be named '*value*' and that the function return the entire object:

```
me <- list(name = "Chris", age = 25)
`name<-` <- function(obj, value) {
    obj$name <- value
    return(obj)
```

```
}
name(me) <- "Christopher"
```

2.5 Functions as objects

Note that a function is just an object.

```
x <- 3
x(2)

## Error:  could not find function "x"

x <- function(z) z^2
x(2)

## [1] 4
```

We can call a function based on the text name of the function.

```
myFun = "mean"
x = rnorm(10)
eval(as.name(myFun))(x)

## [1] -0.03887
```

We can also pass a function into another function either as the actual function object or as a character vector of length one with the name of the function. Here *match.fun()* is a handy function that extracts a function when the function is passed in as an argument of a function. It looks in the calling environment for the function and can handle when the function is passed in as a function object or as a character vector of length 1 giving the function name.

```
f <- function(fxn, x) {
    match.fun(fxn)(x)
}
f("mean", x)

## [1] -0.03887
```

This allows us to write functions in which the user passes in the function (as an example, this works when using *outer()*). Caution: one may need to think carefully about scoping issues in such contexts.

Function objects contain three components: an argument list, a body (a parsed R statement), and an environment.

```
f <- function(x) x
f2 <- function(x) y <- x^2
f3 <- function(x) {
    y <- x^2
    z <- x^3
    return(list(y, z))
}
class(f)

## [1] "function"

typeof(body(f))

## [1] "symbol"

class(body(f))

## [1] "name"

typeof(body(f2))

## [1] "language"

class(body(f2))

## [1] "<-"

typeof(body(f3))

## [1] "language"

class(body(f3))

## [1] "{"
```

We'll see more about objects relating to the R language and parsed code in a later section. For now, just realize that the parsed code itself is treated as an object(s) with certain types and certain classes.

We can extract the argument object as

```
f4 <- function(x, y = 2, z = 1/y) {
    x + y + z
}
args <- formals(f4)
class(args)

## [1] "pairlist"
```

A *pairlist* is like a list, but with pairing that in this case pairs argument names with default values.

2.6 Promises and lazy evaluation

One additional concept that it's useful to be aware of is the idea of a *promise* object. In function calls, when R matches user input arguments to formal argument names, it does not (usually) evaluate the arguments until they are needed, which is called *lazy evaluation*. Instead the formal arguments are of a special type called a *promise*. Let's see lazy evaluation in action. Do you think the following code will run?

```
f <- function(a, b = c) {
    c <- log(a)
    return(a * b)
}
f(7)
```

What's strange about this?

Another example:

```
f <- function(x) print("hi")
system.time(mean(rnorm(1e+06)))

##    user  system elapsed
##   0.068   0.000   0.068
```

```
system.time(f(3))

## [1] "hi"

##    user  system elapsed
##       0       0       0

system.time(f(mean(rnorm(1e+06))))

## [1] "hi"

##    user  system elapsed
##       0       0       0
```

3 Evaluating memory use

We've seen the use of *gc()* to assess total memory use and *object.size()* to see how much memory objects are using. The main things to remember when thinking about memory use are (1) numeric vectors take 8 bytes per element and (2) to keep track of when large objects are created, including in the frames of functions.

Note that *rm()* does not immediately free up memory; it merely disassociates the symbol/name pointing to the memory location from that memory, though this should happen fairly soon after an object is removed. You can call *gc()* to force the garbage collection to occur.

3.1 Hidden uses of memory

- Replacement functions can hide the use of additional memory.

```
x <- rnorm(1e+07)
gc()
dim(x) <- c(10000, 1000)
diag(x) <- 1
gc()
```

How much memory was used?

- Not all replacement functions actually involve creating a new object and replacing the original object.

```
x <- rnorm(1e+07)
.Internal(inspect(x))
x[5] <- 7
.Internal(inspect(x))
gc()
```

- Indexing large subsets can involve a lot of memory use.

```
x <- rnorm(1e+07)
gc()
y <- x[1:(length(x) - 1)]
gc()
```

Why was more memory used than just for x and y? Note that this is a limitation of R, which relates to it being a scripting language. Note that R could be designed to avoid this problem.

3.2 Passing objects to compiled code

As we've already discussed, when R objects are passed to compiled code (e.g., C or C++), they are passed as pointers and the compiled code uses the memory allocated by R (though it could also allocate additional memory if allocation is part of the code). So calling a C function from R will generally not have much memory overhead.

However we need to be aware of any casting that occurs, because the compiled code requires that the R object types match those that the function in the compiled code is expecting.

Here's an example of calling compiled code:

```
res <- .C("fastcount", PACKAGE="GCcorrect", tablex = as.integer(tablex),
tabley = as.integer(tabley), as.integer(xvar), as.integer(yvar),
as.integer(useline), as.integer(length(xvar)))
```

Let's consider when copies are made in casts:

```
f <- function(x) {
    print(.Internal(inspect(x)))
    return(mean(x))
```

```
}
x <- rnorm(1e+07)
class(x)
debug(f)
f(x)
f(as.numeric(x))
f(as.integer(x))
```

3.3 Lazy evaluation, delayed copying (copy-on-change) and memory use

Now let's consider how lazy evaluation affects memory use. We'll also see that something like lazy evaluation occurs outside of functions as well.

Let's see what goes on within a function in terms of memory use in different situations.

```
f <- function(x) {
    print(gc())
    z <- x[1]
    .Internal(inspect(x))
}
y <- rnorm(1e+07)
gc()
.Internal(inspect(y))
f(y)
```

In fact, this occurs outside function calls as well. Copies of objects are not made until one of the objects is actually modified. Initially, the copy points to the same memory location as the original object.

```
y <- rnorm(1e+07)
gc()
.Internal(inspect(y))
x <- y
gc()
.Internal(inspect(x))
x[1] <- 5
gc()
```

```
.Internal(inspect(x))
rm(x)
x <- y
.Internal(inspect(x))
.Internal(inspect(y))
y[1] <- 5
.Internal(inspect(x))
.Internal(inspect(y))
```

Challenge: How much memory is used in the following calculation?

```
x <- rnorm(1e+07)
myfun <- function(y) {
    z <- y
    return(mean(z))
}
myfun(x)

## [1] -0.0001403
```

How about here? What is going on?

```
x <- c(NA, x)
myfun <- function(y) {
    return(mean(y, na.rm = TRUE))
}
```

This makes sense if we look at *mean.default()*. Consider where additional memory is used.

3.4 Strategies for saving memory

A couple basic strategies for saving memory include:

- Avoiding unnecessary copies
- Removing objects that are not being used and, if necessary, do a *gc()* call.

If you're really trying to optimize memory use, you may also consider:

- Using reference classes and similar strategies to pass by reference
- Substituting integer and logical vectors for numeric vectors when possible

3.5 Example

Let's work through a real example where we keep a running tally of current memory in use and maximum memory used in a function call. We'll want to consider hidden uses of memory, passing objects to compiled code, and lazy evaluation. This code is courtesy of Yuval Benjamini. For our purposes here, let's assume that *xvar* and *yvar* are very long vectors using a lot of memory.

```
fastcount <- function(xvar, yvar) {
    naline <- is.na(xvar)
    naline[is.na(yvar)] = TRUE
    xvar[naline] <- 0
    yvar[naline] <- 0
    useline <- !naline
    # Table must be initialized for -1's
    tablex <- numeric(max(xvar) + 1)
    tabley <- numeric(max(xvar) + 1)
    stopifnot(length(xvar) == length(yvar))
    res <- .C("fastcount", PACKAGE = "GCcorrect", tablex = as.integer(tablex
        tabley = as.integer(tabley), as.integer(xvar),
        as.integer(yvar), as.integer(useline), as.integer(length(xvar)))
    xuse <- which(res$tablex > 0)
    xnames <- xuse - 1
    resb <- rbind(res$tablex[xuse], res$tabley[xuse])
    colnames(resb) <- xnames
    return(resb)
}
```

4 Object-oriented programming (OOP)

Popular languages that use OOP include C++, Java, and Python. In fact C++ is the object-oriented version of C. Different languages implement OOP in different ways.

The idea of OOP is that all operations are built around objects, which have a class, and methods that operate on objects in the class. Classes are constructed to build on (inherit from) each other, so that one class may be a specialized form of another class, extending the components and methods of the simpler class (e.g., *lm* and *glm* objects).

Note that in more formal OOP languages, all functions are associated with a class, while in R, only some are.

Often when you get to the point of developing OOP code in R, you're doing more serious programming, and you're going to be acting as a software engineer. It's a good idea to think carefully in advance about the design of the classes and methods.

4.1 S3 approach

S3 classes are widely-used, in particular for statistical models in the *stats* package. S3 classes are very informal in that there's not a formal definition for an S3 class. Instead, an S3 object is just a primitive R object such as a list or vector with additional attributes including a class name.

Inheritance Let's look at the *lm* class, which builds on lists, and *glm* class, which builds on the *lm* class. Here *mod* is an object (an instance) of class *lm*. An analogy is the difference between a random variable and a realization of that random variable.

```
library(methods)
yb <- sample(c(0, 1), 10, replace = TRUE)
yc <- rnorm(10)
x <- rnorm(10)
mod1 <- lm(yc ~ x)
mod2 <- glm(yb ~ x, family = binomial)
class(mod1)

## [1] "lm"

class(mod2)

## [1] "glm" "lm"

is.list(mod1)

## [1] TRUE

names(mod1)

##  [1] "coefficients"  "residuals"
##  [3] "effects"       "rank"
##  [5] "fitted.values" "assign"
```

```
##  [7] "qr"            "df.residual"
##  [9] "xlevels"       "call"
## [11] "terms"         "model"

is(mod2, "lm")

## [1] TRUE

methods(class = "lm")

##  [1] add1.lm*           alias.lm*
##  [3] anova.lm           case.names.lm*
##  [5] confint.lm*        cooks.distance.lm*
##  [7] deviance.lm*       dfbeta.lm*
##  [9] dfbetas.lm*        drop1.lm*
## [11] dummy.coef.lm*     effects.lm*
## [13] extractAIC.lm*     family.lm*
## [15] formula.lm*        hatvalues.lm
## [17] influence.lm*      kappa.lm
## [19] labels.lm*         logLik.lm*
## [21] model.frame.lm     model.matrix.lm
## [23] nobs.lm*           plot.lm
## [25] predict.lm         print.lm
## [27] proj.lm*           qr.lm*
## [29] residuals.lm       rstandard.lm
## [31] rstudent.lm        simulate.lm*
## [33] summary.lm         variable.names.lm*
## [35] vcov.lm*
##
##    Non-visible functions are asterisked
```

Often S3 classes inherit from lists (i.e., are special cases of lists), so you can obtain components of the object using the $ operator.

Creating our own class We can create an object with a new class as follows:

```
me <- list(firstname = "Chris", surname = "Paciorek",
    age = NA)
class(me) <- "indiv"  # there is already a 'person' class in R
```

Actually, if we want to create a new class that we'll use again, we want to create a *constructor* function that initializes new individuals:

```
indiv <- function(firstname = NA, surname = NA, age = NA) {
    # constructor for 'indiv' class
    obj <- list(firstname = firstname, surname = surname,
        age = age)
    class(obj) <- "indiv"
    return(obj)
}
me <- indiv("Chris", "Paciorek")
```

For those of you used to more formal OOP, the following is probably disconcerting:

```
class(me) <- "silly"
class(me) <- "indiv"
```

Methods The real power of OOP comes from defining *methods*. For example,

```
mod <- lm(yc ~ x)
summary(mod)
gmod <- glm(yb ~ x, family = "binomial")
summary(gmod)
```

Here *summary()* is a generic method (or generic function) that, based on the type of object given to it (the first argument), dispatches a class-specific function (method) that operates on the object. This is convenient for working with objects using familiar functions. Consider the generic methods *plot()*, *print()*, *summary()*, *'['*, and others. We can look at a function and easily see that it is a generic method. We can also see what classes have methods for a given generic method.

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x3f67300>
## <environment: namespace:base>

methods(mean)

## [1] mean.data.frame mean.Date
## [3] mean.default    mean.difftime
## [5] mean.POSIXct    mean.POSIXlt
```

In many cases there will be a default method (here, *mean.default()*), so if no method is defined for the class, R uses the default. Sidenote: arguments to a generic method are passed along to the selected method by passing along the calling environment.

We can define new generic methods:

```
summarize <- function(object, ...) UseMethod("summarize")
```

Once *UseMethod()* is called, R searches for the specific method associated with the class of *object* and calls that method, without ever returning to the generic method. Let's try this out on our *indiv* class. In reality, we'd write either *summary.indiv()* or *print.indiv()* (and of course the generics for *summary* and *print* already exist) but for illustration, I wanted to show how we would write both the generic and the specific method.

```
summarize.indiv <- function(object) return(with(object,
    cat("Individual of age ", age, " whose name is ",
        firstname, " ", surname, ".\n", sep = "")))
summarize(me)

## Individual of age NA whose name is Chris Paciorek.
```

Note that the *print()* function is what is called when you simply type the name of the object, so we can have object information printed out in a structured way. Recall that the output when we type the name of an *lm* object is NOT simply a regurgitation of the elements of the list - rather *print.lm()* is called.

Similarly, when we used `print(object.size(x))` we were invoking the *object_size*-specific print method which gets the value of the size and then formats it. So there's actually a fair amount going on behind the scenes.

Surprisingly, the *summary()* method generally doesn't actually print out information; rather it computes things not stored in the original object and returns it as a new class (e.g., class *summary.lm*), which is then automatically printed, per my comment above, using *print.summary.lm()*, unless one assigns it to a new object. Note that *print.summary.lm()* is hidden from user view.

```
out <- summary(mod)
out
print(out)
getS3method(f = "print", class = "summary.lm")
```

More on inheritance As noted with *lm* and *glm* objects, we can assign more than one class to an object. Here *summarize()* still works, even though the primary class is *BerkeleyIndiv*.

```
class(me) <- c("BerkeleyIndiv", "indiv")
summarize(me)

## Individual of age NA whose name is Chris Paciorek.
```

The classes should nest within one another with the more specific classes to the left, e.g., here a *BerkeleyIndiv* would have some additional objects on top of those of an individual, perhaps *CalnetID*, and perhaps additional or modified methods. *BerkeleyIndiv* inherits from *indiv*, and R uses methods for the first class before methods for the next class(es), unless no such method is defined for the first class. If no methods are defined for any of the classes, R looks for *method.default()*, e.g., *print.default()*, *plot.default()*, etc..

Class-specific operators We can also use operators with our classes. The following example will be a bit silly (it would make more sense with a class that is a mathematical object) but indicates the power of having methods.

```
methods(`+`)

## [1] +.Date   +.POSIXt

`+.indiv` <- function(object, incr) {
    object$age <- object$age + incr
    return(object)
}
old.me <- me + 15
```

Class-specific replacement functions We can use replacement functions with our classes.

This is again a bit silly but we could do the following. We need to define the generic replacement function and then the class-specific one.

```
`age<-` <- function(x, ...) UseMethod("age<-")
`age<-.indiv` <- function(object, value) {
    object$age <- value
    return(object)
}
age(old.me) <- 60
```

Why use class-specific methods? We could have implemented different functionality (e.g., for *summary()*) for different objects using a bunch of *if* statements (or *switch()*) to figure out what class of object is the input, but then we need to have all that checking. Furthermore, we don't control the *summary()* function, so we would have no way of adding the additional conditions in a big if-else statement. The OOP framework makes things *extensible*, so we can build our own new functionality on what is already in R.

Final thoughts Recall the *Date* class we discussed in Unit 4. This is an example of an S3 class, with methods such as *julian()*, *weekdays()*, etc.

Challenge: how would you get R to quit immediately, without asking for any more information, when you simply type '*q*' (no parentheses!)?

What we've just discussed are the old-style R (and S) object orientation, called S3 methods. The new style is called S4 and we'll discuss it next. S3 is still commonly used, in part because S4 can be slow (or at least it was when I last looked into this a few years ago). S4 is more structured than S3.

4.2 S4 approach

S4 methods are used a lot in *bioconductor*. They're also used in *lme4*, among other packages. Tools for working with S4 classes are in the *methods* package.

Note that components of S4 objects are obtained as `object@component` so they do not use the usual list syntax. The components are called *slots*, and there is careful checking that the slots are specified and valid when a new object of a class is created. You can use the *prototype* argument to *setClass()* to set default values for the slots. The default constructor (the function is *initialize()*), but you can modify. One can create methods for operators and for replacement functions too. For

S4 classes, there is a default method invoked when *print()* is called on an object in the class (either explicitly or implicitly) - the method is actually called *show()* and it can also be modified. Let's reconsider our *indiv* class example in the S4 context.

```
library(methods)
setClass("indiv", representation(name = "character",
    age = "numeric", birthday = "Date"))
me <- new("indiv", name = "Chris", age = 20, birthday = as.Date("91-08-03"))
# next notice the missing age slot
me <- new("indiv", name = "Chris", birthday = as.Date("91-08-03"))
# finally, apparently there's not a default
# object of class Date
me <- new("indiv", name = "Chris", age = 20)

## Error:  invalid class "indiv" object:  invalid object for slot "birthday"
in class "indiv":  got class "S4", should be or extend class "Date"

me

## An object of class "indiv"
## Slot "name":
## [1] "Chris"
##
## Slot "age":
## numeric(0)
##
## Slot "birthday":
## [1] "91-08-03"

me@age <- 60
```

S4 methods are designed to be more structured than S3, with careful checking of the slots.

```
setValidity("indiv", function(object) {
    if (!(object@age > 0 && object@age < 130))
        return("error: age must be between 0 and 130")
    if (length(grep("[0-9]", object@name)))
        return("error: name contains digits")
```

```
    return(TRUE)
    # what other validity check would make sense
    # given the slots?
})

## Class "indiv" [in ".GlobalEnv"]
##
## Slots:
##
## Name:       name       age  birthday
## Class: character   numeric      Date

me <- new("indiv", name = "5z%a", age = 20, birthday = as.Date("91-08-03"))

## Error:  invalid class "indiv" object:  error:  name contains digits

me <- new("indiv", name = "Z%a B''*", age = 20, birthday = as.Date("91-08-0
me@age <- 150  # so our validity check is not foolproof
```

To deal with this latter issue of the user mucking with the slots, it's recommended when using OOP that slots only be accessible through methods that operate on the object, e.g., a *setAge()* method, and then check the validity of the supplied age within *setAge()*.

Here's how we create generic and class-specific methods. Note that in some cases the generic will already exist.

```
# generic method
setGeneric("voter", function(object, ...) {
    standardGeneric("voter")
})

## [1] "voter"

# class-specific method
voter.indiv = function(object) {
    if (object@age > 17) {
        cat(object@name, "is of voting age.\n")
    } else cat(object@name, "is not of voting age.\n")
}
setMethod(voter, signature = c("indiv"), definition = voter.indiv)
```

```
## [1] "voter"
## attr(,"package")
## [1] ".GlobalEnv"

voter(me)

## Z%a B''* is of voting age.
```

We can have method signatures involve multiple objects. Here's some syntax where we'd fill in the function body with appropriate code - perhaps the plus operator would create a child.

```
  setMethod(`+`, signature = c("indiv", "indiv"),
definition = function(indiv1, indiv2) { }
```

As with S3, classes can inherit from one or more other classes. Chambers calls the class that is being inherited from a *superclass*.

```
setClass("BerkeleyIndiv", representation(CalID = "character"),
    contains = "indiv")
me <- new("BerkeleyIndiv", name = "Chris", age = 20,
    birthday = as.Date("91-08-03"), CalID = "01349542")
voter(me)

## Chris is of voting age.

is(me, "indiv")

## [1] TRUE
```

For a more relevant example suppose we had spatially-indexed time series. We could have a time series class, a spatial location class, and a "location time series" class that inherits from both. Be careful that there are not conflicts in the slots or methods from the multiple classes. For conflicting methods, you can define a method specific to the new class to deal with this. Also, if you define your own *initialize()* method, you'll need to be careful that you account for any initialization of the superclass(es) and for any classes that might inherit from your class (see help on *new()* and Chambers, p. 360).

You can inherit from other S4 classes (which need to be defined or imported into the environment in which your class is created), but not S3 classes. You can inherit (at most one) of the basic R types, but not environments, symbols, or other non-standard types. You can use S3 classes in slots, but this requires that the S3 class be declared as an S4 class. To do this, you create S4 versions of

S3 classes use *setOldClass()* - this creates a virtual class. This has been done, for example, for the *data.frame* class:

```
showClass("data.frame")

## Class "data.frame" [package "methods"]
##
## Slots:
##
## Name:                .Data               names
## Class:                list           character
##
## Name:            row.names            .S3Class
## Class: data.frameRowLabels           character
##
## Extends:
## Class "list", from data part
## Class "oldClass", directly
## Class "vector", by class "list", distance 2
```

You can use *setClassUnion()* to create what Adler calls *superclass* and what Chambers calls a *virtual class* that allows for methods that apply to multiple classes. So if you have a person class and a pet class, you could create a "named lifeform" virtual class that has methods for working with name and age slots, since both people and pets would have those slots. You can't directly create an object in the virtual class.

4.3 Reference Classes

Reference classes are a new construct in R. They are classes somewhat similar to S4 that allow us to access their fields by reference. Importantly, they behave like pointers (the fields in the objects are 'mutable'). Let's work through an example where we set up the fields of the class (like S4 slots) and class methods, including a constructor. Note that one cannot add fields to an already existing class.

Here's the initial definition of the class.

```
tsSimClass <- setRefClass("tsSimClass", fields = list(n = "numeric",
    times = "numeric", corMat = "matrix", lagMat = "matrix",
```

```
    corParam = "numeric", U = "matrix", currentU = "logical"),
    methods = list(initialize = function(times = 1:10,
        corParam = 1, ...) {
        # we seem to need default values for the copy()
        # method to function properly
        require(fields)
        times <<- times  # field assignment requires using <<-
        n <<- length(times)
        corParam <<- corParam
        currentU <<- FALSE
        calcMats()
        callSuper(...)  # calls initializer of base class (envRefClass)
    }, calcMats = function() {
        # Python-style doc string
        " calculates correlation matrix and Cholesky factor "
        lagMat <- rdist(times)  # local variable
        corMat <<- exp(-lagMat/corParam)  # field assignment
        U <<- chol(corMat)  # field assignment
        cat("Done updating correlation matrix and Cholesky factor\n")
        currentU <<- TRUE
    }, changeTimes = function(newTimes) {
        times <<- newTimes
        calcMats()
    }, show = function() {
        # 'print' method
        cat("Object of class 'tsSimClass' with ", n,
            " time points.\n", sep = "")
    }))

## Warning:  Local assignment to field name will not change the field:
## lagMat <- rdist(times)
## Did you mean to use "«-"?  ( in method "calcMats" for class "tsSimClass"
```

We can add methods after defining the class.

```
tsSimClass$methods(list(simulate = function() {
    " simulates random processes from the model "
    if (!currentU) calcMats()
    return(crossprod(U, rnorm(n)))
}))
```

Now let's see how we would use the class.

```
master <- tsSimClass$new(1:100, 10)
master
tsSimClass$help("calcMats")
devs <- master$simulate()
plot(master$times, devs, type = "l")
mycopy <- master
myDeepCopy <- master$copy()
master$changeTimes(seq(0, 1, length = 100))
mycopy$times[1:5]
myDeepCopy$times[1:5]
```

A few additional points:

- As we just saw, a copy of an object is just a pointer to the original object, unless we explicitly invoke the *copy()* method.
- As with S3 and S4, classes can inherit from other classes. E.g., if we had a *simClass* and we wanted the *tsSimClass* to inherit from it:

  ```
  setRefClass("tsSimClass", contains = "simClass")
  ```

  - We can call a method inherited from the superclass from within a method of the same name with *callSuper(...)*, as we saw for our *initialize()* method.
- If we need to refer to a field or change a field we can do so without hard-coding the field name as:

```
master$field("times")[1:5]
# the next line is dangerous in this case, since
# currentU will no longer be accurate
master$field("times", 1:10)
```

- Note that reference classes have Python style doc strings. We get help on a class with *class$help()*, e.g. `tsSimClass$help()`. This prints out information, including the doc strings.
- If you need to refer to the entire object within an object method, you refer to it as *.self*. E.g., with our *tsSimClass* object, `.self$U` would refer to the Cholesky factor. This is sometimes necessary to distinguish a class field from an argument to a method.

5 Creating and working in an environment

We've already talked extensively about the environments that R creates. Occasionally you may want to create an environment in which to store objects.

```
e <- new.env()
assign("x", 3, envir = e)  # same as e$x = 3
e$x

## [1] 3

get("x", envir = e, inherits = FALSE)

## [1] 3

# the FALSE avoids looking for x in the enclosing
# environments
e$y <- 5
objects(e)

## [1] "x" "y"

rm("x", envir = e)
parent.env(e)

## <environment: R_GlobalEnv>
```

Before the existence of Reference Classes, using an environment was one way to pass objects by reference, avoiding having to re-assign the output. Here's an example where we iteratively update a random walk.

```
myWalk <- new.env()
myWalk$pos = 0
nextStep <- function(walk) walk$pos = walk$pos + sample(c(-1,
    1), size = 1)
nextStep(myWalk)
```

We can use *eval()* to evaluate some code within a specified environment. By default, it evaluates in the result of *parent.frame()*, which amounts to evaluating in the frame from which *eval()* was called. *evalq()* avoids having to use *quote()*.

```
eval(quote(pos <- pos + sample(c(-1, 1), 1)), envir = myWalk)
evalq(pos <- pos + sample(c(-1, 1), 1), envir = myWalk)
```

6 Computing on the language

6.1 The R interpreter

Parsing When you run R, the R interpreter takes the code you type or the lines of code that are read in a batch session and parses each statement, translating the text into functional form. It substitutes objects for the symbols (names) that represent those objects and evaluates the statement, returning the resulting object. For complicated R code, this may be recursive.

Since everything in R is an object, the result of parsing is an object that we'll be able to investigate, and the result of evaluating the parsed statement is an object.

We'll see more on parsing in the next section.

.Primitive and .Internal Some functionality is implemented internally within the C implementation that lies at the heart of R. If you see *.Internal* or *.Primitive*, in the code of a function, you know it's implemented internally (and therefore generally very quickly). Unfortunately, it also means that you don't get to see R code that implements the functionality, though Chambers p. 465 describes how you can look into the C source code. Basically you need to download the source code for the relevant package off of CRAN.

```
plot.xy

## function (xy, type, pch = par("pch"), lty = par("lty"), col = par("col"),
##     bg = NA, cex = 1, lwd = par("lwd"), ...)
```

```
## .Internal(plot.xy(xy, type, pch, lty, col, bg, cex, lwd, ...))
## <bytecode: 0x1be2780>
## <environment: namespace:graphics>

print(`%*%`)

## function (x, y)  .Primitive("%*%")
```

6.2 Parsing code and understanding language objects

R code is just text and we can actually write R code that will create or manipulate R code. We can then evaluate that R code using *eval()*.

quote() will parse R code, but not evaluate it. This allows you to work with the code rather than the results of evaluating that code. The *print()* method for language objects is not very helpful! But we can see the parsed code by treating the result as a list.

```
obj <- quote(if (x > 1) "orange" else "apple")
as.list(obj)

## [[1]]
## `if`
##
## [[2]]
## x > 1
##
## [[3]]
## [1] "orange"
##
## [[4]]
## [1] "apple"

class(obj)

## [1] "if"

weirdObj <- quote(`if`(x > 1, 'orange', 'apple'))
identical(obj, weirdObj)

## [1] TRUE
```

Recall that to access symbols that involve special syntax (such as special characters), you use backquotes.

Officially, the name that you assign to an object (including functions) is a *symbol*.

```
x <- 3
typeof(quote(x))

## [1] "symbol"
```

We can create an *expression* object that contains R code as

```
myExpr <- expression(x <- 3)
eval(myExpr)
typeof(myExpr)

## [1] "expression"
```

The difference between *quote()* and *expression()* is basically that *quote()* works with a single statement, while expression() can deal with multiple statements, returning a list-like object of parsed statements. Both of them parse R code.

Table 1 shows the *language* objects in R; note that there are three classes of language objects: *expressions*, *calls*, and *names*.

	Example syntax to create	Class	Type
functions	function() {}	function	closure
object names	quote(x)	name	symbol (language)
expressions	expression(x <- 3)	expression	expression (language)
function calls	quote(f())	call	language
if statements	quote(if(x < 3) y=5)	if (call)	language
for statement	quote(for(i in 1:5) {})	for (call)	language
assignments	quote(x <- 3)	<- (call)	language
operators	quote(3 + 7)	call	language

Basically any standard function, operator, *if* statement, *for* statement, assignment, etc. are function calls and inherit from the *call* class.

Objects of type language are not officially lists, but they can be queried as such. You can convert between language objects and lists with *as.list()* and *as.call()*.

An official expression is one or more syntactically correct R statements. When we use *quote()*, we're working with a single statement, while *expression()* will create a list of separate statements

(essentially separate call objects). I'm trying to use the term *statement* to refer colloquially to R code, rather than using the term *expression*, since that has formal definition in this context.

Let's take a look at some examples of language objects and parsing.

```
e0 <- quote(3)
e1 <- expression(x <- 3)
e1m <- expression({x <- 3; y <- 5})
e2 <- quote(x <- 3)
e3 <- quote(rnorm(3))
print(c(class(e0), typeof(e0)))

## [1] "numeric" "double"

print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

print(c(class(e1[[1]]), typeof(e1[[1]])))

## [1] "<-"       "language"

print(c(class(e1m), typeof(e1m)))

## [1] "expression" "expression"

print(c(class(e2), typeof(e2)))

## [1] "<-"       "language"

identical(e1[[1]], e2)

## [1] TRUE

print(c(class(e3), typeof(e3)))

## [1] "call"     "language"

e4 <- quote(-7)
print(c(class(e4), typeof(e4))) # huh? what does this imply?

## [1] "call"     "language"
```

```
as.list(e4)

## [[1]]
## `-`
##
## [[2]]
## [1] 7
```

We can evaluate language types using *eval()*:

```
rm(x)
eval(e1)
rm(x)
eval(e2)
e1mlist <- as.list(e1m)
e2list <- as.list(e2)
eval(as.call(e2list))
# here's how to do it if the language object is
# actually a list
eval(as.expression(e1mlist))
```

Now let's look in more detail at the components of R expressions. We'll be able to get a sense from this of how R evaluates code. We see that when R evaluates a parse tree, the first element says what function to use and the remaining elements are the arguments. But in many cases one or more arguments will themselves be call objects, so there's recursion.

```
e1 = expression(x <- 3)
# e1 is one-element list with the element an
# object of class '<-'
print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

e1[[1]]

## x <- 3

as.list(e1[[1]])
```

```
## [[1]]
## `<-`
##
## [[2]]
## x
##
## [[3]]
## [1] 3
```

```
lapply(e1[[1]], class)
```

```
## [[1]]
## [1] "name"
##
## [[2]]
## [1] "name"
##
## [[3]]
## [1] "numeric"
```

```
y = rnorm(5)
e3 = quote(mean(y))
print(c(class(e3), typeof(e3)))
```

```
## [1] "call"     "language"
```

```
e3[[1]]
```

```
## mean
```

```
print(c(class(e3[[1]]), typeof(e3[[1]])))
```

```
## [1] "name"   "symbol"
```

```
e3[[2]]
```

```
## y
```

```
print(c(class(e3[[2]]), typeof(e3[[2]])))
```

```
## [1] "name"   "symbol"

# we have recursion
e3 = quote(mean(c(12, 13, 15)))
as.list(e3)

## [[1]]
## mean
##
## [[2]]
## c(12, 13, 15)

as.list(e3[[2]])

## [[1]]
## c
##
## [[2]]
## [1] 12
##
## [[3]]
## [1] 13
##
## [[4]]
## [1] 15
```

6.3 Manipulating the parse tree

Of course since the parsed code is just an object, we can manipulate it, i.e., *compute on the language*:

```
out <- quote(y <- 3)
out[[3]] <- 4
eval(out)
y

## [1] 4
```

Here's another example:

```
e1 <- quote(4 + 5)
e2 <- quote(plot(x, y))
e2[[1]] <- `+`
eval(e2)

## [1] 7

e1[[3]] <- e2
e1

## 4 + .Primitive("+")(x, y)

class(e1[[3]])  # note the nesting

## [1] "call"

eval(e1)  # what should I get?

