

The *vi* Editor

Eric Behr, NIU Mathematical Sciences

`behr@math.niu.edu`

First draft: February 1994

Last revision: May 1995

1 Introduction

One of the oldest Unix editors, *ex*, is very powerful but quite hard to use. It is oriented towards line-by-line editing, which was the only option in the days of teletype consoles. When screen terminals became common, *vi* (which stands for ‘visual’) was created. It is simply an add-on, an interface built on top of *ex*.

The *vi* editor is relatively simple, but powerful. It is not a model of user-friendliness, but it strikes a relatively good balance between power and ease of use. Unix gurus will prefer *emacs*, while very casual users may want to look at *ce*. See the on-line manual pages for both.

We will use `<ESC>` to denote the *escape key*, usually located near the top left corner of the keyboard. The abbreviation `<RET>` will stand for hitting the Return key, `<SP>` is the spacebar, and `<BSP>`– the Backspace key. The symbol `^C` means “hold down the Control key and press C”.

Proper terminal emulation is essential for correct operation of *vi*. If after logging on to the computer you see a warning such as `type dec-vt220 unknown`, *vi* will not function the way it should. Try to think of a simpler, similar name (in this case `vt220`), and type `set term=vt220` (this may or may not work depending on how good your guess was); or ask the system manager.

2 Starting Up

Enter the command `vi` at the Unix prompt. This will start up the editor with an empty, unnamed file. The screen will be nearly blank, with empty lines marked by tilde characters (`~`) in the first column. You will have to give the new file a name when you save it.

You can specify a filename as an argument, as in `vi filename`; if the file exists, it will be opened for editing. If it doesn’t exist, a new empty file will be opened (and actually created on the disk when you next save your work). You can use wildcards here: for example, `vi a*` will tell the editor to process all files whose names begin with `a`, one by one. You can also specify pathnames, as in `vi ~/.elm/elmrc`, which will open your `elm` configuration file (if it exists).

Note: some software packages, such as `elm`, invoke *vi* automatically when an editor needs to be used. In that case, you don’t need to type the `vi` command, and you don’t have to worry about saving the work in a separate file – the invoking application will take care of it.

Typing `vi -R filename` or simply `view filename` will open the file read-only; this is useful if – say – the file is a template which should not be modified.

If the editor is invoked as `vi +22 filename`, it will open the file so that line 22 will be on top of the screen. In general, the form `vi +command ...` will start up the editor and immediately execute one of the valid ‘long commands’ described in Section 4.

Vi has two major modes. When it starts up, the editor enters the *command mode*, places the cursor at the beginning of the first line, and waits for the user to tell it what to do next. The cursor can be moved around with the arrow keys, or – if the keyboard doesn’t have them, or if the terminal emulation is less than perfect, with certain keystrokes described below.

The *insert* (or *text input*) mode is started with `i`, `a` or one of nine other similar commands (see Section 9). In the insert mode, all characters typed by the user go into the file being edited, except `<ESC>`, which ends the insert mode and resumes the command mode. If you cannot locate the escape key, try `^C`, but this should be avoided. The passionate feeling of contempt or even hate that some people have towards this editor seems

to result from the fact that there is no clear indicator that would tell the user which state *vi* is currently in – it’s up to you to keep track (Section 8 will tell you how to change that!) If you get confused, just press `<ESC>` until *vi* beeps – then you can be certain that you are in the command mode.

3 Entering Text

The most likely task to be performed after opening or creating a file is, obviously, to add some text to it. Press `i`, and start typing. The characters, inserted just before the initial position of the cursor, will show up on the screen. Depending on the various settings, *vi* may do word-wrapping (automatically inserting the ‘new line’ characters when the next word doesn’t fit), or it may not, in which case the lines will be broken in mid-word at the end of the screen, but in the file they will form one looong line. Many Unix programs don’t tolerate lines longer than 255 characters – stick to 72 or so if you can. See Section 8 if you want to make automatic word-wrap to be the default.

Practice a bit by typing some text now. Remember, you are in the insert mode: don’t use the arrows and other special keys just yet.

4 Command Mode

In the command mode, the editor is waiting for instructions such as: “move the cursor”, “delete this line”, “search for a pattern”, “save to a file”, “begin inserting characters”, or “quit”.

Two types of commands can be issued at this stage: simple keystrokes which are *not* followed by `<RET>`, and longer commands which are prefixed by a special character (the colon `:` by default). For lack of better terms we will refer to them as *short* and *long* commands. Technically speaking, the colon instructs *vi* to execute one of the commands valid in its sibling, *ex*, which shares many features with *vi*, except it is line-based rather than screen-oriented.

