Introductions to Computing

MJ Rutter
mjr19@cam

Lent 2009

Miscellaneous Commands

bc

The Basic Calculator. Much more useful than the even more basic `xcalc`.

The one surprise about `bc` is that it is a fixed point calculator, and it defaults to zero decimal places.

```
$ bc
3/4
0
```

This is fixed either by launching as '`bc -l`', which will default to twenty decimal places, or by setting the scale variable.

```
$ bc
scale=20
1/7
.14285714285714285714
```

It obeys usual precidence rules: $3 + 4/5$ is 3.8, not 1.4.

`bc` and Functions

Just one function is built in to `bc`: `sqrt`.

However, `bc` is a programmable calculator, and a maths library exists which defines several more common functions. Restrictions in the original `bc` force these functions to have single letter names.

`a(x)`	$\arctan(x)$	`e(x)`	$\exp(x)$
`c(x)`	$\cos(x)$	`l(x)`	$\ln(x)$
`s(x)`	$\sin(x)$	`j(n,x)`	$J_n(x)$

```
$ bc -l
4*a(1)
3.14159265358979323844
```

`bc` and Base Conversion

One can specify independently the input base and output base in `bc`.

```
obase=16
63
3F
ibase=16
3F+2
41
obase=10
41
41
obase=A
41
65
```

Note that the argument to the base commands is in the current input base. The command `ibase=10` will never have any effect.

Why `bc`**?**

It's free.
It runs in any text terminal (and will thus accept cut and paste as long as that terminal does so).
It is easy to use in scripts.
You need to know how many zeros are at the end of 30!

`tail` and `tee`

These commands are useful for examining the output of a program which is also saving its output to a file.

`tail file` prints the last few lines of the given file.

`tail -f file` likewise, but it continues to monitor the file and prints anything else added to it (unlike control C is pressed).

This can be used in conjunction with `grep`

```
$ tail -f output | grep Iteration
```

(Monitor file called output, and print every line which appears and contains the string ”Iteration”.)

tee

`tee file` copies `stdin` to both `stdout` and the file given. If a command writes a lot to `stdout`, then

```
$ verbose_cmd | tee output | less
```

will save a copy of its output as well as paging a copy to the terminal via `less`. Of course, `less` could be replaced with `grep` as above.

If one wishes to capture `stderr` along with `stdout`, then

```
$ verbose_cmd 2>&1 | tee output | less
```

will do so. (Not in `csh`/`tcsh`.)

less

{space}	next page
{enter}	next line
d	scroll about half a page
/text	search for next occurrence of `text`
?text	search for previous occurrence of `text`
n	repeat previous search
i	toggle case sensitivity in searches
v	start `vi`
q	quit
{ctrl}L	redraw screen
b	previous page
j	previous line
u	reverse scroll c. half a page
G	goto end of file
numG	goto line number `num` (1G for beginning)
{ctrl}G	display current position in file
F	act like `tail -f`

(Cursor keys should also work for movement. On Linux all of the above work whilst reading man pages.)

wc

WordCount. Actually counts characters, words or lines from `stdin` or files given.

How big is that compressed file?

```
$ bunzip2 -c foo.bz2 | wc -c
   32451
```

(Note that `gunzip` has a `-l` flag for doing the above more efficiently.)

How small can I compress that file?

```
$ wc -c cmds.tex
   17680 cmds.tex
$ compress -c cmds.tex | wc -c
    9360
$ gzip -c cmds.tex | wc -c
    6919
$ bzip2 -c cmds.tex | wc -c
    6667
```

man

The manual command, giving documentation which is often correct. The style of this manual is compact, technical and of most use to those who already know the answers! Consider first the man page of a command with which you are familiar.

WC(1) FSF WC(1)

NAME

wc - print the number of bytes, words, and lines in files

SYNOPSIS

wc [OPTION]... [FILE]...