## [1] 11
```

We can also turn it back into standard R code, as a character, using *deparse()*, which turns the parse tree back into R code as text. *deparse()* is like *quote()* but it takes the code in the form of a string rather than an actual expression:

```
codeText <- deparse(out)
parsedCode <- parse(text = codeText)  # works like quote except on the code
eval(parsedCode)
deparse(quote(if (x > 1) "orange" else "apple"))

## [1] "if (x > 1) \"orange\" else \"apple\""
```

Note that the quotes have been escaped since they're inside a string.

It can be very useful to be able to convert names of objects as text to names that R interprets as symbols referring to objects:

```
x3 <- 7
i <- 3
as.name(paste("x", i, sep = ""))
```

```
## x3

eval(as.name(paste("x", i, sep = "")))

## [1] 7
```

6.4 Parsing replacement expressions

Let's consider replacement expressions.

```
animals = c("cat", "dog", "rat", "mouse")
out1 = quote(animals[4] <- "rat")
out2 = quote(animals[4] <- "rat")
out3 = quote(`[<-`(animals, 4, "rat"))
as.list(out1)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
## [[3]]
## [1] "rat"

as.list(out2)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
## [[3]]
## [1] "rat"

identical(out1, out2)
```

```
## [1] TRUE
```

```
as.list(out3)
```

```
## [[1]]
## `[<-`
##
## [[2]]
## animals
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] "rat"
```

```
identical(out1, out3)
```

```
## [1] FALSE
```

```
typeof(out1[[2]])  # language
```

```
## [1] "language"
```

```
class(out1[[2]])  # call
```

```
## [1] "call"
```

The parse tree for *out3* is different than those for *out1* and *out2*, but when *out3* is evaluated the result is the same as for *out1* and *out2*:

```
eval(out1)
animals
```

```
## [1] "cat" "dog" "rat" "rat"
```

```
animals[4] = "dog"
eval(out3)
```

```
## [1] "cat" "dog" "rat" "rat"
```

```
animals  # both do the same thing

## [1] "cat" "dog" "rat" "rat"
```

Why? When R evaluates a call to '<-', if the first argument is a name, then it does the assignment, but if the first argument (i.e. what's on the left-hand side of the "assignment") is a call then it calls the appropriate replacement function. The second argument (the value being assigned) is evaluated first. Ultimately in all of these cases, the replacement function is used.

6.5 substitute()

The substitute function acts like *quote()*:

```
identical(quote(z <- x^2), substitute(z <- x^2))

## [1] TRUE
```

But if you also pass *substitute()* an environment, it will replace symbols with their object values in that environment.

```
e <- new.env()
e$x <- 3
substitute(z <- x^2, e)

## z <- 3^2
```

This can do non-sensical stuff:

```
e$z <- 5
substitute(z <- x^2, e)

## 5 <- 3^2
```

Let's see a practical example of substituting for variables in statements:

```
plot(x = rnorm(5), y = rgamma(5, 1)) # how does plot get the axis label names?
```

In the *plot()* function, you can see this syntax:

```
xlabel <- if(!missing(x)) deparse(substitute(x))
```

So what's going on is that within *plot.default()*, it substitutes in for '*x*' with the statement that was

passed in as the *x* argument, and then uses *deparse()* to convert to character. The fact that *x* still has *rnorm(5)* associated with it rather than the five numerical values from evaluating *rnorm()* has to do with lazy evaluation and promises. Here's the same idea in action in a stripped down example:

```
f <- function(obj) {
    objName <- deparse(substitute(obj))
    print(objName)
}
f(y)

## [1] "y"
```

More generally, we can substitute into *expression* and *call* objects by providing a named list (or an environment) - the substition happens within the context of this list.

```
substitute(a + b, list(a = 1, b = quote(x)))

## 1 + x
```

Things can get intricate quickly:

```
e1 <- quote(x + y)
e2 <- substitute(e1, list(x = 3))
```

The problem is that *substitute()* doesn't evaluate its first argument, "*e1*", so it can't replace the parsed elements in *e1*. Instead, we'd need to do the following, where we force the evaluation of *e1*:

```
e2 <- substitute(substitute(e, list(x = 3)), list(e = e1))
substitute(substitute(e, list(x = 3)), list(e = e1))

## substitute(x + y, list(x = 3))

# so e1 is substituted as an evaluated object,
# which then allows for substitution for 'x'
e2

## substitute(x + y, list(x = 3))

eval(e2)

## 3 + y
```

If this subsection is confusing, let me assure you that it has confused me too. The indirection going on here is very involved.

6.6 Final thoughts

Challenge: figure out how a *for* loop is parsed in R. See how a *for* loop with one statement within the loop differs from one with two or more statements.

We'll see *expression()* again when we talk about inserting mathematical notation in plots.

7 Programming concepts

A few thoughts on what we've learned that is germane beyond R include:

- variable types
- passing by reference and by value
- variable scope
- the call stack
- flow control
- object-oriented programming
- matrix storage concepts
- parsing

If you go to learn other languages, these and other concepts that we've covered should be helpful.

Parallel Processing

October 17, 2012

Reference:

Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U (2009). "State of the Art in Parallel Computing with R." Journal of Statistical Software, 31(1), 127.

http://www.jstatsoft.org/v31/i01

That reference is a bit old, and I've pulled material from a variety of sources, often presentations, and not done a good job documenting this, so I don't have a good list of references for this topic.

1 Computer architecture

Computers now come with multiple processors for doing computation. Basically, physical constraints have made it harder to keep increasing the speed of individual processors, so the industry is now putting multiple processing units in a given computer and trying/hoping to rely on implementing computations in a way that takes advantage of the multiple processors.

Everyday personal computers often have more than one processor (more than one chip) and on a given processor, often have more than one core (multi-core). A multi-core processor has multiple processors on a single computer chip. On personal computers, all the processors and cores share the same memory.

Supercomputers and computer clusters generally have tens, hundreds, or thousands of 'nodes', linked by a fast local network. Each node is essentially a computer with its own processor(s) and memory. Memory is local to each node (distributed memory). One basic principle is that communication between a processor and its memory is much faster than communication between processors with different memory. An example of a modern supercomputer is the Jaguar supercomputer at Oak Ridge National Lab, which has 18,688 nodes, each with two processors and each processor with 6 cores, giving 224,256 total processing cores. Each node has 16 Gb of memory for a total of 300 Tb.

There is little practical distinction between multi-processor and multi-core situations. The main issue is whether processes share memory or not. In general, I won't distinguish between cores and processors.

1.1 Distributed vs. shared memory

There are two basic flavors of parallel processing (leaving aside GPUs): distributed memory and shared memory. With shared memory, multiple processors (which I'll call cores) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. You can think of each node as a separate computer connected by a fast network.

Parallel programming for distributed memory parallelism requires passing messages between the different nodes. The standard protocol for doing this is MPI, of which there are various versions, including *openMPI*. The R package *Rmpi* implements MPI in R.

With shared memory parallelism, each core is accessing the same memory so there is no need to pass messages. But one needs to be careful that activity on different cores doesn't mistakenly overwrite places in memory that are used by other cores. We'll focus on shared memory parallelism here in this unit, though *Rmpi* will come up briefly.

1.2 Graphics processing units (GPUs)

GPUs were formerly a special piece of hardware used by gamers and the like for quickly rendering (displaying) graphics on a computer. They do this by having hundreds of processing units and breaking up the computations involved in an embarrassingly parallel fashion (i.e., without inter-processor communication). For particular tasks, GPUs are very fast. Nowadays GPUs are generally built onto PC motherboards whereas previously they were on a video card.

Researchers (including some statisticians) are increasingly looking into using add-on GPUs to do massively parallel computations with inexpensive hardware that one can easily add to one's existing machine. This has promise. However, some drawbacks in current implementations include the need to learn a specific programming language (similar to C) and limitations in terms of transferring data to the GPU and holding information in the memory of the GPU.

1.3 Cloud computing

Amazon (through its EC2 service) and other companies (Google and Microsoft now have offerings) offer computing through the cloud. The basic idea is that they rent out their servers on a pay-as-you-go basis. You get access to a virtual machine that can run various versions of Linux or Microsoft Windows server and where you choose the number of processing cores you want. You

configure the virtual machine with the applications, libraries, and data you need and then treat the virtual machine as if it were a physical machine that you log into as usual.

2 Parallelization

2.1 Overview

A lot of parallel processing follows a master-slave paradigm. There is one master process that controls one or more slave processes. The master process sends out tasks and data to the slave processes and collects results back.

One comment about parallelized code is that it can be difficult to debug because communication problems can occur in addition to standard bugs. Debugging may require some understanding of the communication that goes on between processors. If you can first debug your code on one processor, that can be helpful.

2.2 Threading

One form of parallel processing is threading. Here an algorithm is implemented across multiple "light-weight" processes called threads in a shared memory situation. Threads are multiple paths of execution within a single process. One can write one's own code to make use of threading, e.g., using the *openMP* protocol for C/C++/Fortran. For our purpose we'll focus on using pre-existing code or libraries that are threaded, specifically threaded versions of the BLAS.

In R, the basic strategy is to make sure that the R installation uses a threaded BLAS so that standard linear algebra computations are done in a threaded fashion. We can do this by linking R to a threaded BLAS shared object library (i.e., a *.so* file). Details can be found in Section A.3.1.5 of the R administration manual or talk to your system administrator.

Note that in R, the threading only helps with linear algebra operations. In contrast, Matlab uses threading for a broader range of calculations.

2.2.1 The BLAS

The BLAS is the library of basic linear algebra operations (written in Fortran or C). A fast BLAS can greatly speed up linear algebra relative to the default BLAS on a machine. Some fast BLAS libraries are Intel's MKL, AMD's ACML, and the open source (and free) openBLAS (formerly GotoBLAS). All of these BLAS libraries are now threaded - if your computer has multiple cores and there are free resources, your linear algebra will use multiple cores, provided your program is linked against the specific BLAS. Using *top* (on a machine other than the cluster), you'll see

the process using more than 100% of CPU. inconceivable! The default BLAS on the SCF Linux compute servers is openBLAS. On Macs, the threaded BLAS is generally implemented by default, so you don't need to do anything.

```
X <- matrix(rnorm(8000^2), 8000)
system.time({
    X <- crossprod(X)  # X^t X produces pos.def. matrix
    U <- chol(x)
})  # U^t U = X
# exit R, execute: 'export OMP_NUM_THREADS=1', and restart R
system.time({
    X <- crossprod(X)
    U <- chol(X)
})
```

2.2.2 Fixing the number of threads (cores used)

In general, if you want to limit the number of threads used, you can set the OMP_NUM_THREADS environment variable, as indicated in the previous example. This can be used in the context of R or C code that uses BLAS or your own threaded C code, but this does not work with Matlab. In the UNIX bash shell, you'd do this as follows (e.g. to limit to 3 cores) (do this before starting R):

```
export OMP_NUM_THREADS=3 # or "setenv OMP_NUM_THREADS 1" if using csh/tcsh
```

2.3 Embarrassingly parallel (EP) problems

An EP problem is one that can be solved by doing independent computations as separate processes without communication between the processes. You can get the answer by doing separate tasks and then collecting the results. Examples in statistics include

1. simulations with many independent replicates
2. bootstrapping
3. stratified analyses

The standard setup is that we have the same code running on different datasets. (Note that different processes may need different random number streams, as we will discuss in the Simulation Unit.)

To do parallel processing in this context, you need to have control of multiple processes. In a system with a queueing setup, this will generally mean requesting access to a certain number of processors and then running your job in such a way that you use multiple processors.

In general, except for some modest overhead, an EP problem can be solved with $1/p$ the amount of time for the non-parallel implementation, given p processors. This gives us a speedup of p, which is called linear speedup (basically anytime the speedup is of the form cp for some constant p).

One difficulty is load balancing. We'd like to make sure each slave process finishes at the same time. Often we can give each process the same amount of work, but if we have a mix of faster and slower processors, things become more difficult. To the extent it is possible to break up a job into many small tasks and have processors start new tasks as they finish off old tasks, this can be effective, but may involve some parallel programming.

In the next section, we'll see a few approaches in R for dealing with EP problems.

2.4 Parallelization with communication

If we do not have an EP problem, we have one that involves some sort of serial calculation. As a result, different processes need to communicate with each other. There are standard protocols for such communication, with *MPI* being most common. You can use C libraries that implement these protocols. While MPI has many functions, a core of 6-10 functions (basic functions for functionality such as sending and receiving data between processes - either master-slave or slave-slave) are what we mostly need.

R provides the *Rmpi* library, which allows you to do message passing in R. It has some drawbacks, but may be worth exploring if you have a non-EP problem and don't want to learn C. Installing Rmpi may be tricky and on institutional machines will require you talk to your systems administrator. *Rmpi* is a basic building block for other parallel processing functionality such as *foreach* and *SNOW*.

For non-EP problems, the primary question is how the speed of the computation scales with p. This will generally be much worse than $1/p$ and as p increases, if communication must increase as well, then the speedup can be much worse.

3 Explicit parallel code in R

Before we get into some functionality, let's define some terms more explicitly.

- *threading*: multiple paths of execution within a single process; the OS sees the threads as a single process, but one can think of them as 'lightweight' processes

- *forking*: child processes are spawned that are identical to the parent, but with different process IDs and their own memory
- *sockets*: some of R's parallel functionality involves creating new R processes and communicating with them via a communication technology called sockets

3.1 *foreach*

A simple way to exploit parallelism in R when you have an EP problem is to use the *foreach* package to do a for loop in parallel. For example, bootstrapping, random forests, simulation studies, cross-validation and many other statistical methods can be handled in this way. You would not want to use *foreach* if the iterations were not independent of each other.

The *foreach* package provides a *foreach* command that allows you to do this easily. *foreach* can use a variety of parallel "back-ends". It can use *Rmpi* to access cores in a distributed memory setting or (our focus here) the *parallel* or *multicore* packages to use shared memory cores. When using *parallel* or *multicore* as the back-end, you should see multiple processes (as many as you registered; ideally each at 100%) when you look at *top*. The multiple processes are generally created by forking.

```
require(parallel)  # one of the core R packages
require(doParallel)
# require(multicore); require(doMC) # alternative to parallel/doParallel
# require(Rmpi); require(doMPI) # when Rmpi is available as the back-end
library(foreach)
library(iterators)
taskFun <- function() {
    mn <- mean(rnorm(1e+07))
    return(mn)
}
nCores <- 4
registerDoParallel(nCores)
# registerDoMC(nCores) # alternative to registerDoParallel
#
# cl <- startMPIcluster(nCores); registerDoMPI(cl) # when using Rmpi as
# the back-end
out <- foreach(i = 1:100, .combine = c) %dopar% {
    cat("Starting ", i, "th job.\n", sep = "")
```

```
    outSub <- taskFun()
    cat("Finishing ", i, "th job.\n", sep = "")
    outSub  # this will become part of the out object
}
```

The result of *foreach* will generally be a list, unless *foreach* is able to put it into a simpler R object. Here I've explicitly told foreach to combine the results with *c()* (*cbind()* and *rbind()* are other common choices), but it will often be smart enough to figure it out on its own. Note that *foreach* also provides some additional functionality for collecting and managing the results that mean that you don't have to do some of the bookkeeping you would need to do if writing your own for loop.

You can debug by running serially using *%do%* rather than *%dopar%*.

Note that you may need to load packages within the *foreach* code block to ensure a package is available to all of the calculations.

Warning: There is some sort of conflict between foreach and the threaded BLAS on the SCF Linux compute servers, so before running an R job that does linear algebra within a call to *foreach*, you may need to set *OMP_NUM_THREADS* to 1 to prevent the BLAS from doing threaded calculations. Hopefully we'll be able to fix this in the future.

Caution: Note that I didn't pay any attention to possible danger in generating random numbers in separate processes. The developers of *foreach* are aware of this issue, but I can't tell from the documentation how they handle it. More on an approach that is explicit about this in the unit on simulation.

3.2 Parallel apply and vectorization (parallel package)

The *parallel* package has the ability to parallelize the various *apply()* functions (apply, lapply, sapply, etc.) and parallelize vectorized functions. The *multicore* package also has this ability and *parallel* is built upon *multicore*. *parallel* is a core R package so we'll explore the functionality in that setting. Here's the vignette for the parallel package – it's hard to find because *parallel* is not listed as one of the contributed packages on CRAN.

First let's consider parallel apply.

```
require(parallel)
nCores <- 4
### using sockets
###
### ?clusterApply
```

```
cl <- makeCluster(nCores)  # by default this uses sockets
nSims <- 60
testFun <- function(i) {
    mn <- mean(rnorm(1e+06))
    return(mn)
}
# if the processes need objects (x and y, here) from the master's
# workspace: clusterExport(cl, c('x', 'y'))
system.time(res <- parSapply(cl, 1:nSims, testFun))
system.time(res2 <- sapply(1:nSims, testFun))
myList <- as.list(1:nSims)
res <- parLapply(cl, myList, testFun)
### using forking
system.time(res <- mclapply(seq_len(nSims), testFun, mc.cores = nCores))
```

Now let's consider parallel evaluation of a vectorized function.

```
require(parallel)
nCores <- 4
cl <- makeCluster(nCores)
library(fields)
ds <- runif(6e+06, 0.1, 10)
system.time(corVals <- pvec(ds, Matern, 0.1, 2, mc.cores = nCores))
system.time(corVals <- Matern(ds, 0.1, 2))
```

Note that some R packages can directly interact with the parallelization packages to work with multiple cores. E.g., the *boot* package can make use of the *multicore* package directly.

3.3 Explicit parallel programming in R: mcparallel and forking

Now let's discuss some functionality in which one more explicitly controls the parallelization.

3.3.1 Using mcparallel to dispatch blocks of code to different processes

First one can use *mcparallel()* in the *parallel* package to send different chunks of code to different processes.

```
library(parallel)
n <- 1e+07
system.time({
    p <- mcparallel(mean(rnorm(n)))
    q <- mcparallel(mean(rgamma(n, shape = 1)))
    res <- mccollect(list(p, q))
})
system.time({
    p <- mean(rnorm(n))
    q <- mean(rgamma(n, shape = 1))
})
```

3.3.2 Explicitly forking code in R

The *fork* package and *fork()* function in R provide an implementation of the UNIX *fork* system call for forking a process. Note that the code here does not handle passing information back from the child very well. One approach is to use sockets – the help page for *fork()* has a bit more information.

```
library(fork)
# mode 1
pid <- fork(slave = myfun)
# mode 2
{
    # this set of braces is REQUIRED when you don't pass a function
    # to the slave argument of fork()
    pid <- fork(slave = NULL)
    if (pid == 0) {
        cat("Starting child process execution.\n")
        tmpChild <- mean(rnorm(1e+07))
        cat("Result is ", tmpChild, "\n", sep = "")
        save(tmpChild, file = "child.RData")  # clunky
        cat("Finishing child process execution.\n")
        exit()
    } else {
        cat("Starting parent process execution.\n")
```

```
        tmpParent <- mean(rnorm(1e+07))
        cat("Finishing parent process execution.\n")
        wait(pid)  # wait til child is finished so can read
        # in updated child.RData below
    }
}
load("child.RData")  # clunky
print(c(tmpParent, tmpChild))
```

Note that if we were really running the above code, we'd want to be careful about the random number generation (RNG). As it stands, it will use the same random numbers in both child and parent processes.

Computer numbers

October 19, 2012

References:

- Gentle, Computational Statistics, Chapter 2.
- http://www.lahey.com/float.htm
- And for more gory detail, see Monahan, Chapter 2.

A quick note that R's version of scientific notation is *XeY*, which means $X \cdot 10^Y$.

A second note is that the concepts developed here apply outside of R, but we'll illustrate the principles of computer numbers using R.

1 Basic representations

Everything in computer memory or on disk is stored in terms of bits. A *bit* is essentially a switch than can be either on or off. Thus everything is encoded as numbers in base 2, i.e., 0s and 1s. 8 bits make up a *byte*. For information stored as plain text (ASCII), each byte is used to encode a single character. One way to represent a byte is to write it in hexadecimal, rather than as 8 0/1 bits. Since there are $2^8 = 256$ possible values in a byte, we can represent it more compactly as 2 base-16 numbers, such as "3e" or "a0" or "ba". A file format is nothing more than a way of interpreting the bytes in a file.

We can think about how we'd store an integer in terms of bytes. With two bytes, we could encode any value from $0, \ldots, 2^{16} - 1 = 65535$. This is an unsigned integer representation. To store negative numbers as well, we can use one bit for the sign, giving us the ability to encode -32767 - 32767 ($\pm 2^{15} - 1$). Note that in general, rather than be stored simply as the sign and then a number in base 2, integers are actually stored in a different binary encoding to facilitate arithmetic. Finally note that the set of computer integers is not closed under arithmetic:

```
a <- as.integer(3423333)
a * a

## Warning:  NAs produced by integer overflow

## [1] NA
```

and R reports an overflow (i.e., a result that is too large to be stored as an integer).

Real numbers (or *floating points*) use a minimum of 4 bytes, for single precision floating points. In general 8 bytes are used to represent real numbers and these are called *double precision floating points* or *doubles*. Let's see some examples in R of how much space different types of variables take up.

Let's see how this plays out in terms of memory use in R.

```
x <- rnorm(1e+05)
y <- 1:1e+05
z <- rep("a", 1e+05)
object.size(x)  # 800040 bytes

## 800040 bytes

object.size(y)  # 400040 bytes - so how many bytes per integer in R?

## 400040 bytes

object.size(z)  # 800088 bytes - hmm, what's going on here?

## 800088 bytes
```

We can easily calculate the number of megabytes (Mb) a vector of floating points (in double precision) will use as the number of elements times 8 (bytes/double) divided by 10^6 to convert from bytes to megabytes. (In some cases when considering computer memory, a megabyte is $1,048,576 = 2^{20}$ bytes so slightly different than 10^6). Finally, R has a special object that tells us about the characteristics of computer numbers on the machine that R is running on called *.Machine*. For example, *.Machine$integer.max* is $2147483647 = 2^{31} - 1$, which confirms how many bytes R is using for each integer (and that R is using a bit for the sign of the integer).

2 Floating point basics

2.1 Representing real numbers

Reals (also called floating points) are stored on the computer as an approximation, albeit a very precise approximation. As an example, with a double, the error in the distance from the earth to the sun is around a millimeter. However, we need to be very careful if we're trying to do a calculation that produces a very small (or very large number) and particularly when we want to see if numbers are equal to each other.

```
0.3 - 0.2 == 0.1
```

```
## [1] FALSE
```

```
0.3
```

```
## [1] 0.3
```

```
0.2
```

```
## [1] 0.2
```

```
0.1  # Hmmm...
```

```
## [1] 0.1
```

```
options(digits = 22)
a <- 0.3
b <- 0.2
a - b
```

```
## [1] 0.09999999999999997779554
```

```
0.1
```

```
## [1] 0.10000000000000000055511
```

```
1/3
```

```
## [1] 0.33333333333333333148296
```

Notice that we can represent the result accurately only up to the 16th decimal place. This suggests no need to show more than 16 decimal places and no need to print out any more when writing to a file. And of course, often we don't need anywhere near that many. *Machine epsilon* is the term used for indicating the accuracy of real numbers and it is defined as the smallest float, x, such that $1 + x \neq 1$:

```
1e-16 + 1

## [1] 1

1e-15 + 1

## [1] 1.0000000000000001110223

2e-16 + 1

## [1] 1.0000000000000000222045

.Machine$double.eps

## [1] 2.220446049250313080847e-16
```

Floating point refers to the decimal point (or radix point since we'll be working with base 2 and *decimal* relates to 10). Consider Avogadro's number in terms of scientific notation: $+6.023 \times 10^{23}$. A real number on a computer is stored in what is basically scientific notation:

$$\pm 0.d_1 d_2 \dots d_p \times b^e \tag{1}$$

where b is the base, e is an integer and $d_i \in \{0, \dots, b-1\}$. First, we need to choose the number of bits to represent e so that we can represent sufficiently large and small numbers. Second we need to choose the number of bits, p, to allocate to $d = d_1 d_2 \dots d_p$, which determines the accuracy of any computer representation of a real. The great thing about floating points is that we can represent numbers that range from incredibly small to very large while maintaining good precision. The floating point floats to adjust to the size of the number. Suppose we had only three digits to use and were in base 10. In floating point notation we can express $0.12 \times 0.12 = 0.0144$ as $(1.20 \times 10^{-1}) \times (1.20 \times 10^{-1}) = 1.44 \times 10^{-2}$, but if we had fixed the decimal point, we'd have $0.120 \times 0.120 = 0.014$ and we'd have lost a digit of accuracy.

More specifically, the actual storage of a number on a computer these days is generally as a

double in the form:

$$(-1)^S \times 1.d \times 2^{e-1023}$$

where the computer uses base 2, $b = 2$, because base-2 arithmetic is faster than base-10 arithmetic. The leading 1 normalizes the number; i.e., ensures there is a unique representation for a given computer number. This avoids representing any number in multiple ways, e.g., either $1 = 1.0 \times 2^0 = 0.1 \times 2^1 = 0.01 \times 2^2$. For a double, we have 8 bytes=64 bits. Consider our representation as (S, d, e) where S is the sign. The leading 1 is the *hidden bit*. In general e is represented using 11 bits ($2^{11} = 2048$), and the subtraction takes the place of having a sign bit for the exponent. This leaves $p = 52$ bits for d.

Let's consider what can be represented exactly:

```
0.1

## [1] 0.10000000000000055511

0.5

## [1] 0.5

0.25

## [1] 0.25

0.26

## [1] 0.26000000000000088818

1/32

## [1] 0.03125

1/33

## [1] 0.030303030303030387138
```

So why is 0.5 stored exactly and 0.1 not stored exactly? By analogy, consider the difficulty with representing 1/3 in base 10.

2.2 Overflow and underflow

The largest and smallest numbers we can represent are $2^{e_{\max}}$ and $2^{e_{\min}}$ where $e_{\max}$ and $e_{\min}$ are the smallest and largest possible values of the exponent. Let's consider the exponent and what we can infer about the range of possible numbers. With 11 bits for e, we can represent $\pm 2^{10} = \pm 1024$ different exponent values (see *.Machine$double.max.exp*) (why is *.Machine$double.min.exp* only -1022?). So the largest number we could represent is 2^{1024}. What is this in base 10?

```
log10(2^1024)  # whoops ... we've actually just barely overflowed

## [1] Inf

log10(2^1023)

## [1] 307.95368556425273709 46
```

We could have been smarter about that calculation: $\log_{10} 2^{1024} = \log_2 2^{1024} / \log_2 10 = 1024/3.32 \approx 308$. Analogously for the smallest number, so we have that floating points can range between 1×10^{-308} and 1×10^{308}. Take a look at *.Machine$double.xmax* and *.Machine.double.xmin*. Producing something larger of smaller in magnitude than these values is called overflow and underflow respectively. When we overflow, R gives back an Inf or -Inf (and in other cases we might get an error message). When we underflow, we get back 0, which in particular can be a problem if we try to divide by the value.

2.3 Integers or floats?

Values stored as integers should overflow if they exceed *.Machine$integer.max*.

Should 2^{45} overflow?

```
x <- 2^45
z <- 25
class(x)

## [1] "numeric"

class(z)

## [1] "numeric"

as.integer(x)
```

```
## Warning:  NAs introduced by coercion

## [1] NA

as.integer(z)

## [1] 25

1e308

## [1] 1.0000000000000000010979e+308

as.integer(1e308)

## Warning:  NAs introduced by coercion

## [1] NA

1e309

## [1] Inf
```

In R, numbers are generally stored as doubles. We've basically already seen why - consider the maximum integer when using 4 bytes and the maximum floating point value. Representing integers as floats isn't generally a problem, in part because integers will be stored exactly in base two provided the absolute value is less than 2^{53}. Why 2^{53}?

However, you can get storage as integers in a few ways: values generated based on *seq()*, specified with an "L", or explicitly coerced:

```
x <- 3
typeof(x)

## [1] "double"

x <- as.integer(3)
typeof(x)

## [1] "integer"

x <- 3L
typeof(x)

## [1] "integer"
```

2.4 Precision

Consider our representation as (*S, d, e*) where we have $p = 52$ bits for d. Since we have $2^{52} \approx 0.5 \times 10^{16}$, we can represent about that many discrete values, which means we can accurately represent about 16 digits (in base 10). The result is that floats on a computer are actually discrete (we have a finite number of bits), and if we get a number that is in one of the gaps (there are uncountably many reals), it's approximated by the nearest discrete value. The accuracy of our representation is to within 1/2 of the gap between the two discrete values bracketing the true number. Let's consider the implications for accuracy in working with large and small numbers. By changing e we can change the magnitude of a number. So regardless of whether we have a very large or small number, we have about 16 digits of accuracy, since the absolute spacing depends on what value is represented by the least significant digit (the *ulp*, or *unit in the last place*) in d, i.e., the $p = 52$nd one, or in terms of base 10, the 16th digit. Let's explore this:

```
options(digits = 22)
.1234123412341234

## [1] 0.12341234123412339607 21

1234.1234123412341234 # not accurate to 16 places

## [1] 1234.1234123412341241

123412341234.123412341234 # only accurate to 4 places

## [1] 123412341234.1234130859

1234123412341234.123412341234 # no places!

## [1] 1234123412341234

12341234123412341234 # fewer than no places!

## [1] 12341234123412340736
```

We can see the implications of this in the context of calculations:

```
1234567812345678 - 1234567812345677

## [1] 1
```

```
12345678123456788888 - 12345678123456788887

## [1] 0

.1234567812345678 - .1234567812345677

## [1] 9.714451465470119728707e-17

.12345678123456788888 - .12345678123456788887

## [1] 0

.000012345678123456788888 - .000012345678123456788887

## [1] 0

# the above is not as close as we'd expect, should be 1e-20
.0000123456781234567888888 - .0000123456781234567888887

## [1] 0
```

Suppose we try this calculation: $123456781234 - .0000123456781234$. How many decimal places do we expect to be accurate?

The spacing of possible computer numbers that have a magnitude of about 1 leads us to another definition of *machine epsilon* (an alternative, but essentially equivalent definition to that given above). Machine epsilon tells us also about the relative spacing of numbers. First let's consider numbers of magnitude one. The difference between $1 = 1.00...00 \times 2^0$ and $1.000...01 \times 2^0$ is $1 \times 2^{-52} \approx 2.2 \times 10^{-16}$. Machine epsilon gives the *absolute spacing* for numbers near 1 and the *relative spacing* for numbers with a different order of magnitude and therefore a different absolute magnitude of the error in representing a real. The relative spacing at x is

$$\frac{(1+\epsilon)x - x}{x} = \epsilon$$

since the next largest number from x is given by $(1+\epsilon)x$. Suppose $x = 1 \times 10^6$. Then the absolute error in representing a number of this magnitude is $x\epsilon \approx 2 \times 10^{-10}$. We can see by looking at the numbers in decimal form, where we are accurate to the order 10^{-10} but not 10^{-11}.

```
1000000.1
```

```
## [1] 1000000.099999999976717
```

Let's see what we arithmetic we can do exactly with integers stored as doubles and how that relates to the absolute spacing of numbers we've just seen:

```
2^52
```

```
## [1] 4503599627370496
```

```
2^52 + 1
```

```
## [1] 4503599627370497
```

```
2^53
```

```
## [1] 9007199254740992
```

```
2^53 + 1
```

```
## [1] 9007199254740992
```

```
2^53 + 2
```

```
## [1] 9007199254740994
```

```
2^54
```

```
## [1] 18014398509481984
```

```
2^54 + 2
```

```
## [1] 18014398509481984
```

```
2^54 + 4
```

```
## [1] 18014398509481988
```

The absolute spacing is $x\epsilon$, so $2^{52} \times 2^{-52} = 1$, $2^{53} \times 2^{-52} = 2$, $2^{54} \times 2^{-52} = 4$.

3 Implications for calculations and comparisons

3.1 Computer arithmetic is not mathematical arithmetic!

As mentioned for integers, computer number arithmetic is not closed, unlike real arithmetic. For example, if we multiply two computer floating points, we can overflow and not get back another computer floating point. One term that is used, which might pop up in an error message (though probably not in R) is that an "exception" is "thrown". Another mathematical concept we should consider here is that computer arithmetic does not obey the associative and distribute laws, i.e., $(a + b) + c$ may not equal $a + (b + c)$ on a computer and $a(b + c)$ may not be the same as $ab + ac$. Here's an example:

```
val1 <- 1/10
val2 <- 0.31
val3 <- 0.57
res1 <- val1 * val2 * val3
res2 <- val3 * val2 * val1
identical(res1, res2)

## [1] FALSE

res1

## [1] 0.017669999999999982081

res2

## [1] 0.017670000000000000167755
```

3.2 Calculating with integers vs. floating points

It's important to note that operations with integers are fast and exact (but can easily overflow) while operations with floating points are slower and approximate. Because of this slowness, floating point operations (*flops*) dominate calculation intensity and are used as the metric for the amount of work being done - a multiplication (or division) combined with an addition (or subtraction) is one flop. We'll talk a lot about flops in the next unit on linear algebra.

3.3 Comparisons

As we saw, we should never test `a==b` unless *a* and *b* are represented as integers in R or are integers stored as doubles that are small enough that they can be stored exactly) or unless they are decimal numbers that have been created in the same way (e.g., `0.1==0.1` vs. `0.1==0.4-0.3`. Similarly we should never test `a==0`. One nice approach to checking for approximate equality is to make use of *machine epsilon*. If the relative spacing of two numbers is less than *machine epsilon* then for our computer approximation, we say they are the same. Here's an implementation that relies on the absolute error being $x\epsilon$ (see above):

```
if(abs(a - b) < .Machine$double.eps * abs(a + b)) print("approximately equal")
```

Actually, we probably want to use a number slightly larger than *.Machine$double.eps* to be safe. You can also take a look at the R function *all.equal()*.

3.4 Calculations

Given the limited *precision* of computer numbers, we need to be careful when:

- Subtracting large numbers that are nearly equal (or adding negative and positive numbers of the same magnitude). You won't have the precision in the answer that you would like.