4.1 ‘Short’ commands

We are back in the command mode, ready to make some corrections. Most of them will require moving the cursor to the appropriate place on the screen. Use the arrow keys for that. On the Suns they are on the keypad on the right side of the keyboard. If this fails, use `<SP>` or `l` to move right, `<BSP>` or `h` to move left, `<RET>` or `j` to go to the next line, and `k` to move up.

Here are a few commands you can try now. Hitting `x` will delete the character under the cursor; `r` will replace this character with the one you type next; `u` un-does the last action *vi* took. Most commands which modify text can be repeated by typing the period. If the last such action was performed with the command `rg`, then hitting period will replace the current character with ‘g’ (even if the cursor has been moved).

Pressing `$` moves you to the end of the current line, while `0` jumps to the beginning of the line. Going to the next word is as easy as hitting `w` or `W` (those two have a slightly different notion of what a “word” is – see for yourself!), `e` or `E` move to the end of the current word, and moving between sentences is accomplished with `)` and `(`.

Pressing `dd` (the ‘d’ key twice), or `d` followed by return, erases the line where the cursor sits. If you follow `d` by a command that moves the cursor somewhere, it will delete all the characters in between; for example, `d$` erases all characters between the cursor and the end of line, and `wde` jumps to the beginning of the next word and deletes that word.

Most *vi* commands can be preceded with a number, which will repeat the command that many times. For example, `4x` will erase the character under the cursor and the next three; `2dd` erases two lines; `7r` will replace seven consecutive characters with the next keystroke you type.

The commands which delete things (e.g. `x` or `dd`) make the deleted stuff go into *vi*’s *buffer*. Until next such command overwrites the buffer, you can use it with the “put” commands `p` or `P` to paste that material after

or before the current cursor position. If the deleted material was in several lines, it will get pasted as new lines; if it consisted of characters, it will be pasted into the current line.

Experiment with all that – practice makes perfect! You won’t be happy with vi until the basic commands are automatically flowing from your head to the fingertips.

4.2 ‘Long’ commands

The second kind of commands are the ‘global’ actions, such as writing the text to a file, substituting strings, etc. Press the command character (the colon), and the cursor will jump to the bottom of the screen. Now type the write command, `w`, and press return (this writes the text to a Unix disk file). If you started vi without giving it the name of the file you are going to edit, it will complain now – you have to say something like `:w myfile` after the colon. Note that as opposed to ‘short’ commands described above, to which the editor reacts immediately after you hit a key, ‘long’ commands typed on the bottom line have to end with a return (we will neglect to mention it explicitly in what follows).

As we mentioned, if vi was invoked from within – say – `elm`, it is already operating on some temporary file; when you type `:w`, you may even catch the glimpse of something like “saved /tmp/snd3212” flashing on the screen.

Once a disk file has been associated with the editor session, you can get out of vi by typing `:q`. If the changes you made haven’t been saved, the editor will complain again. Do `:wq` (for ‘write and quit’). If you changed your mind and don’t want to save the changes, type `:q!`, which forces vi to quit no matter what.

An handy feature is the ability to leave vi momentarily to execute some Unix commands, and then get back to the editor. For example, if you want to see what files you have in the current directory before saving the document, you can use the *shell escape* command `:! ls -l`. To get into a Unix shell for a longer time (e.g. to read the last e-mail message you received) use `:shell`; the command interpreter takes over, while vi is waiting in the background until you type `exit`.

Other global commands are used to delete several lines, to move the cursor to the specified line (as in `:26<RET>`), to make a series of text substitutions, and so on. One of their features is that you can specify a range of lines to which they apply. This is done with a clause such as `3,25`, meaning lines number 3 through 25 (inclusive). You can see which line you are at by typing `:.=`; the period means “the current line” in all commands, while `$` refers to the last one. As an example, `:1,.d` deletes all lines from the beginning to the current one, while `:$=` will print the total number of lines in the file. We will talk more about this in the later sections.

4.3 Marking locations

The editor lets you mark any place in the file with single-character labels. Typing `mchar`, where *char* is a letter, assigns such a label to the current position of the cursor. These marks can then be used in moving around as follows: `’char` brings the cursor back to the precise location of the label *char*, no matter where you moved in the meantime. Similarly, `’char` will make the cursor jump to the beginning of the line which contains the mark.

This can be used to ‘cut and paste’ as follows: move to the end of the text you want to move elsewhere, and type `ma`; then move to the beginning of your selection, and type `d’a`, thus deleting the entire block. Next move the cursor where you want the text to be and hit `p`. All this sounds pretty complicated, especially if you are used to mouse-based editing, but in practice it isn’t too bad. Remember that if you make a mistake, you can always quickly hit `u` to undo it.

5 Searching

Searching for patterns is done by pressing `/` (which makes the cursor jump to the bottom of the screen, just like `:`) and then typing the search pattern followed by `<RET>`. The editor will scan the file, and if it finds

the pattern in it, the cursor will jump to that place. For example, `/continuous<RET>` should get you to the next occurrence of “continuous”.