DESCRIPTION

Print line, word, and byte counts for each FILE, and a
total line if more than one FILE is specified. With
no FILE, or when FILE is –, read standard input.

–c, **– –bytes**

print the byte counts

–l, **– –lines**

print the newline counts

[etc.]

grep

The trivial use of `grep` is to search for fixed strings in files.

```
$ grep Energy output
```

will display all lines containing the string ‘Energy’ in the file output.

Three useful flags at this level are:

`-c` print number of lines found
`-i` ignore case
`-v` display/count lines which don’t match.

Like many simple commands, if given no filename, `grep` will read from `stdin` instead.

Quotes

The first argument not starting with a hyphen is the thing grep will search for.

```
$ ./wonder_code | grep Total Energy
```

is almost certainly an error, with

```
$ ./wonder_code | grep 'Total Energy'
```

being what was intended. The use of quotes is more and more likely to be required as we progress...

(There is much more on the use of quotes in the next lecture.)

Regular Expressions

In fact `grep` searches for *regular expressions*, not fixed strings. These are unlike the shell wildcards already discussed.

For simple matches, the special characters most worth remembering are:

{dot}	matches any character
`*`	any number (including zero) of previous item
`^`	the start of a line
`$`	the end of a line

This enables us to do:

`grep -i '^c' foo.f` (find comments in F77 code)

`grep 'a.*e.*i.*o.*u' /usr/share/dict/words` (find words containing all five vowels in order)

`grep independ.nt' /usr/share/dict/words` (how is it spelt?)

Text searches in `less` are regular expression searches. Awk, emacs, perl, python, sed and vi all use regular expressions too.

Alternatives

The final useful and common piece of regular expression syntax is to provide a list of alternative characters. This is simply done with square brackets. It is particularly useful for searching for numbers:

```
$ grep '[0123456789]' output.dat
$ grep '[0-9]' output.dat
```

The above two lines are equivalent.

If the first character is a `^`, the list is negated. So, for five vowels in order with no stray vowels intervening,

```
$ grep 'a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u'
```

The above expression matches on ‘abstemiousness’, whereas the expression on the previous page also matches on ‘adventitious’.

Beware Collation Orders

The sane world uses ASCII for its collation order when talking to computers.

0–9	48–57
A–Z	65–90
a–z	97–122

The insane world doesn't. It matters because in the sane world `[A-Z]` means all capital letters. In the insane world, it could mean anything, but will probably mean all letters excluding 'z' (from a belief that letters are ordered AaBbCc...).

This mess was caused by an insignificant island south of the Isle of Wight whose inhabitants abandoned Latin in favour of some language in which e, E, è, é, ê, ë, È, É, Ê and Ë are expected to remain together when sorted. The ASCII/ISO codes for that lot are 101, 69, 232 to 236 and 200 to 203, which is less random than could have been the case.

The environment variables `$LANG`, `$LC_ALL` and `LC_COLLATE` can all influence this behaviour.

Remember Anchors

The command

```
$ grep '[^aeiou]*' /usr/share/dict/words
```

might be an attempt to find words with no vowels. It fails. It finds lines (words, in the case of that file) which contain at least zero characters which are not vowels. In other words, it will match absolutely anything.

```
$ grep -i '^[^aeiou]*$' /usr/share/dict/words
```

is a rather better version.

awk

Awk is a simple but powerful filter which is particularly good at numerics, as it supports all the standard numeric functions and has just one precision, equivalent to double. It operates on files line by line, with optional blocks for initialisation and finalisation.

Some simple examples follow.

`awk`, add third column as sum of first two

```
$ cat test.dat
1  5
2  4
3  3.1415
3  2E1
 27.5  -15
$ awk '{print $1,$2,$1+$2;}' test.dat
1 5 6
2 4 6
3 3.1415 6.1415
3 2E1 23
27.5 -15 12.5
```

Those familiar with C's printf can do pretty formatting tricks

```
$ awk '{printf "%8g %8g %8g\n",$1,$2,$1+$2;}' test.dat
       1        5        6
       2        4        6
       3   3.1415   6.1415
       3       20       23
    27.5      -15     12.5
```

`awk`, sum/average second column

```
$ awk '{t+=$2;} END {print t;}' test.dat
17.1415
$ awk '{t+=$2; c++;} END {print t/c;}' test.dat
3.4283
```

Variables do not need to be declared, and will be initialised to zero.