```
123456781234.56 - 123456781234.00
```

```
## [1] 0.55999755859375
```

The absolute error here, based on the larger value (which has the fewest error-free decimal places) is of the order $\epsilon x = 2.2 \times 10^{-16} \cdot 1 \times 10^{12} \approx 1 \times 10^{-4} = .0001$, so while we might think that the result is close to the value 1 and should have error of about machine epsilon, we actually only have about four significant digits in our result.

This is called *catastrophic cancellation*, because most of the digits that are left represent rounding error - all the significant digits have cancelled with each other.
Here's catastrophic cancellation with small numbers. The right answer here is EXACTLY .000000000000000000001234.

```
a = .000000000000123412341234
```

```
b = .000000000000123412340000

a - b

## [1] 1.233999993151397490851e-21
```

But the result is accurate only to 8 places + 21 = 29 places, as expected from a machine precision-based calculation, since the "1" is in the 13th position (13+16=29). Ideally, we would have accuracy to 37 places (16 + the 21), but we've lost 8 digits to catastrophic cancellation.

It's best to do any subtraction on numbers that are not too large. For example, we can get catastrophic cancellation when computing a sum of squares in a naive way:

$$s^2 = \sum x_i^2 - n\bar{x}^2$$

```
x <- c(-1, 0, 1) + 1e8

n <- length(x)

sum(x^2)-n*mean(x)^2 # that's not good!

## [1] 0


sum((x - mean(x))^2)

## [1] 2
```

A good principle to take away is to subtract off a number similar in magnitude to the values (in this case $\bar{x}$ is obviously ideal) and adjust your calculation accordingly. In general, you can sometimes rearrange your calculation to avoid catastrophic cancellation. Another example involves the quadratic formula for finding a root (p. 101 of Gentle).

- Adding or subtracting numbers that are very different in magnitude. The precision will be that of the large magnitude number, since we can only represent that number to a certain absolute accuracy, which is much less than the absolute accuracy of the smaller number:

```
123456781234 - 1e-06
```

```
## [1] 123456781234
```

The absolute error in representing the larger number is around 1×10^{-4} and the smaller number is smaller than this.

A work-around is to add a set of numbers in increasing order. However, if the numbers are all of similar magnitude, then by the time you add ones later in the summation, the partial sum will be much larger than the new term. A work-around is to add the numbers in a tree-like fashion, so that each addition involves a summation of numbers of similar size.

Given the limited *range* of computer numbers, be careful when you are:

- Multiplying or dividing many numbers, particularly large or small ones. Never take the product of many large or small numbers as this can cause over- or under-flow. Rather compute on the log scale and only at the end of your computations should you exponentiate. E.g.,

$$\prod_i x_i / \prod_j y_j = \exp(\sum_i \log x_i - \sum_j \log y_j)$$

- Challenge: Let's think about how we can handle the following calculation. Suppose I want to calculate a predictive density (e.g., in a model comparison in a Bayesian context):

$$\begin{aligned} f(y^*|y,x) &= \int f(y^*|y,x,\theta)\pi(\theta|y,x)d\theta \\ &\approx \frac{1}{M}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|x,\theta_j) \\ &= \frac{1}{M}\sum_{j=1}^{m}\exp\sum_{i=1}^{n}\log f(y_i^*|x,\theta_j) \\ &\equiv \frac{1}{M}\sum_{j=1}^{m}\exp(v_j) \end{aligned}$$

First, why do I use the log conditional predictive density? Second, let's work with an estimate of the unconditional predictive density on the log scale, $\log f(y^*|y,x) \approx \log \frac{1}{M}\sum_{j=1}^{m}\exp(v_j)$. Now note that e^{v_j} may be quite small as v_j is the sum of log likelihoods. So what happens if we have terms something like e^{-1000}? So we can't exponentiate each individual v_j. Thoughts? I have one solution in mind, but there might be other approaches.

Numerical issues come up frequently in linear algebra. For example, they come up in working with positive definite and semi-positive-definite matrices, such as covariance matrices. You can easily

get negative numerical eigenvalues even if all the eigenvalues are positive or non-negative. Here's an example where we use an squared exponential correlation as a function of time (or distance in 1-d), which is *mathematically* positive definite but not numerically positive definite:

```
xs <- 1:100
dists <- rdist(xs)
corMat <- exp(-(dists/10)^2)
eigen(corMat)$values[80:100]

##  [1]  2.025087032040071293067e-18 -3.266419741215397140920e-17
##  [3] -3.444200677004415898082e-17 -4.886954483578307325434e-17
##  [5] -6.129347918638579386910e-17 -9.880603825772825889419e-17
##  [7] -9.967343900132262641741e-17 -1.230143695483269682612e-16
##  [9] -1.248024408367381367725e-16 -1.292974005668460125397e-16
## [11] -1.331124664942472191787e-16 -1.651346230025272135916e-16
## [13] -1.951061360969111230889e-16 -1.990648753104187720567e-16
## [15] -2.015924870201480734054e-16 -2.257013487287240792239e-16
## [17] -2.335683529300324037256e-16 -2.719929490669187250991e-16
## [19] -2.882703020809833099805e-16 -3.057847173103147957185e-16
## [21] -4.411825302647757790411e-16
```

3.5 Final note

How the computer actually does arithmetic with the floating point representation in base 2 gets pretty complicated, and we won't go into the details. These rules of thumb should be enough for our practical purposes. Monahan and the URL reference have many of the gory details.

4 Some additional details

In computing, we often encounter the use of an unusual integer as a symbol for missing values. E.g., a datafile might store missing values as -9999. Testing for this using == in R should generally be ok:

```
x [ x == -9999 ] <- NA
```

but only because integers of this magnitude are stored exactly. To be really careful, you can read in as character type and do the assessment before converting to numeric.

Finally, be wary of

```
x[ x < 0 ] <- NA
```

if what you are looking for is values that might be *mathematically* less than zero, rather than whatever is *numerically* less than zero.

Numerical linear algebra

November 2, 2012

References:

- Gentle: Numerical Linear Algebra for Applications in Statistics (my notes here are based primarily on this source) [Gentle-NLA]
  - Unfortunately, this is not in the UCB library system - I have a copy (on loan from a colleague) that you could take a look at.
- Gentle: Computational Statistics [Gentle-CS]
- Lange: Numerical Analysis for Statisticians
- Monahan: Numerical Methods of Statistics

In working through how to compute something or understanding an algorithm, it can be very helpful to depict the matrices and vectors graphically. We'll see this on the board in class.

1 Preliminaries

1.1 Goals

Here's what I'd like you to get out of this unit:

1. How to think about the computational order (number of computations involved) of a problem
2. How to choose a computational approach to a given linear algebra calculation you need to do.
3. An understanding of how issues with computer numbers play out in terms of linear algebra.

1.2 Key principle

The form of a mathematical expression and how it should be evaluated on a computer may be very different. Better computational approaches can increase speed and improve the numerical properties of the calculation.

Example 1: We do not compute $(X^\top X)^{-1}X^\top Y$ by computing $X^\top X$ and finding its inverse. In fact, perhaps more surprisingly, we may never actually form $X^\top X$ in some implementations.

Example 2: Suppose I have a matrix A, and I want to permute (switch) two rows. I can do this with a permutation matrix, P, which is mostly zeroes. On a computer, in general I wouldn't need to even change the values of A in memory in some cases. Why not?

1.3 Computational complexity

We can assess the computational complexity of a linear algebra calculation by counting the number multiplys/divides and the number of adds/subtracts. Sidenote: addition is a bit faster than multiplication, so some algorithms attempt to trade multiplication for addition.

In general we do not try to count the actual number of calculations, but just their order, though in some cases in this unit we'll actually get a more exact count. In general, we denote this as $O(f(n))$ which means that the number of calculations approaches $cf(n)$ as $n \to \infty$ (i.e., we know the calculation is approximately proportional to $f(n)$). Consider matrix multiplication, AB, with matrices of size $a \times b$ and $b \times c$. Each column of the second matrix is multiplied by all the rows of the first. For any given inner product of a row by a column, we have b multiplies. We repeat these operations for each column and then for each row, so we have abc multiplies so $O(abc)$ operations. We could count the additions as well, but there's usually an addition for each multiply, so we can usually just count the multiplys and then say there are such and such {multiply and add}s. This is Monahan's approach, but you may see other counting approaches where one counts the multiplys and the adds separately.

For two symmetric, $n \times n$ matrices, this is $O(n^3)$. Similarly, matrix factorization (e.g., the Cholesky decomposition) is $O(n^3)$ unless the matrix has special structure, such as being sparse. As matrices get large, the speed of calculations decreases drastically because of the scaling as n^3 and memory use increases drastically. In terms of memory use, to hold the result of the multiply indicated above, we need to hold $ab + bc + ac$ total elements, which for symmetric matrices sums to $3n^2$. So for a matrix with $n = 10000$, we have $3 \cdot 10000^2 \cdot 8/1e6 = 2.4$Gb.

When we have $O(n^q)$ this is known as polynomial time. Much worse is $O(b^n)$ (exponential time), while much better is $O(\log n)$ (log time). Computer scientists talk about NP-complete problems; these are essentially problems for which there is not a polynomial time algorithm - it turns out all such problems can be rewritten such that they are equivalent to one another.

In real calculations, it's possible to have the actual time ordering of two approaches differ from what the order approximations tell us. For example, something that involves n^2 operations may be faster than one that involves $1000(n \log n + n)$ even though the former is $O(n^2)$ and the latter $O(n \log n)$. The problem is that the constant, $c = 1000$, can matter (depending on how big n is), as can the extra calculations from the lower order term(s), in this case $1000n$.

A note on terminology: ***flops*** stands for both floating point operations (the number of operations required) and floating point operations per second, the speed of calculation.

1.4 Notation and dimensions

I'll try to use capital letters for matrices, A, and lower-case for vectors, x. Then x_i is the ith element of x, A_{ij} is the ith row, jth column element, and $A_{\cdot j}$ is the jth column and $A_{i\cdot}$ the ith row. By default, we'll consider a vector, x, to be a one-column matrix, and $x^\top$ to be a one-row matrix. Some of the textbook resources also use a_{ij} for A_{ij} and a_j for the jth column.

Throughout, we'll need to be careful that the matrices involved in an operation are conformable: for $A + B$ both matrices need to be of the same dimension, while for AB the number of columns of A must match the number of rows of B. Note that this allows for B to be a column vector, with only one column, Ab. Just checking dimensions is a good way to catch many errors. Example: is $\text{Cov}(Ax) = A\text{Cov}(x)A^\top$ or $\text{Cov}(Ax) = A^\top\text{Cov}(x)A$? Well, if A is $m \times n$, it must be the former, as the latter is not conformable.

The inner product of two vectors is $\sum_i x_i y_i = x^\top y \equiv \langle x, y \rangle \equiv x \cdot y$. The outer product is $xy^\top$, which comes from all pairwise products of the elements.

When the indices of summation should be obvious, I'll sometimes leave them implicit. Ask me if it's not clear.

1.5 Norms

$\|x\|_p = (\sum_i |x_i|^p)^{1/p}$ and the standard (Euclidean) norm is $\|x\|_2 = \sqrt{\sum x_i^2} = \sqrt{x^\top x}$, just the length of the vector in Euclidean space, which we'll refer to as $\|x\|$, unless noted otherwise. The standard norm for a matrix is the Frobenius norm, $\|A\|_F = (\sum_{i,j} a_{ij}^2)^{1/2}$. There is also the induced matrix norm, corresponding to any chosen vector norm,

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

So we have

$$\|A\|_2 = \sup_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sup_{\|x\|_2 = 1} \|Ax\|_2$$

A property of any legitimate matrix norm (including the induced norm) is that $\|AB\| \leq \|A\|\|B\|$. Recall that norms must obey the triangle inequality, $\|A + B\| \leq \|A\| + \|B\|$.

A normalized vector is one with "length", i.e., Euclidean norm, of one. We can easily normalize a vector: $\tilde{x} = x/\|x\|$

The angle between two vectors is

$$\theta = \cos^{-1}\left(\frac{\langle x, y\rangle}{\sqrt{\langle x, x\rangle\langle y, y\rangle}}\right)$$

1.6 Orthogonality

Two vectors are orthogonal i4f $x^\top y = 0$, in which case we say $x \perp y$. An orthogonal matrix is a matrix in which all of the columns are orthogonal to each other and normalized. Orthogonal matrices can be shown to have full rank. Furthermore if A is orthogonal, $A^\top A = I$, so $A^{-1} = A^\top$. Given all this, the determinant of orthogonal A is either 1 or -1. Finally the product of two orthogonal matrices, A and B, is also orthogonal since $(AB)^\top AB = B^\top A^\top AB = I$.

Permutations Sometimes we make use of matrices that permute two rows (or two columns) of another matrix when multiplied. Such a matrix is known as an elementary permutation matrix and is an orthogonal matrix with a determinant of -1. You can multiply such matrices to get more general permutation matrices that are also orthogonal. If you premultiply by P, you permute rows, and if you postmultiply by P you permute columns. Note that on a computer, you wouldn't need to actually do the multiply (and if you did, you should use a sparse matrix routine), but rather one can often just rework index values that indicate where relevant pieces of the matrix are stored (more in the next section).

1.7 Some vector and matrix properties

$AB \neq BA$ but $A + B = B + A$ and $A(BC) = (AB)C$.

In R, recall the syntax is

```
A + B
A %*% B
```

You don't need the spaces, but they're nice for code readability.

1.8 Trace and determinant of square matrices

For square matrices, $\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$, $\text{tr}(A) = \text{tr}(A^\top)$.

We also have $\text{tr}(ABC) = \text{tr}(CAB) = \text{tr}(BCA)$ - basically you can move a matrix from the beginning to the end or end to beginning. This is helpful for a couple reasons:

1. We can find the ordering that reduces computation the most if the individual matrices are not square.

2. $x^\top Ax = \text{tr}(x^\top Ax)$ since the quadratic form is a scalar, and this is equal to $\text{tr}(xx^\top A)$. It can be helpful to be able to go back and forth between a scalar and a trace in some statistical calculations.

For square matrices, the determinant exists and we have $|AB| = |A||B|$ and therefore, $|A^{-1}| = 1/|A|$ since $|I| = |AA^{-1}| = 1$. Also $|A| = |A^\top|$.

Other matrix multiplications The Hadamard or direct product is simply multiplication of the correspoding elements of two matrices by each other. In R this is simply

```
A * B
```

Challenge: How can I find $\text{tr}(AB)$ without using `A %*% B` ?

The Kronecker product is the product of each element of one matrix with the entire other matrix"

$$A \otimes B = \begin{pmatrix} A_{11}B & \cdots & A_{1m}B \\ \vdots & \ddots & \vdots \\ A_{n1}B & \cdots & A_{nm}B \end{pmatrix}$$

The inverse of a Kronecker product is the Kronecker product of the inverses,

$$B^{-1} \otimes A^{-1}$$

which is obviously quite a bit faster because the inverse (i.e., solving a system of equations) in this special case is $O(n^3 + m^3)$ rather than the naive approach being $O((nm)^3)$.

1.9 Linear independence, rank, and basis vectors

A set of vectors, $v_1, \ldots v_n$, is linearly independent (LIN) when none of the vectors can be represented as a linear combination, $\sum c_i v_i$, of the others. If we have vectors of length n, we can have at most n linearly independent vectors. The rank of a matrix is the number of linearly independent rows (or columns - it's the same), and is at most the minimum of the number of rows and columns. We'll generally think about it in terms of the dimension of the column space - so we can just think about the number of linearly independent columns.

Any set of linearly independent vectors (say $v_1, \ldots, v_n$) span a space made up of all linear combinations of those vectors ($\sum_{i=1}^n c_i v_i$). The spanning vectors are known as basis vectors. We can express a vector x that is in the space with respect to (as a linear combination of) normalized basis vectors using the inner product: $x = \sum_i c_i v_i$ where we can find the weights as $c_k = \langle x, v_k \rangle$.

Consider a regression context. We have p covariates (p columns in the design matrix, X), of which q are linearly independent covariates. This means that $p - q$ of the vectors can be written as linear combos of the q vectors. The space spanned by the covariate vectors is of dimension q, rather than p, and $X^\top X$ has $p - q$ eigenvalues that are zero. The q LIN vectors are basis vectors for the space - we can represent any point in the space as a linear combination of the basis vectors. You can think of the basis vectors as being like the axes of the space, except that the basis vectors are not orthogonal. So it's like denoting a point in $\Re^q$ as a set of q numbers telling us where on each of the axes we are - this is the same as a linear combination of axis-oriented vectors. When we have $n \leq q$, a vector of n observations can be represented exactly as a linear combination of the q basis vectors, so there is no residual. If $n = q$, then we have a single unique solution, while if $n < q$ we have multiple possible solutions and the system is ill-determined (under-determined). Of course we usually have $n > p$, so the system is overdetermined - there is no exact solution, but regression is all about finding solutions that minimize some criterion about the differences between the observations and linear combinations of the columns of the X matrix (such as least squares or penalized least squares). In standard regression, we project the observation vector onto the space spanned by the columns of the X matrix, so we find the point in the space closest to the observation vector.

1.10 Invertibility, singularity, rank, and positive definiteness

For square matrices, let's consider how invertibility, singularity, rank and positive (or non-negative) definiteness relate.

Square matrices that are "regular" have an eigendecomposition, $A = \Gamma \Lambda \Gamma^{-1}$ where Γ is a matrix with the eigenvectors as the columns and Λ is a diagonal matrix of eigenvalues, $\Lambda_{ii} = \lambda_i$. Symmetric matrices and matrices with unique eigenvalues are regular, as are some other matrices. The number of non-zero eigenvalues is the same as the rank of the matrix. Square matrices that have an inverse are also called nonsingular, and this is equivalent to having full rank. If the matrix is symmetric, the eigenvectors and eigenvalues are real and Γ is orthogonal, so we have $A = \Gamma \Lambda \Gamma^\top$. The determinant of the matrix is the product of the eigenvalues (why?), which is zero if it is less than full rank. Note that if none of the eigenvalues are zero then $A^{-1} = \Gamma \Lambda^{-1} \Gamma^\top$.

Let's focus on symmetric matrices. The symmetric matrices that tend to arise in statistics are either positive definite (p.d.) or non-negative definite (n.n.d.). If a matrix is positive definite, then

by definition $x^\top Ax > 0$ for any x. Note that $x^\top Ax = \text{Cov}(x^\top y) = \text{Var}(x^\top y)$ if $\text{Cov}(y) = A$, so positive definiteness amounts to having linear combinations of random variables having positive variance. So we must have that positive definite matrices are equivalent to variance-covariance matrices (I'll just refer to this as a variance matrix or as a covariance matrix). If A is p.d. then it has all positive eigenvalues and it must have an inverse, though as we'll see, from a numerical perspective, we may not be able to compute it if some of the eigenvalues are very close to zero. In R, `eigen(A)$vectors` is Γ, with each column a vector, and `eigen(A)$values` contains the ordered eigenvalues.

Let's interpret the eigendecomposition in a generative context as a way of generating random vectors. We can generate y s.t. $\text{Cov}(y) = A$ if we generate $y = \Gamma\Lambda^{1/2}z$ where $\text{Cov}(z) = I$ and $\Lambda^{1/2}$ is formed by taking the square roots of the eigenvalues. So $\sqrt{\lambda_i}$ is the standard deviation associated with the basis vector $\Gamma_{\cdot i}$. That is, the z's provide the weights on the basis vectors, with scaling based on the eigenvalues. So y is produced as a linear combination of eigenvectors as basis vectors, with the variance attributable to the basis vectors determined by the eigenvalues.

If $x^\top Ax \geq 0$ then A is nonnegative definite (also called positive semi-definite). In this case one or more eigenvalues can be zero. Let's interpret this a bit more in the context of generating random vectors based on non-negative definite matrices, $y = \Gamma\Lambda^{1/2}z$ where $\text{Cov}(z) = I$. Questions:

1. What does it mean when one or more eigenvalue (i.e., $\lambda_i = \Lambda_{ii}$) is zero?

2. Suppose I have an eigenvalue that is very small and I set it to zero? What will be the impact upon y and $\text{Cov}(y)$?

3. Now let's consider the inverse of a covariance matrix, known as the precision matrix, $A^{-1} = \Gamma\Lambda^{-1}\Gamma^\top$. What does it mean if a $(\Lambda^{-1})_{ii}$ is very large? What if $(\Lambda^{-1})_{ii}$ is very small?

Consider an arbitrary $n \times p$ matrix, X. Any crossproduct or sum of squares matrix, such as $X^\top X$ is positive definite (non-negative definite if $p > n$). This makes sense as it's just a scaling of an empirical covariance matrix.

1.11 Generalized inverses

Suppose I want to find x such that $Ax = b$. Mathematically the answer (provided A is invertible, i.e. of full rank) is $x = A^{-1}b$.

Generalized inverses arise in solving equations when A is not full rank. A generalized inverse is a matrix, A^- s.t. $AA^-A = A$. The Moore-Penrose inverse (the pseudo-inverse), A^+, is a (unique) generalized inverse that also satisfies some additional properties. $x = A^+b$ is the solution to the linear system, $Ax = b$, that has the shortest length for x.

We can find the pseudo-inverse based on an eigendecomposition (or an SVD) as $\Gamma\Lambda^{+}\Gamma^{\top}$. We obtain Λ^{+} from Λ as follows. For values $\lambda_i > 0$, compute $1/\lambda_i$. All other values are set to 0. Let's interpret this statistically. Suppose we have a precision matrix with one or more zero eigenvalues and we want to find the covariance matrix. A zero eigenvalue means we have no precision, or infinite variance, for some linear combination (i.e., for some basis vector). We take the pseudo-inverse and assign that linear combination zero variance.

Let's consider a specific example. Autoregressive models are often used for smoothing (in time, in space, and in covariates). A first order autoregressive model for $y_1, y_2, \ldots, y_T$ has $E(y_i|y_{-i}) = \frac{1}{2}(y_{i-1}+y_{i+1})$. A second order autoregressive model has $E(y_i|y_{-i}) = \frac{1}{6}(4y_{i-1}+4y_{i+1}-y_{i-2}-y_{i+2})$. These constructions basically state that each value should be a smoothed version of its neighbors. One can figure out that the **precision** matrix for y in the first order model is

$$\begin{pmatrix} \ddots & & \vdots & & \\ -1 & 2 & -1 & 0 & \\ \cdots & -1 & 2 & -1 & \ldots \\ & 0 & -1 & 2 & -1 \\ & & \vdots & & \ddots \end{pmatrix}$$

and in the second order model is

$$\begin{pmatrix} \ddots & & & \vdots & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ \cdots & 1 & -4 & 6 & -4 & 1 & \cdots \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & \vdots & & & \end{pmatrix}.$$

If we look at the eigendecomposition of such matrices, we see that in the first order case, the eigenvalue corresponding to the constant eigenvector is zero.

```
precMat <- matrix(c(1,-1,0,0,0,-1,2,-1,0,0,0,-1,2,-1,
    0,0,0,-1,2,-1,0,0,0,-1,1), 5)
e <- eigen(precMat)
e$values
```

```
## [1] 3.618 2.618 1.382 0.382 0.000
```

```
e$vectors[ , 5]
```

```
## [1] 0.4472 0.4472 0.4472 0.4472 0.4472
```

```
# generate a realization
e$values[1:4] <- 1 / e$values[1:4]
y <- e$vec %*% (e$values * rnorm(5))
sum(y)

## [1] -5.246e-15
```

This means we have no information about the overall level of y. So how would we generate sample y vectors? We can't put infinite variance on the constant basis vector and still generate samples. Instead we use the pseudo-inverse and assign ZERO variance to the constant basis vector. This corresponds to generating realizations under the constraint that $\sum y_i$ has no variation, i.e., $\sum y_i = \bar{y} = 0$ - you can see this by seeing that $\text{Var}(\Gamma_{\cdot i}^\top y) = 0$ when $\lambda_i = 0$. In the second order case, we have two non-identifiabilities: for the sum and for the linear component of the variation in y (linear in the indices of y). I could parameterize a statistical model as $\mu + y$ where y has covariance that is the generalized inverse discussed above. Then I allow for both a non-zero mean and for smooth variation governed by the autoregressive structure. In the second-order case, I would need to add a linear component as well, given the second non-identifiability.

1.12 Matrices arising in regression

In regression, we work with $X^\top X$. Some properties of this matrix are that it is symmetric and non-negative definite (hence our use of $(X^\top X)^{-1}$ in the OLS estimator). When is it not positive definite?

Fitted values are $X\hat{\beta} = X(X^\top X)^{-1}X^\top Y = HY$. The "hat" matrix, H, projects Y into the column space of X. H is idempotent: $HH = H$, which makes sense - once you've projected into the space, any subsequent projection just gives you the same thing back. H is singular. Why? Also, under what special circumstance would it not be singular?

2 Computational issues

2.1 Storing matrices

We've discussed column-major and row-major storage of matrices. First, retrieval of matrix elements from memory is quickest when multiple elements are contiguous in memory. So in a column-major language (e.g., R, Fortran), it is best to work with values in a common column (or entire columns) while in a row-major language (e.g., C) for values in a common row.

In some cases, one can save space (and potentially speed) by overwriting the output from a matrix calculation into the space occupied by an input. This occurs in some clever implementations of matrix factorizations.

2.2 Algorithms

Good algorithms can change the efficiency of an algorithm by one or more orders of magnitude, and many of the improvements in computational speed over recent decades have been in algorithms rather than in computer speed.

Most matrix algebra calculations can be done in multiple ways. For example, we could compute $b = Ax$ in either of the following ways, denoted here in pseudocode.

1. Stack the inner products of the rows of A with x.

```
for(i=1:n){
        b_i = 0
        for(j=1:m){
                b_i = b_i + a_{ij} x_j
        }
}
```

2. Take the linear combination (based on x) of the columns of A

```
for(i=1:n){
        b_i = 0
}
for(j=1:m){
        for(i = 1:n){
                b_i = b_i + a_{ij} x_j
        }
}
```

In this case the two approaches involve the same number of operations but the first might be better for row-major matrices (so might be how we would implement in C) and the second for column-major (so might be how we would implement in Fortran). **Challenge**: check whether the second approach is faster in R. (Write the code just doing the outer loop and doing the inner loop using vectorized calculation.)

General computational issues The same caveats we discussed in terms of computer arithmetic hold naturally for linear algebra, since this involves arithmetic with many elements. Good implementations of algorithms are aware of the danger of catastrophic cancellation and of the possibility of dividing by zero or by values that are near zero.

2.3 Ill-conditioned problems

Basics A problem is ill-conditioned if small changes to values in the computation result in large changes in the result. This is quantified by something called the *condition number* of a calculation. For different operations there are different condition numbers.

Ill-conditionedness arises most often in terms of matrix inversion, so the standard condition number is the "condition number with respect to inversion", which when using the L_2 norm is the ratio of the absolute values of the largest to smallest eigenvalue. Here's an example:

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}.$$

The solution of $Ax = b$ for $b = (32, 23, 33, 31)$ is $x = (1, 1, 1, 1)$, while the solution for $b + \delta b = (32.1, 22.9, 33.1, 30.9)$ is $x + \delta x = (9.2, -12.6, 4.5, -1.1)$, where δ is notation for a perturbation to the vector or matrix. What's going on?

```
norm2 <- function(x) sqrt(sum(x^2))
A <- matrix(c(10, 7, 8, 7, 7, 5, 6, 5, 8, 6, 10, 9, 7, 5, 9, 10), 4)
e <- eigen(A)
b <- c(32, 23, 33, 31)
bPerturb <- c(32.1, 22.9, 33.1, 30.9)
x <- solve(A, b)
xPerturb <- solve(A, bPerturb)
norm2(x - xPerturb)

## [1] 16.4

norm2(b - bPerturb)

## [1] 0.2

norm2(x - xPerturb)/norm2(x)
```

```
## [1] 8.198
```

```
(e$val[1]/e$val[4]) * norm2(b - bPerturb)/norm2(b)
```

```
## [1] 9.943
```

Some manipulations with inequalities involving the induced matrix norm (for any chosen vector norm, but we might as well just think about the Euclidean norm) (see Gentle-CS Sec. 5.1) give

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|}$$

where we define the condition number w.r.t. inversion as $\text{cond}(A) \equiv \|A\|\|A^{-1}\|$. We'll generally work with the L_2 norm, and for a nonsingular square matrix the result is that the condition number is the ratio of the absolute values of the largest and smallest magnitude eigenvalues. This makes sense since $\|A\|_2$ is the absolute value of the largest magnitude eigenvalue of A and $\|A^{-1}\|_2$ that of the inverse of the absolute value of the smallest magnitude eigenvalue of A. We see in the code above that the large disparity in eigenvalues of A leads to an effect predictable from our inequality above, with the condition number helping us find an upper bound.

The main use of these ideas for our purposes is in thinking about the numerical accuracy of a linear system solution (Gentle-NLA Sec 3.4). On a computer we have the system

$$(A + \delta A)(x + \delta x) = b + \delta b$$

where the 'perturbation' is from the inaccuracy of computer numbers. Our exploration of computer numbers tells us that

$$\frac{\|\delta b\|}{\|b\|} \approx 10^{-p}; \quad \frac{\|\delta A\|}{\|A\|} \approx 10^{-p}$$

where $p = 16$ for standard double precision floating points. Following Gentle, one gets the approximation

$$\frac{\|\delta x\|}{\|x\|} \approx \text{cond}(A)10^{-p},$$

so if $\text{cond}(A) \approx 10^t$, we have accuracy of order 10^{t-p} instead of 10^{-p}. (Gentle cautions that this holds only if $10^{t-p} \ll 1$). So we can think of the condition number as giving us the number of digits of accuracy lost during a computation relative to the precision of numbers on the computer. E.g., a condition number of 10^8 means we lose 8 digits of accuracy relative to our original 16 on standard systems. One issue is that estimating the condition number is itself subject to numerical error and requires computation of A^{-1} (albeit not in the case of L_2 norm with square, nonsingular

A) but see Golub and van Loan (1996; p. 76-78) for an algorithm.

Improving conditioning Ill-conditioned problems in statistics often arise from collinearity of regressors. Often the best solution is not a numerical one, but re-thinking the modeling approach, as this generally indicates statistical issues beyond just the numerical difficulties.

A general comment on improving conditioning is that we want to avoid large differences in the magnitudes of numbers involved in a calculation. In some contexts such as regression, we can center and scale the columns to avoid such differences - this will improve the condition of the problem. E.g., in simple quadratic regression with $x = \{1990, \ldots, 2010\}$ (e.g., regressing on calendar years), we see that centering and scaling the matrix columns makes a huge difference on the condition number

```
x1 <- 1990:2010
x2 <- x1 - 2000  # centered
x3 <- x2/10  # centered and scaled
X1 <- cbind(rep(1, 21), x1, x1^2)
X2 <- cbind(rep(1, 21), x2, x2^2)
X3 <- cbind(rep(1, 21), x3, x3^2)
e1 <- eigen(crossprod(X1))
e1$values

## [1]  3.360e+14  7.699e+02 -3.833e-08

e2 <- eigen(crossprod(X2))
e2$values

## [1] 50677.704   770.000     9.296

e3 <- eigen(crossprod(X3))
e3$values

## [1] 24.113  7.700  1.954
```

The basic story is that simple strategies often solve the problem, and that you should be cognizant of the absolute and relative magnitudes involved in your calculations.

3 Matrix factorizations (decompositions) and solving systems of linear equations

Suppose we want to solve the following linear system:

$$\begin{aligned} Ax &= b \\ x &= A^{-1}b \end{aligned}$$

Numerically, this is never done by finding the inverse and multiplying. Rather we solve the system using a matrix decomposition (or equivalent set of steps). One approach uses Gaussian elimination (equivalent to the LU decomposition), while another uses the Cholesky decomposition. There are also iterative methods that generate a sequence of approximations to the solution but reduce computation (provided they are stopped before the exact solution is found).

Gentle-CS has a nice table overviewing the various factorizations (Table 5.1, page 219).

3.1 Triangular systems

As a preface, let's figure out how to solve $Ax = b$ if A is upper triangular. The basic algorithm proceeds from the bottom up (and therefore is called a 'backsolve'. We solve for x_n trivially, and then move upwards plugging in the known values of x and solving for the remaining unknown in each row (each equation).

1. $x_n = b_n/A_{nn}$
2. Now for $k < n$, use the already computed $\{x_n, x_{n-1}, \ldots, x_{k+1}\}$ to calculate $x_k = \frac{b_k - \sum_{j=k+1}^{n} x_j A_{kj}}{A_{kk}}$.
3. Repeat for all rows.

How many multiplies and adds are done? Solving lower triangular systems is very similar and involves the same number of calculations.

In R, *backsolve()* solves upper triangular systems and *forwardsolve()* solves lower triangular systems:

```
n <- 20
X <- crossprod(matrix(rnorm(n^2), n))
b <- rnorm(n)
U <- chol(crossprod(X))  # U is upper-triangular
L <- t(U)  # L is lower-triangular
```

```
out1 <- backsolve(U, b)
out2 <- forwardsolve(L, b)
all.equal(out1, c(solve(U) %*% b))

## [1] TRUE

all.equal(out2, c(solve(L) %*% b))

