Northwestern University
Earth & Planetary Sciences Linux Computer System
New User Notes
Seth Stein, March 2007

The departmental computing facility consists of two basic components. One is a set of Macintoshes. The second is a Linux workstation network with a number of workstations, printers, a color printer/plotters. All new Linux users are installed on the C-shell, or similar shells, which are command interpreters whose syntax is similar to that of the C programming language.

The assignments in the class are most easily done using the Linux system. You can work either directly on a Linux workstation, or by logging into one from another computer.

Workstations

The workstations in the computer room (second floor) are for student use. Other possibilities include graduate student offices: the occupants have priority.

The Network

All the computers in the network have local names, e.g. "beno", and full names "beno.earth.northwestern.edu"

Logging In

1. Respond to login prompt with your login name and <return>.
2. Respond to password prompt with your password and <return>.
3. If you are logged into one computer, and would like to log onto another, use the “ssh” (secure shell) command. For example, to log onto lothlorien, type “ssh lothlorien”.

Remote access

Users can log in using programs like "Terminal" or "Nifty Telnet" that support ssh and the computers’ full names: "ssh -lusername beno.earth.northwestern.edu"

Learning Unix/Linux

There are many ways to learn Unix/Linux. In addition to the many Linux books, web sites like the introductory tutorial on

http://www.ee.surrey.ac.uk/Teaching/Unix/

are very helpful.

The *man* command provides descriptions of Linux functions on the screen. For example, typing “man mail” will produce a description of the mail program. The *apropos* command finds manual sections appropos to a subject. For example, “apropos string” would print the names of all the sections that are about strings.

Editor

Most users use the "vi" editor. This is described in books and on the web. A short introduction is attached, from

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html

Producing Hard Copy

A file can be printed on the laser printers in the computer room by typing:

enscript filename for text files
lpr filename for text or postscript files

Languages

The following languages are currently on the system: C, C++, FORTRAN 77, Perl. Fortran is the one most commonly used in geophysics classes here.

Fortran

FORTRAN 77 programs must have a filename ending with ".f" To compile type

f77 prog.f

This produces an executable file called "a.out" which can be run by simply typing "a.out". FORTRAN programs traditionally read from logical unit 5 and write to unit 6. These can be any files, as in:

a.out < inputfile > outputfile

If no input or output file is specified, the defaults are the keyboard and the screen, respectively. The name of the executable file can be changed using the "-o" flag, so to compile the program "curve.f" and name it "curve", type

f77 curve.f -o curve

More details may be found in Fortran manuals or experienced users

Programming thoughts

Thus class relies heavily on using computers to solve geophysical problems. This involves *scientific programming*, a programming style used for essentially mathematical applications. This is a lot easier than the object oriented program using to do things like manipulate windows and icons. Most assignments are done in Fortran, a language that is especially suitable for scientific programming and is therefore commonly used in geophysics. Other languages can also be used. There's a general discussion in Stein & Wysession, Section A.8.

Many of the class assignments require *modular programming.* he idea is to divide large programs into smaller subroutines or functions, which can be used like the functions (e.g. sine, square root) supplied by many computer languages. Each subroutine can be tested separately and then used in various programs. Subroutines can handle applications that frequently recur, such as reading or plotting data or carrying out a mathematical operation. This approach saves the time needed to write and debug portions of a program similar to one already available.Moreover, the overall structure of a program containing a set of calls to subroutines is generally easier to understand, because many complexities are isolated into subroutines.

For example:

```
c exsub.f: simple example of using subroutines and functions
c
c       To compile this, type f77 exsub.f -o exsub
c      that makes the executable fi le "exsub"
c      To run type "exsub"
c
c      read two numbers from standard input (unit 5)
       read(5,*) a,b
c      check the input
        write(6,*) 'add two numbers:', a, b
c use subroutine
        call subadd(a,b,c)
        write(6,*) ' subroutine sum of',a,' and ',b,' is ',c

c use function
        d= funadd(a,b)
        write(6,*) ' function sum of',a,' and ',b,' is ',d
        stop
        end

        subroutine subadd(a,b,c)
        c=a+b
        return
        end

        function funadd(a,b)
        funadd=a+b
        return
        end
```

An Extremely Quick and Simple Introduction to the Vi Text Editor

Norm Matloff

(last updated October 25, 2006)

1 Overview

A **text editor** is a program that can be used to create and modify text files. One of the most popular editors on Linux/Unix systems (it is also available on Windows and many other platforms) is **vi**. There are many variations, with the most popular being vim.