C-like operators are recognised, such as ++ for increment by one. Unlike C, ^ exists for exponentiation.

Like C, `x+=y` means `x=x+y`, with corresponding shorthands for `-`, `*`, `/`, `%` (integer division) and `^`.

`awk`, finding oddities

```
$ awk '{if ($2<0) {print "Warn: neg value " $2 " at line " NR;}}' test.dat
Warn: neg value -15 at line 5
```

The above can also be written

```
$ awk '($2<0) {print "Warn: neg value " $2 " at line " NR;}' test.dat
```

Other examples

```
$ awk '($2!=int($2)) {print "Non int value " $2 " at line " NR;}' test.dat
Non int value 3.1415 at line 3
$ awk '(NF!=2) {print "Line " NR " does not have two columns";}' test.dat
```

`awk`, finding flags

```
$ cat test2.dat
It's hello from him
Energy  -26.2
Energy  -28.3
Bananas  17
Energy  -28.2
Energy  -28.5
$ awk '/^Energy/ {t+=$2;c++} END {print t/c;}' test2.dat
-27.8
```

The use of `awk` is particularly appropriate when its string and number processing abilities can be combined in under about 100 characters.

`awk`, unexpected growth

```
$ awk '/^Energy/ {if ($2>Eold) {
   print "Warning! Energy increase from " Eold " to " $2;};
   Eold=$2;}' test2.dat
Warning! Energy increase from -28.3 to -28.2
```

As those line-breaks are within a single-quoted string as far as the shell is concerned, they will be treated as part of the string, not the end of the command.

The line-breaks at those points are acceptable to `awk`, so all is well. A better way of doing this may be found later.

awk, a vague syntax

As may be become clear, each line of the input file is passed through every statement in the awk program. The awk program statement has the form
`condition { action }`
where condition may be absent, it may be an arithmetic expression in brackets, or it may be a regular expression between forward slashes. Finally, the special expressions BEGIN and END precede actions which will execute before the first line is processed and after the last.

Variables set when each line is processed include `$0` (the whole line), `$1` to `$9` (the first nine columns), NF (number of fields in current line), NR (number of current record (line) in file).

Numeric functions available are `exp`, `log`, `sqrt`, `int`, `sin`, `cos`, and `atan2`.

Variables are typeless, and will be interpretted as numeric or string according to context.

`awk` and Hashes

A hash is an array indexed by something other than a small increasing integer. It is a very useful feature of `awk`, `perl` and `python` amongst others.

```
$ cat test3.dat
fred  7
harry 6
fred 2
mike 4
harry 3
$ awk '{t[$1]+=$2;} END {for (ind in t) {print ind, t[ind];}}' test3.dat
fred 9
harry 9
mike 4
```

There is no concept of the ordering of the elements of a hash. If you need to sort them, you may wish to try `perl` or `python`.

This sort of processing is unpleasant in languages which do not have hashes.

`awk` and skipping

Sometimes it is useful to be able to skip certain input lines. This can be achieved using `next`, which immediately abandons processing of the current line.

```
$ cat test.dat
1  5
2  4
# note that exponent syntax is accepted
3  2E1
# 27.5  -15
$ awk '/^#/ {next} {t+=$2; c++;} END {print t/c;}' test.dat
9.66667
```

For reassurance that `awk` does use double precision, replace '`print`' above with '`printf "%.10g\n",`'.

Lots of Awks

Beware that `awk` has suffered much improvement over the years. As usual, Gnu's version is particularly stuffed with extensions, and sometimes goes by the name of `gawk` to distinguish it from the original(s).