## [1] TRUE
```

We can also solve $(U^\top)^{-1}b$ and $(L^\top)^{-1}b$ as

```
backsolve(U, b, transpose = TRUE)
forwardsolve(L, b, transpose = TRUE)
```

To reiterate the distinction between matrix inversion and solving a system of equations, when we write $U^{-1}b$, what we mean on a computer is to carry out the above algorithm, not to find the inverse and then multiply.

3.2 Gaussian elimination (LU decomposition)

Gaussian elimination is a standard way of directly computing a solution for $Ax = b$. It is equivalent to the LU decomposition. LU is primarily done with square matrices, but not always. Also LU decompositions do exist for some singular matrices.

The idea of Gaussian elimination is to convert the problem to a triangular system. In class, we'll walk through Gaussian elimination in detail and see how it relates to the LU decomposition. I'll describe it more briefly here. Following what we learned in algebra when we have multiple equations, we preserve the solution, x, when we add multiples of rows (i.e., add multiples of equations) together. This amounts to doing $L_1Ax = L_1b$ for a lower-triangular matrix L_1 that produces all zeroes in the first column of L_1A except for the first row. We proceed to zero out values below the diagonal for the other columns of A. The result is $L_{n-1}\cdots L_1A \equiv U = L_{n-1}\cdots L_1b \equiv b^*$ where U is upper triangular. This is the forward reduction step of Gaussian elimination. Then the backward elimination step solves $Ux = b^*$.

If we're just looking for the solution of the system, we don't need the lower-triangular factor $L = (L_{n-1}\cdots L_1)^{-1}$ in $A = LU$, but it turns out to have a simple form that is computed as we go along, it is unit lower triangular and the values below the diagonal are the negative of the values below the diagonals in $L_1, \ldots, L_{n-1}$ (note that each L_j has non-zeroes below the diagonal only in

the jth column). As a side note related to storage, it turns out that as we proceed, we can store the elements of L and U in the original A matrix, except for the implicit 1s on the diagonal of L.

If we look at *solve.default()* in R, we see that it uses *dgesv*. A Google search indicates that this is a Lapack routine that does the LU decomposition with partial pivoting and row interchanges (see below on what these are), so R is using the algorithm we've just discussed.

For a problem set problem, you'll compute the computational order of the LU decomposition and compare it to explicitly finding the inverse and multiplying the inverse by one or more vectors.

One additional complexity is that we want to avoid dividing by very small values to avoid introducing numerical inaccuracy (we would get large values that might overwhelm whatever they are being added to, and small errors in the divisor willl have large effects on the result). This can be done on the fly by interchanging equations to use the equation (row) that produces the largest value to divide by. For example in the first step, we would switch the first equation (first row) for whichever of the remaining equations has the largest value in the first column. This is called partial pivoting. The divisors are called pivots. Complete pivoting also considers interchanging columns, and while theoretically better, partial pivoting is generally sufficient and requires fewer computations. Note that partial pivoting can be expressed as multiplying along the way by permutation matrices, $P_1, \ldots P_{n-1}$ that switch rows. Based on pivoting, we have $PA = LU$, where $P = P_{n-1} \cdots P_1$. In the demo code, we'll see a toy example of the impact of pivoting.

Finally $|PA| = |P||A| = |L||U| = |U|$ (why?) so $|A| = |U|/|P|$ and since the determinant of each permutation matrix, P_j is -1 (except when $P_j = I$ because we don't need to switch rows), we just need to multiply by minus one if there is an odd number of permutations. Or if we know the matrix is non-negative definite, we just take the absolute value of $|U|$. So Gaussian elimination provides a fast stable way to find the determinant.

3.3 Cholesky decomposition

When A is p.d., we can use the Cholesky decomposition to solve a system of equations. Positive definite matrices can be decomposed as $U^\top U = A$ where U is upper triangular. U is called a square root matrix and is unique (apart from the sign, which we fix by requiring the diagonals to be positive). One algorithm for computing U is:

1. $U_{11} = \sqrt{A_{11}}$
2. For $j = 2, \ldots, n$, $U_{1j} = A_{1j}/U_{11}$
3. For $i = 2, \ldots, n$,
   - $U_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} U_{ki}^2}$

- for $j = i+1, \ldots, n$: $U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} U_{ki}U_{kj})/U_{ii}$

We can then solve a system of equations as: $U^{-1}(U^{\top -1}b)$, which in R is done as follows:

```
backsolve(U, backsolve(U, b, transpose = TRUE))
backsolve(U, forwardsolve(t(U), b))   # equivalent but less efficient
```

The Cholesky has some nice advantages over the LU: (1) while both are $O(n^3)$, the Cholesky involves only half as many computations, $n^3/6 + O(n^2)$ and (2) the Cholesky factorization has only $(n^2+n)/2$ unique values compared to $n^2 + n$ for the LU. Of course the LU is more broadly applicable. The Cholesky does require computation of square roots, but it turns out this is not too intensive. There is also a method for finding the Cholesky without square roots.

Uses of the Cholesky The standard algorithm for generating $y \sim \mathcal{N}(0, A)$ is:

```
U <- chol(A)
y <- crossprod(U, rnorm(n))   # i.e., t(U)%*%rnorm(n), but much faster
```

Question: where will most of the time in this two-step calculation be spent?

If a regression design matrix, X, is full rank, then $X^\top X$ is positive definite, so we could find $\hat{\beta} = (X^\top X)^{-1} X^\top Y$ using either the Cholesky or Gaussian elimination. **Challenge**: write efficient R code to carry out the OLS solution using either LU or Cholesky factorization.

However, it turns out that the standard approach is to work with X using the QR decomposition rather than working with $X^\top X$; working with X is more numerically stable, though in most situations without extreme collinearity, either of the approaches will be fine.

Numerical issues with eigendecompositions and Cholesky decompositions for positive definite matrices Monahan comments that in general Gaussian elimination and the Cholesky decomposition are very stable. However if the matrix is very ill-conditioned we can get $A_{ii} - \sum_k U_{ki}^2$ being negative and then the algorithm stops when we try to take the square root. In this case, the Cholesky decomposition does not exist numerically although it exists mathematically. It's not all that hard to produce such a matrix, particularly when working with high-dimensional covariance matrices with large correlations.

```
library(fields)
locs <- runif(100)
rho <- 0.1
```

```
C <- exp(-rdist(locs)^2/rho^2)
e <- eigen(C)
e$values[96:100]

## [1] -3.664e-16 -4.240e-16 -6.321e-16 -7.410e-16 -7.625e-16

U <- chol(C)

## Error:  the leading minor of order 26 is not positive definite

vals <- abs(e$values)
max(vals)/min(vals)

## [1] 3.938e+18

U <- chol(C, pivot = TRUE)

## Warning:  matrix not positive definite
```

We can think about the accuracy here as follows. Suppose we have a matrix whose diagonal elements (i.e., the variances) are order of magnitude 1 and that the true value of a U_{ii} is less than 1×10^{-16}. From the given A_{ii} we are subtracting $\sum_k U_{ki}^2$ and trying to calculate this very small number but we know that we can only represent the values A_{ii} and $\sum_k U_{ki}^2$ accurately to 16 places, so the difference is garbage starting in the 17th position and could well be negative. Now realize that $\sum_k U_{ki}^2$ is the result of a potentially large set of arithmetic operations, and is likely represented accurately to fewer than 16 places. Now if the true value of U_{ii} is smaller than the accuracy to which $\sum_k U_{ki}^2$ is represented, we can get a difference that is negative.

Note that when the Cholesky fails, we can still compute an eigendecomposition, but we have negative numeric eigenvalues. Even if all the eigenvalues are numerically positive (or equivalently, we're able to get the Cholesky), errors in small eigenvalues near machine precision could have large effects when we work with the inverse of the matrix. This is what happens when we have columns of the X matrix nearly collinear. We cannot statistically distinguish the effect of two (or more) covariates, and this plays out numerically in terms of unstable results.

A strategy when working with mathematically but not numerically positive definite A is to set eigenvalues or singular values to zero when they get very small, which amounts to using a pseudo-inverse and setting to zero any linear combinations with very small variance. We can also use pivoting with the Cholesky and accumulate zeroes in the last $n - q$ rows (for cases where we try to take the square root of a negative number), corresponding to the columns of A that are numerically

linearly dependent.

3.4 QR decomposition

3.4.1 Introduction

The QR decomposition is available for any matrix, $X = QR$, with Q orthogonal and R upper triangular. If X is non-square, $n \times p$ with $n > p$ then the leading p rows of R provide an upper triangular matrix (R_1) and the remaining rows are 0. (I'm using p because the QR is generally applied to design matrices in regression). In this case we really only need the first p columns of Q, and we have $X = Q_1 R_1$, the 'skinny' QR (this is what R's QR provides). For uniqueness, we can require the diagonals of R to be nonnegative, and then R will be the same as the upper-triangular Cholesky factor of $X^\top X$:

$$\begin{aligned} X^\top X &= R^\top Q^\top Q R \\ &= R^\top R \end{aligned}$$

There are three standard approaches for computing the QR, using (1) reflections (Householder transformations), (2) rotations (Givens transformations), or (3) Gram-Schmidt orthogonalization (see below for details).

For $n \times n$ X, the QR (for the Householder approach) requires $2n^3/3$ flops, so QR is less efficient than LU or Cholesky.

We can also obtain the pseudo-inverse of X from the QR: $X^+ = [R_1^{-1}\, 0]Q^\top$. In the case that X is not full-rank, there is a version of the QR that will work (involving pivoting) and we end up with some additional zeroes on the diagonal of R_1.

3.4.2 Regression and the QR

Often QR is used to fit linear models, including in R. Consider the linear model in the form $Y = X\beta + \epsilon$, finding $\hat{\beta} = (X^\top X)^{-1} X^\top Y$. Let's consider the skinny QR and note that $R^\top$ is invertible. Therefore, we can express the normal equations as

$$\begin{aligned} X^\top X\beta &= X^\top Y \\ R^\top Q^\top Q R\beta &= R^\top Q^\top Y \\ R\beta &= Q^\top Y \end{aligned}$$

and solving for β is just a backsolve since R is upper-triangular. Furthermore the standard regression quantities, such as the hat matrix, the SSE, the residuals, etc. can be easily expressed in terms

of Q and R.

Why use the QR instead of the Cholesky on $X^\top X$? The condition number of A is the square root of that of $X^\top X$, and the QR factorizes X. Monahan has a discussion of the condition of the regression problem, but from a larger perspective, the situations where numerical accuracy is a concern are generally cases where the OLS estimators are not particularly helpful anyway (e.g., highly collinear predictors).

What about computational order of the different approaches to least squares? The Cholesky is $np^2+\frac{1}{3}p^3$, an algorithm called sweeping is np^2+p^3 , the Householder method for QR is $2np^2-\frac{2}{3}p^3$, and the modified Gram-Schmidt approach for QR is $2np^2$. So if $n \gg p$ then Cholesky (and sweeping) are faster than the QR approaches. According to Monahan, modified Gram-Schmidt is most numerically stable and sweeping least. In general, regression is pretty quick unless p is large since it is linear in n, so it may not be worth worrying too much about computational differences of the sort noted here.

3.4.3 Regression and the QR in R

Regression in R uses the QR decomposition via *qr()*, which calls a Fortran function. *qr()* (and the Fortran functions that are called) is specifically designed to output quantities useful in fitting linear models. Note that by default you get the skinny QR, namely only the first p rows of R and the first p columns of Q, where the latter form an orthonormal basis for the column space of X. The remaining columns form an orthonormal basis for the null space of X. The analogy in regression is that we get the basis vectors for the regression, while adding the remaining columns gives us the full n-dimensional space of the observations.

qr() returns the result as a list meant for use by other tools. R stores the R matrix in the upper triangle of *$qr*, while the lower triangle of *$qr* and *$aux* store the information for constructing Q (this relates to the Householder-related vectors u below). One can multiply by Q using *qr.qy()* and by $Q^\top$ using *qr.qty()*. If you want to extract R and Q, the following will work:

```
X.qr = qr(X)
Q = qr.Q(X.qr)
R = qr.R(X.qr)
```

As a side note, there are QR-based functions that provide regression-related quantities, such as *qr.resid()*, *qr.fitted()* and *qr.coef()*. These functions (and their Fortran counterparts) exist because one can work through the various regression quantities of interest and find their expressions in terms of Q and R, with nice properties resulting from Q being orthogonal and R triangular.

3.4.4 Computing the QR decomposition

We'll work through some of the details of the different approaches to the QR, in part because they involve some concepts that may be useful in other contexts.

One approach involves reflections of vectors and a second rotations of vectors. Reflections and rotations are transformations that are performed by orthogonal matrices. The determinant of a reflection matrix is -1 and the determinant of a rotation matrix is 1. We'll see some of the details in the demo code.

Reflections If u and v are orthonormal vectors and x is in the space spanned by u and v, $x = c_1u + c_2v$, then $\tilde{x} = -c_1u + c_2v$ is a reflection (a *Householder* reflection) along the u dimension (since we are using the negative of that basis vector). We can think of this as reflecting across the plane perpendicular to u. This extends simply to higher dimensions with orthonormal vectors, $u, v_1, v_2, \ldots$

Suppose we want to formulate the reflection in terms of a "Householder" matrix, Q. It turns out that

$$Qx = \tilde{x}$$

if $Q = I - 2uu^\top$. Q has the following properties: (1) $Qu = -u$, (2) $Qv = v$ for $u^\top v = 0$, (3) Q is orthogonal and symmetric.

One way to create the QR decomposition is by a series of Householder transformations that create an upper triangular R from X:

$$\begin{aligned} R &= Q_p \cdots Q_1 X \\ Q &= (Q_p \cdots Q_1)^\top \end{aligned}$$

where we make use of the symmetry in defining Q.

Basically Q_1 reflects the first column of X with respect to a carefully chosen u, so that the result is all zeroes except for the first element. We want $Q_1x = \tilde{x} = (||x||, 0, \ldots, 0)$. This can be achieved with $u = \frac{x - \tilde{x}}{||x - \tilde{x}||}$. Then Q_2 makes the last $n - 2$ rows of the second column equal to zero. We'll work through this a bit in class.

In the regression context, as we work through the individual transformations, $Q_j = I - 2u_ju_j^\top$, we apply them to X and Y to create R (note this would not involve doing the full matrix multiplication - think about what calculations are actually needed) and $QY = Q^\top Y$, and then solve $R\beta = Q^\top Y$. To find $\text{Cov}(\hat{\beta}) = (X^\top X)^{-1} = (R^\top R)^{-1} = R^{-1}R^{-\top}$ we do need to invert R, but it's upper-triangular and of dimension $p \times p$. It turns out that $Q^\top Y$ can be partitioned into the first p and the last $n - p$ elements, $z^{(1)}$ and $z^{(2)}$. The SSR is $\|z^{(1)}\|^2$ and SSE is $\|z^{(2)}\|^2$.

Rotations A *Givens* rotation matrix rotates a vector in a two-dimensional subspace to be axis oriented with respect to one of the two dimensions by changing the value of the other dimension. E.g. we can create $\tilde{x} = (x_1, \ldots, \tilde{x}_p, \ldots, 0, \ldots x_n)$ from $x = (x_{1,} \ldots, x_p, \ldots, x_q, \ldots, x_n)$ using a matrix multiplication: $\tilde{x} = Qx$. Q is orthogonal but not symmetric.

We can use a series of Givens rotations to do the QR but unless it is done carefully, more computations are needed than with Householder reflections. The basic story is that we apply a series of Givens rotations to X such that we zero out the lower triangular elements.

$$\begin{aligned} R &= Q_{pn} \cdots Q_{22} Q_{1n} \cdots Q_{13} Q_{12} X \\ Q &= (Q_{pn} \cdots Q_{12})^\top \end{aligned}$$

Note that we create the $n - p$ zero rows in R (because the calculations affect the upper triangle of R), but we can then ignore those rows and the corresponding columns of Q.

Gram-Schmidt Orthogonalization Gram-Schmidt involves finding a set of orthonormal vectors to span the same space as a set of LIN vectors, $x_1, \ldots, x_p$. If we take the LIN vectors to be the columns of X, so that we are discussing the column space of X, then G-S yields the QR decomposition. Here's the algorithm:

1. $\tilde{x}_1 = \frac{x_1}{\|x_1\|}$ (normalize the first vector)
2. Orthogonalize the remaining vectors with respect to $\tilde{x}_1$:
   (a) $\tilde{x}_2 = \frac{x_2 - \tilde{x}_1^\top x_2 \tilde{x}_1}{\|x_2 - \tilde{x}_1^\top x_2 \tilde{x}_1\|}$, which orthogonalizes with respect to $\tilde{x}_1$ and normalizes. Note that $\tilde{x}_1^\top x_2 \tilde{x}_1 = \langle \tilde{x}_1, x_2 \rangle \tilde{x}_1$. So we are finding a scaling, $c\tilde{x}_1$, where c is based on the inner product, to remove the variation in the x_1 direction from x_2.
   (b) For $k > 2$, find interim vectors, $x_k^{(2)}$, by orthogonalizing with respect to $\tilde{x}_1$
3. Proceed for $k = 3, \ldots$, in turn orthogonalizing and normalizing the first of the remaining vectors w.r.t. $\tilde{x}_{k-1}$ and orthogonalizing the remaining vectors w.r.t. $\tilde{x}_{k-1}$ to get new interim vectors

Mathematically, we could instead orthogonalize x_2 w.r.t. $\tilde{x}_1$, then orthogonalize x_3 w.r.t. $\{\tilde{x}_1, \tilde{x}_2\}$, etc. The algorithm above is the *modified* G-S, and is known to be more numerically stable if the columns of X are close to collinear, giving vectors that are closer to orthogonal. The resulting $\tilde{x}$ vectors are the columns of Q. The elements of R are obtained as we proceed: the diagonal values are the the normalization values in the denominators, while the off-diagonals are the inner products with the already-computed columns of Q that are computed as part of the numerators.

Another way to think about this is that $R = Q^\top X$, which is the same as regressing the columns of X on Q, $(Q^\top Q)^{-1}Q^\top X = Q^\top X$. By construction, the first column of X is a scaling of the first column of Q, the second column of X is a linear combination of the first two columns of Q, etc., so R being upper triangular makes sense.

3.5 Determinants

The absolute value of the determinant of a square matrix can be found from the product of the diagonals of the triangular matrix in any factorization that gives a triangular (including diagonal) matrix times an orthogonal matrix (or matrices) since the determinant of an orthogonal matrix is either one or minus one.

$|A| = |QR| = |Q||R| = \pm|R|$

$|A^\top A| = |(QR)^\top QR| = |R^\top R| = |R_1^\top R_1| = |R_1|^2$

In R, the following will do it (on the log scale), since R is stored in the upper triangle of the *$qr* element.

```
myqr = qr(A)
magn = sum(log(abs(diag(myqr$qr))))
sign = prod(sign(diag(myqr$qr)))
```

An alternative is the product of the diagonal elements of D (the singular values) in the SVD factorization, $A = UDV^\top$.

For non-negative definite matrices, we know the determinant is non-negative, so the uncertainty about the sign is not an issue. For positive definite matrices, a good approach is to use the product of the diagonal elements of the Cholesky decomposition.

One can also use the product of the eigenvalues: $|A| = |\Gamma\Lambda\Gamma^{-1}| = |\Gamma||\Gamma^{-1}||\Lambda| = |\Lambda|$

Computation Computing from any of these diagonal or triangular matrices as the product of the diagonals is prone to overflow and underflow, so we **always** work on the log scale as the sum of the log of the values. When some of these may be negative, we can always keep track of the number of negative values and take the log of the absolute values.

Often we will have the factorization as a result of other parts of the computation, so we get the determinant for free.

R's *determinant()* uses the LU decomposition. Supposedly *det()* just wraps determinant, but I can't seem to pass the *logarithm* argument into *det()*, so *determinant()* seems more useful.

4 Eigendecomposition and SVD

4.1 Eigendecomposition

The eigendecomposition (spectral decomposition) is useful in considering convergence of algorithms and of course for statistical decompositions such as PCA. We think of decomposing the components of variation into orthogonal patterns (the eigenvectors) with variances (eigenvalues) associated with each pattern.

Square symmetric matrices have real eigenvectors and eigenvalues, with the factorization into orthogonal Γ and diagonal Λ, $A = \Gamma\Lambda\Gamma^\top$, where the eigenvalues on the diagonal of Λ are ordered in decreasing value. Of course this is equivalent to the definition of an eigenvalue/eigenvector pair as a pair such that $Ax = \lambda x$ where x is the eigenvector and λ is a scalar, the eigenvalue. The inverse of the eigendecomposition is simply $\Gamma\Lambda^{-1}\Gamma^\top$. On a similar note, we can create a square root matrix, $\Gamma\Lambda^{1/2}$, by taking the square roots of the eigenvalues.

The spectral radius of A, denoted $\rho(A)$, is the maximum of the absolute values of the eigenvalues. As we saw when talking about ill-conditionedness, for symmetric matrices, this maximum is the induced norm, so we have $\rho(A) = \|A\|_2$. It turns out that $\rho(A) \leq \|A\|$ for any induced matrix norm. The spectral radius comes up in determining the rate of convergence of some iterative algorithms.

Computation There are several methods for eigenvalues; a common one for doing the full eigendecomposition is the *QR algorithm*. The first step is to reduce A to upper Hessenburg form, which is an upper triangular matrix except that the first subdiagonal in the lower triangular part can be non-zero. For symmetric matrices, the result is actually tridiagonal. We can do the reduction using Householder reflections or Givens rotations. At this point the QR decomposition (using Givens rotations) is applied iteratively (to a version of the matrix in which the diagonals are shifted), and the result converges to a diagonal matrix, which provides the eigenvalues. It's more work to get the eigenvectors, but they are obtained as a product of Householder matrices (required for the initial reduction) multiplied by the product of the Q matrices from the successive QR decompositions.

We won't go into the algorithm in detail, but note that it involves manipulations and ideas we've seen already.

If only the largest (or the first few largest) eigenvalues and their eigenvectors are needed, which can come up in time series and Markov chain contexts, the problem is easier and can be solved by the *power method*. E.g., in a Markov chain context, steady state is reached through $x_t = A^t x_0$. One can find the largest eigenvector by multiplying by A many times, normalizing at each step. $v^{(k)} = Az^{(k-1)}$ and $z^{(k)} = v^{(k)}/\|v^{(k)}\|$. There is an extension to find the p largest eigenvalues and their vectors. See the demo code for an implementation.

4.2 Singular value decomposition

Let's consider an $n \times m$ matrix, A, with $n \geq m$ (if $m > n$, we can always work with $A^\top$). This often is a matrix representing m features of n observations. We could have n documents and m words, or n gene expression levels and m experimental conditions, etc. A can always be decomposed as

$$A = UDV^\top$$

where U and V are matrices with orthonormal columns (left and right eigenvectors) and D is diagonal with non-negative values (which correspond to eigenvalues in the case of square A and to squared eigenvalues of $A^\top A$).

The SVD can be represented in more than one way. One representation is

$$A_{n\times m} = U_{n\times k} D_{k\times k} V^\top_{k\times m} = \sum_{j=1}^{k} D_{jj} u_j v_j^\top$$

where u_j and v_j are the columns of U and V and where k is the rank of A (which is at most the minimum of n and m of course). The diagonal elements of D are the singular values.

If A is positive semi-definite, the eigendecomposition is an SVD. Furthermore, $A^\top A = VD^2V^\top$ and $AA^\top = UD^2U^\top$, so we can find the eigendecomposition of such matrices using the SVD of A. Note that the squares of the singular values of A are the eigenvalues of $A^\top A$ and $AA^\top$.

We can also fill out the matrices to get

$$A = U_{n\times n} D_{n\times m} V^\top_{m\times m}$$

where the added rows and columns of D are zero with the upper left block the $D_{k\times k}$ from above.

Uses The SVD is an excellent way to determine a matrix rank and to construct a pseudo-inverse ($A^+ = VD^+U^\top$).

We can use the SVD to approximate A by taking $A \approx \tilde{A} = \sum_{j=1}^{p} D_{jj} u_j v_j^\top$ for $p < m$. This approximation holds in terms of the Frobenius norm for $A - \tilde{A}$. As an example if we have a large image of dimension $n \times m$, we could hold a compressed version by a rank-p approximation using the SVD. The SVD is used a lot in clustering problems. For example, the Netflix prize was won based on a variant of SVD (in fact all of the top methods used variants on SVD, I believe).

Computation The basic algorithm is similar to the QR method for the eigendecomposition. We use a series of Householder transformations on the left and right to reduce A to a bidiagonal matrix, $A^{(0)}$. The post-multiplications generate the zeros in the upper triangle. (A bidiagonal matrix is one

with non-zeroes only on the diagonal and first subdiagonal above the diagonal). Then the algorithm produces a series of upper bidiagonal matrices, $A^{(0)}$, $A^{(1)}$, etc. that converge to a diagonal matrix, D . Each step is carried out by a sequence of Givens transformations:

$$\begin{aligned} A^{(j+1)} &= R_{m-2}^{\top} R_{m-3}^{\top} \cdots R_0^{\top} A^{(j)} T_0 T_1 \cdots T_{m-2} \\ &= R A^{(j)} T \end{aligned}$$

Note that the multiplication on the right is required to produce zeroes in the kth column and the kth row. This eventually gives $A^{(\cdots)} = D$ and by construction, U (the product of the pre-multiplied Householder matrices and the R matrices) and V (the product of the post-multiplied Householder matrices and the T matrices) are orthogonal. The result is then transformed by a diagonal matrix to make the elements of D non-negative and by permutation matrices to order the elements of D in nonincreasing order.

5 Computation

5.1 Linear algebra in R

Speedups and storage savings can be obtained by working with matrices stored in special formats when the matrices have special structure. E.g., we might store a symmetric matrix as a full matrix but only use the upper or lower triangle. Banded matrices and block diagonal matrices are other common formats. Banded matrices are all zero except for $A_{i,i+c_k}$ for some small number of integers, c_k. Viewed as an image, these have bands. The bands are known as co-diagonals.

Note that for many matrix decompositions, you can change whether all of the aspects of the decomposition are returned, or just some, which may speed calculations.

Some useful packages in R for matrices are *Matrix*, *spam*, and *bdsmatrix*. *Matrix* can represent a variety of rectangular matrices, including triangular, orthogonal, diagonal, etc. and provides methods for various matrix calculations that are specific to the matrix type. *spam* handles general sparse matrices with fast matrix calculations, in particular a fast Cholesky decomposition. *bdsmatrix* focuses on block-diagonal matrices, which arise frequently in contexts where there is clustering that induces within-cluster correlation and cross-cluster independence.

In general, matrix operations in R go to compiled C or Fortran code without much intermediate R code, so they can actually be pretty efficient and are based on the best algorithms developed by numerical experts. The core libraries that are used are LAPACK and BLAS (the Linear Algebra PACKage and the Basic Linear Algebra Subroutines). As we've discussed in the parallelization unit, one way to speed up code that relies heavily on linear algebra is to make sure you have a

BLAS library tuned to your machine. These include GotoBLAS (free), Intel's MKL, and AMD's ACML. These can be installed and R can be linked to the shared object library file (*.so* file) for the fast BLAS. These BLAS libraries are also available in threaded versions that farm out the calculations across multiple cores or processors that share memory.

BLAS routines do vector operations (level 1), matrix-vector operations (level 2), and dense matrix-matrix operations (level 3). Often the name of the routine has as its first letter "d", "s", "c" to indicate the routine is double precision, single precision, or complex. LAPACK builds on BLAS to implement standard linear algebra routines such as eigendecomposition, solutions of linear systems, a variety of factorizations, etc.

5.2 Sparse matrices

As an example of exploiting sparsity, here's how the *spam* package in R stores a sparse matrix. Consider the matrix to be row-major and store the non-zero elements in order in a vector called *value*. Then create a vector called *rowptr* that stores the position of the first element of each row. Finally, have a vector, *colindices* that tells the column identity of each element.

We can do a fast matrix multiply, Ab, as follows in pseudo-code:

```
for(i in 1:nrows(A)){
        x[i] = 0
        for(j in (rowptr[i]:rowptr[i+1]-1) {
                x[i] = x[i] + value[j] * b[colindices[j]]
        }
}
```

How many computations have we done? Only k multiplies and $O(k)$ additions where k is the number of non-zero elements of A. Compare this to the usual $O(n^2)$ for dense multiplication.

Note that for the Cholesky of a sparse matrix, if the sparsity pattern is fixed, but the entries change, one can precompute an optimal re-ordering that retains as much sparsity in U as possible. Then multiple Cholesky decompositions can be done more quickly as the entries change.

Banded matrices Suppose we have a banded matrix A where the lower bandwidth is p, namely $A_{ij} = 0$ for $i > j + p$ and the upper bandwidth is q ($A_{ij} = 0$ for $j > i + q$). An alternative to reducing to $Ux = b^*$ is to compute $A = LU$ and then do two solutions, $U^{-1}(L^{-1}b)$. One can show that the computational complexity of the LU factorization is $O(npq)$ for banded matrices, while solving the two triangular systems is $O(np + nq)$, so for small p and q, the speedup can be dramatic.

Banded matrices come up in time series analysis. E.g., MA models produce banded covariance structures because the covariance is zero after a certain number of lags.

5.3 Low rank updates

A transformation of the form $A - uv^\top$ is a rank-one update because $uv^\top$ is of rank one.

More generally a low rank update of A is $\tilde{A} = A - UV^\top$ where U and V are $n \times m$ with $n \geq m$. The Sherman-Morrison-Woodbury formula tells us that

$$\tilde{A}^{-1} = A^{-1} + A^{-1}U(I_m - V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

so if we know $x_0 = A^{-1}b$, then the solution to $\tilde{A}x = b$ is $x + A^{-1}U(I_m - V^\top A^{-1}U)^{-1}V^\top x$. Provided m is not too large, and particularly if we already have a factorization of A, then $A^{-1}U$ is not too bad computationally, and $I_m - V^\top A^{-1}U$ is $m \times m$. As a result $A^{-1}(U(\cdots)^{-1}V^\top x)$ isn't too bad.

This also comes up in working with precision matrices in Bayesian problems where we may have A^{-1} but not A (we often add precision matrices to find conditional normal distributions). An alternative expression for the formula is $\tilde{A} = A + UCV^\top$, and the identity tells us

$$\tilde{A}^{-1} = A^{-1} - A^{-1}U(C^{-1} + V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

Basically Sherman-Morrison-Woodbury gives us matrix identities that we can use in combination with our knowledge of smart ways of solving systems of equations.

6 Iterative solutions of linear systems

Gauss-Seidel Suppose we want to iteratively solve $Ax = b$. Here's the algorithm, which sequentially updates each element of x in turn.

- Start with an initial approximation, $x^{(0)}$.
- Hold all but $x_1^{(0)}$ constant and solve to find $x_1^{(1)} = \frac{1}{a_{11}}(b_1 - \sum_{j=2}^n a_{1j}x_j^{(0)})$.
- Repeat for the other rows of A (i.e., the other elements of x), finding $x^{(1)}$.
- Now iterate to get $x^{(2)}$, $x^{(3)}$, etc. until a convergence criterion is achieved, such as $\|x^{(k)} - x^{(k-1)}\| \leq \epsilon$ or $\|r^{(k)} - r^{(k-1)}\| \leq \epsilon$ for $r^{(k)} = b - Ax^{(k)}$.

Let's consider how many operations are involved in a single update: $O(n)$ for each element, so $O(n^2)$ for each update. Thus if we can stop well before n iterations, we've saved computation relative to exact methods.

If we decompose $A = L + D + U$ where L is strictly lower triangular, U is strictly upper triangular, then Gauss-Seidel is equivalent to solving

$$(L + D)x^{(k+1)} = b - Ux^{(k)}$$

and we know that solving the lower triangular system is $O(n^2)$.

It turns out that the rate of convergence depends on the spectral radius of $(L + D)^{-1}U$.

Gauss-Seidel amounts to optimizing by moving in axis-oriented directions, so it can be slow in some cases.

Conjugate gradient For positive definite A, conjugate gradient (CG) reexpresses the solution to $Ax = b$ as an optimization problem, minimizing

$$f(x) = \frac{1}{2}x^\top Ax - x^\top b,$$

since the derivative of $f(x)$ is $Ax - b$ and at the minimum this gives $Ax - b = 0$.

Instead of finding the minimum by following the gradient at each step (so-called steepest descent, which can give slow convergence - we'll see a demonstration of this in the optimization unit), CG chooses directions that are mutually conjugate w.r.t. A, $d_i^\top Ad_j = 0$ for $i \neq j$. The method successively chooses vectors giving the direction, d_k, in which to move down towards the minimum and a scaling of how much to move, α_k. If we start at $x_{(0)}$, the kth point we move to is $x_{(k)} = x_{(k-1)} + \alpha_k d_k$ so we have

$$x_{(k)} = x_{(0)} + \sum_{j \leq k} \alpha_j d_j$$

and we use a convergence criterion such as given above for Gauss-Seidel. The directions are chosen to be the residuals, $b - Ax_{(k)}$. Here's the basic algorithm:

- Choose $x_{(0)}$ and define the residual, $r_{(0)} = b - Ax_{(0)}$ (the error on the scale of b) and the direction, $d_0 = r_{(0)}$ and set$k = 0$.
- Then iterate
  - $\alpha_k = \frac{r_{(k)}^\top r_{(k)}}{d_k^\top Ad_k}$ (choose step size so next error will be orthogonal to current direction - which we can express in terms of the residual, which is easily computable)

- $x_{(k+1)} = x_{(k)} + \alpha_k d_k$ (update current value)
- $r_{(k+1)} = r_{(k)} - \alpha_k A d_k$ (update current residual)
- $d_{k+1} = r_{(k+1)} + \frac{r_{(k+1)}^\top r_{(k+1)}}{r_{(k)}^\top r_{(k)}} d_k$ (choose next direction by conjugate Gram-Schmidt, starting with $r_{(k+1)}$ and removing components that are not A-orthogonal to previous directions, but it turns out that $r_{(k+1)}$ is already A-orthogonal to all but d_k).

- Stop when $\|r^{(k+1)}\|$ is sufficiently small.

The convergence of the algorithm depends in a complicated way on the eigenvalues, but in general convergence is faster when the condition number is smaller (the eigenvalues are not too spread out). CG will in principle give the exact answer in n steps (where A is $n \times n$). However, computationally we lose accuracy and interest in the algorithm is really as an iterative approximation where we stop before n steps. The approach basically amounts to moving in axis-oriented directions in a space stretched by A.

In general, CG is used for large sparse systems.

See the extensive description from Shewchuk for more details and for the figures shown in class, as well as the use of CG when A is not positive definite.

Updating a solution Sometimes we have solved a system, $Ax = b$ and then need to solve $Ax = c$. If we have solved the initial system using a factorization, we can reuse that factorization and solve the new system in $O(n^2)$. Iterative approaches can do a nice job if $c = b + \delta b$. Start with the solution x for $Ax = b$ as $x^{(0)}$ and use one of the methods above.

Simulation

November 1, 2012

References:

- Gentle: Computational Statistics
- Monahan: Numerical Methods of Statistics

Many (most?) statistical papers include a simulation (i.e., Monte Carlo) study. The basic idea is that closed form analysis of the properties of a statistical method/model is often hard to do. Even if possible, it usually involves approximations or simplifications. A canonical situation is that we have an asymptotic result and we want to know what happens in finite samples, but often we do not even have the asymptotic result. Instead, we can estimate mathematical expressions using random numbers. So we design a simulation study to evaluate the method/model or compare multiple methods. The result is that the statistician carries out an experiment, generally varying different factors to see what has an effect on the outcome of interest.

The basic strategy generally involves simulating data and then using the method(s) on the simulated data, summarizing the results to assess/compare the method(s).

Most simulation studies aim to approximate an integral, generally an expected value (mean, bias, variance, MSE, probability, etc.). In low dimensions, methods such as Gaussian quadrature are best for estimating an integral but these methods don't scale well [we'll discuss this in the next unit on integration/differentiation], so in higher dimensions we often use Monte Carlo techniques.

1 Monte Carlo considerations

1.1 Monte Carlo basics

The basic idea is that we often want to estimate $\mu \equiv E_f(h(X))$ for $X \sim f$. Note that if h is an indicator function, this includes estimation of probabilities, e.g., $p = P(X \leq x) = F(x) =$

$\int_{-\infty}^{x} f(t)dt = \int I(t \leq x)f(t)dt = E_f(I(X \leq x))$. We would estimate variances or MSEs by having h involve squared terms.

We get an MC estimate of μ based on an iid sample of a large number of values of X from f:

$$\hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} h(X_i),$$

which is justified by the Law of Large Numbers. The simulation variance of $\hat{\mu}$ is $\text{Var}(\hat{\mu}) = \sigma^2/m$, with $\sigma^2 = \text{Var}(h(X))$. An estimator of $\sigma^2 = E_f((h(X) - \mu)^2)$ is $\hat{\sigma}^2 = \frac{1}{m-1}\sum(h(X_i) - \hat{\mu})^2$. So our MC simulation error is based on

$$\widehat{\text{Var}}(\hat{\mu}) = \frac{1}{m(m-1)}\sum_{i=1}^{m}(h(X_i) - \hat{\mu})^2.$$

Sometimes the X_i are generated in a dependent fashion (e.g., sequential MC or MCMC), in which case this variance estimator does not hold because the samples are not IID, but the estimator $\hat{\mu}$ is still correct. [As a sidenote, a common misperception with MCMC is that you should thin your chains because of dependence of the samples. This is not correct - the only reason to thin a chain is if you want to save on computer storage or processing.]

Note that in this case the randomness in the system is very well-defined (as it is in survey sampling, but unlike in most other applications of statistics), because it comes from the RNG that we perform as part of our attempt to estimate μ.

Happily, we are in control of m, so in principle we can reduce the simulation error to as little as we desire. Unhappily, as usual, the standard error goes down with the square root of m.

1.2 Simple example

Suppose we've come up with a fabulous new estimator for the mean of a distribution. The estimator is to take the middle value of the sorted observations as our estimate of the mean of the entire distribution. We work out some theory to show that this estimator is robust to outlying observations and we come up with a snazzy new name for our estimator. We call it the 'median'. Let's denote it as $\tilde{X}$.

Unfortunately, we have no good way of estimating $\text{Var}(\tilde{X}) = E((\tilde{X} - E(\tilde{X}))^2)$ analytically. We decide to use a Monte Carlo estimate

$$\frac{1}{m}\sum_{i=1}^{m}(\tilde{X}_i - \bar{\tilde{X}})^2$$

where $\bar{\tilde{X}} = \frac{1}{m}\sum \tilde{X}_i$. Each $\tilde{X}_i$ in this case is generated by generating a dataset and calculating the

median. In evaluating the variance of the median and comparing it to our standard estimator, the sample mean, what decisions do we have to make in our Monte Carlo procedure?

1.3 Variance reduction

There are some tools for variance reduction in MC settings. One is importance sampling (see Section 3). Others are the use of control variates and antithetic sampling. I haven't personally run across these latter in practice, so I'm not sure how widely used they are and won't go into them here.

In some cases we can set up natural strata, for which we know the probability of being in each stratum. Then we would estimate μ for each stratum and combine the estimates based on the probabilities. The intuition is that we remove the variability in sampling amongst the strata from our simulation.

Another strategy that comes up in MCMC contexts is *Rao-Blackwellization*. Suppose we want to know $E(h(X))$ where $X = \{X_1, X_2\}$. Iterated expectation tells us that $E(h(X)) = E(E(h(X)|X_2)$. If we can compute $E(h(X)|X_2) = \int h(x_1, x_2)f(x_1|x_2)dx_1$ then we should avoid introducing stochasticity related to the X_1 draw (since we can analytically integrate over that) and only average over stochasticity from the X_2 draw by estimating $E_{X_2}(E(h(X)|X_2)$. The estimator is

$$\hat{\mu}_{RB} = \frac{1}{m}\sum_{i=1}^{m} E(h(X)|X_{2,i})$$

where we either draw from the marginal distribution of X_2, or equivalently, draw X, but only use X_2. Our MC estimator averages over the simulated values of X_2. This is called Rao-Blackwellization because it relates to the idea of conditioning on a sufficient statistic. It has lower variance because the variance of each term in the sum of the Rao-Blackwellized estimator is Var$(E(h(X)|X_2)$, which is less than the variance in the usual MC estimator, Var$(h(X))$, based on the usual iterated variance formula: $V(X) = E(V(X|Y)) + V(E(X|Y)) \Rightarrow V(E(X|Y)) < V(X)$.

2 Random number generation (RNG)

At the core of simulations is the ability to generate random numbers, and based on that, random variables. On a computer, our goal is to generate sequences of pseudo-random numbers that behave like random numbers but are replicable. The reason that replicability is important is so that we can reproduce the simulation.

2.1 Generating random uniforms on a computer

Generating a sequence of random standard uniforms is the basis for all generation of random variables, since random uniforms (either a single one or more than one) can be used to generate values from other distributions. Most random numbers on a computer are pseudo-random. The numbers are chosen from a deterministic stream of numbers that behave like random numbers but are actually a finite sequence (recall that both integers and real numbers on a computer are actually discrete and there are finitely many distinct values), so it's actually possible to get repeats. The seed of a RNG is the place within that sequence where you start to use the pseudo-random numbers.

Many RNG methods are sequential congruential methods. The basic idea is that the next value is

$$u_k = f(u_{k-1}, \ldots, u_{k-j}) \text{mod } m$$

for some function, f, and some positive integer m . Often $j = 1$. *mod* just means to take the remainder after dividing by m. One then generates the random standard uniform value as u_k/m, which by construction is in $[0, 1]$.

Given the construction, such sequences are periodic if the subsequence ever reappears, which is of course guaranteed because there is a finite number of possible values given that all the values are remainders of divisions by a fixed number . One key to a good random number generator (RNG) is to have a very long period.

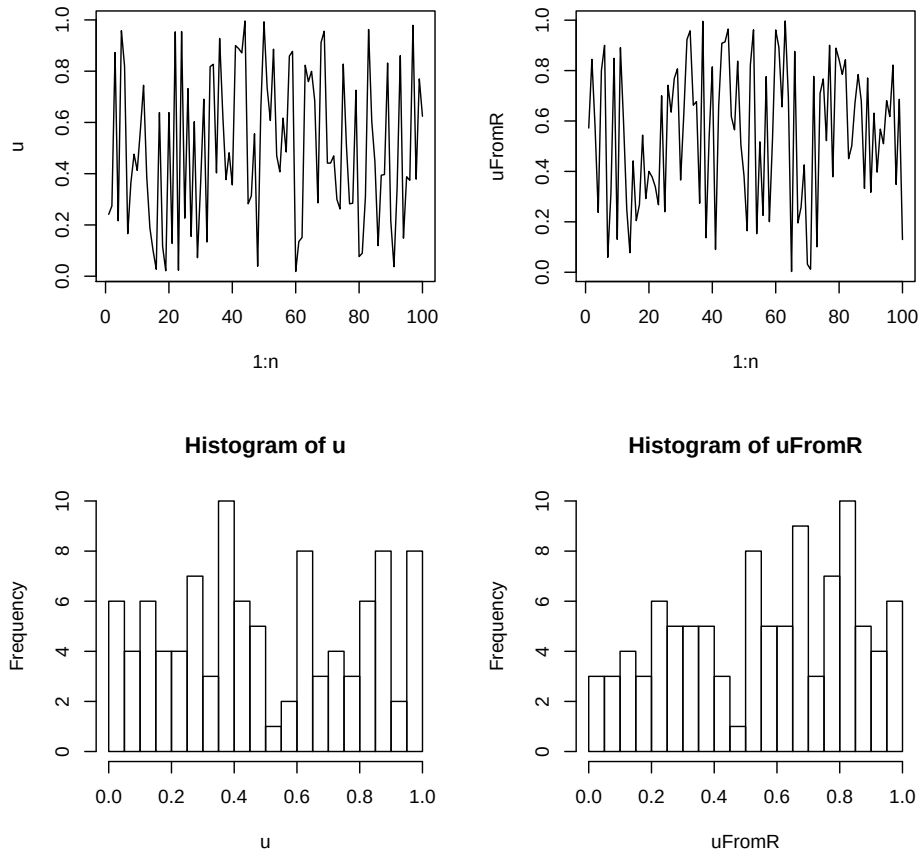
An example of a sequential congruential method is a basic linear congruential generator:

$$u_k = (au_{k-1}) \text{mod } m$$

with integer a, m, and u_i values. Here the periodicity can't exceed $m - 1$ (the method is set up so that we never get $u_k = 0$ as this causes the algorithm to break), so we only have $m - 1$ possible values. The seed is the initial state, u_0 - i.e., the point in the sequence at which we start. By setting the seed you guarantee reproducibility since given a starting value, the sequence is deterministic. In general a and m are chosen to be large, but of course they can't be too large if they are to be represented as computer integers. The standard values of m are Mersenne primes, which have the form $2^p - 1$ (but these are not prime for all p), with $m = 2^{31} - 1$ common. Here's an example of a linear congruential sampler:

```
n <- 100
a <- 171
m <- 30269
u <- rep(NA, n)
u[1] <- 7306
```

```
for (i in 2:n) u[i] <- (a * u[i - 1])%%m
u <- u/m
uFromR <- runif(n)
par(mfrow = c(2, 2))
plot(1:n, u, type = "l")
plot(1:n, uFromR, type = "l")
hist(u, nclass = 25)
hist(uFromR, nclass = 25)
```



A wide variety of different RNG have been proposed. Many have turned out to have substantial defects based on tests designed to assess if the behavior of the RNG mimics true randomness. Some of the behavior we want to ensure is uniformity of each individual random deviate, independence of sequences of deviates, and multivariate uniformity of subsequences. One test of a RNG that many RNGs don't perform well on is to assess the properties of k-tuples - subsequences of length k, which should be independently distributed in the k-dimensional unit hypercube. Unfortunately, linear congruential methods produce values that lie on a simple lattice in k-space, i.e., the points

are not selected from q^k uniformly spaced points, where q is the the number of unique values. Instead, points often lie on parallel lines in the hypercube.

Combining generators can yield better generators. The Wichmann-Hill is an option in R and is a combination of three linear congruential generators with $a = \{171, 172, 170\}$, $m = \{30269, 30307, 30323\}$, and $u_i = (x_i/30269 + y_i/30307 + z_i/30323) \bmod 1$ where x, y, and z are generated from the three individual generators. Let's mimic the Wichmann-Hill manually:

```
RNGkind("Wichmann-Hill")
set.seed(0)
saveSeed <- .Random.seed
uFromR <- runif(10)
a <- c(171, 172, 170)
m <- c(30269, 30307, 30323)
xyz <- matrix(NA, nr = 10, nc = 3)
xyz[1, ] <- (a * saveSeed[2:4])%%m
for (i in 2:10) xyz[i, ] <- (a * xyz[i - 1, ])%%m
for (i in 1:10) print(c(uFromR[i], sum(xyz[i, ]/m)%%1))
```

```
## [1] 0.4626 0.4626
## [1] 0.2658 0.2658
## [1] 0.5772 0.5772
## [1] 0.5108 0.5108
## [1] 0.3376 0.3376
## [1] 0.3576 0.3576
## [1] 0.413 0.413
## [1] 0.1329 0.1329
## [1] 0.255 0.255
## [1] 0.9202 0.9202
```

```
# we should be able to recover the current value of the seed
xyz[10, ]
```

```
## [1] 20696  2593  4576
```

```
.Random.seed[2:4]
```

```
## [1] 20696  2593  4576
```

By default R uses something called the Mersenne twister, which is in the class of generalized feedback shift registers (GFSR). The basic idea of a GFSR is to come up with a deterministic generator of bits (i.e., a way to generate sequences of 0s and 1s), B_i, $i = 1, 2, 3, \ldots$. The pseudo-random numbers are then determined as sequential subsequences of length L from $\{B_i\}$, considered as a base-2 number and dividing by 2^L to get a number in $(0, 1)$. In general the sequence of bits is generated by taking B_i to be the *exclusive or* [i.e., 0+0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 0] summation of two previous bits further back in the sequence where the lengths of the lags are carefully chosen.

Additional notes Generators should give you the same sequence of random numbers, starting at a given seed, whether you ask for a bunch of numbers at once, or sequentially ask for individual numbers.

When one invokes a RNG without a seed, they generally have a method for choosing a seed, often based on the system clock.

There have been some attempts to generate truly random numbers based on physical randomness. One that is based on quantum physics is http://www.idquantique.com/true-random-number-generator/quantis-usb-pcie-pci.html. Another approach is based on lava lamps!

2.2 RNG in R

We can change the RNG in R using *RNGkind()*. We can set the seed with *set.seed()*. The seed is stored in *.Random.seed*. The first element indicates the type of RNG (and the type of normal RV generator). The remaining values are specific to the RNG. In the demo code, we've seen that for Wichmann-Hill, the remaining three numbers are the current values of $\{x, y, z\}$.

In R the default RNG is the Mersenne twister (`?RNGkind`), which is considered to be state-of-the-art – it has some theoretical support, has performed reasonably on standard tests of pseudorandom numbers and has been used without evidence of serious failure. Plus it's fast (because bitwise operations are fast). In fact this points out one of the nice features of R, which is that for something as important as this, the default is generally carefully chosen by R's developers. The particular Mersenne twister used has a periodicity of $2^{19937} - 1 \approx 10^{6000}$. Practically speaking this means that if we generated one random uniform per nanosecond for 10 billion years, then we would generate 10^{25} numbers, well short of the period. So we don't need to worry about the periodicity! The seed for the Mersenne twister is a set of 624 32-bit integers plus a position in the set, where the position is `.Random.seed[2]`.

We can set the seed by passing an integer to *set.seed()*, which then sets as many actual seeds as required for a given generator. Here I'll refer to the integer passed to *set.seed()* as *the* seed. Ideally, nearby seeds generally should not correspond to getting sequences from the stream that are

closer to each other than far away seeds. According to Gentle (CS, p. 327) the input to *set.seed()* should be an integer, $i \in \{0, \ldots, 1023\}$, and each of these 1024 values produces positions in the RNG sequence that are "far away" from each other. I don't see any mention of this in the R documentation for *set.seed()* and furthermore, you can pass integers larger than 1023 to *set.seed()*, so I'm not sure how much to trust Gentle's claim. More on generating parallel streams of random numbers below.

So we get replicability by setting the seed to a specific value at the beginning of our simulation. We can then set the seed to that same value when we want to replicate the simulation.

```
set.seed(0)
rnorm(10)

##  [1] -0.09400  0.19476 -0.41913 -0.21971 -0.65887 -0.55566  0.08172
##  [8]  0.20599  0.97703 -0.07111

set.seed(0)
rnorm(10)

##  [1] -0.09400  0.19476 -0.41913 -0.21971 -0.65887 -0.55566  0.08172
##  [8]  0.20599  0.97703 -0.07111
```

We can also save the state of the RNG and pick up where we left off. So this code will pick where you had left off, ignoring what happened in between saving to *savedSeed* and resetting.

```
set.seed(0)
rnorm(5)

## [1] -0.0940  0.1948 -0.4191 -0.2197 -0.6589

savedSeed <- .Random.seed
tmp <- sample(1:50, 2000, replace = TRUE)
.Random.seed <- savedSeed
rnorm(5)

## [1] -0.55566  0.08172  0.20599  0.97703 -0.07111
```

In some cases you might want to reset the seed upon exit from a function so that a user's random number stream is unaffected:

```
f = function(args) {
    oldseed <- .Random.seed
    # other code
    .Random.seed <<- oldseed
}
```

Note the need to reassign to the global variable *.Random.seed*.

RNGversion() allows you to revert to RNG from previous versions of R, which is very helpful for reproducibility.

The RNGs in R generally return 32-bit (4-byte) integers converted to doubles, so there are at most 2^{32} distinct values. This means you could get duplicated values in long runs, but this does not violate the comment about the periodicity because the two values after the two duplicated numbers will not be duplicates of each other – note there is a distinction between the values as presented to the user and the values as generated by the RNG algorithm.

One way to proceed if you're using both R and C is to have C use the R RNG, using the *unif_rand* (corresponding to *runif()*) and *norm_rand* (corresponding to *rnorm()*) functions in the *Rmath* library. This way you have a consistent source of random numbers and don't need to worry about issues with RNG in C. If you call C from R, this should approach should also work; you could also generate all the random numbers you need in R and pass them to the C function.

Note that whenever a random number is generated, the software needs to retain information about what has been generated, so this is an example where a function must have a side effect not observed by the user. R frowns upon this sort of thing, but it's necessary in this case.

2.3 Random slippage

If the exact floating point representations of a random number sequence differ, even in the 14th, 15th, 16th decimal places, if you run a simulation long enough, such a difference can be enough to change the result of some conditional calculation. Suppose your code involves:

```
> if(x>0) { } else{ }
```

As soon as a small difference changes the result of testing *x>0*, the remainder of the simulation can change entirely. This happened to me in my thesis as a result of the difference of an AMD and Intel processor, and took a while to figure out.

2.4 RNG in parallel

We can generally rely on the RNG in R to give a reasonable set of values. One time when we want to think harder is when doing work with RNG in parallel on multiple processors. The worst

thing that could happen is that one sets things up in such a way that every process is using the same sequence of random numbers. This could happen if you mistakenly set the same seed in each process, e.g., using *set.seed(mySeed)* in R on every process.

The naive approach The naive approach is to use a different seed for each process. E.g., if your processes are numbered id=1,...,p, with *id* unique to a process, using `set.seed(id)` on each process. This is likely not to cause problems, but raises the danger that two (or more sequences) might overlap. For an algorithm with dependence on the full sequence, such as an MCMC, this probably won't cause big problems (though you likely wouldn't know if it did), but for something like simple simulation studies, some of your 'independent' samples could be exact replicates of a sample on another process.

The clunky but effective approach One approach to avoid the problem is to do all your RNG on one process and distribute the random deviates to the other processes, but this can be infeasible with many random numbers.

The sophisticated approach More generally to avoid this problem, the key is to use an algorithm that ensures sequences that do not overlap. In R, there are two packages that deal with this, *rlecuyer* and *rsprng*. We'll go over *rlecuyer*, as I've heard that rsprng is deprecated (though there is no evidence of this on CRAN) and *rsprng* is (at the moment) not available for the Mac.

The L'Ecuyer algorithm has a period of 2^{191}, which it divides into subsequences of length 2^{127}.

Here's how you initialize independent sequences on different processes when using the *parallel* package's parallel apply functionality (illustrated here with *parSapply()*.

```
require(parallel)

## Loading required package:  parallel

require(rlecuyer)

## Loading required package:  rlecuyer

nSims <- 250
testFun <- function(i) {
    val <- runif(1)
    return(val)
}
```

```
nCores <- 4
RNGkind()

## [1] "Wichmann-Hill" "Inversion"

cl <- makeCluster(nCores)
iseed <- 0
clusterSetRNGStream(cl = cl, iseed = iseed)
RNGkind()

## [1] "Wichmann-Hill" "Inversion"

# hmmm... clusterSetRNGStream() should set as 'L'Ecuyer-CMRG'
res <- parSapply(cl, 1:nSims, testFun)
clusterSetRNGStream(cl = cl, iseed = iseed)
res2 <- parSapply(cl, 1:nSims, testFun)
identical(res, res2)

## [1] TRUE

stopCluster(cl)
```

If you want to explicitly move from stream to stream, you can use *nextRNGStream()*. For example:

```
RNGkind("L'Ecuyer-CMRG")
seed <- 0
set.seed(seed)  ## now start M workers
s <- .Random.seed
for (i in 1:M) {
    s <- nextRNGStream(s)
    # now send s to worker i as .Random.seed
}
```

When using *mclapply()*, you can use the *mc.set.seed* argument as follows (note that *mc.set.seed* is TRUE by default, but one needs to invoke *RNGkind()*.

```
require(parallel)
require(rlecuyer)
RNGkind("L'Ecuyer-CMRG")
res <- mclapply(seq_len(nSims), testFun,
mc.cores = nSlots, mc.set.seed = TRUE)
# this also seems to auotmatically reset the seed when it is run
res2 <- mclapply(seq_len(nSims), testFun,
mc.cores = nSlots, mc.set.seed = TRUE)
identical(res,res2)
```

foreach One question is whether *foreach* deals with RNG correctly. I don't see any documentation on this, but the developers (Revolution Analytics) are well aware of RNG issues, so it seems plausible that *foreach* handles things for you. **Challenge**: can we think of a way to test whether foreach is dealing with RNG correctly? What are the difficulties in doing so?

There is also a package called *doRNG* that ensures that *foreach* loops are reproducible, but in my basic test, it appears that *foreach* loops are reproducible without *doRNG*, so I'm not sure what the deal is here.

3 Generating random variables

There are a variety of methods for generating from common distributions (normal, gamma, beta, Poisson, t, etc.). Since these tend to be built into R and presumably use good algorithms, we won't go into them. A variety of statistical computing and Monte Carlo books describe the various methods. Many are built on the relationships between different distributions - e.g., a beta random variable (RV) can be generated from two gamma RVs.

Also note that you can call the C functions that implement the R distribution functions as a library (*Rmath*), so if you're coding in C or another language, you should be able to make use of the standard functions: *{r,p,q,d}{norm,t,gamma,binom,pois,etc.}* (as well as a variety of other R math functions, which can be seen in *Rmath.h*). Phil Spector has a writeup on this (http://www.stat.berkeley.edu/classes/s243/rmath.html) and material can also be found in the *Writing R Extensions* manual on CRAN (section 6.16).

3.1 Multivariate distributions

The *mvtnorm* package supplies code for working with the density and CDF of multivariate normal and t distributions.

To generate a multivariate normal, we've seen the standard method based on the Cholesky decomposition:

```
U <- chol(covMat)
crossprod(U, nrow(covMat))
```

For a singular covariance matrix we can use the Cholesky with pivoting, setting as many rows to zero as the rank deficiency. Then when we generate the multivariate normals, they respect the constraints implicit in the rank deficiency.

3.2 Inverse CDF

Most of you know the inverse CDF method. To generate $X \sim F$ where F is a CDF and is an invertible function, first generate $Z \sim \mathcal{U}(0,1)$, then $x = F^{-1}(z)$. For discrete CDFs, one can work with a discretized version. For multivariate distributions, one can work with a univariate marginal and then a sequence of univariate conditionals: $f(x_1)f(x_2|x_1)\cdots f(x_k|x_{k-1},\ldots,x_1)$, when the distribution allows this analytic decomposition.

3.3 Rejection sampling

The basic idea of rejection sampling (RS) relies on the introduction of an auxiliary variable, u. Suppose $X \sim F$. Then we can write $f(x) = \int_0^{f(x)} du$. Thus X is the marginal density of X in the joint density, $(X,U) \sim \mathcal{U}\{(x,u) : 0 < u < f(x)\}$. Now we'd like to use this in a way that relies only on evaluating $f(x)$ without having to draw from f.

To implement this we draw from a larger set and then only keep draws for which $u < f(x)$. We choose a density, g, that is easy to draw from and that can *majorize* f, which means there exists a constant c s.t. , $cg(x) \geq f(x)\ \forall x$. In other words we have that $cg(x)$ is an upper envelope for $f(x)$. The algorithm is

1. generate $x \sim g$
2. generate $u \sim \mathcal{U}(0,1)$
3. if $u \leq f(x)/cg(x)$ then use x; otherwise go back to step 1

The intuition here is graphical: we generate from under a curve that is always above $f(x)$ and accept only when u puts us under $f(x)$ relative to the majorizing density. A key here is that the majorizing density have fatter tails than the density of interest, so that the constant c can exist. So we could use a t to generate from a normal but not the reverse. We'd like c to be small to reduce

the number of rejections because it turns out that $\frac{1}{c} = \frac{\int f(x)dx}{\int cg(x)dx}$ is the acceptance probability. This approach works in principle for multivariate densities but as the dimension increases, the proportion of rejections grows, because more of the volume under $cg(x)$ is above $f(x)$.

If f is costly to evaluate, we can sometimes reduce calculation using a lower bound on f. In this case we accept if $u \leq f_{\text{low}}(y)/cg_Y(y)$. If it is not, then we need to evaluate the ratio in the usual rejection sampling algorithm. This is called squeezing.

One example of RS is to sample from a truncated normal. Of course we can just sample from the normal and then reject, but this can be inefficient, particularly if the truncation is far in the tail (a case in which inverse CDF suffers from numerical difficulties). Suppose the truncation point is greater than zero. Working with the standardized version of the normal, you can use an translated exponential with lower end point equal to the truncation point as the majorizing density (Robert 1995; Statistics and Computing, and see calculations in the demo code). For truncation less than zero, just make the values negative.

3.4 Adaptive rejection sampling

The difficulty of RS is finding a good enveloping function. Adaptive rejection sampling refines the envelope as the draws occur, in the case of a continuous, differentiable, log-concave density. The basic idea considers the log of the density and involves using tangents or secants to define an upper envelope and secants to define a lower envelope for a set of points in the support of the distribution. The result is that we have piecewise exponentials (since we are exponentiating from straight lines on the log scale) as the bounds. We can sample from the upper envelope based on sampling from a discrete distribution and then the appropriate exponential. The lower envelope is used for squeezing. We add points to the set that defines the envelopes whenever we accept a point that requires us to evaluate $f(x)$ (the points that are accepted based on squeezing are not added to the set). We'll talk this through some in class.

3.5 Importance sampling

Importance sampling (IS) allows us to estimate expected values, with some commonalities with rejection sampling.

$$E_f(h(X)) = \int h(x)\frac{f(x)}{g(x)}g(x)dx$$

so $\hat{\mu} = \frac{1}{m}\sum_i h(x_i)\frac{f(x_i)}{g(x_i)}$ for x_i drawn from $g(x)$, where $w_i^* = f(x_i)/g(x_i)$ act as weights. Often in Bayesian contexts, we know $f(x)$ only up to a normalizing constant. In this case we need to use

$w_i = w_i^* / \sum_i w_i^*$.

Here we don't require the majorizing property, just that the densities have common support, but things can be badly behaved if we sample from a density with lighter tails than the density of interest. So in general we want g to have heavier tails. More specifically for a low variance estimator of μ, we would want that $f(x_i)/g(x_i)$ is large only when $h(x_i)$ is very small, to avoid having overly influential points.

This suggests we can reduce variance in an IS context by oversampling x for which $h(x)$ is large and undersampling when it is small, since $\text{Var}(\hat{\mu}) = \frac{1}{m}\text{Var}(h(X)\frac{f(X)}{g(X)})$. An example is that if h is an indicator function that is 1 only for rare events, we should oversample rare events and then the IS estimator corrects for the oversampling.

What if we actually want a sample from f as opposed to estimating the expected value above? We can draw x from the unweighted sample, $\{x_i\}$, with weights $\{w_i\}$. This is called sampling importance resampling (SIR).

3.6 Ratio of uniforms

If U and V are uniform in $C = \{(u,v) : 0 \leq u \leq \sqrt{f(v/u)}$ then $X = V/U$ has density proportion to f. The basic algorithm is to choose a rectangle that encloses C and sample until we find $u \leq f(v/u)$. Then we use $x = v/u$ as our RV. The larger region enclosing C is the majorizing region and a simple approach (if $f(x)$and $x^2 f(x)$ are bounded in C) is to choose the rectangle, $0 \leq u \leq \sup_x \sqrt{f(x)}$, $\inf_x x\sqrt{f(x)} \leq v \leq \sup_x x\sqrt{f(x)}$.

One can also consider truncating the rectangular region, depending on the features of f.

Monahan recommends the ratio of uniforms, particularly a version for discrete distributions (p. 323 of the 2nd edition).

4 Design of simulation studies

Let's pose a concrete example. This is based on a JASA paper that you'll look at more carefully in a problem set. Suppose one is modeling data as being from a mixture of normal distributions:

$$f(y;\theta) = \sum_{h=1}^{m} w_k f(y;\mu_h,\sigma_h)$$

where $f(y;\mu_h,\sigma_h)$ is a normal density with mean μ_h and s.d. σ_h. A statistician has developed methodology for carrying out a hypothesis test for $H_0 : m = m_0$ vs. $H_a : m > m_0$.

First, what are the key issues that need to be assessed to evaluate their methodology? What do we want to know to assess a hypothesis test?

Second, what do we need to consider in carrying out a simulation study to address those issues?

4.1 Basic steps of a simulation study

1. Specify what makes up an individual experiment: sample size, distributions, parameters, statistic of interest, etc.

2. Determine what inputs, if any, to vary; e.g., sample sizes, parameters, data generating mechanisms

3. Write code to carry out the individual experiment and return the quantity of interest

4. For each combination of inputs, repeat the experiment m times. Note this is an embarrassingly parallel calculation (in both the data generating dimension and the inputs dimension(s).

5. Summarize the results for each combination of interest, quantifying simulation uncertainty

6. Report the results in graphical or tabular form

4.2 Overview

Since a simulation study is an experiment, we should use the same principles of design and analysis we would recommend when advising a practicioner on setting up an experiment.

These include efficiency, reporting of uncertainty, reproducibility and documentation.

In generating the data for a simulation study, we want to think about what structure real data would have that we want to mimic in the simulation study: distributional assumptions, parameter values, dependence structure, outliers, random effects, sample size (n), etc.

All of these may become input variables in a simulation study. Often we compare two or more statistical methods conditioning on the data context and then assess whether the differences between methods vary with the data context choices. E.g., if we compare an MLE to a robust estimator, which is better under a given set of choices about the data generating mechanism and how sensitive is the comparison to changing the features of the data generating mechanism? So the "treatment variable" is the choice of statistical method. We're then interested in sensitivity to the conditions.

Often we can have a large number of replicates (m) because the simulation is fast on a computer, so we can sometimes reduce the simulation error to essentially zero and thereby avoid reporting uncertainty. To do this, we need to calculate the simulation standard error, generally, $s/\sqrt{m}$ and see how it compares to the effect sizes. This is particularly important when reporting on the bias of a statistical method.

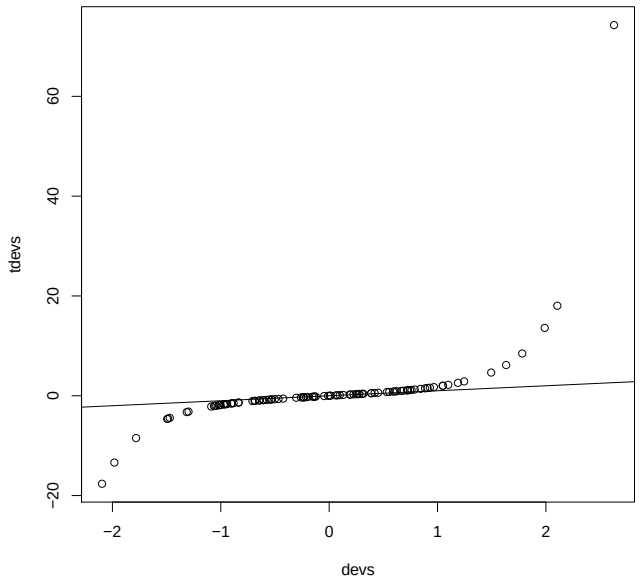
We might denote the data, which could be the statistical estimator under each of two methods as Y_{ijklq}, where i indexes treatment, j, k, l index different additional input variables, and $q \in \{1, \ldots, m\}$ indexes the replicate. E.g., j might index whether the data are from a t or normal, k the value of a parameter, and l the dataset sample size (i.e., different levels of n).

One can think about choosing m based on a basic power calculation, though since we can always generate more replicates, one might just proceed sequentially and stop when the precision of the results is sufficient.

When comparing methods, it's best to use the same simulated datasets for each level of the treatment variable and to do an analysis that controls for the dataset (i.e., for the random numbers used), thereby removing some variability from the error term. A simple example is to do a paired analysis, where we look at differences between the outcome for two statistical methods, pairing based on the simulated dataset.

One can even use the "same" random number generation for the replicates under different conditions. E.g., in assessing sensitivity to a t vs. normal data generating mechanism, we might generate the normal RVs and then for the t use the same random numbers, in the sense of using the same quantiles of the t as were generated for the normal - this is pretty easy, as seen below. This helps to control for random differences between the datasets.