2 5-Minute Introduction

As a brief introduction to vi, go through the following: First, type

```
vi x
```

at the Unix prompt. Assuming you did not already have a file named x, this command will create one. (If you have tried this example before, x will already exist, and **vi** will work on it. If you wish to start the example from scratch, simply remove x first.)

The file will of course initially be empty. To put something in it, type the letter **i** (it stands for "insert-text mode"), and type the following (including hitting the Enter key at the end of each of the three lines):

```
The quick
brown
fox will return.
```

Then hit the Escape key, to end insert-text-mode.

This mode-oriented aspect of the vi editor differs from many other editors in this respect. With modeless editors such as joe and emacs, for instance, to insert text at the cursor position, one simply starts typing, and to stop inserting, one just stops typing! However, that means that in order to perform most commands, one needs to use the Control key (in order to distinguish a command from text to be inserted). This has given rise to jokes that heavy users of modeless editors develop gnarled fingers.

Now save the file and exit **vi**, by typing **ZZ** (note the capitals).

Again, the key to learning vi is to keep in mind always the difference between insert-text mode and command mode. In the latter mode, as its name implies, one issues commands, such as the ZZ above, which we issued to save the file and exit **vi**. The characters you type will appear on the screen if you are in insert-text mode, whereas they will not appear on the screen while you are in command

mode. By far the most frequent problem new **vi** users have is that they forget they are in insert-text mode, and so their commands are not obeyed.

For example, suppose a new user wants to type ZZ, to save the file and exit **vi**, but he has forgotten to hit the Escape key to terminate insert-text mode. Then the ZZ will appear on the screen, and will become part of the text of the file-and the ZZ command will not be obeyed.

You now have a file named x. You can check its contents by typing (at the Unix shell prompt)

```
more x
```

which will yield

```
The quick
brown
fox will return.
```

just as expected.

Now let's see how we can use **vi** again to modify that file. Type

```
vi x
```

again, and make the following changes.

First, suppose we wish to say the fox will *not* return: We need to first move the cursor to the word "return". To do this, type **/re** and hit the Enter key, which instructs **vi** to move the cursor to the first instance of **/re** relative to the current cursor position. (Note that typing only **/r**' would have moved the cursor to the first instance of `r', which would be the `r' in `brown', not what we want.)

Now use the **i** command again: Hit **i**, then type **not** (note the space), and then hit Escape.

Next, let's delete the word `brown'. Type **b** to move the cursor there, and then hit **x** five times, to delete each of the five letters in `brown'. (This will still leave us with a blank line. If we did not want this, we could have used the **dd**' command, which would have deleted the entire line.)

Now type **ZZ** to save the file and exit vi. Use the **more** command again to convince yourself that you did indeed modify the file.

At this point you know the basics. You may wish to print or constantly display the excellent, clever and colorful VIM Graphical Cheat Sheet.

3 Going Further: Other Frequently-Used Commands

You now know how to use **vi** to insert text, move the cursor to text, and delete text. Technically, the bare-bones set of commands introduced above is sufficient for any use of vi. However, if you limit yourself to these few commands, you will be doing a large amount of unnecessary, tiresome typing.

So, you should also learn at least some of these other frequently-used **vi** commands:

```
h               move cursor one character to left
j               move cursor one line down
k               move cursor one line up
l               move cursor one character to right
```

```
w               move cursor one word to right
b               move cursor one word to left
0               move cursor to beginning of line
$               move cursor to end of line
nG              move cursor to line n
control-f       scroll forward one screen
control-b       scroll backward one screen

i               insert to left of current cursor position (end with ESC)
a               append to right of current cursor position (end with ESC)
dw              delete current word (end with ESC)
cw              change current word (end with ESC)
r               change current character
~               change case (upper-, lower-) of current character

dd              delete current line
D               delete portion of current line to right of the cursor
x               delete current character
ma              mark currrent position
d`a             delete everything from the marked position to here
`a              go back to the marked position
p               dump out at current place your last deletion (``paste'')

u               undo the last command
.               repeat the last command

J               combine (``join'') next line with this one

:w              write file to disk, stay in vi
:q!             quit VI, do not write file to disk,
ZZ              write file to disk, quit vi

:r filename     read in a copy of the specified file to the current
                buffer

/string         search forward for string (end with Enter)
?string         search backward for string (end with Enter)
n               repeat the last search (``next search'')

:s/s1/s2        replace (``substitute'') (the first) s1 in this line by s2
:lr/s/s1/s2/g   replace all instances of s1 in the line range lr by s2
                (lr is of form `a,b', where a and b are either explicit
                line numbers, or . (current line) or $ (last line)
:map k s        map the key k to a string of vi commands s (see below)
:abb s1 s2      expand the string s1 in append/insert mode to a string
                s2 (see below)
%               go to the "mate," if one exists, of this parenthesis
                or brace or bracket (very useful for programmers!)
```

All of the `:' commands end with your hitting the Enter key. (By the way, these are called "ex" commands, after the name of the simpler editor from which **vi** is descended.)

