Navigating Linux System Commands

A guide for beginners to the Linux Shell and GNU coreutils

Sayan Ghosh

June 11, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on June 11, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

List of Figures

List of Tables

Command Line Editors 1

1.1 Introduction

Now that we know how to go about navigating the linux based operating systems, we might want to view and edit files. This is where command line editors come in.

Definition 1.1.1 A **command line editor** is a type of text editor that operates entirely from the command line interface. They usually do not require a graphical user interface or a mouse. [a]

[a] This means that CLI editors are the only way to edit files when you are connected to a remote server. Since remote servers do not have any graphical server like X11, you cannot use graphical editors like gedit or kate.

1.1.1 Types of Editors

- **Graphical Editors:** These are editors that require a graphical user interface. Examples include *gedit* * , *kate* † , *vs code*, etc.
- **Command Line Editors:** These are editors that operate entirely from the command line interface.

We will be only discussing command line editors in this chapter.

1.1.2 Why Command Line Editors?

Command line editors are very powerful and efficient. They let you edit files without having to leave the terminal. This is usually faster than opening a graphical editor. In many cases like sshing [1] into a remote server, command line editors are the only way to edit files. Another reason for the popularity of command line editors is that they are very lightweight.

1: SSH stands for Secure Shell. It is a cryptographic network protocol for operating network services securely over an unsecured network. This will be discussed in detail in a later chapter.

1.1.3 Mouse Support

Most command line editors do not have mouse support, and others do not encourage it. But wont it be difficult to navigate without a mouse? Not really. Once you get used to the keyboard shortcuts, you will find that you can navigate way faster than with a mouse.

Mouse editors usually require the user to click on certain buttons, or to follow multi-click procedures in nested menus to perform certain tasks.

* Gedit is the default text editor in GNOME desktop environment.
† Kate is the default text editor in KDE desktop environment.

Whereas in keyboard based editors, all the actions that can be performed are mapped to some keyboard shortcuts.

Modern CLI editors usually also allow the user to totally customize the keyboard shortcuts.

This being said, most modern CLI editors do have mouse support as well if the user is running them in a terminal emulator that supports mouse over a X11 or Wayland display server. [2]

2: X11 and Wayland are display servers that are used to render graphical applications. Although not directly covered, these can be explored in details on the internet.

1.1.4 Editor war

Although there are many command line editors available, the most popular ones are *vim* and *emacs*.

Definition 1.1.2 The editor war is the rivalry between users of the Emacs and vi (now usually Vim, or more recently Neovim) text editors. The rivalry has become an enduring part of hacker culture and the free software community. [3]

3: More on this including the history and the humor can be found on the internet.

Vim

Vim is a modal editor, meaning that it has different modes for different tasks. Most editors are modeless, this makes vim a bit difficult to learn. However, once familiar with it, it is very powerful and efficient. Vim heavily relies on alphanumeric keys for navigation and editing. Vim keybindings are so popular that many other editors and even some browsers [4] have vim-like keybindings.

4: *qutebrowser* is a browser that uses vim-like keybindings. Firefox and Chromium based browsers also have extensions that provide vim-like keybindings. These allow the user to navigate the browser using vim-like keybindings and without ever touching the mouse.

Emacs

Emacs is a modeless editor, meaning that it does not have different modes for different tasks. Emacs is also very powerful and efficient. It uses multi-key combinations for navigation and editing.

1.1.5 Differences between Vim and Emacs

Keystroke execution

Emacs commands are key combinations for which modifier keys are held down while other keys are pressed; a command gets executed once completely typed.

Vim retains each permutation of typed keys (e.g. order matters). This creates a path in the decision tree which unambiguously identifies any command.

Memory usage and customizability

Emacs executes many actions on startup, many of which may execute arbitrary user code. This makes Emacs take longer to start up (even compared to vim) and require more memory. However, it is highly customizable and includes a large number of features, as it is essentially an execution environment for a Lisp program designed for text-editing.

Vi is a smaller and faster program, but with less capacity for customization. vim has evolved from vi to provide significantly more functionality and customization than vi, making it comparable to Emacs.

User environment

Emacs, while also initially designed for use on a console, had X11 GUI support added in Emacs 18, and made the default in version 19. Current Emacs GUIs include full support for proportional spacing and font-size variation. Emacs also supports embedded images and hypertext.

Vi, like emacs, was originally exclusively used inside of a text-mode console, offering no graphical user interface (GUI). Many modern vi derivatives, e.g. MacVim and gVim, include GUIs. However, support for proportionally spaced fonts remains absent. Also lacking is support for different sized fonts in the same document.

Function/navigation interface

Emacs uses metakey chords. Keys or key chords can be defined as prefix keys, which put Emacs into a mode where it waits for additional key presses that constitute a key binding. Key bindings can be mode-specific, further customizing the interaction style. Emacs provides a command line accessed by M-x that can be configured to autocomplete in various ways. Emacs also provides a defalias macro, allowing alternate names for commands.

Vi uses distinct editing modes. Under "insert mode", keys insert characters into the document. Under "normal mode" (also known as "command mode", not to be confused with "command-line mode", which allows the user to enter commands), bare keypresses execute vi commands.