```
devs <- rnorm(100)
tdevs <- qt(pnorm(devs), df = 1)
plot(devs, tdevs)
abline(0, 1)
```



4.3 Experimental Design

A typical context is that one wants to know the effect of multiple input variables on some outcome. Often, scientists, and even statisticians doing simulation studies will vary one input variable at a time. As we know from standard experimental design, this is inefficient.

The standard strategy is to discretize the inputs, each into a small number of levels. If we have a small enough number of inputs and of levels, we can do a full factorial design (potentially with replication). For example if we have three inputs and three levels each, we have 3^3 different treatment combinations. Choosing the levels in a reasonable way is obviously important.

As the number of inputs and/or levels increases to the point that we can't carry out the full factorial, a fractional factorial is an option. This carefully chooses which treatment combinations to omit. The goal is to achieve balance across the levels in a way that allows us to estimate lower level effects (in particular main effects) but not all high-order interactions. What happens is that high-order interactions are aliased to (confounded with) lower-order effects. For example you might choose a fractional factorial design so that you can estimate main effects and two-way interactions but not higher-order interactions.

In interpreting the results, I suggest focusing on the decomposition of sums of squares and not on statistical significance. In most cases, we expect the inputs to have at least some effect on the outcome, so the null hypothesis is a straw man. Better to assess the magnitude of the impacts of the different inputs.

When one has a very large number of inputs, one can use the Latin hypercube approach to sample in the input space in a uniform way, spreading the points out so that each input is sampled uniformly. Assume that each input is $\mathcal{U}(0, 1)$ (one can easily transform to whatever marginal distributions you want). Suppose that you can run m samples. Then for each input variable, we divide the unit interval into m bins and randomly choose the order of bins and the position within each bin. This is done independently for each variable and then combined to give m samples from the input space. We would then analyze main effects and perhaps two-way interactions to assess which inputs seem to be most important.

5 Implementation of simulation studies

5.1 Computational efficiency

Parallel processing is often helpful for simulation studies. The reason is that simulation studies are embarrassingly parallel - we can send each replicate to a different computer processor and then collect the results back, and the speedup should scale directly with the number of processors we used. Since we often need to some sort of looping, writing code in C/C++ and compiling and

linking to the code from R may also be a good strategy, albeit one not covered in this course.

Handy functions in R include *expand.grid()* to get all combinations of a set of vectors and the *replicate()* function in R, which will carry out the same R expression (often a function call) repeated times. This can replace the use of a *for* loop with some gains in cleanliness of your code. Storing results in an array is a natural approach.

```
thetaLevels <- c("low", "med", "hi")
n <- c(10, 100, 1000)
tVsNorm <- c("t", "norm")
levels <- expand.grid(thetaLevels, tVsNorm, n)
# example of replicate() -- generate m sets correlated normals
set.seed(0)
genFun <- function(n, theta = 1) {
    u <- rnorm(n)
    x <- runif(n)
    Cov <- exp(-rdist(x)/theta)
    U <- chol(Cov)
    return(cbind(x, crossprod(U, u)))
}
m <- 20
simData <- replicate(m, genFun(100, 1))
dim(simData)  # 100 observations by {x, y} values by 20 replicates

## [1] 100   2  20
```

5.2 Analysis and reporting

Often results are reported simply in tables, but it can be helpful to think through whether a graphical representation is more informative (sometimes it's not or it's worse, but in some cases it may be much better).

You should set the seed when you start the experiment, so that it's possible to replicate it. It's also a good idea to save the current value of the seed whenever you save interim results, so that you can restart simulations (this is particularly helpful for MCMC) at the exact point you left off, including the random number sequence.

To enhance reproducibility, it's good practice to post your simulation code (and potentially data) on your website or as supplementary material with the journal. One should report sample sizes and information about the random number generator.

Here are JASA's requirements on documenting computations:

"Results Based on Computation - Papers reporting results based on computation should provide enough information so that readers can evaluate the quality of the results. Such information includes estimated accuracy of results, as well as descriptions of pseudorandom-number generators, numerical algorithms, computers, programming languages, and major software components that were used."

Numerical Integration and Differentiation

November 8, 2012

References:

- Gentle: Computational Statistics
- Monahan: Numerical Methods of Statistics
- Givens and Hoeting: Computational Statistics

Our goal here is to understand the basics of numerical (and symbolic) approaches to approximating derivatives and integrals on a computer. Derivatives are useful primarily for optimization. Integrals arise in approximating expected values and in various places where we need to integrate over an unknown random variable (e.g., Bayesian contexts, random effects models, missing data contexts).

1 Differentiation

1.1 Numerical differentiation

There's not much to this topic. The basic idea is to approximate the derivative of interest using finite differences.

A standard discrete approximation of the derivative is the forward difference

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

A more accurate approach is the central difference

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Provided we already have computed $f(x)$, the forward difference takes half as much computing as the central difference. However, the central difference has an error of $O(h^2)$ while the forward difference has error of $O(h)$.

For second derivatives, if we apply the above approximations to $f'(x)$ and $f'(x+h)$, we get an approximation of the second derivative based on second differences:

$$f''(x) \approx \frac{f'(x+h) - f'(x)}{h} \approx \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2}.$$

The corresponding central difference approximation is

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.$$

For multivariate x, we need to compute directional derivatives. In general these will be in axis-oriented directions (e.g., for the Hessian), but they can be in other directions. The basic idea is to find $f(x+he)$ in expressions such as those above where e is a unit length vector giving the direction. For axis oriented directions, we have e_i being a vector with a one in the ith position and zeroes in the other positions,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x+he_i) - f(x)}{h}.$$

Note that for mixed partial derivatives, we need to use e_i and e_j, so the second difference approximation gets a bit more complicated,

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \approx \frac{f(x+he_j+he_i) - f(x+he_j) - f(x+he_i) + f(x)}{h^2}.$$

We would have analogous quantities for central difference approximations.

Numerical issues Ideally we would take h very small and get a highly accurate estimate of the derivative. However, the limits of machine precision mean that the difference estimator can behave badly for very small h, since we lose accuracy in computing differences such as between $f(x+h)$ and $f(x-h)$ and from dividing by small h. Therefore we accept a bias in the estimate by not using h so small, often by taking h to be square root of machine epsilon (i.e., about 1×10^{-8} on most systems). Actually, we need to account for the order of magnitude of x, so what we really want is $h = \sqrt{\epsilon}|x|$ - i.e., we want it to be in terms relative to the magnitude of x. As an example, recall that if $x = 1 \times 10^9$ and we did $x + h = 1 \times 10^9 + 1 \times 10^{-8}$, we would get $x + h = 1 \times 10^9 = x$ because we can only represent 7 decimal places with precision.

Givens and Hoeting and Monahan point out that some sources recommend the cube root of machine epsilon (about 5×10^{-6} on most systems), in particular when approximating second

derivatives.

Let's assess these recommendations empirically in R. We'll use a test function, $\log \Gamma(x)$, for which we can obtain the derivatives with high accuracy using built-in R functions. This is a modification of Monahan's example from his *numdif.r* code.

```
# compute first and second derivatives of log(gamma(x)) at x=1/2
options(digits = 9, width = 120)
h <- 10^(-(1:15))
x <- 1/2
fx <- lgamma(x)
# targets: actual derivatives can be computed very accurately using
# built-in R functions:
digamma(x)  # accurate first derivative

## [1] -1.96351003

trigamma(x)  # accurate second derivative

## [1] 4.9348022

# calculate discrete differences
fxph <- lgamma(x + h)
fxmh <- lgamma(x - h)
fxp2h <- lgamma(x + 2 * h)
fxm2h <- lgamma(x - 2 * h)
# now find numerical derivatives
fp1 <- (fxph - fx)/h  # forward difference
fp2 <- (fxph - fxmh)/(2 * h)  # central difference
# second derivatives
fpp1 <- (fxp2h - 2 * fxph + fx)/(h * h)  # forward difference
fpp2 <- (fxph - 2 * fx + fxmh)/(h * h)  # central difference
# table of results
cbind(h, fp1, fp2, fpp1, fpp2)

##            h         fp1         fp2             fpp1             fpp2
##  [1,] 1e-01 -1.74131085 -1.99221980  3.67644733e+00  5.01817899e+00
##  [2,] 1e-02 -1.93911250 -1.96379057  4.77200996e+00  4.93561416e+00
##  [3,] 1e-03 -1.96104543 -1.96351283  4.91803003e+00  4.93481032e+00
```

```
##  [4,] 1e-04 -1.96326331 -1.96351005  4.93311987e+00  4.93480230e+00
##  [5,] 1e-05 -1.96348535 -1.96351003  4.93463270e+00  4.93480257e+00
##  [6,] 1e-06 -1.96350756 -1.96351003  4.93505237e+00  4.93483032e+00
##  [7,] 1e-07 -1.96350978 -1.96351003  4.91828800e+00  4.95159469e+00
##  [8,] 1e-08 -1.96351001 -1.96351003  7.77156117e+00  3.33066907e+00
##  [9,] 1e-09 -1.96351002 -1.96351002 -1.11022302e+02  0.00000000e+00
## [10,] 1e-10 -1.96351047 -1.96351047  2.22044605e+04  0.00000000e+00
## [11,] 1e-11 -1.96349603 -1.96350158 -3.33066907e+06  1.11022302e+06
## [12,] 1e-12 -1.96342942 -1.96348493 -1.11022302e+08  1.11022302e+08
## [13,] 1e-13 -1.96398453 -1.96398453  2.22044605e+10  0.00000000e+00
## [14,] 1e-14 -1.96509475 -1.97064587  1.11022302e+12  1.11022302e+12
## [15,] 1e-15 -1.99840144 -1.94289029  0.00000000e+00 -1.11022302e+14
```

What do we conclude about the advice about using h proportional to either the square root or cube root of machine epsilon?

1.2 Numerical differentiation in R

There are multiple numerical derivative functions in R. *numericDeriv()* will do the first derivative. It requires an expression rather than a function as the form in which the function is input, which in some cases might be inconvenient. The functions in the *numDeriv* package will compute the gradient and Hessian, either in the standard way (using the argument `method = 'simple'`) or with a more accurate approximation (using the argument `method = 'Richardson'`). For optimization, one might use the simple option, assuming that is faster, while the more accurate approximation might be good for computing the Hessian to approximate the information matrix for getting an asymptotic covariance. (Although in this case, the statistical uncertainty generally will ovewhelm any numerical uncertainty.)

```
x <- 1/2
numericDeriv(quote(lgamma(x)), "x")

## [1] 0.572364943
## attr(,"gradient")
##             [,1]
## [1,] -1.96351001
```

Note that by default, if you rely on numerical derivatives in *optim()*, it uses $h = 0.001$ (the *ndeps* sub-argument to *control*), which might not be appropriate if the parameters vary on a small

scale. This relatively large value of h is probably chosen based on *optim()* assuming that you've scaled the parameters as described in the text describing the *parscale* argument.

1.3 Symbolic differentiation

We've seen that we often need the first and second derivatives for optimization. Numerical differentiation is fine, but if we can readily compute the derivatives in closed form, that can improve our optimization. (Venables and Ripley comment that this is particularly the case for the first derivative, but not as much for the second.)

In general, using a computer program to do the analytic differentiation is recommended as it's easy to make errors in doing differentiation by hand. Monahan points out that one of the main causes of error in optimization is human error in coding analytic derivatives, so it's good practice to avoid this. R has a simple differentiation ability in the *deriv()* function (which handles the gradient and the Hessian). However it can only handle a limited number of functions. Here's an example of using *deriv()* and then embedding the resulting R code in a user-defined function. This can be quite handy, though the format of the result in terms of attributes is not the most handy, so you might want to monkey around with the code more in practice.

```
deriv(quote(atan(x)), "x")  # derivative of simple expression

## expression({
##     .value <- atan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## })

# derivative of a function; note we need to pass in an expression,
# not the entire function
f <- function(x,y) sin(x * y)+x^3+exp(y)
newBody <- deriv(body(f), c("x", "y"), hessian = TRUE)
# now create a new version of f that provides gradient
# and hessian as attributes of the output,
# in addition to the function value as the return value
f <- function(x, y) {} # function template
body(f) <- newBody
```

```
# try out the new function
f(3,1)

## [1] 29.8594018
## attr(,"gradient")
##                x            y
## [1,] 26.0100075 -0.251695661
## attr(,"hessian")
## , , x
##
##              x           y
## [1,] 17.85888 -1.41335252
##
## , , y
##
##                 x          y
## [1,] -1.41335252 1.44820176

attr(f(3,1), "gradient")

##                x            y
## [1,] 26.0100075 -0.251695661

attr(f(3,1), "hessian")

## , , x
##
##              x           y
## [1,] 17.85888 -1.41335252
##
## , , y
##
##                 x          y
## [1,] -1.41335252 1.44820176
```

For more complicated functions, both Maple and Mathematica do symbolic differentiation. Here are some examples in Mathematica, which is available on the SCF machines and through campus: http://ist.berkeley.edu/software-central:

```
# first partial derivative wrt x
D[ Exp[x^n] - Cos[x y], x]
# second partial derivative
D[ Exp[x^n] - Cos[x y], {x, 2}]
# partials
D[ Exp[x^n] - Cos[x y], x, y]
# trig function example
D[ ArcTan[x], x]
```

2 Integration

We've actually already discussed numerical integration extensively in the simulation unit, where we considered Monte Carlo approximation of high-dimensional integrals. In the case where we have an integral in just one or two dimensions, MC is fine, but we can get highly-accurate, very fast approximations by numerical integration methods known as quadrature. Unfortunately such approximations scale very badly as the dimension grows, while MC methods scale well, so MC is recommended in higher dimensions. Here's an empirical example in R, where the MC estimator is

$$\int_0^\pi \sin(x)dx = \int_0^\pi \pi \sin(x) \left(\frac{1}{\pi} \cdot 1\right) dx = E_f(\pi \sin(x))$$

for $f = \mathcal{U}(0, \pi)$:

```r
f <- function(x) sin(x)
# mathematically, the integral from 0 to pi is 2
# quadrature through integrate()
integrate(f, 0, pi)

## 2 with absolute error < 2.2e-14

system.time(integrate(f, 0, pi))

##    user  system elapsed
##       0       0       0

# MC estimate
ninteg <- function(n) mean(sin(runif(n, 0, pi))*pi)
n <- 1000
ninteg(n)
```

```
## [1] 1.93447152

system.time(ninteg(n))

##    user  system elapsed
##       0       0       0

n <- 10000
ninteg(n)

## [1] 1.98920619

system.time(ninteg(n))

##    user  system elapsed
##   0.000   0.000   0.001

n <- 1000000
ninteg(n)

## [1] 1.99924616

system.time(ninteg(n))

##    user  system elapsed
##   0.140   0.000   0.141

# that was fairly slow,
# especially if you need to do a lot of individual integrals
```

More on this issue below.

2.1 Numerical integration methods

The basic idea is to break the domain into pieces and approximate the integral within each piece:

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx,$$

where we then approximate $\int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{j=0}^{m} A_{ij} f(x_{ij}^*)$ where x_{ij}^* are the *nodes*.

2.1.1 Newton-Cotes quadrature

Newton-Cotes quadrature has equal length intervals of length $h = (b-a)/n$, with the same number of nodes in each interval. $f(x)$ is replaced with a polynomial approximation in each interval and A_{ij} are chosen so that the sum equals the integral of the polynomial approximation on the interval.

A basic example is the *Riemann rule*, which takes a single node, $x_i^* = x_i$ and the "polynomial" is a constant, $f(x_i^*)$, so we have

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx (x_{i+1} - x_i)f(x_i).$$

Of course using a piecewise constant to approximate $f(x)$ is not likely to give us high accuracy.

The *trapezoidal rule* takes $x_{i0}^* = x_i$, $x_{i1}^* = x_{i+1}$ and uses a linear interpolation between $f(x_{i0}^*)$ and $f(x_{i1}^*)$ to give

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \left(\frac{x_{i+1} - x_i}{2}\right)(f(x_i) + f(x_{i+1})).$$

Simpson's rule uses a quadratic interpolation at the points $x_{i0}^* = x_i$, $x_{i1}^* = (x_i + x_{i+1})/2$, $x_{i2}^* = x_{i+1}$ to give

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \left(\frac{x_{i+1} - x_i}{6}\right)\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right).$$

The error of various rules is often quantified as a power of $h = x_{i+1} - x_i$. The trapezoid rule gives $O(h^2)$ while Simpson's rule gives $O(h^4)$.

Romberg quadrature There is an extension of Newton-Cotes quadrature that takes combinations of estimates based on different numbers of intervals. This is called Richardson extrapolation and when used with the trapezoidal rule is called *Romberg quadrature*. The result is greatly increased accuracy. A simple example of this is as follows. Let $\hat{T}(h)$ be the trapezoidal rule approximation of the integral when the length of each interval is h. Then $\frac{4\hat{T}(h/2)-\hat{T}(h)}{3}$ results in an approximation with error of $O(h^4)$ because the differencing is cleverly chosen to kill off the error term that is $O(h^2)$. In fact this approximation is Simpson's rule with intervals of length $h/2$, with the advantage that we don't have to do as many function evaluations ($2n$ vs. $4n$). Even better, one can iterate this approach for more accuracy as described in detail in Givens and Hoeting.

Note that at some point, simply making intervals smaller in quadrature will not improve accuracy because of errors introduced by the imprecision of computer numbers.

2.1.2 Gaussian quadrature

Here the idea is to relax the constraints of equally-spaced intervals and nodes within intervals. We want to put more nodes where the function is larger in magnitude.

Gaussian quadrature approximates integrals that are in the form of an expected value as

$$\int_a^b f(x)\mu(x)dx \approx \sum_{i=0}^{m} w_i f(x_i)$$

where $\mu(x)$ is a probability density, with the requirement that $\int x^k \mu(x)dx = E_\mu X^k < \infty$ for $k \geq 0$. Note that it can also deal with indefinite integrals where $a = -\infty$ and/or $b = \infty$. Typically μ is non-uniform, so the nodes (the quadrature points) cluster in areas of high density.The choice of node locations depends on understanding orthogonal polynomials, which we won't go into here.

It turns out this approach can exactly integrate polynomials of degree $2m + 1$ (or lower). The advantage is that for smooth functions that can be approximated well by a single polynomial, we get highly accurate results. The downside is that if the function is not well approximated by such a polynomial, the result may not be so good. The Romberg approach is more robust.

Note that if the problem is not in the form $\int_a^b f(x)\mu(x)dx$, but rather $\int_a^b f(x)dx$, we can reexpress as $\int_a^b \frac{f(x)}{\mu(x)}\mu(x)dx$.

Note that the trapezoidal rule amounts to μ being the uniform distribution with the points equally spaced.

2.1.3 Adaptive quadrature

Adaptive quadrature chooses interval lengths based on the behavior of the integrand. The goal is to have shorter intervals where the function varies more and longer intervals where it varies less. The reason for avoiding short intervals everywhere involves the extra computation and greater opportunity for rounding error.

2.1.4 Higher dimensions

For rectangular regions, one can use the techniques described above over squares instead of intervals, but things become more difficult with more complicated regions of integration.

The basic result for Monte Carlo integration (i.e., Unit 10 on simulation) is that the error of the MC estimator scales as $O(m^{-1/2})$, where m is the number of MC samples, regardless of dimensionality. Let's consider how the error of quadrature scales. We've seen that the error is often quantified as $O(h^q)$. In d dimensions, the error is the same as a function of h, but if in one dimension we need n function evaluations to get intervals of length h, in d dimensions, we need n^d function evaluations to get hypercubes with sides of length h. Let's re-express the error in terms

of n rather than h based on $h = c/n$ for a constant c (such as $c = b - a$), which gives us error of $O(n^{-q})$ for one-dimensional integration. In d dimensions we have $n^{1/d}$ function evaluations per dimension, so the error for fixed n is $O((n^{1/d})^{-q}) = O(n^{-q/d})$ which scales as $n^{-1/d}$. As an example, suppose $d = 10$ and we have $n = 1000$ function evaluations. This gives us an accuracy comparable to one-dimensional integration with $n = 1000^{1/10} \approx 2$, which is awful. Even with only $d = 4$, we get $n = 1000^{1/4} \approx 6$, which is pretty bad. This is one version of the curse of dimensionality.

2.2 Numerical integration in R

R implements an adaptive version of Gaussian quadrature in *integrate()*. The '...' argument allows you to pass additional arguments to the function that is being integrated. The function must be vectorized (i.e., accept a vector of inputs and evaluate and return the function value for each input as a vector of outputs).

Note that the domain of integration can be unbounded and if either the upper or lower limit is unbounded, you should enter **`Inf`** or **`-Inf`** respectively.

```
integrate(dnorm, -Inf, Inf, 0, 0.1)

## 1 with absolute error < 6.1e-07

integrate(dnorm, -Inf, Inf, 0, 0.001)

## 1 with absolute error < 2.1e-06

integrate(dnorm, -Inf, Inf, 0, 1e-04)  # THIS FAILS!

## 0 with absolute error < 0
```

2.3 Singularities and infinite ranges

A singularity occurs when the function is unbounded, which can cause difficulties with numerical integration. For example, $\int_0^1 \frac{1}{\sqrt{x}} = 2$, but $f(0) = \infty$. One strategy is a change of variables. For example, to find $\int_0^1 \frac{\exp(x)}{\sqrt{x}} dx$, let $u = \sqrt{x}$, which gives the integral, $2\int_0^1 \exp(u^2) du$.

Another strategy is to subtract off the singularity. E.g., in the example above, reexpress as

$$\int_0^1 \frac{\exp(x) - 1}{\sqrt{x}} dx + \int_0^1 \frac{1}{\sqrt{x}} dx = \int_0^1 \frac{\exp(x) - 1}{\sqrt{x}} dx + 2$$

where we do the second integral analytically. It turns out that the first integral is well-behaved at 0.

It turns out that R's *integrate()* function can handle $\int_0^1 \frac{\exp(x)}{\sqrt{x}} dx$ directly without us changing the problem statement analytically. Perhaps this has something to do with the use of adaptive quadrature, but I'm not sure.

```
# doing it directly with integrate()
f <- function(x) exp(x)/sqrt(x)
integrate(f, 0, 1)

## 2.92530349 with absolute error < 9.4e-06

# subtracting off the singularity
f <- function(x) (exp(x) - 1)/sqrt(x)
x <- seq(0, 1, len = 200)
integrate(f, 0, 1)

## 0.925303567 with absolute error < 7.6e-05


# analytic change of variables, followed by numeric integration
f <- function(u) 2 * exp(u^2)
integrate(f, 0, 1)

## 2.92530349 with absolute error < 3.2e-14
```

Infinite ranges Gaussian quadrature deals with the case that $a = -\infty$ and/or $b = \infty$. Another possibility is change of variables using transformations such as $1/x$, $\exp(x)/(1 + \exp(x))$, $\exp(-x)$, and $x/(1 + x)$.

2.4 Symbolic integration

Mathematica and Maple are able to do symbolic integration for many problems that are very hard to do by hand (and with the same concerns as when doing differentiation by hand). So this may be worth a try.

```
# one-dimensional integration
Integrate[Sin[x]^2, x]
```

```
# two-dimensional integration
Integrate[Sin[x] Exp[-y^2], x, y]
```

Optimization

November 29, 2012

References:

- Gentle: *Computational Statistics*
- Lange: *Optimization*
- Monahan: *Numerical Methods of Statistics*
- Givens and Hoeting: *Computational Statistics*
- Materials online from Stanford's EE364a course on convex optimization, including Boyd and Vandenberghe's (online) book *Convex Optimization,* which is linked to from the course webpage.

1 Notation

We'll make use of the first derivative (the gradient) and second derivative (the Hessian) of functions. We'll generally denote univariate and multivariate functions (without distinguishing between them) as $f(x)$ with $x = (x_1, \ldots, x_p)$. The (column) vector of first partial derivatives (the gradient) is $f'(x) = \nabla f(x) = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_p})^\top$ and the matrix of second partial derivatives (the Hessian) is

$$f''(x) = \nabla_f^2(x) = H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_p} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_p} \\ \vdots & \vdots & \ddots & \\ \frac{\partial^2 f}{\partial x_1 \partial x_p} & \frac{\partial^2 f}{\partial x_2 \partial x_p} & \cdots & \frac{\partial^2 f}{\partial x_p^2} \end{pmatrix}.$$

In considering iterative algorithms, I'll use x_0, $x_1, \ldots, x_t$, x_{t+1} to indicate the sequence of values as we search for the optimum, denoted x^*. x_0 is the starting point, which we must choose (sometimes carefully). If it's unclear at any point whether I mean a value of x in the sequence or

a sub-element of the x vector, let me know, but hopefully it will be clear from context most of the time.

I'll try to use x (or if we're talking explicitly about a likelihood, θ) to indicate the argument with respect to which we're optimizing and Y to indicate data involved in a likelihood. I'll try to use z to indicate covariates/regressors so there's no confusion with x.

2 Overview

The basic goal here is to optimize a function numerically when we cannot find the maximum (or minimum) analytically. Some examples:

1. Finding the MLE for a GLM
2. Finding least squares estimates for a nonlinear regression model,

$$Y_i \sim \mathcal{N}(g(z_i; \beta), \sigma^2)$$

   where $g(\cdot)$ is nonlinear and we seek to find the value of $\theta = (\beta, \sigma^2)$ that best fits the data.

3. Maximizing a likelihood under constraints

Maximum likelihood estimation and variants thereof is a standard situation in which optimization comes up.

We'll focus on **minimization**, since any maximization of f can be treated as minimization of $-f$. The basic setup is to find

$$\arg\min_{x \in D} f(x)$$

where D is the domain. Sometimes $D = \Re^p$ but other times it imposes constraints on x. When there are no constraints, this is unconstrained optimization, where any x for which $f(x)$ is defined is a possible solution. We'll assume that f is continuous as there's little that can be done systematically if we're dealing with a discontinuous function.

In one dimension, minimization is the same as root-finding with the derivative function, since the minimum of a differentiable function can only occur at a point at which the derivative is zero. So with differentiable functions we'll seek to find x s.t. $f'(x) = \nabla f(x) = 0$. To ensure a minimum, we want that for all y in a neighborhood of x^*, $f(y) \leq f(x^*)$, or (for twice differentiable functions) $f''(x^*) = \nabla^2 f(x^*) = H_f(x^*) \geq 0$, i.e., that the Hessian is positive semi-definite.

Different strategies are used depending on whether D is discrete and countable, or continuous, dense and uncountable. We'll concentrate on the continuous case but the discrete case can arise in statistics, such as in doing variable selection.

In general we rely on the fact that we can evaluate f. Often we make use of analytic or numerical derivatives of f as well.

To some degree, optimization is a solved problem, with good software implementations, so it raises the question of how much to discuss in this class. The basic motivation for going into some of the basic classes of optimization strategies is that the function being optimized changes with each problem and can be tricky to optimize, and I want you to know something about how to choose a good approach when you find yourself with a problem requiring optimization. Finding global, as opposed to local, minima can also be an issue.

Note that I'm not going to cover MCMC (Markov chain Monte Carlo) methods, which are used for approximating integrals and sampling from posterior distributions in a Bayesian context and in a variety of ways for optimization. If you take a Bayesian course you'll cover this in detail, and if you don't do Bayesian work, you probably won't have much need for MCMC, though it comes up in MCEM (Monte Carlo EM) and simulated annealing, among other places.

Goals for the unit Optimization is a big topic. Here's what I would like you to get out of this:

1. an understanding of line searches,
2. an understanding of multivariate derivative-based optimization and how line searches are useful within this,
3. an understanding of derivative-free methods,
4. an understanding of the methods used in R's optimization routines, their strengths and weaknesses, and various tricks for doing better optimization in R, and
5. a basic idea of what convex optimization is and when you might want to go learn more about it.

3 Univariate function optimization

We'll start with some strategies for univariate functions. These can be useful later on in dealing with multivariate functions.

3.1 Golden section search

This strategy requires only that the function be unimodal.

Assume we have a single minimum, in $[a, b]$. We choose two points in the interval and evaluate them, $f(x_1)$ and $f(x_2)$. If $f(x_1) < f(x_2)$ then the minimum must be in $[a, x_2]$, and if the converse

in $[x_1, b]$. We proceed by choosing a new point in the new, smaller interval and iterate. At each step we reduce the length of the interval in which the minimum must lie. The primary question involves what is an efficient rule to use to choose the new point at each iteration.

Suppose we start with x_1 and x_2 s.t. they divide $[a, b]$ into three equal segments. Then we use $f(x_1)$ and $f(x_2)$ to rule out either the leftmost or rightmost segment based on whether $f(x_1) < f(x_2)$. If we have divided equally, we cannot place the next point very efficiently because either x_1 or x_2 equally divides the remaining space, so we are forced to divide the remaining space into relative lengths of 0.25, 0.25, and 0.5. The next time around, we may only rule out the shorter segment, which leads to inefficiency.

The efficient strategy is to maintain the *golden ratio* between the distances between the points using $\phi = (\sqrt{5} - 1)/2 \approx .618$, the golden ratio. We start with $x_1 = a + (1 - \phi)(b - a)$ and $x_2 = a + \phi(b - a)$. Then suppose $f(x_1) < f(x_2)$. We now choose to place x_3 s.t. it uses the golden ratio in the interval $[a, x_1]$: $x_3 = a + (1 - \phi)(x_2 - a)$. Because of the way we've set it up, we once again have the third subinterval, $[x_1, x_2]$, of equal length as the first subinterval, $[a, x_3]$. The careful choice allows us to narrow the search interval by an equal proportion,$1 - \phi$, in each iteration. Eventually we have narrowed the minimum to between x_{t-1} and x_t, where the difference $|x_t - x_{t-1}|$ is sufficiently small (within some tolerance - see Section 4 for details), and we report $(x_t + x_{t-1})/2$. We'll see an example of this on the board in class.

3.2 Bisection method

The bisection method requires the existence of the first derivative but has the advantage over the golden section search of halving the interval at each step. We again assume unimodality.

We start with an initial interval (a_0, b_0) and proceed to shrink the interval. Let's choose a_0 and b_0, and set x_0 to be the mean of these endpoints. Now we update according to the following algorithm, assuming our current interval is $[a_t, b_t]$:

$$[a_{t+1}, b_{t+1}] = \begin{cases} [a_t, x_t] & \text{if} f'(a_t)f'(x_t) < 0 \\ [x_t, b_t] & \text{if} f'(a_t)f'(x_t) > 0 \end{cases}$$

and set x_{t+1} to the mean of a_{t+1} and b_{t+1}. The basic idea is that if the derivative at both a_t and x_t is negative, then the minimum must be between x_t and b_t, based on the intermediate value theorem. If the derivatives at a_t and x_t are of different signs, then the minimum must be between a_t and x_t.

Since the bisection method reduces the size of the search space by one-half at each iteration, one can work out that each decimal place of precision requires 3-4 iterations. Obviously bisection is more efficient than the golden section search because we reduce by $0.5 > 0.382 = 1 - \phi$, so we've gained information by using the derivative. It requires an evaluation of the derivative

however, while golden section just requires an evaluation of the original function.

Bisection is an example of a *bracketing* method, in which we trap the minimum within a nested sequence of intervals of decreasing length. These tend to be slow, but if the first derivative is continuous, they are robust and don't require that a second derivative exist.

3.3 Newton-Raphson (Newton's method)

3.3.1 Overview

We'll talk about Newton-Raphson (N-R) as an optimization method rather than a root-finding method, but they're just different perspectives on the same algorithm.

For N-R, we need two continuous derivatives that we can evaluate. The benefit is speed, relative to bracketing methods. We again assume the function is unimodal. The minimum must occur at x^* s.t. $f'(x^*) = 0$, provided the second derivative is non-negative at x^*. So we aim to find a zero (a root) of the first derivative function. Assuming that we have an initial value x_0 that is close to x^*, we have the Taylor series approximation

$$f'(x) \approx f'(x_0) + (x - x_0)f''(x_0).$$

Now set $f'(x) = 0$, since that is the condition we desire (the condition that holds when we are at x^*), and solve for x to get

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)},$$

and iterate, giving us updates of the form $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$. What are we doing intuitively? Basically we are taking the tangent to $f(x)$ at x_0 and extrapolating along that line to where it crosses the x-axis to find x_1. We then reevaluate $f(x_1)$ and continue to travel along the tangents.

One can prove that if $f'(x)$ is twice continuously differentiable, is convex, and has a root, then N-R converges from any starting point.

Note that we can also interpret the N-R update as finding the analytic minimum of the quadratic Taylor series approximation to $f(x)$.

Newton's method converges very quickly (as we'll discuss in Section 4), but if you start too far from the minimum, you can run into serious problems.

3.3.2 Secant method variation on N-R

Suppose we don't want to calculate the second derivative required in the divisor of N-R. We might replace the analytic derivative with a discrete difference approximation based on the secant line

joining $(x_t, f'(x_t))$ and $(x_{t-1}, f'(x_{t-1}))$, giving an approximate second derivative:

$$f''(x_t) \approx \frac{f'(x_t) - f'(x_{t-1})}{x_t - x_{t-1}}.$$

For this variant on N-R, we need two starting points, x_0 and x_1.

An alternative to the secant-based approximation is to use a standard discrete approximation of the derivative such as

$$f''(x_t) \approx \frac{f'(x_t + h) - f'(x_t - h)}{2h}.$$

3.3.3 How can Newton's method go wrong?

Let's think about what can go wrong - namely when we could have $f(x_{t+1}) > f(x_t)$? Basically, if $f'(x_t)$ is relatively flat, we can get that $|x_{t+1} - x^*| > |x_t - x^*|$. We'll see an example on the board and the demo code . Newton's method can also go uphill when the second derivative is negative, with the method searching for a maximum.