The **a** command, which puts text to the right of the cursor, does put you in insert-text mode, just like the **i** command does.

By the way, if you need to insert a control character while in append/insert mode, hit control-v first. For example, to insert control-g into the file being edited, type control-v then control-g.

One of **vi**'s advantages is easy cursor movement. Since the keys h,j,k,l are adjacent and easily

accessible with the fingers of your right hand, you can quickly reach them to move the cursor, instead of fumbling around for the arrow keys as with many other editors (though they can be used in **vi** too). You will find that this use of h,j,k,l become second nature to you very quickly, very much increasing your speed, efficiency and enjoyment of text editing.

Many of the commands can be prefixed by a number. For example, 3dd means to delete (consecutive) three lines, starting with the current one. As an another example, 4cw will delete the next four words.

The p command can be used for "cut-and-paste" and copy operations. For example, to move three lines from place A to place B:

1. Move the cursor to A.

2. Type **3dd**.

3. Move the cursor to B.

4. Type **p**.

The same steps can be used to copy text, except that p must be used twice, the first time being immediately after Step 2 (to put back the text just deleted).

Note that you can do operations like cut-and-paste, cursor movement, and so on, much more easily using a mouse. This requires a GUI version of **vi**, which we will discuss later in this document.

4 Advanced Topics

4.1 Macros

When you are using **vi**, you can use the **map** and **abb** commands to save a lot of typing. For example, I often accidentally transpose two letters when I am typing fast, say typing `taht' instead of `that'. Since I do this so often, I place the command

```
:map v xp
```

which means that the **v** key now performs the operations **x**and **p**, (try `xp' yourself and you will see it work), in my

```
~/.exrc
```

file ((without the colon; see below).

Also, since I often edit HTML files, I save myself typing by including lines like

```
abb cg <FONT color=green>
```

in my .exrc file. This means that whenever I am vi's insert/append mode and type "cg" and then hit the space bar, **vi** will automatically expand it to "<FONT color=green>".

Here are some more examples:

```
map ; $
map - 1G
```

```
map \ $G

map ^K ~

map ^X :.,$d^M

map! ^P ^[a. ^[hbmmi?\<^[2h"zdt.@z^Mywmx`mP xi
map! ^N ^[a. ^[hbmmi/\<^[2h"zdt.@z^Mywmx`mP xi

abb taht that
abb wb http://heather.cs.ucdavis.edu/~matloff
```

The first three simply perform cursor movement (to end-of-line, start-of-file, end-of-file) Most of them only saves one keystroke, but they require much less finger movement (for the standard touch-typing hand position) and since they are such frequently-used operations they are worthwhile. The fourth map is for case change, again (for me) a frequent operation.

The fifth map deletes all material from the current cursor position to the end of the file. I often find this useful, when editing a reply to an e-mail message for instance, or when I use :r to import another file into the one I am editing.

The sixth and seventh maps, which are labeled "map!" instead of "map" to indicate that they operate during append or insert mode, are modifications of some macros which are "famous" in the vi user community. They are used for "word completion," an extremely useful trick to save typing. Suppose for example I am currently in append/insert mode and I wish to type the word "investigation," and that I have used the word previously. If I just type, say, "inv" and then control-p, **vi** will search for a word earlier in my file which began with "inv" and complete my "inv" to that word, in this case "investigation ". Typing control-n will do the same thing, except that it will search forward instead of backwards.

Note again that in typing these macros in one's .exrc file, one must hit control-v first. For example, to insert control-g into the file being edited during append/insert mode, type control-v then control-g.

4.2 The .exrc Startup File

When you invoke the **vi** editor, it will look for the file

```
~/.exrc
```

and obey any "ex" commands it finds there. For example, I have lines in my startup file corresponding to the map and abb examples in the last section:

```
map v xp
abb cg <FONT color=green>
```

(Note that in the .exrc file we omit the colon, i.e. we type "map" instead of ":map", because **vi** assumes these are all "ex" commands.) That way I have those settings (and many others) permanently set, rather than my needing to type them in again each time I use vi.