Keyboard

The expansion of one of Emacs' backronyms is Escape, Meta, Alt, Control, Shift, which neatly summarizes most of the modifier keys it uses, only leaving out Super. Emacs was developed on Space-cadet keyboards that had more key modifiers than modern layouts. There are multiple emacs packages, such as spacemacs or ergoemacs that replace these key combinations with ones easier to type, or customization can be done ad hoc by the user.

Vi does not use the Alt key and seldom uses the Ctrl key. vi's keyset is mainly restricted to the alphanumeric keys, and the escape key. This is an enduring relic of its teletype heritage, but has the effect of making most of vi's functionality accessible without frequent awkward finger reaches.

Language and script support

Emacs has full support for all Unicode-compatible writing systems and allows multiple scripts to be freely intermixed.

Vi has rudimentary support for languages other than English. Modern Vim supports Unicode if used with a terminal that supports Unicode.

1.1.6 Nano: The peacemaker amidst the editor war

Nano is a simple command line editor that is easy to use. It does not have the steep learning curve of vim or emacs. But it is not as powerful as vim or emacs as well. It is a common choice for beginners who just want to append a few lines to a file or make a few changes. It is also a non-modal editor like editor which uses modifier chording like emacs. However, it mostly uses the control key for this purpose and has only simple keybindings such as *Ctrl+O* to save and *Ctrl+X* to exit.

1.2 Vim

1.2.1 History

The history of Vim is a very long and interesting one.

Table 1.1: History of Vim

1967	QED text editor by Butler Lampson and Peter Deutsch for Berkeley Timesharing System.
1967	Ken Thompson and Dennis Ritchie's QED for MIT CTSS, Multics, and GE-TSS.
1969	Ken Thompson releases ed - The Standard Text Editor.
1976	George Coulouris and Patrick Mullaney release em - The Editor for Mortals.
1976	Bill Joy and Chuck Haley build upon em to make en, which later becomes ex.
1977	Bill Joy adds visual mode to ex.
1979	Bill Joy creates a hardlink 'vi' for ex's visual mode.
1987	Tim Thompson develops a vi clone for the Atari ST named STevie (ST editor for VI enthusiasts).
1988	Bram Moolenaar makes a stevie clone for the Amiga named Vim (Vi IMitation).

Teletypes

Definition 1.2.1 A **teletype** (TTY) or a teleprinter is a device that can send and receive typed messages from a distance.

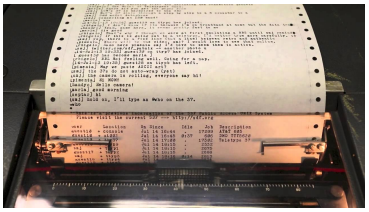

Figure 1.1: A Teletype

Very early computers used to use teletypes as the output device. These were devices that used ink and paper to actually *print* the output of the computer. These did not have an *automatic* refresh rate like modern monitors. Only when the computer sent a signal to the teletype, would the teletype print the output.

Due to these restrictions it was not economical or practical to print the entire file on the screen. Thus most editors used to print only one line at a time on the screen and did not have updating graphics.

QED

QED was a text editor developed by Butler Lampson and Peter Deutsch in 1967 for the Berkeley Timesharing System. It was a character-oriented editor that was used to create and edit text files. It used to print or edit only one character at a time on the screen. This is because the computers at that time used to use a teletype machine as the output device, and not a monitor.

Figure 1.2: Ken Thompson

Ken Thompson used this QED at Berkeley before he came to Bell Labs, and among the first things he did on arriving was to write a new version for the MIT CTSS system. Written in IBM 7090 assembly language, it differed from the Berkeley version most notably in introducing regular expressions [5] for specifying strings to seek within the document being edited, and to specify a substring for which a substitution should be made. Until that time, text editors could search for a literal string, and substitute for one, but not specify more general strings.

5: Regular expressions are a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithms for "find" or "find and replace" operations on strings. This will be discussed in detail in a later chapter.

Ken not only introduced a new idea, he found an inventive implementation: on-the-fly compiling. Ken's QED compiled machine code for each regular expression that created a NDFA (non-deterministic finite automaton) to do the search. He published this in C. ACM 11 #6, and also received a patent for the technique: US Patent #3568156.

While the Berkeley QED was character-oriented, the CTSS version was line-oriented. Ken's CTSS qed adopted from the Berkeley one the notion of multiple buffers to edit several files simultaneously and to move and copy text among them, and also the idea of executing a given buffer as editor commands, thus providing programmability.

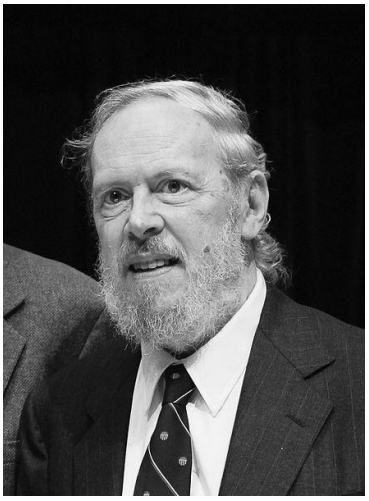

Figure 1.3: Dennis Ritchie

When developing the MULTICS project, Ken Thompson wrote yet another version of QED for that system, now in BCPL [6] and now created trees for regular expressions instead of compiling to machine language.