```
options(width = 55)
par(mfrow = c(1, 2))
fp <- function(x, theta = 1) {
    exp(x * theta)/(1 + exp(x * theta)) - 0.5
}
fpp <- function(x, theta = 1) {
    exp(x * theta)/((1 + exp(x * theta))^2)
}
xs <- seq(-15, 15, len = 300)
plot(xs, fp(xs), type = "l")
lines(xs, fpp(xs), lty = 2)
# good starting point
x0 <- 2
xvals <- c(x0, rep(NA, 9))
for (t in 2:10) {
    xvals[t] = xvals[t - 1] - fp(xvals[t - 1])/fpp(xvals[t - 1])
}
print(xvals)
```

```
##  [1]  2.000e+00 -1.627e+00  8.188e-01 -9.461e-02
##  [5]  1.412e-04 -4.691e-13  6.142e-17  6.142e-17
##  [9]  6.142e-17  6.142e-17
```

```r
points(xvals, fp(xvals), pch = as.character(1:length(xvals)))
# bad starting point
x0 <- 2.5
xvals <- c(x0, rep(NA, 9))
for (t in 2:10) {
    xvals[t] = xvals[t - 1] - fp(xvals[t - 1])/fpp(xvals[t - 1])
}
print(xvals)
```

```
##  [1]       2.50      -3.55      13.85 -515287.63
##  [5]        Inf        NaN        NaN        NaN
##  [9]        NaN        NaN
```

```r
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = "red")
# whoops
#
# mistakenly climbing uphill
f <- function(x) cos(x)
fp <- function(x) -sin(x)
fpp <- function(x) -cos(x)
xs <- seq(0, 2 * pi, len = 300)
plot(xs, f(xs), type = "l", lwd = 2)
lines(xs, fp(xs))
lines(xs, fpp(xs), lty = 2)
x0 <- 0.2  # starting point
fp(x0)  # negative
```

```
## [1] -0.1987
```

```r
fpp(x0)  # negative
```

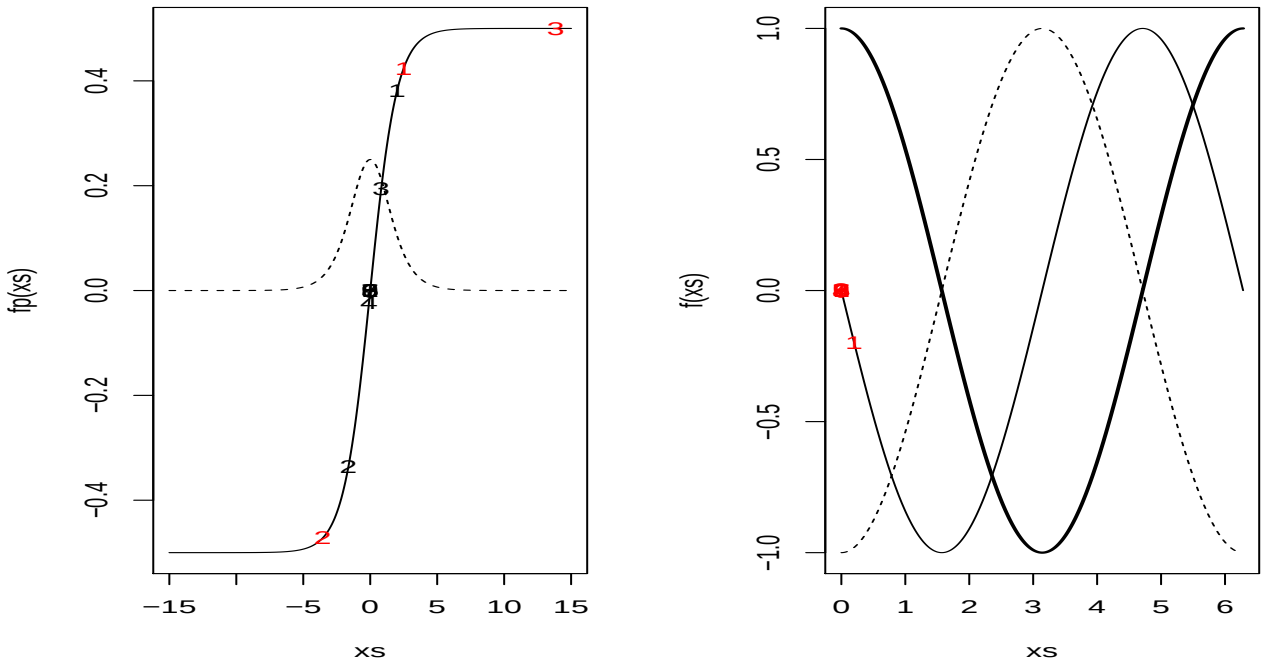
```
## [1] -0.9801
```

```r
x1 <- x0 - fp(x0)/fpp(x0)  # whoops, we've gone uphill
# because of the negative second derivative
xvals <- c(x0, rep(NA, 9))
for (t in 2:10) {
    xvals[t] = xvals[t - 1] - fp(xvals[t - 1])/fpp(xvals[t - 1])
```

```
}
xvals

##  [1]  2.000e-01 -2.710e-03  6.634e-09  0.000e+00
##  [5]  0.000e+00  0.000e+00  0.000e+00  0.000e+00
##  [9]  0.000e+00  0.000e+00

points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = "red")
```




```
# and we've found a maximum rather than a minimum...
```

One nice, general idea is to use a fast method such as Newton's method *safeguarded* by a robust, but slower method. Here's how one can do this for N-R, safeguarding with a bracketing method such as bisection. Basically, we check the N-R proposed move to see if N-R is proposing a step outside of where the root is known to lie based on the previous steps and the gradient values for those steps. If so, we could choose the next step based on bisection.

Another approach is backtracking. If a new value is proposed that yields a larger value of the function, backtrack to find a value that reduces the function. One possibility is a line search but given that we're trying to reduce computation, a full line search is unwise computationally (also in the multivariate Newton's method, we are in the middle of an iterative algorithm for which we will

just be going off in another direction anyway at the next iteration). A basic approach is to keep backtracking in halves. A nice alternative is to fit a polynomial to the known information about that slice of the function, namely $f(x_{t+1})$, $f(x_t)$, $f'(x_t)$ and $f''(x_t)$ and find the minimum of the polynomial approximation.

4 Convergence ideas

4.1 Convergence metrics

We might choose to assess whether $f'(x_t)$ is near zero, which should assure that we have reached the critical point. However, in parts of the domain where $f(x)$ is fairly flat, we may find the derivative is near zero even though we are far from the optimum. Instead, we generally monitor $|x_{t+1} - x_t|$ (for the moment, assume x is scalar). We might consider absolute convergence: $|x_{t+1} - x_t| < \epsilon$ or relative convergence, $\frac{|x_{t+1}-x_t|}{|x_t|} < \epsilon$. Relative convergence is appealing because it accounts for the scale of x, but it can run into problems when x_t is near zero, in which case one can use $\frac{|x_{t+1}-x_t|}{|x_t|+\epsilon} < \epsilon$. We would want to account for machine precision in thinking about setting ϵ. For relative convergence a reasonable choice of ϵ would be to use the square root of machine epsilon or about 1×10^{-8}. This is the *reltol* argument in *optim()* in R.

Problems with the optimization may show up in a convergence measure that fails to decrease or cycles (oscillates). Software generally has a stopping rule that stops the algorithm after a fixed number of iterations; these can generally be changed by the user. When an algorithm stops because of the stopping rule before the convergence criterion is met, we say the algorithm has failed to converge. Sometimes we just need to run it longer, but often it indicates a problem with the function being optimized or with your starting value.

For multivariate optimization, we use a distance metric between x_{t+1} and x_t, such as $\|x_{t+1} - x_t\|_p$, often with $p = 1$ or $p = 2$.

4.2 Starting values

Good starting values are important because they can improve the speed of optimization, prevent divergence or cycling, and prevent finding local optima.

Using random or selected multiple starting values can help with multiple optima.

4.3 Convergence rates

Let $\epsilon_t = |x_t - x^*|$. If the limit

$$\lim_{t\to\infty} \frac{|\epsilon_{t+1}|}{|\epsilon_t|^\beta} = c$$

exists for $\beta > 0$ and $c \neq 0$, then a method is said to have order of convergence β. This basically measures how big the error at the $t+1$th iteration is relative to that at the tth iteration.

Bisection doesn't formally satisfy the criterion needed to make use of this definition, but roughly speaking it has linear convergence ($\beta = 1$). Next we'll see that N-R has quadratic convergence ($\beta = 2$), which is fast.

To analyze convergence of N-R, consider a Taylor expansion of the gradient at the minimum, x^*, around the current value, x_t:

$$f'(x^*) = f'(x_t) + (x^* - x_t)f''(x_t) + \frac{1}{2}(x^* - x_t)^2 f'''(\xi) = 0,$$

for some $\xi_t \in [x_t, x^*]$. Making use of the N-R update equation: $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$, and some algebra, we have

$$\frac{|x^* - x_{t+1}|}{(x^* - x_t)^2} = \frac{1}{2}\frac{f'''(\xi_t)}{f''(x_t)}.$$

If the limit of the ratio on the right hand side exists and is equal to c, then we see that $\beta = 2$. If c were one, then we see that if we have k digits of accuracy at t, we'd have $2k$ digits at $t+1$, which justifies the characterization of quadratic convergence being fast. In practice c will moderate the rate of convergence. The smaller c the better, so we'd like to have the second derivative be large and the third derivative be small. The expression also indicates we'll have a problem if $f''(x_t) = 0$ at any point [think about what this corresponds to graphically - what is our next step when $f''(x_t) = 0$?]. The characteristics of the derivatives determine the domain of attraction (the region in which we'll converge rather than diverge) of the minimum.

Givens and Hoeting show that using the secant-based approximation to the second derivative in N-R has order of convergence, $\beta \approx 1.62$.

Here's an example of convergence comparing bisection and N-R:

```
options(digits = 10)
f <- function(x) cos(x)
fp <- function(x) -sin(x)
fpp <- function(x) -cos(x)
xstar <- pi  # known minimum
# N-R
x0 <- 2
xvals <- c(x0, rep(NA, 9))
```

```
for (t in 2:10) {
    xvals[t] <- xvals[t - 1] - fp(xvals[t - 1])/fpp(xvals[t -
        1])
}
print(xvals)
```

```
##  [1] 2.000000000 4.185039863 2.467893675 3.266186278
##  [5] 3.140943912 3.141592654 3.141592654 3.141592654
##  [9] 3.141592654 3.141592654
```

```
# bisection
bisecStep <- function(interval, fp) {
    xt <- mean(interval)
    if (fp(interval[1]) * fp(xt) <= 0)
        interval[2] <- xt else interval[1] <- xt
    return(interval)
}
nIt <- 30
a0 <- 2
b0 <- (3 * pi/2) - (xstar - a0)
# have b0 be as far from min as a0 for fair comparison
# with N-R
interval <- matrix(NA, nr = nIt, nc = 2)
interval[1, ] <- c(a0, b0)
for (t in 2:nIt) {
    interval[t, ] <- bisecStep(interval[t - 1, ], fp)
}
rowMeans(interval)
```

```
##  [1] 2.785398163 3.178097245 2.981747704 3.079922475
##  [5] 3.129009860 3.153553552 3.141281706 3.147417629
##  [9] 3.144349668 3.142815687 3.142048697 3.141665201
## [13] 3.141473454 3.141569328 3.141617264 3.141593296
## [17] 3.141581312 3.141587304 3.141590300 3.141591798
## [21] 3.141592547 3.141592922 3.141592734 3.141592641
## [25] 3.141592687 3.141592664 3.141592652 3.141592658
## [29] 3.141592655 3.141592654
```

5 Multivariate optimization

Optimizing as the dimension of the space gets larger becomes increasingly difficult. First we'll discuss the idea of profiling to reduce dimensionality and then we'll talk about various numerical techniques, many of which build off of Newton's method.

5.1 Profiling

A core technique for likelihood optimization is to analytically maximize over any parameters for which this is possible. Suppose we have two sets of parameters, θ_1 and θ_2, and we can analytically maximize w.r.t θ_2. This will give us $\hat{\theta}_2(\theta_1)$, a function of the remaining parameters over which analytic maximization is not possible. Plugging in $\hat{\theta}_2(\theta_1)$ into the objective function (in this case generally the likelihood or log likelihood) gives us the profile (log) likelihood solely in terms of the obstinant parameters. For example, suppose we have the regression likelihood with correlated errors:

$$Y \sim \mathcal{N}(X\beta, \sigma^2\Sigma(\rho)),$$

where $\Sigma(\rho)$ is a correlation matrix that is a function of a parameter, ρ. The maximum w.r.t. β is easily seen to be the GLS estimator $\hat{\beta}(\rho) = (X^\top\Sigma(\rho)^{-1}X)^{-1}X^\top\Sigma(\rho)^{-1}Y$. In general such a maximum is a function of all of the other parameters, but conveniently it's only a function of ρ here. This gives us the initial profile likelihood

$$\frac{1}{(\sigma^2)^{n/2}|\Sigma(\rho)|^{1/2}} \exp\left(-\frac{(Y - X\hat{\beta}(\rho))^{-\top}\Sigma(\rho)^{-1}(Y - X\hat{\beta}(\rho))}{2\sigma^2}\right).$$

We then notice that the likelihood is maximized w.r.t. σ^2 at

$$\hat{\sigma^2}(\rho) = \frac{(Y - X\hat{\beta}(\rho))^\top\Sigma(\rho)^{-1}(Y - X\hat{\beta}(\rho))}{n}.$$

This gives us the final profile likelihood,

$$\frac{1}{|\Sigma(\rho)|^{1/2}} \frac{1}{(\hat{\sigma^2}(\rho))^{n/2}} \exp(-\frac{1}{2}n),$$

a function of ρ only, for which numerical optimization is much simpler.

5.2 Newton-Raphson (Newton's method)

For multivariate x we have the Newton-Raphson update $x_{t+1} = x_t - f''(x_t)^{-1} f'(x_t)$, or in our other notation,

$$x_{t+1} = x_t - H_f(x_t)^{-1} \nabla f(x_t).$$

In class we'll use the demo code for an example of finding the nonlinear least squares fit to some weight loss data to fit the model

$$Y_i = \beta_0 + \beta_1 2^{-t_i/\beta_2} + \epsilon_i.$$

Some of the things we need to worry about with Newton's method in general about are (1) good starting values, (2) positive definiteness of the Hessian, and (3) avoiding errors in finding the derivatives.

A note on the positive definiteness: since the Hessian may not be positive definite (although it may well be, provided the function is approximately locally quadratic), one can consider modifying the Cholesky decomposition of the Hessian to enforce positive definiteness by adding diagonal elements to H_f as necessary.

5.3 Fisher scoring variant on N-R

The Fisher information (FI) is the expected value of the outer product of the gradient of the log-likelihood with itself

$$I(\theta) = E_f(\nabla f(y) \nabla f(y)^\top),$$

where the expected value is with respect to the data distribution. Under regularity conditions (true for exponential families), the expectation of the Hessian of the log-likelihood is minus the Fisher information, $E_f H_f(y) = -I(\theta)$. We get the observed Fisher information by plugging the data values into either expression instead of taking the expected value.

Thus, standard N-R can be thought of as using the observed Fisher information to find the updates. Instead, if we can compute the expectation, we can use minus the FI in place of the Hessian. The result is the Fisher scoring (FS) algorithm. Basically instead of using the Hessian for a given set of data, we are using the FI, which we can think of as the average Hessian over repeated samples of data from the data distribution. FS and N-R have the same convergence properties (i.e., quadratic convergence) but in a given problem, one may be computationally or analytically easier. Givens and Hoeting comment that FS works better for rapid improvements at the beginning of iterations and N-R better for refinement at the end.

In the demo code, we try out Fisher scoring in the weight loss example.

The Gauss-Newton algorithm for nonlinear least squares involves using the FI in place of the Hessian in determining a Newton-like step. *nls()* in R uses this approach. Note that this is not exactly the same updating as our manual coding of FS for the weight loss example.

Connections between statistical uncertainty and ill-conditionedness When either the observed or expected FI matrix is nearly singular this means we have a small eigenvalue in the inverse covariance (the precision), which means a large eigenvalue in the covariance matrix. This indicates some linear combination of the parameters has low precision (high variance), and that in that direction the likelihood is nearly flat. As we've seen with N-R, convergence slows with shallow gradients, and we may have numerical problems in determining good optimization steps when the likelihood is sufficiently flat. So convergence problems and statistical uncertainty go hand in hand. One, but not the only, example of this occurs when we have nearly collinear regressors.

5.4 IRLS (IWLS) for GLMs

As most of you know, iterative reweighted least squares (also called iterative weighted least squares) is the standard method for estimation with GLMs. It involves linearizing the model and using working weights and working variances and solving a weighted least squares (WLS) problem (the generic WLS solution is $\hat{\beta} = (X^\top WX)^{-1}X^\top WY$).

Exponential families can be expressed as

$$f(y;\theta,\phi) = \exp((y\theta - b(\theta))/a(\phi) + c(y,\phi)),$$

with $E(Y) = b'(\theta)$ and $\text{Var}(Y) = b''(\theta)$. If we have a GLM in the canonical parameterization (log link for Poisson data, logit for binomial), we have the natural parameter θ equal to the linear predictor, $\theta = \eta$. A standard linear predictor would simply be $\eta = X\beta$.

Considering N-R for a GLM in the canonical parameterization (and ignoring $a(\phi)$, which is one for logistic and Poisson regression), we find that the gradient is the inner product of the covariates and a residual vector, $\nabla f(\beta) = (Y - E(Y))^\top X$, and the Hessian is $\nabla^2 f(\beta) = -X^\top WX$ where W is a diagonal matrix with $\{\text{Var}(Y_i)\}$ on the diagonal (the working weights). Note that both $E(Y)$ and the variances in W depend on β, so these will change as we iteratively update β. Therefore, the N-R update is

$$\beta_{t+1} = \beta_t + (X^\top W_t X)^{-1}X^\top(Y - E(Y)_t)$$

where $E(Y)_t$ and W_t are the values at the current parameter estimate, β_t . For example, for logistic regression, $W_{t,ii} = p_{ti}(1-p_{ti})$ and $E(Y)_{ti} = p_{ti}$ where $p_{ti} = \frac{\exp(X_i^\top \beta_t)}{1+\exp(X_i^\top \beta_t)}$. In the canonical parameterization of a GLM, the Hessian does not depend on the data, so the observed and expected

FI are the same, and therefore N-R and FS are the same.

The update above can be rewritten in the standard form of IRLS as a WLS problem,

$$\beta_{t+1} = (X^\top W_t X)^{-1} X^\top W_t \tilde{Y}_t,$$

where the so-called working observations are $\tilde{Y}_t = X\beta_t + W_t^{-1}(Y - E(Y)_t)$. Note that these are on the scale of the linear predictor.

While Fisher scoring is standard for GLMs, you can also use general purpose optimization routines.

IRLS is a special case of the general Gauss-Newton method for nonlinear least squares.

5.5 Descent methods and Newton-like methods

More generally a Newton-like method has updates of the form

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t).$$

We can choose M_t in various ways, including as an approximation to the second derivative.

This opens up several possibilities:

1. using more computationally efficient approximations to the second derivative,
2. avoiding steps that do not go in the correct direction (i.e., go uphill when minimizing), and
3. scaling by α_t so as not to step too far.

Let's consider a variety of strategies.

5.5.1 Descent methods

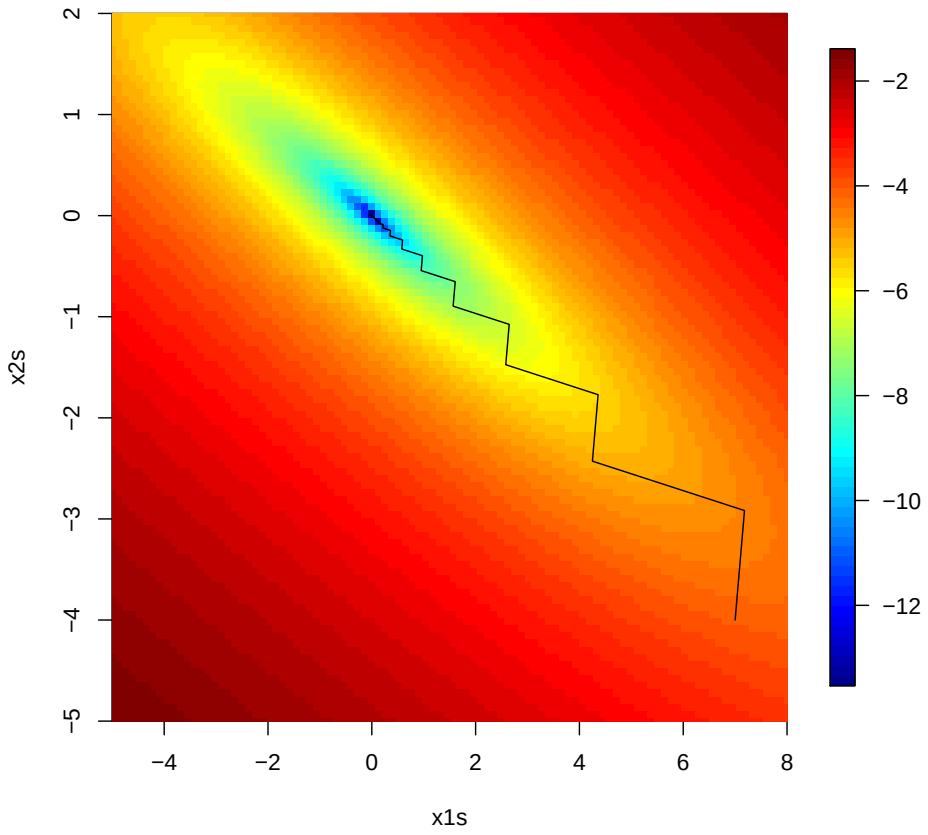
The basic strategy is to choose a good direction and then choose the longest step for which the function continues to decrease. Suppose we have a direction, p_t. Then we need to move $x_{t+1} = x_t + \alpha_t p_t$, where α_t is a scalar, choosing a good α_t. We might use a line search (e.g., bisection or golden section search) to find the local minimum of $f(x_t + \alpha_t p_t)$ with respect to α_t. However, we often would not want to run to convergence, since we'll be taking additional steps anyway.

Steepest descent chooses the direction as the steepest direction downhill, setting $M_t = I$, since the gradient gives the steepest direction uphill (the negative sign in the equation below has us move directly downhill rather than directly uphill). A better approach is to scale the step

$$x_{t+1} = x_t - \alpha_t f'(x_t)$$

where the contraction, or step length, parameter α_t is chosen sufficiently small to ensure that we descend, via some sort of line search. The critical downside to steepest ascent is that when the contours are elliptical, it tends to zigzag; here's an example. Note that I do a full line search (using the golden section method via *optimize()*) at each step in the direction of steepest descent - this is generally computationally wasteful, but I just want to illustrate how steepest descent can go wrong, even if you go the "right" amount in each direction.

```
f <- function(x) {
    x[1]^2/1000 + 4 * x[1] * x[2]/1000 + 5 * x[2]^2/1000
}
fp <- function(x) {
    c(2 * x[1]/1000 + 4 * x[2]/1000, 4 * x[1]/1000 + 10 *
        x[2]/1000)
}
lineSearch <- function(alpha, xCurrent, direction, FUN) {
    newx <- xCurrent + alpha * direction
    FUN(newx)
}
nIt <- 50
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
for (t in 2:50) {
    newalpha <- optimize(lineSearch, interval = c(-5000,
        5000), xCurrent = xvals[t - 1, ], direction = fp(xvals[t -
        1, ]), FUN = f)$minimum
    xvals[t, ] <- xvals[t - 1, ] + newalpha * fp(xvals[t -
        1, ])
}
x1s <- seq(-5, 8, len = 100)
x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
# plot f(x) surface on log scale
image.plot(x1s, x2s, matrix(log(fx), 100, 100), xlim = c(-5,
    8), ylim = c(-5, 2))
lines(xvals)  # overlay optimization path
```




```
# kind of slow
```

If the contours are circular, steepest descent works well. Newton's method deforms elliptical contours based on the Hessian. Another way to think about this is that steepest descent does not take account of the rate of change in the gradient, while Newton's method does.

The general descent algorithm is

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t),$$

where M_t is generally chose to approximate the Hessian and α_t allows us to adjust the step in a smart way. Basically, since the negative gradient tells us the direction that descends (at least within a small neighborhood), if we don't go too far, we should be fine and should work our way downhill. One can work this out formally using a Taylor approximation to $f(x_{t+1}) - f(x_t)$ and see that we make use of M_t being positive definite. (Unfortunately backtracking with positive definite M_t does not give a theoretical guarantee that the method will converge. We also need to make sure that the steps descend sufficiently quickly and that the algorithm does not step along a level contour of f.)

The conjugate gradient algorithm for iteratively solving large systems of equations is all about choosing the direction and the step size in a smart way given the optimization problem at hand.

5.5.2 Quasi-Newton methods such as BFGS

Other replacements for the Hessian matrix include estimates that do not vary with t and finite difference approximations. When calculating the Hessian is expensive, it can be very helpful to substitute an approximation.

A basic finite difference approximation requires us to compute finite differences in each dimension, but this could be computationally burdensome. A more efficient strategy for choosing M_{t+1} is to (1) make use of M_t and (2) make use of the most recent step to learn about the curvature of $f'(x)$ in the direction of travel. One approach is to use a rank one update to M_t.

A basic strategy is to choose M_{t+1} such that the secant condition is satisfied:

$$M_{t+1}(x_{t+1} - x_t) = \nabla f(x_{t+1}) - \nabla f(x_t),$$

which is motivated by the fact that the secant approximates the gradient in the direction of travel. Basically this says to modify M_t in such a way that we incorporate what we've learned about the gradient from the most recent step. M_{t+1} is not fully determined based on this, and we generally impose other conditions, in particular that M_{t+1} is symmetric and positive definite. Defining $s_t = x_{t+1} - x_t$ and $y_t = \nabla f(x_{t+1}) - \nabla f(x_t)$, the unique, symmetric rank one update (why is the following a rank one update?) that satisfies the secant condition is

$$M_{t+1} = M_t + \frac{(y_t - M_t s_t)(y_t - M_t s_t)^\top}{(y_t - M_t s_t)^\top s_t}.$$

If the denominator is positive, M_{t+1} may not be positive definite, but this is guaranteed for non-positive values of the denominator. One can also show that one can achieve positive definiteness by shrinking the denominator toward zero sufficiently.

A standard approach to updating M_t is a commonly-used rank two update that generally results in M_{t+1} being positive definite is

$$M_{t+1} = M_t - \frac{M_t s_t (M_t s_t)^\top}{s_t^\top M_t s_t} + \frac{y_t y_t^\top}{s_t^\top y_t},$$

which is known as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. This is one of the methods used in R in *optim()*.

Question: how can we update M_t^{-1} to M_{t+1}^{-1} efficiently? It turns out there is a way to update the Cholesky of M_t efficiently and this is a better approach than updating the inverse.

The order of convergence of quasi-Newton methods is generally slower than the quadratic convergence of N-R because of the approximations but still faster than linear. In general, quasi-Newton methods will do much better if the scales of the elements of x are similar. Lange suggests using a starting point for which one can compute the expected information, to provide a good starting value M_0.

Note that for estimating a covariance based on the numerical information matrix, we would not want to rely on M_t from the final iteration, as the approximation may be poor. Rather we would spend the effort to better estimate the Hessian directly at x^*.

5.6 Gauss-Seidel

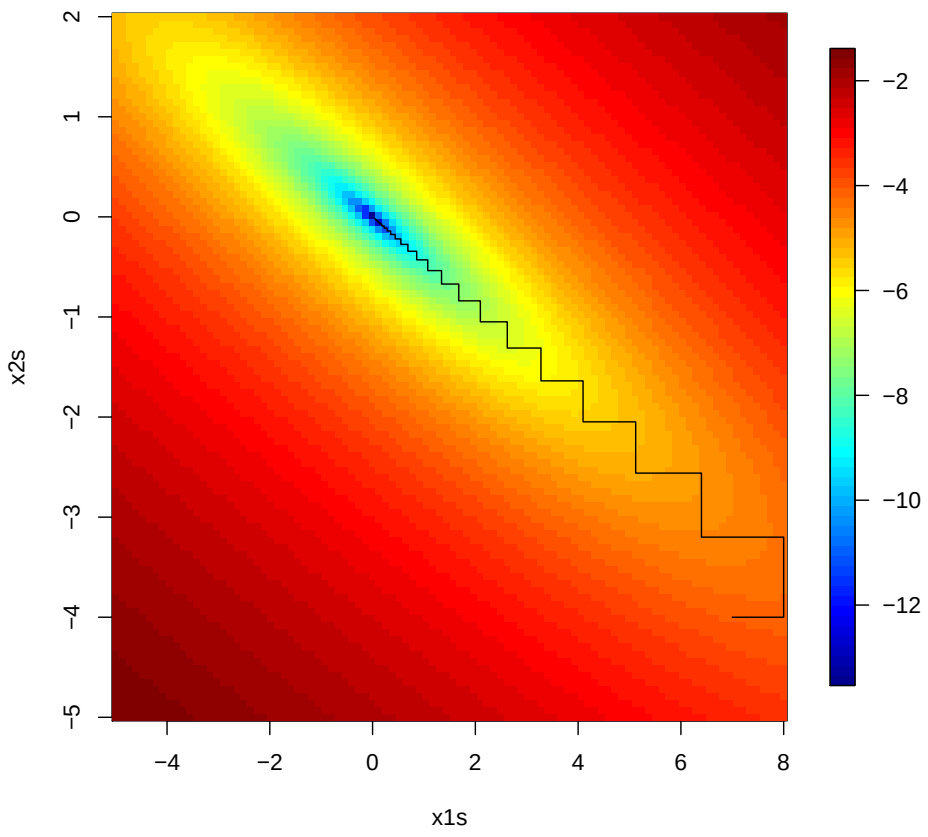
Gauss-Seidel is also known a back-fitting or cyclic coordinate descent. The basic idea is to work element by element rather than having to choose a direction for each step. For example backfitting used to be used to fit generalized additive models of the form $E(Y) = f_1(z_1)+f_2(z_2)+\ldots+f_p(z_p)$.

The basic strategy is to consider the jth component of $f'(x)$ as a univariate function of x_j only and find the root, $x_{j,t+1}$ that gives $f_j'(x_{j,t+1}) = 0$. One cycles through each element of x to complete a single cycle and then iterates. The appeal is that univariate root-finding/minimization is easy, often more stable than multivariate, and quick.

However, Gauss-Seidel can zigzag, since you only take steps in one dimension at a time, as we see here.