4.3 Mainly for Programmers

There are a number of editing commands available in **vi** and most other sophisticated text editors which are especially useful for programmers. They are described in

(click here)

`http://heather.cs.ucdavis.edu/~matloff/progedit.html`

The reader is urged to make daily use of these, which can really save a lot of time and effort.

4.4 Lots and Lots About Vi

There is a large compendium of information about **vi** at

(click here)

`http://www.math.fu-berlin.de/~guckes/vi/`

A nice compact reference for **vi** commands is available at

(click here)

`http://www.ungerhu.com/jxh/vi.html`

5 Other Editors, Including Other Versions of Vi

Arguments over the pros and cons of various editors become almost "religious" in their ferocity. I have tried all of the major Unix text editors, but have always come back to **vi**.

In my view, the best versions of **vi** currently available are **elvis** and **vim**. Both have really good features, especially their X11 GUI versions. It is much easier to do a cut-and-paste operation, for example, using the mouse instead of "by hand."

I have Web pages on both of these versions of vi, at

(click here)

`http://heather.cs.ucdavis.edu/~matloff/vim.html`

and

(click here)

`http://heather.cs.ucdavis.edu/~matloff/elvis.html`

But in the end it is a matter of taste. I have used **vi** as the introductory editor here because it is so prevalent in the Unix world, but you may wish to give others, say emacs or some of the X11-only editors, a try.

File translated from TEX by TTH, version 3.30.
On 18 Oct 2003, 16:54.

Introduction to Plotting with nplot

Seth Stein

March 2002

nplot is a FORTRAN-callable subroutine that makes it fairly easy to produce line and point plots of curves. It can also be used for other, more complex uses with a little bit of effort. It's less capable, but easier to use for simple plots, than GMT. This program, an old department standby, has been ported to the linux machines and works, at least in simple forms.

In a program, the **nplot** call looks like this:

```
call nplot(n, x, y, xlabel, ylabel, title, para).
```

n is an integer that tells the routine how many points to plot. *x* and *y* are real arrays that contain the x and y coordinates of the points to be plotted, such that the coordinates of the *i*th point are (*x*(*i*), *y*(*i*)). *xlabel*, *ylabel*, and *title* are character strings used to label the plot. In the simplest form, these are character constants (strings enclosed by single quotes). These strings are terminated, due to a Linux "feature", by the string "//char(0)". There are also several character string manipulation routines that can be used produce more complex labels; these will be discussed later. *Para* is an array of 22 real numbers that control the size and style of the plot. The simplest way to use this is to set all the elements of *para* to zero at the beginning of a program. In this case, a line plot of a standard size will result. As discussed later, more sophisticated plots can be done by setting various elements of *para*.

Example I

Let's look at a simple example, a program to plot the function $y = ax^2 + b$

```
c program curve.f: plots y=a*(x**2)+b
      dimension x(10),y(10),para(22)
c parameters
      n=10
      dx=1.0
      a=2.
      b=3.
c initialize control array to defaults
      do 80 i=1,22
      para(i)=0.
80    continue
c generate curve
      do 85 i=1,n
      x(i)=(i-1)*dx
      y(i)=a*(x(i)**2) + b
85    continue
c plot curve
      call nplot(n,x,y,'xlabel'//char(0),'ylabel'//char(0),
     x'nplot example'//char(0),para)
      stop
      end
```

The program is simple: we set the *para* array to zero, generate the *n* points in the *x* and *y* arrays, and call **nplot.**

Using nplot

To compile the program and name it “curve”, we invoke the fortran compiler using the *nplot* option:

```
f77-s -O curve.f -lnplot -o curve
```

We can then run the program by simply typing

```
curve
```

Programs containing the **nplot** routines produce a file called *pltfil* The output plotfile, called "pltfil" can be converted to postscript using

```
bpen -v -t pltfil > pltfil.ps
```

which is aliased as "psmake". To plot this file on the laser printers, use

```
lpr pltfil.ps
```

and to preview the plot on a workstation use the command

```
xv pltfil.ps
```

Line and Point Plots

Two different plot types are available: a line plot, which connects the points in the *x* and *y* arrays, and a point plot, which puts a single symbol at each point. The plot type is controlled by *para*(13). *Para*(13) = 0.0 produces a line plot. Positive values of *para*(13) give a character plot using the character specified by the ASCII value of *para*(13) as the point symbol. (A commonly used value is 42.0, which is an asterisk.) Negative values of *para*(13) produce a plot with solid polygonal point symbols. The currently defined symbols are:

-1.0: upward-pointing equilateral triangle
-2.0: downward-pointing triangle
-3.0: rightward-pointing triangle
-4.0: leftward-pointing triangle
-5.0: square
-6.0: diamond
-7.0: hexagon

When making a character/point plot, *para*(14) controls the size of the point symbol. The default value used when *para*(14) = 0.0 is rather large; *para*(14) = 4.0 produces good-looking plots.