The very original, written by Aho, Weinberger and Kernighan (hence the name) appeared in 1977. It benefited from significant revisions at the end of the 1980s. It was standardised by POSIX in 1992.

If a piece of `awk` code is to be portable to POSIX versions of `awk`, then it will run correctly with Gnu's `awk` and the `--posix` flag.

The late eighties version used to be called `nawk` (New awk) to distinguish it from earlier versions. This distinction has been dropped, as it would now be surprising to find a pre-`nawk` version.

sed

The name `sed` stands for Stream EDitor. It can perform simple operations on a file line–by–line. Being command-line based, it can be easier to use in scripts than a full editor.

Its most common use is probably to make substitutions. These can be trivial:

```
$ sed 's/color/colour/g' < us.txt > mid_atlantic.txt
```

The option 'g' specifies that the substitution should not simply apply to the first occurance on the line, but to all occurances.

`sed` and Numeric Addresses

A `sed` command can be prefixed by an address, or an address range.

```
$ sed '10,100s/color/colour/g' < us.txt > mid_atlantic.txt
```

operate on lines 10 to 100 inclusive

```
$ sed '50s/color/colour/g' < us.txt > mid_atlantic.txt
```

operate on line 50

```
$ sed '25,$s/color/colour/g' < us.txt > mid_atlantic.txt
```

operate on lines 25 to the end.

`sed` and Regular Expression Addresses

```
$ cat in
Some preamble
A line containing Final Energy 1223
More text
$ sed '1,/Final Energy/d' < in
More text
```

Possibly more useful is the form

```
$ sed -e '/Final Energy/,$p' -e 'd' < in
A line containing Final Energy 1223
More text
```

To specify multiple `sed` command as arguments, each must be preceeded by `-e`.

d – delete line, print nothing
p – print current line. This is the default action when all commands have been processed.

```
$ cat in
Start
1
End
2
Start
3
End
4
$ sed '/Start/,/End/d' < in
2
4
```

Once the lines between ‘Start’ and ‘End’ are processed (deleted), the expression looks for the next occurance of ‘Start’.

`sed` and Backreferences

```
$ cat in
Is it not Dreadfully annoying when 2.34D5 or 6.7D-5
appears and C-like programs fail? Don't even think of 1.D+03.
$ sed 's/\([0-9.]\)D\([-+0-9]\)/\1e\2/g' <in
Is it not Dreadfully annoying when 2.34e5 or 6.7e-5
appears and C-like programs fail? Don't even think of 1.e+03.
```

So what does that nasty line mean? Think first of
`[0-9.]D[-+0-9]`
This will catch a capital D with a digit or point on its left, and a digit or sign on its right.

The escaped brackets, `\(` and `\)` then save the parts of the string which the fragments they enclose match. These two saved items are then referred to in the substitution string, `\1` being the first, and `\2` the second.

So the first match on the first line is of 4D5, with `[0-9.]` matching 4 and `[-+0-9]` matching 5. The substitution string is thus constructed as 4e5.

More Backreferences

Just before Michaelmas 2009, TCM suffered the pain of renaming its PCs from names of the form
`tcmpc20.phy.cam.ac.uk` to `pc20.tcm.phy.cam.ac.uk`.

```
$ sed 's/tcm\(pc[0-9]*\)\.phy/\1.tcm.phy/g' <old >new
```

Unfortunately `sed` operates only on files and streams. It is not possible to pipe medium-sized planets into it. Nor is it able to operate on a large number of files simultaneously.

Note that on the lhs `\.` is needed to match only a dot, for a bare dot would match any character. No such quoting of dots (or stars) is needed on the rhs, for the rhs is not a regular expression.