Note that even though the search is performed in the forward direction, it ‘cycles back’, i.e. after reaching the end of the file the editor continues to search from the beginning. You can also search backwards using `?` instead of `/`. Typing `<RET>` (or `?<RET>`) will simply repeat the last search, using the same pattern. Single-letter commands `n` and `N` have the same function, except `N` *reverses* the direction of the last search.

Several characters have special, reserved meaning in this context. For instance, in search patterns the caret means “beginning of a line”, so that `/^<RET>` will move to the start of the next line, instead of finding the next caret character! Similarly, `$` means “end of line”, and the period denotes “any character”. If you want to search for such characters, you have to force vi to understand them literally, which is done by ‘escaping’ them with a backslash: `?\$<RET>` will search backwards for dollar signs in the text.

Patterns can be more complicated than simple strings – they are so-called *regular expressions*, whose structure is too complex to be described in this note. To give a couple of examples, `/^[A-Z][a-z]*$` will find the first line which starts with a capital letter, followed by zero or more lowercase letters stretching to the end of the line; `/ [a][a-z]*\.` in turn finds the next lowercase word starting with the letter a, preceded by a space and ending with a period. To learn more, see the on-line manual pages for the Unix editor ‘ed’ (type `man ed`).

6 Pattern Substitution

To substitute the first occurrence of a pattern *p1* on the current line with another string, say *p2*, use the command `:s/p1/p2/`; here *p1* is again a regular expression, as described above, and vi uses it to find the fragment of the file that is to be replaced with *p2*. The delimiter `/` is quite arbitrary; if the search and replace strings themselves contain `/`, vi would obviously be confused. You can use `+`, or `#` or some other delimiter in place of the slash, as long as it will clearly separate *p1* and *p2*. If you are lazy, the third delimiter can be skipped if no options follow it.

The substitute command can be given an ‘address’, i.e. a line or range of lines where the replacements will be made. For example, `:2 s /it's/its` will make a change in the first pattern it finds in line number 2, while `:1,. s/Bush/Clinton` will search for the first occurrence of ‘Bush’ in the lines up to and including the current one.

If it is necessary to replace *all* instances of *p1*, rather than just the first one, the “global” option can be used: `:1,$ s/Iowa/Illinois/g` does the trick.

Before doing a substitution which can alter the file in dozens of places, you may want to see the lines which will be affected; this can be done with the `ex` command `:global /p1/ print`, or simply `:g /p1/ p`. Forcing such a substitution to do exactly what you want may require a few trials and errors in choosing the search pattern.

7 Problems

One potential source of trouble with vi is bad terminal emulation, or improper timing of incoming keystrokes. When the terminal (e.g. a home computer which you use to connect with a modem) and the host executing the vi commands (e.g. a Sun workstation) do not agree on the precise meaning of signals that pass between them, or if those signals are for some reason distorted, the likely result is a garbled screen, characters appearing in incorrect places, etc.

The reason for this is that every terminal type which can be used with vi must be able to perform *screen addressing*, i.e. make the cursor jump to random places (instead of just moving it sequentially) when required, make the characters appear “black on white”, etc. This in turn is accomplished by transmitting special sequences of characters that the other side must not only understand, but also receive them within a specified period of time. For example, the right arrow key transmitted properly will move the cursor to the right; but if the line is slow and noisy, it may show up as a sequence of three unrelated characters, one

of them an ‘A’, which can turn your screen into chaos.

There are no good remedies here, other than understanding what’s happening. If your screen always looks distorted (lines beginning in the middle of the screen, or <ESC> not working) then you are either using an improper terminal emulation, or your communications software isn’t good enough. Make sure that you fix the **unknown terminal type** messages mentioned in the beginning, and that you use a decent software package to communicate with the Unix host.

If the problems are intermittent, you may be the victim of network overload (e.g. when you use telnet to a heavily loaded computer in a small Transylvanian town), or bad telephone installation in your house, an unusually bad phone connection, or a bad modem. Bear with it, using `:p<RET>` or `^L` to repaint the screen whenever necessary.

The other problem which you may encounter from time to time is that vi crashes, or the network or modem connection gets broken, when you are in the middle of editing an important document which hasn’t been saved. The obvious solution is to develop a habit of saving the changes you’ve made every now and then, so if something happens, you won’t have to re-type 25 pages.

Fortunately, most Unix implementations of vi are quite good at saving the edit session in temporary files which can usually be recovered later. To see whether vi or ex saved any backups during a crash, type `vi -L` or `vi -r`. If the response indicates some saved files, type `vi -r filename` to open the saved file. If it is indeed what you were looking for, you should immediately save it – if you don’t, and quit this vi session, the file will be gone for good.