```
f <- function(x) {
    return(x[1]^2/1000 + 4 * x[1] * x[2]/1000 + 5 * x[2]^2/1000)
}
f1 <- function(x1, x2) {
    # f(x) as a function of x1
    return(x1^2/1000 + 4 * x1 * x2/1000 + 5 * x2^2/1000)
}
f2 <- function(x2, x1) {
    # f(x) as a function of x2
    return(x1^2/1000 + 4 * x1 * x2/1000 + 5 * x2^2/1000)
}
x1s <- seq(-5, 8, len = 100)
x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
image.plot(x1s, x2s, matrix(log(fx), 100, 100))
nIt <- 49
```

```
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
# 5, -10
for (t in seq(2, nIt, by = 2)) {
    newx1 <- optimize(f1, x2 = xvals[t - 1, 2], interval = c(-40,
        40))$minimum
    xvals[t, ] <- c(newx1, xvals[t - 1, 2])
    newx2 <- optimize(f2, x1 = newx1, interval = c(-40,
        40))$minimum
    xvals[t + 1, ] <- c(newx1, newx2)
}
lines(xvals)
```



In the notes for Unit 9 on linear algebra, I discussed the use of Gauss-Seidel to iteratively solve $Ax = b$ in situations where factorizing A (which of course is $O(n^3)$) is too computationally expensive.

The lasso The lasso uses an L1 penalty in regression and related contexts. A standard formulation for the lasso in regression is to minimize

$$\|Y - X\beta\|_2^2 + \lambda \sum_j |\beta_j|$$

to find $\hat{\beta}(\lambda)$ for a given value of the penalty parameter, λ. A standard strategy to solve this problem is to use coordinate descent, either cyclically, or by using directional derivatives to choose the coordinate likely to decrease the objective function the most (a greedy strategy). We need to use directional derivatives because the penalty function is not differentiable, but does have directional derivatives in each direction. The directional derivative of the objective function for β_j is

$$-2x_j \sum_i (Y_i - X\beta) \pm \lambda$$

where we add λ if $\beta_j \geq 0$ and you subtract λ if $\beta_j < 0$. If $\beta_{j,t}$ is 0, then a step in either direction contributes $+\lambda$ to the derivative as the contribution of the penalty.

Once we have chosen a coordinate, we set the directional derivative to zero and solve for β_j to obtain $\beta_{j,t+1}$.

The LARS (least angle regression) algorithm uses a similar strategy that allows one to compute $\hat{\beta}_\lambda$ for all values of λ at once.

The lasso can also be formulated as the constrained minimization of $\|Y - X\beta\|_2^2$ s.t. $\sum_j |\beta_j| \leq c$, with c now playing the role of the penalty parameter. Solving this minimization problem would take us in the direction of quadratic programming, a special case of convex programming, discussed in Section 9.

5.7 Nelder-Mead

This approach avoids using derivatives or approximations to derivatives. This makes it robust, but also slower than Newton-like methods. The basic strategy is to use a simplex, a polytope of $p + 1$ points in p dimensions (e.g., a triangle when searching in two dimensions, tetrahedron in three dimensions...) to explore the space, choosing to shift, expand, or contract the polytope based on the evaluation of f at the points.

The algorithm relies on four tuning factors: a reflection factor, $\alpha > 0$; an expansion factor, $\gamma > 1$; a contraction factor, $0 < \beta < 1$; and a shrinkage factor, $0 < \delta < 1$. First one chooses an initial simplex: $p + 1$ points that serve as the vertices of a convex hull.

1. Evaluate and order the points, $x_1, \ldots, x_{p+1}$ based on $f(x_1) \leq \ldots \leq f(x_{p+1})$. Let $\bar{x}$ be the average of the first p x's.

2. (Reflection) Reflect x_{p+1} across the hyperplane (a line when $p+1=3$) formed by the other points to get x_r, based on α.

   - $x_r = (1+\alpha)\bar{x} - \alpha x_{p+1}$

3. If $f(x_r)$ is between the best and worst of the other points, the iteration is done, with x_r replacing x_{p+1}. We've found a good direction to move.

4. (Expansion) If $f(x_r)$ is better than all of the other points, expand by extending x_r to x_e based on γ, because this indicates the optimum may be further in the direction of reflection. If $f(x_e)$ is better than $f(x_r)$, use x_e in place of x_{p+1}. If not, use x_r. The iteration is done.

   - $x_e = \gamma x_r + (1-\gamma)\bar{x}$

5. If $f(x_r)$ is worse than all the other points, but better than $f(x_{p+1})$, let $x_h = x_r$. Otherwise $f(x_r)$ is worse than $f(x_{p+1})$ so let $x_h = x_{p+1}$. In either case, we want to concentrate our polytope toward the other points.

   (a) (Contraction) Contract x_h toward the hyperplane formed by the other points, based on β, to get x_c. If the result improves upon $f(x_h)$ replace x_{p+1} with x_c. Basically, we haven't found a new point that is better than the other points, so we want to contract the simplex away from the bad point.

   - $x_c = \beta x_h + (1-\beta)\bar{x}$

   (b) (Shrinkage) Otherwise (if x_c is not better than x_{p+1}) shrink the simplex toward x_1. Basically this suggests our step sizes are too large and we should shrink the simplex, shrinking towards the best point.

   - $x_i = \delta x_i + (1-\delta)x_1$for $i = 2, \ldots, p+1$

Convergence is assessed based on the sample variance of the function values at the points, the total of the norms of the differences between the points in the new and old simplexes, or the size of the simplex.

This is the default in *optim()* in R, however it is relatively slow, so you may want to try one of the alternatives, such as BFGS.

5.8 Simulated annealing (SA)

Simulated annealing is a *stochastic* descent algorithm, unlike the deterministic algorithms we've already discussed. It has a couple critical features that set it aside from other approaches. First,

uphill moves are allowed; second, whether a move is accepted is stochastic, and finally, as the iterations proceed the algorithm becomes less likely to accept uphill moves.

Assume we are minimizing a negative log likelihood as a function of θ, $f(\theta)$.

The basic idea of simulated annealing is that one modifies the objective function, f in this case, to make it less peaked at the beginning, using a "temperature" variable that changes over time. This helps to allow moves away from local minima, when combined with the ability to move uphill. The name comes from an analogy to heating up a solid to its melting temperature and cooling it slowly - as it cools the atoms go through rearrangements and slowly freeze into the crystal configuration that is at the lowest energy level.

Here's the algorithm. We divide up iterations into stages, $j = 1, 2, \ldots$ in which the temperature variable, τ_j, is constant. Like MCMC, we require a proposal distribution to propose new values of θ.

1. Propose to move from θ_t to $\tilde{\theta}$ from a proposal density, $g_t(\cdot|\theta_t)$, such as a normal distribution centered at θ_t.

2. Accept $\tilde{\theta}$ as θ_{t+1} according to the probability $\min(1, \exp((f(\theta_t) - f(\tilde{\theta}))/\tau_j)$ - i.e., accept if a uniform random deviate is less than that probability. Otherwise set $\theta_{t+1} = \theta_t$. Notice that for larger values of τ_j the differences between the function values at the two locations are reduced (just like a large standard deviation spreads out a distribution). So the exponentiation smooths out the objective function when τ_j is large.

3. Repeat steps 1 and 2 m_j times.

4. Increment the temperature and cooling schedule: $\tau_j = \alpha(\tau_{j-1})$ and $m_j = \beta(m_{j-1})$. Back to step 1.

The temperature should slowly decrease to 0 while the number of iterations, m_j, should be large. Choosing these 'schedules' is at the core of implementing SA. Note that we always accept downhill moves in step 2 but we sometimes accept uphill moves as well.

For each temperature, SA produces an MCMC based on the Metropolis algorithm. So if m_j is long enough, we should sample from the stationary distribution of the Markov chain, $\exp(-f(\theta)/\tau_j))$. Provided we can move between local minima, the chain should gravitate toward the global minima because these are increasingly deep (low values) relative to the local minima as the temperature drops. Then as the temperature cools, θ_t should get trapped in an increasingly deep well centered on the global minimum. There is a danger that we will get trapped in a local minimum and not be able to get out as the temperature drops, so the temperature schedule is quite important in trying to avoid this.

A wide variety of schedules have been tried. One approach is to set $m_j = 1 \forall j$ and $\alpha(\tau_{j-1}) = \frac{\tau_{j-1}}{1+a\tau_{j-1}}$ for a small a. For a given problem it can take a lot of experimentation to choose τ_0 and m_0 and the values for the scheduling functions. For the initial temperature, it's a good idea to choose it large enough that $\exp((f(\theta_i) - f(\theta_j))/\tau_0) \approx 1$ for any pair $\{\theta_i, \theta_j\}$ in the domain, so that the algorithm can visit the entire space initially.

Simulated annealing can converge slowly. Multiple random starting points or stratified starting points can be helpful for finding a global minimum. However, given the slow convergence, these can also be computationally burdensome.

6 Basic optimization in R

6.1 Core optimization functions

R has several optimization functions.

- *optimize()* is good for 1-d optimization: "The method used is a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions."

- Another option is *uniroot()* for finding the zero of a function, which you can use to minimize a function if you can compute the derivative.

- For more than one variable, *optim()* uses a variety of optimization methods including the robust Nelder-Mead method, the BFGS quasi-Newton method and simulated annealing. You can choose which method you prefer and can try multiple methods. You can supply a gradient function to *optim()* for use with the Newton-related methods but it can also calculate numerical derivatives on the fly. You can have *optim()* return the Hessian at the optimum (based on a numerical estimate), which then allows straighforward calculation of asymptotic variances based on the information matrix.

- Also for multivariate optimization, *nlm()* uses a Newton-style method, for which you can supply analytic gradient and Hessian, or it will estimate these numerically. *nlm()* can also return the Hessian at the optimum.

- The *optimx* package provides *optimx()*, which is a wrapper for a variety of optimization methods (including many of those in *optim()*, as well as *nlm()*. One nice feature is that it allow you to use multiple methods in the same function call.

In the demo code, we'll work our way through a real example of optimizing a likelihood for some climate data on extreme precipitation.

6.2 Various considerations in using the R functions

As we've seen, initial values are important both for avoiding divergence (e.g., in N-R), for increasing speed of convergence, and for helping to avoid local optima. So it is well worth the time to try to figure out a good starting value or multiple starting values for a given problem.

Scaling can be important. One useful step is to make sure the problem is well-scaled, namely that a unit step in any parameter has a comparable change in the objective function, preferably approximately a unit change at the optimum. *optim()* allows you to supply scaling information through the *parscale* component of the *control* argument. Basically if x_j is varying at p orders of magnitude smaller than the other xs, we want to reparameterize to $x_j^* = x_j \cdot 10^p$ and then convert back to the original scale after finding the answer. Or we may want to work on the log scale for some variables, reparameterizing as $x_j^* = \log(x_j)$. We could make such changes manually in our expression for the objective function or make use of arguments such as *parscale*.

If the function itself gives very large or small values near the solution, you may want to rescale the entire function to avoid calculations with very large or small numbers. This can avoid problems such as having apparent convergence because a gradient is near zero, simply because the scale of the function is small. In *optim()* this can be controlled with the *fnscale* component of *control*.

Always consider your answer and make sure it makes sense, in particular that you haven't 'converged' to an extreme value on the boundary of the space.

Venables and Ripley suggest that it is often worth supplying analytic first derivatives rather than having a routine calculate numerical derivatives but not worth supplying analytic second derivatives. As we've seen R can do symbolic (i.e., analytic) differentiation to find first and second derivatives using *deriv()*.

In general for software development it's obviously worth putting more time into figuring out the best optimization approach and supplying derivatives. For a one-off analysis, you can try a few different approaches and assess sensitivity.

The nice thing about likelihood optimization is that the asymptotic theory tells us that with large samples, the likelihood is approximately quadratic (i.e., the asymptotic normality of MLEs), which makes for a nice surface over which to do optimization. When optimizing with respect to variance components and other parameters that are non-negative, one approach to dealing with the constraints is to optimize with respect to the log of the parameter.

7 Combinatorial optimization over discrete spaces

Many statistical optimization problems involve continuous domains, but sometimes there are problems in which the domain is discrete. Variable selection is an example of this.

Simulated annealing can be used for optimizing in a discrete space. Another approach uses *genetic algorithms*, in which one sets up the dimensions as loci grouped on a chromosome and has mutation and crossover steps in which two potential solutions reproduce. An example would be in high-dimensional variable selection.

Stochastic search variable selection is a popular Bayesian technique for variable selection that involves MCMC.

8 Convexity

Many optimization problems involve (or can be transformed into) convex functions. Convex optimization (also called convex programming) is a big topic and one that we'll only brush the surface of in Sections 8 and 9. The goal here is to give you enough of a sense of the topic that you know when you're working on a problem that might involve convex optimization, in which case you'll need to go learn more.

Optimization for convex functions is simpler than for ordinary functions because we don't have to worry about local optima - any stationary point (point where the gradient is zero) is a global minimum. A set S in $\Re^p$ is convex if any line segment between two points in S lies entirely within S. More generally, S is convex if any convex combination is itself in S, i.e., $\sum_{i=1}^m \alpha_i x_i \in S$ for non-negative weights, α_i, that sum to 1. Convex functions are defined on convex sets - f is convex if for points in a convex set, $x_i \in S$, we have $f(\sum_{i=1}^m \alpha_i x_i) \leq \sum_{i=1}^m \alpha_i f(x_i)$. Strict convexity is when the inequality is strict (no equality).

The first-order convexity condition relates a convex function to its first derivative: f is convex if and only if $f(x) \geq f(y) + \nabla f(y)^\top (x - y)$ for y and x in the domain of f. We can interpret this as saying that the first order Taylor approximation to f is tangent to and below (or touching) the function at all points.

The second-order convexity condition is that a function is convex if (provided its first derivative exists), the derivative is non-decreasing, in which case we have $f''(x) \geq 0 \ \forall x$ (for univariate functions). If we have $f''(x) \leq 0 \ \forall x$ (a concave, or convex down function) we can always consider $-f(x)$, which is convex. Convexity in multiple dimensions means that the gradient is nondecreasing in all dimensions. If f is twice differentiable, then if the Hessian is positive semi-definite, f is convex.

There are a variety of results that allow us to recognize and construct convex functions based on knowing what operations create and preserve convexity. The Boyd book is a good source for material on such operations. Note that norms are convex functions (based on the triangle inequality), $\| \sum_{i=1}^n \alpha_i x_i \| \leq \sum_{i=1}^n \alpha_i \|x_i\|$.

We'll talk about a general algorithm that works for convex functions (the MM algorithm) and

about the EM algorithm that is well-known in statistics, and is a special case of MM.

8.1 MM algorithm

The MM algorithm is really more of a principle for constructing problem specific algorithms. MM stands for majorize-minorize. We'll use the majorize part of it to minimize functions - the minorize part is the counterpart for maximizing functions.

Suppose we want to minimize a convex function, $f(x)$. The idea is to construct a majorizing function, at x_t, which we'll call g. g majorizes f at x_t if $f(x_t) = g(x_t)$ and $f(x) \leq g(x) \forall x$.

The iterative algorithm is as follows. Given x_t, construct a majorizing function $g(x_t)$. Then minimize g w.r.t. x (or at least move downhill, such as with a modified Newton step) to find x_{t+1}. Then we iterate, finding the next majorizing function. The algorithm is obviously guaranteed to go downhill, and ideally we use a function g that is easy to work with (i.e., to minimize or go downhill with respect to). Note that we haven't done any matrix inversions or computed any derivatives of f. Furthermore, the algorithm is numerically stable - it does not over- or undershoot the optimum. The downside is that convergence can be quite slow.

The tricky part is finding a good majorizing function. Basically one needs to gain some skill in working with inequalities. The Lange book has some discussion of this.

An example is for estimating regression coefficients for median regression (aka least absolute deviation regression), which minimizes $f(\theta) = \sum_{i=1}^{n} |y_i - z_i^\top \theta| = \sum_{i=1}^{n} |r_i(\theta)|$. Note that $f(\theta)$ is convex because affine functions (in this case $y_i - z_i^\top \theta$) are convex, convex functions of affine functions are convex, and the summation preserves the convexity. We'll work through this example in class. We'll make use the following (commonly-used) inequality, which holds for any concave function, f:

$$f(x) \leq f(y) + f'(y)(x - y).$$

8.2 Expectation-Maximization (EM)

It turns out the EM algorithm that many of you have heard about is a special case of MM. For our purpose here, we'll consider maximization.

The EM algorithm is most readily motivated from a missing data perspective. Suppose you want to maximize $L(\theta|X = x) = f(x|\theta)$ based on available data in a missing data context. Denote the complete data as $Y = (X, Z)$ with Z is missing. As we'll see, in many cases, Z is actually a set of latent variables that we introduce into the problem to formulate it so we can use EM. The canonical example is when Z are membership indicators in a mixture modeling context.

In general, $L(\theta|x)$ may be hard to optimize because it involves an integral over the missing

data, Z:

$$f(x|\theta) = \int f(x, z|\theta)dz,$$

but $L(\theta|y) = f(x, z|\theta)$ may be straightforward to optimize.

The algorithm is as follows. Let θ_t be the current value of θ. Then define

$$Q(\theta|\theta_t) = E(\log L(\theta|Y)|x, \theta_t)$$

The algorithm is

1. E step: Compute $Q(\theta|\theta_t)$, ideally calculating the expectation over the missing data in closed form. Note that $\log L(\theta|Y)$ is a function of θ so $Q(\theta|\theta_t)$ will involve both θ and θ_t.

2. M step: Maximize $Q(\theta|\theta_t)$ with respect to θ, finding θ_{t+1}.

3. Continue until convergence.

Ideally both the E and M steps can be done analytically. When the M step cannot be done analytically, one can employ some of the numerical optimization tools we've already seen. When the E step cannot be done analytically, one standard approach is to estimate the expectation by Monte Carlo, which produces Monte Carlo EM (MCEM). The strategy is to draw from z_j from $f(z|x, \theta_t)$ and approximate Q as a Monte Carlo average of $\log f(x, z_j|\theta)$, and then optimize over this approximation to the expectation. If one can't draw in closed form from the conditional density, one strategy is to do a short MCMC to draw a (correlated) sample.

EM can be show to increase the value of the function at each step using Jensen's inequality (equivalent to the information inequality that holds with regard to the Kullback-Leibler divergence between two distributions) (Givens and Hoeting, p. 95, go through the details). Furthermore, one can show that it amounts, at each step, to maximizing a minorizing function for $\log L(\theta)$ - the minorizing function (effectively Q) is tangent to $\log L(\theta)$ at θ_t and lies below $\log L(\theta)$.

A standard example is a mixture model. Suppose we have

$$f(x) = \sum_{k=1}^{K} \pi_k f_k(x; \phi_k)$$

where we have K mixture components and π_k are the (marginal) probabilities of being in each component. The complete parameter vector is $\theta = \{\{\pi_k\}, \{\phi_k\}\}$. Note that the likelihood is a complicated product (over observations) over the sum (over components), so maximization may be difficult. Furthermore, such likelihoods are well-known to be multimodal because of label switching.

To use EM, we take the group membership indicators for each observation as the missing data. For the ith observation, we have $z_i \in \{1, 2, \ldots, K\}$. Introducing these indicators "breaks the mixture". If we know the memberships for all the observations, it's often easy to estimate the parameters for each group based on the observations from that group. For example if the $\{f_k\}$'s were normal densities, then we can estimate the mean and variance of each normal density using the sample mean and sample variance of the x_i's that belong to each mixture component. EM will give us a variation on this that uses "soft" (i.e., probabilistic) weighting.

The complete log likelihood given z and x is

$$\log \prod_i f(x_i|z_i, \theta) \text{Pr}(Z_i = z_i|\pi)$$

which can be expressed as

$$\log L(x, z|\theta) = \sum_i \sum_k I(z_i = k)(\log f_k(x_i|\phi_k) + \log \pi_k)$$

with Q equal to

$$Q(\theta|\theta_t) = \sum_i \sum_k E(I(z_i = k)|x_i, \theta_t)(\log f_k(y_i|\phi_k) + \log \pi_k)$$

where $E(I(z_i = k)|x_i, \theta_t)$ is equal to the probability that the ith observation is in the kth group given x_i and θ_t, which is calculated from Bayes theorem as

$$p_{ikt} = \frac{\pi_{k,t} f_k(x_i|\theta_t)}{\sum_j \pi_{j,t} f_j(x_i|\theta_t)}$$

We can now separately maximize $Q(\theta|\theta_t)$ with respect to π_k and ϕ_k to find $\pi_{k,t+1}$ and $\phi_{k,t+1}$, since the expression is the sum of a term involving the parameters of the distributions and a term involving the mixture probabilities. In the latter case, if the f_k are normal distributions, you end up with a weighted sum of normal distributions, for which the estimators of the mean and variance parameters are the weighted mean of the observations and the weighted variance.

9 Optimization under constraints

Constrained optimization is harder than unconstrained, and inequality constraints harder to deal with than equality constraints.

Constrained optimization can sometimes be avoided by reparameterizing. E.g., to optimize w.r.t. a variance component or other non-negative parameter, you can work on the log scale.

Optimization under constraints often goes under the name of 'programming', with different types of programming for different types of objective functions combined with different types of constraints.

9.1 Convex optimization (convex programming)

Convex programming minimizes $f(x)$ s.t. $h_i(x) \leq 0$, $i = 1, \ldots, m$ and $a_i^\top x = b_i$, $i = 1, \ldots, q$, where both f and the constraint functions are convex. Note that this includes more general equality constraints, as we can write $g(x) = b$ as two inequalities $g(x) \leq b$ and $g(x) \geq b$. It also includes $h_i(x) \geq b_i$ by taking $-h_i(x)$. Also note that we can always have $h_i(x) \leq b_i$ and convert to the above form by subtracting b_i from each side (note that this preserves convexity). A vector x is said to be feasible, or in the feasible set, if all the constraints are satisfied for x.

There are good algorithms for convex programming, and it's possible to find solutions when we have hundreds or thousands of variables and constraints. It is often difficult to recognize if one has a convex program (i.e., if f and the constraint functions are convex), but there are many tricks to transform a problem into a convex program and many problems can be solved through convex programming. So the basic challenge is in recognizing or transforming a problem to one of convex optimization; once you've done that, you can rely on existing methods to find the solution.

Linear programming, quadratic programming, second order cone programming and semidefinite programming are all special cases of convex programming. In general, these types of optimization are progressively more computationally complex.

First let's see some of the special cases and then discuss the more general problem.

9.2 Linear programming: Linear system, linear constraints

Linear programming seeks to minimize

$$f(x) = c^\top x$$

subject to a system of m inequality constraints, $a_i^\top x \leq b_i$ for $i = 1, \ldots, m$, where A is of full row rank. This can also be written in terms of generalized inequality notation, $Ax \preceq b$. There are standard algorithms for solving linear programs, including the simplex method and interior point methods.

Note that each equation in the set of equations $Ax = b$ defines a hyperplane, so each inequality in $Ax \preceq b$ defines a half-space. Minimizing a linear function (presuming that the minimum exists) must mean that we push in the correct direction towards the boundaries formed by the hyperplanes, with the solution occuring at a corner (vertex) of the solid formed by the hyperplanes. The sim-

plex algorithm starts with a feasible solution at a corner and moves along edges in directions that improve the objective function.

9.3 General system, equality constraints

Suppose we have an objective function $f(x)$ and we have equality constraints, $Ax = b$. We can manipulate this into an unconstrained problem. The null space of A is the set of x s.t. $Ax = 0$. So if we start with a candidate x_c s.t. $Ax_c = b$ (e.g., by using the pseudo inverse, A^+b), we can form all other candidates (a candidate is an x s.t. $Ax = b$) as $x = x_c + Bz$ where B is a set of column basis functions for the null space of A and $z \in \Re^{p-m}$. Consider $h(z) = f(x_c + Bz)$ and note that h is a function of $p - m$ rather than p inputs. Namely, we are working in a reduced dimension space with no constraints. If we assume differentiability of f, we can express $\nabla h(z) = B^\top \nabla f(x_c + Bz)$ and $H_h(z) = B^\top H_f(x_c + Bz)B$. Then we can use unconstrained methods to find the point at which $\nabla h(z) = 0$.

How do we find B? One option is to use the $p - m$ columns of V in the SVD of A that correspond to singular values that are zero. A second option is to take the QR decomposition of $A^\top$. Then B is the columns of Q_2, where these are the columns of the (non-skinny) Q matrix corresponding to the rows of R that are zero.

For more general (nonlinear) equality constraints, $g_i(x) = b_i$, $i = 1, \ldots, q$, we can use the Lagrange multiplier approach to define a new objective function,

$$L(x, \lambda) = f(x) + \lambda^\top (g(x) - b)$$

for which, if we set the derivative (with respect to both x and the Lagrange multiplier vector, λ) equal to zero, we have a critical point of the original function and we respect the constraints.

An example occurs with quadratic programming, under the simplification of affine equality constraints (quadratic programming in general optimizes a quadratic function under affine inequality constraints - i.e., constraints of the form $Ax - b \preceq 0$). For example we might solve a least squares problem subject to linear equality constraints, $f(x) = \frac{1}{2}x^\top Qx + m^\top x + c$ s.t. $Ax = b$, where Q is positive semi-definite. The Lagrange multiplier approach gives the objective function

$$L(x, \lambda) = \frac{1}{2}x^\top Qx + m^\top x + c + \lambda^\top (Ax - b)$$

and differentiating gives the equations

$$\begin{aligned} \frac{\partial L(x,\lambda)}{\partial x} = m + Qx + A^\top \lambda &= 0 \\ \frac{\partial L(x,\lambda)}{\partial \lambda} = Ax &= b, \end{aligned}$$

which leads to the solution

$$\begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} Q & A^\top \\ A & 0 \end{pmatrix}^{-1} \begin{pmatrix} -m \\ b \end{pmatrix} \tag{1}$$

which gives us $x^* = -Q^{-1}m + Q^{-1}A^\top (AQ^{-1}A^\top)^{-1}(AQ^{-1}m + b)$.

Under inequality constraints there are a variety of methods but we won't go into them.

9.4 The dual problem

Sometimes a reformulation of the problem eases the optimization. There are different kinds of dual problems, but we'll just deal with the Lagrangian dual. Let $f(x)$ be the function we want to minimize, under constraints $g_i(x) = 0; i = 1, \ldots, q$ and $h_j(x) \leq 0; j = 1, \ldots, m$. Here I've explicitly written out the equality constraints to follow the notation in Lange. Consider the Langrangian,

$$L(x, \lambda, \mu) = f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x).$$

Its Lagrange dual function is

$$d(\lambda, \mu) = \inf_x L(x, \lambda, \mu).$$

For $\mu \succeq 0$, one can easily show that $d(\lambda, \mu) \leq f(x^*)$ for the minimizing value x^* (p. 216 of the Boyd book), so the challenge is then to find the best lower bound.

Thus, the Lagrange dual problem is to maximize $d(\lambda, \mu)$ subject to $\mu \succeq 0$, which is always a convex optimization problem because $d(\lambda, \mu)$ is concave (because $d(\lambda, \mu)$ is a pointwise infimum of a family of affine functions of (λ, μ)). If the optima of the primal (original) problem and that of the dual do not coincide, there is said to be a "duality gap". For convex programming, if certain conditions are satisfied (called *constraint qualifications*), then there is no duality gap, and one can solve the dual problem to solve the primal problem. Usually with the standard form of convex programming, there is no duality gap. Provided we can find the infimum over x in closed form we then maximize $d(\lambda, \mu)$ w.r.t. the Lagrangian multipliers in a new constrained problem that is sometimes easier to solve, giving us (λ^*, μ^*).

One can show (p. 242 of the Boyd book) that $\mu_i^* = 0$ unless the ith constraint is active at the

optimum x^* and that x^* minimizes $L(x, \lambda^*, \mu^*)$. So once one has (λ^*, μ^*), one is in the position of minimizing an unconstrained convex function. If $L(x, \lambda^*, \mu^*)$ is strictly convex, then x^* is the unique optimum provided x^* satisfies the constraints, and no optimum exists if it does not.

Here's a simple example: suppose we want to minimize $x^\top x$ s.t. $Ax = b$. The Lagrangian is $L(x, \lambda) = x^\top x + \lambda^\top (Ax - b)$. Since $L(x, \lambda)$ is quadratic in x, the infimum is found by setting $\nabla_x L(x, \lambda) = 2x + A^\top \lambda = 0$, yielding $x = -\frac{1}{2} A^\top \lambda$. So the dual function is obtained by plugging this value of x into $L(x, \lambda)$, which gives

$$g(\lambda) = -\frac{1}{4} \lambda^\top A A^\top \lambda - b^\top \lambda,$$

which is concave quadratic. In this case we can solve the original constrained problem in terms of this unconstrained dual problem.

9.5 KKT conditions

Karush-Kuhn-Tucker (KKT) theory provides sufficient conditions under which a constrained optimization problem has a minimum, generalizing the Lagrange multiplier approach. The Lange and Boyd books have whole sections on this topic.

Suppose that the function and the constraint functions are continuously differentiable near x^* and that we have the Lagrangian as before:

$$L(x, \lambda, \mu) = f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x).$$

For nonconvex problems, if x^* and (λ^*, μ^*) are the primal and dual optimal points and there is no duality gap, then the KKT conditions hold:

$$\begin{aligned} h_j(x^*) &\leq 0 \\ g_i(x^*) &= 0 \\ \mu_j^* &\geq 0 \\ \mu_j^* h_j(x) &= 0 \\ \nabla f(x^*) + \sum_i \lambda_i \nabla g_i(x^*) + \sum_j \mu_j \nabla h_j(x^*) &= 0. \end{aligned}$$

For convex problems, we also have that if the KKT conditions hold, then x^* and (λ^*, μ^*) are primal and dual optimal and there is no duality gap.

We can consider this from a slightly different perspective, in this case requiring that the Lagrangian be twice differentiable.

First we need a definition. A *tangent direction*, w, with respect to $g(x)$, is a vector for which $\nabla g_i(x)^\top w = 0$. If we are at a point, x^*, at which the constraint is satisfied, $g_i(x^*) = 0$, then we can move in the tangent direction (orthogonal to the gradient of the constraint function) (i.e., along the level curve) and still satisfy the constraint. This is the only kind of movement that is legitimate (gives us a feasible solution).

If the gradient of the Lagrangian with respect to x is equal to 0,

$$\nabla f(x^*) + \sum_i \lambda_i \nabla g_i(x^*) + \sum_j \mu_j \nabla h_j(x^*) = 0,$$

and if $w^\top H_L(x^*, \lambda, \mu) w > 0$ for all vectors w s.t. $\nabla g(x^*)^\top w = 0$ and, for all active constraints,$\nabla h(x^*)^\top w = 0$, then x^* is a local minimum. An active constraint is an inequality for which $h_j(x^*) = 0$ (rather than $h_j(x^*) < 0$, in which case it is inactive). Basically we only need to worry about the inequality constraints when we are on the boundary, so the goal is to keep the constraints inactive.

Some basic intuition is that we need positive definiteness only for directions that stay in the feasible region. That is, our only possible directions of movement (the tangent directions) keep us in the feasible region, and for these directions, we need the objective function to be increasing to have a minimum. If we were to move in a direction that goes outside the feasible region, it's ok for the quadratic form involving the Hessian to be negative.

Many algorithms for convex optimization can be interpreted as methods for solving the KKT conditions.

9.6 Interior-point methods

We'll briefly discuss one of the standard methods for solving a convex optimization problem. The barrier method is one type of interior-point algorithm. It turns out that Newton's method can be used to solve a constrained optimization problem, with twice-differentiable f and linear equality constraints. So the basic strategy of the barrier method is to turn the more complicated constraint problem into one with only linear equality constraints.

Recall our previous notation, in which convex programming minimizes $f(x)$ s.t. $h_i(x) \leq 0$, $i = 1, \ldots, m$ and $a_i^\top x = b_i$, $i = 1, \ldots, q$, where both f and the constraint functions are convex. The strategy begins with moving the inequality constraints into the objective function:

$$f(x) + \sum_{i=1}^{m} I_-(h_i(x))$$

where $I_-(u) = 0$ if $u \leq 0$ and $I_-(u) = \infty$ if $u > 0$.

This is fine, but the new objective function is not differentiable so we can't use a Newton-like

approach. Instead, we approximate the indicator function with a logarithmic function, giving the new objective function

$$\tilde{f}(x) = f(x) + \sum_{i=1}^{m} -(1/t)\log(-h_i(x)),$$

which is convex and differentiable. The new term pushes down the value of the overall objective function when x approaches the boundary, nearing points for which the inequality constraints are not met. The $-\sum(1/t)\log(-h_i(x))$ term is called the log barrier, since it keeps the solution in the feasible set (i.e., the set where the inequality constraints are satisfied), provided we start at a point in the feasible set. Newton's method with equality constraints ($Ax = b$) is then applied. The key thing is then to have t get larger as the iterations proceed, which allows the solution to get closer to the boundary if that is indeed where the minimum lies.

The basic ideas behind Newton's method with equality constraints are (1) start at a feasible point, x_0, such that $Ax_0 = b$, and (2) make sure that each step is in a feasible direction, $A(x_{t+1} - x_t) = 0$. To make sure the step is in a feasible direction we have to solve a linear system similar to that in the simplified quadratic programming problem (1):

$$\begin{pmatrix} x_{t+1} - x_t \\ \lambda \end{pmatrix} = \begin{pmatrix} H_{\tilde{f}}(x_t) & A^\top \\ A & 0 \end{pmatrix}^{-1} \begin{pmatrix} -\nabla\tilde{f}(x_t) \\ 0 \end{pmatrix},$$

which shouldn't be surprising since the whole idea of Newton's method is to substitute a quadratic approximation for the actual objective function.

10 Summary

The different methods of optimization have different advantages and disadvantages.

According to Lange, MM and EM are numerically stable and computationally simple but can converge very slowly. Newton's method shows very fast convergence but has the downsides we've discussed. Quasi-Newton methods fall in between. Convex optimization generally comes up when optimizing under constraints.

One caution about optimizing under constraints is that you just get a point estimate; quantifying uncertainty in your estimator is more difficult. One strategy is to ignore the inactive inequality constraints and reparameterize (based on the active equality constraints) to get an unconstrained problem in a lower-dimensional space.