Plot size

It is possible to change the size of the plot produced by **nplot** by changing the value of *para*(1) and *para*(2). The first element contains the x size of the plot in inches, and the second element controls the y size. If either of these is set to 0.0, the default size of 5.5 inches is used.

X and Y Range

Normally, on the first call to **nplot**, the plot is scaled so that the array to be graphed will just fit within the plotting rectangle. To change this (*e.g.* if you want to plot a bigger curve later), you can set the x and y ranges of the plot before calling *nplot*. *Para*(3) and *para*(4) are the x- and y-coordinates of the lower-left corner of the plot, and *para*(5) and *para*(6) are the coordinates of the upper-right corner. After the first call to **nplot**, these numbers should not be changed while the same graph is being drawn.

Note that it is possible to have *para(3)* > *para(5)* or *para(4)* > *para*(6), that is, to have the values decrease from left to right or bottom to top. This could be used to plot depth increasing downward, for example.

Labeling Plots

The simplest form of plot labeling available is that produced by the nplot call itself. The *xlabel*, *ylabel*, and *title* strings are put beneath, to the left of, and above the plot, respectively. In addition, there is are two routines, **nlabel** and **plabel** that can be used to place additional label strings on a plot. They are called as follows:

```
call nlabel(xp, yp, string, para)
call plabel(xp, yp, string, para)
```

string is a character string, and *para* is the same 22 element real array that was used in an earlier **nplot** call. For **nlabel** *xp* and *yp* are the x- and y-coordinates, *in inches* from the lower-left corner of the plot, of the point where the lower-left corner of the first character of the label string will be plotted. For **plabel** *xp* and *yp* are the x- and y-coordinates, *in the units of the plot itself,* of the point where the lower-left corner of the first character of the label string will be plotted. **nplot** *must* be called at least once before **nlabel** or **plabel** are used.

There are several character-string manipulation routines provided as part of the nplot package. These routines all handle FORTRAN character strings that are declared by a statement like:

```
character*40 label
```

where, in this case, 40 is the length of the string, and *label* is the variable name. The routines are as called as follows:

```
call conzro(s, m)
call const(s, t)
call conum(s, x, p)
```

conzro initializes a string *s* of length *m*, and should be used before anything else is done to the string. *const* appends a string constant *t* to the string *s*. conum appends a decimal number, *x*, to *p* decimal places, to the string *s*. The size of characters used for labels is controlled by *para*(11).

These ideas are illustrated by the example below:

```
c program curve2.f : plots both
c y=a*(x**2)+b as a line plot and y= a*x + b as a point plot
c also demonstrates labeler
      character lab(40)
      dimension x(10),y(10),yy(10),para(22)
c parameters
      n=10
     dx=1.0
      a=2.
      b=3.
c initialize control array to defaults
      do 80 i=1,22
      para(i)=0.
80    continue
c generate line and curve
      do 85 i=1,n
      x(i)=(i-1)*dx
      y(i)=a*x(i)**2 + b
      yy(i)=a*x(i) + b
85    continue
c plot curve
      call nplot(n,x,y,'x'//char(0),'y'//char(0),
   x'nplot example'//char(0),para)
c change control array to plot points
      para(13)=42.
      para(14)=8.
c plot line
      call nplot(n,x,yy,' ',' ','',para)
c generate label
      call conzro(lab,40)
      call const(lab,' y ='//char(0))
      call conum(lab,a,2)
      call const(lab,'*x + '//char(0))
      call conum(lab,b,2)
      call const(lab,' y= '//char(0))
      call conum(lab,a,2)
      call const(lab,' x**2 + '//char(0))
      call conum(lab,b,2)
c plot label
      call nlabel(0.,-.9,lab,para)
      stop
      end
```

Fancier things

nplot can do much more, including fill regions with patterns or shading, and make color plots. For full details, use the online manual by typing

man nplot

or get a hard copy by typing

man -t nplot