Regular Expressions and Greedy Stars

The asterix will always match as many characters as possible, but never so many that it causes the whole expression to fail to match when otherwise it would have done so.

```
$ echo "Hello World" | sed 's/l.*l//'
Hed
$ echo "Hello Cambridge" | sed 's/l.*l//'
Heo Cambridge
```

Rather than using a bare asterix, one can specify a range of repetition counts explicitly.

```
$ echo "Hello World" | sed 's/l.\{0,3\}l//'
Heo World
$ echo "Hello World" | sed 's/l.\{3,\}l//'
Hed
```

The possible counts are of the form ‘n’, ‘n,’ or ‘n,m’.

Space Killing

Do your ASCII data files suffer from leading spaces, double spaces and trailing spaces? Let

```
$ sed -e 's/^ *//' -e 's/  */ /g' -e 's/ *$//' < old > new
```

Sort them out!

Do your shell scripts have comments?

```
$ sed -e '1p' -e '/^#/d' <old >new
```

Not now...(It is assumed that such scripts will start `#!` and need the first line preserving.)

Not `sed`

Regular expression search and replace can be useful in editors too.

In `vi` one can move to a line, and type ‘ma’ to set a mark called ‘a’, and then move elsewhere and type
`,'as/^/!/`
which will, from the current position to the mark ‘a’, substitute (at) the begining of the line a ‘!’ – in other words, it will comment out a block of Fortran.

Emacs will do a regular expression search and replace on pressing {ctrl}{meta}{%}. Setting a mark by first pressing {ctrl}{space} might restrict it to a given region depending on precisely which mode it is in.

Regular Expression Surprises

There are, unfortunately, two competing syntaxes for regular expressions. The old BRE (Basic Regular Expression) and the new ERE (ExtendedRE). Neither looks in much danger of dying.

BRE	ERE	
`\( \)`	`( )`	backreferences
`\{ \}`	`{ }`	repeat counts
	`|`	alternatives
`\{1,\}`	`+`	non-zero repetition
`\{0,1\}`	`?`	
`^` (not 1st char)	`\^`	a literal `^`
`$` (not last char)	`\$`	a literal `$`

Some commands, such as `grep` can use either syntax. In the case of `grep`, the command line switch `-E` switches to extended regular expressions.

Chaos

```
$ echo 'wom^bat' | grep 'm^'
wom^bat
$ echo 'wom^bat' | grep -E 'm^'
$ echo 'wom^bat' | grep '+'
$ echo 'wom^bat' | grep -E '+'
wom^bat
$ echo 'wom^bat' | grep 'bat|frog'
$ echo 'wom^bat' | grep -E 'bat|frog'
wom^bat
$ echo 'wom^bat' | grep '?'
$ echo 'wom^bat' | grep -E '?'
wom^bat
$ echo '(wombat)' | grep '('
(wombat)
$ echo '(wombat)' | grep -E '('
grep: Unmatched ( or \(
```

Whereas with BREs only `.`, `*`, `[` and `\` are special (also `^` and `$` and the begining and end of expressions), in EREs the list is rather longer, including `+`, `?`, `(` `|` and `{`, and `^` and `$` anywhere. This is more likely to confuse people looking for fixed strings.

Sorting Chaos

```
$ cat test.dat
aardvark
Alfred
geese
ants
$ sort test.dat
Alfred
aardvark
ants
geese
$ LC_ALL=en_GB sort test.dat
aardvark
Alfred
ants
geese
```

Flags exist for picking the column to use as the key (`-k`), for sorting in reverse order (`-r`), and and for sorting into numeric order (`-n`).

Note that '`var=value cmd`' sets `var` to `value` and exports it for the command given only, returning it to its original status for the next command. This trick is unavailable in `[t]csh`.

Sorting Numbers

```
$ cat test2.dat
3       40
22      2
7    34
2          35
12      12
$ sort test2.dat
12      12
2          35
22      2
3       40
7    34
$ sort -n test2.dat
2          35
3       40
7    34
12      12
22      2
$ sort -n -k2 test2.dat
22      2
12      12
7    34
2          35
3       40
```