8 Customizing the Settings

Vi gets its settings from several places. They are:

- environment variable `EXINIT`, if it is set
- the file `.exrc` in your home directory, if present, and
- the file `.exrc` in the current directory, if present

For example, if the `.login` file contains the assignment `setenv EXINIT 'set wm=8 showmode'`, vi will operate in the ‘autowrap’ mode, keeping ends of lines at least 8 characters away from the right margin (so line length will be at most 72), and will show a message at the bottom of the screen whenever you enter the text input mode.

In the directory where I work on C programs, I have a file `.exrc` with the line `set nu wrapmargin=0`; this makes vi show line numbers, which is handy while debugging programs, and disables the automatic line wrapping.

Settings can be changed temporarily within vi as well; typing the command `:set nu` will display line numbers, and `:set nonu` will make them disappear.

Here are a few parameters and settings that might be of interest:

Name and abbreviation	Effect
<code>autoindent, ai</code>	<RET> maintains indentation
<code>list</code>	display tabs and line endings
<code>number, nu</code>	show line numbers
<code>showmatch, sm</code>	when } or) is entered, show the matching left one
<code>wrapmargin, wm (= integer)</code>	set right margin for auto-wrap
<code>nowrapscan, nows</code>	prevent searches from ‘wrapping around’ the start of the document

9 Commonly Used *vi* Commands

Note: {n} stands for the optional ‘count’ parameter which may precede some of the commands.

Actions

Moving around

move cursor right
move cursor left
move cursor down
move cursor up
move cursor to column number n
move ‘home’ – to the first character on the screen
move to the middle line of the screen
move to the last line of the screen
move to the start of line
move to the end of line
move to the next/previous character c on the current line
repeat the last character search
repeat the last character search in reverse direction
move to line number n
move to the last line
move to next word (or ‘long word’)
move to previous word (or ‘long word’)
move to next/previous sentence
move to next/previous paragraph
move to the next/previous screen
mark the current position with label (letter) c
move to character marked by label c
move to line marked by label c

Commands which begin insert mode

start inserting before/after the cursor
start inserting at the start/end of line
start replacing characters under cursor (various flavors)
start replacing characters under cursor to end of word
erase current line and start inserting on it
start inserting in a new line below/above the current one

Commands which apply in insert mode

erase the last word entered
end insert mode, return to command mode

Replacing, deleting, copying and pasting

replace the current character with c
change case of the current character
delete the character under cursor
delete to end of line
delete word
delete the current line
delete lines 4 through 12
delete lines from the current one to line 43
yank (copy) the current line
place deleted or copied material after/before the cursor (or line)

Commands

{n} right arrow, <SP>, or l
{n} left arrow, <BSP>, or h
{n} down arrow, <RET>, or k
{n} up arrow or j
{n} |
H
M
L
O (zero)
\$
{n} fc, Fc
{n} ;
{n} ,
:n or nG
:\$ or G
{n} w or W
{n} b or B
{n}), (
{n} }, {
{n} ^F, ^B
mc
'c
'c

{n} i, a
{n} I, A
{n} s, R, C
{n} cw
{n} cc
{n} o, O

~W
<ESC> or ^C

{n} rc
{n} ~
{n} x
D
{n} dw or dW
{n} dd
:4,12 d
:. ,43 d
{n} yy or Y
{n} p, P

Actions

Miscellaneous commands

- undo the last operation
- undo all recent operations on the current line
- repeat the last command which modified text
- display the current line number
- display total number of lines in the file
- re-display (repaint) the screen, to clean it up
- remove the newline from the current line (join lines)

Searching and substitutions

- search for next occurrence of pattern **pat**
- search for previous occurrence of pattern
- repeat previous search
- repeat previous search, reversing its direction
- substitute pattern **pat2** for the first **pat1** in lines 1 through last
- substitute **pat2** for all occurrences of **pat1** in lines 1 through 5
- replace **pat1** with **pat2** everywhere, asking for confirmation

Files, shell and quitting

- write (save) to a file **filename**
- save to a previously named file and quit
- quit without saving
- read from a file and insert it after the current line
- read from a file and insert after line 25
- open the next file for editing (if any)
- execute a Unix command **com**
- start a new Unix shell, temporarily leaving vi

Commands

- u**
- U**
- .**
- :.=**
- :=**
- :p** or **^L**
- {n}** **J** or **:j**

- {n}** **/pat**
- {n}** **?pat**
- n**
- N**
- :1,\$ s/pat1/pat2/**
- :1,5 s/pat1/pat2/g**
- :1,\$ s/pat1/pat2/gc**

- :w filename**
- :wq** or **ZZ**
- :q!**
- :r filename**
- :25 r filename**
- :n**
- :!com**
- :shell**