6: BCPL ("Basic Combined Programming Language") is a procedural, imperative, and structured programming language. Originally intended for writing compilers for other languages, BCPL is no longer in common use.

In 1967 when Dennis Ritchie joined the project, Bell Labs had slowly started to move away from Multics. While he was developing the initial stages of Unix, he rewrote QED yet again, this time for the GE-TSS system in Assembly language. This was well documented, and was originally intented to be published as a paper. [7]

ED

After their experience with multiple implementations of QED, Ken Thompson wrote **ed** in 1969.

This was now written in the newly developed B language, a predecessor to C. This implementation was much simpler than QED, and was line oriented. It stripped out much of regular expression support, and only had the support for $*$. It also got rid of multiple buffers and executing contents of buffer.

Slowly, with time, Dennis Ritchie created the C language, which is widely in use even today.

Ken Thompson re-wrote **ed** in C, and added back some of the complex features of QED, like back references in regular expressions.

Ed ended up being the **Standard Text Editor** for Unix systems.

Remark 1.2.1 Since all of Bell-Labs and AT&T's software was proprietary, the source code for ed was not available to the public. Thus, the *ed* editor accessible today in **GNU/Linux**, is another implementation of the original *ed* editor by the GNU project.

However, **ed** was not very user friendly and it was very terse. Although this was originally intented, since it would be very slow to print a lot of diagnostic messages on a teletype, slowly, as people moved to faster computers and monitors, they wanted a more user friendly editor.

VDU Terminals

Definition 1.2.2 A terminal that uses video display technology like cathode rat tubes (CRT) or liquid crystal displays (LCD) to display the terminal output is called a **VDU terminal**. (Video Display Unit)

These terminals were able to show video output, instead of just printing the output on paper. Although initially these were very expensive, and were not a household item, they were present in the research parks like Xerox PARC.

EM

George Coulouris (not the actor) was one of the people who had access to these terminals in his work at the Queen Mary College in London.

The drawbacks of **ed** were very apparent to him when using on these machines.

He found that the UNIX's **raw** mode, which was at that time totally unused, could be used to give some of the convenience and immediacy of feedback for text editing.

He claimed that although the **ed** editor was groundbreaking in its time, it was not very user friendly. He termed it as not being an editor for mortals.

He thus wrote **em** in 1976, which was an **Editor for Mortals**. [8]

Although em added a lot of features to ed, it was still a line editor, that is, you could only see one line at a time. The difference from ed was that it allowed visual editing, meaning you can see the state of the line as you are editing it.

7: At that time, systems did not have a standardized CPU architecture or a generalized low level compiler. Due to this, applications were not portable across systems. Each machine needed its own version of the application to be written from the scratch, mostly in assembly language.

The reference manual for GE-TSS QED can still be found on Dennis Ritchie's website Much of this information is taken from his blog.

Figure 1.4: Xerox Alto, one of the first VDU terminals with a GUI, released in 1973

8: George named em as Editor for Mortals because Ken Thompson visited his lab at QMC while he was developing it and said something like: "yeah, I've seen editors like that, but I don't feel a need for them, I don't want to see the state of the file when I'm editing". This made George think that Ken was not a mortal, and thus he named it Editor for Mortals.

Figure 1.5: A first generation Dectape (bottom right corner, white round tape) being used with a PDP-11 computer

Whereas most of the development of Multics and Unix was done in the United States, the development of **em** was done in the United Kingdom, in the Queen Mary College, which was the first college in the UK to have UNIX.

EN

In the summer of 1976, George Coulouris was a visiting professor at the University of California, Berkeley. With him, he had brought a copy of his **em** editor on a Dectape [9] and had installed it there on their departmental computers which were still using teletype terminals. Although em was designed for VDU terminals, it was still able to run (albeit slowly) on the teletype terminals by printing the current line every time.

9: A Dectape is a magnetic tape storage device that was used in the 1970s. It was used to store data and programs.

There he met Bill Joy, who was a PhD student at Berkeley. On showing him the editor, Bill Joy was very impressed and wanted to use it on the PDP-11 computers at Berkeley. The system support team at Berkley were using PDP-11 which used VDU Terminals, an environment where em would really shine.

Figure 1.6: George Coulouris

He explained that 'em' was an extension of 'ed' that gave key-stroke level interaction for editing within a single line, displaying the up-to-date line on the screen (a sort of single-line screen editor). This was achieved by setting the terminal mode to 'raw' so that single characters could be read as they were typed - an eccentric thing for a program to do in 1976.

Although the system support team at Berkeley were impressed by this editor, they knew that if this was made available to the general public, it would take up too much resources by going to the raw mode on every keypress. But Bill and the team took a copy of the source code just to see if they might use it.

George then took a vacation for a few weeks, but when he returned, he found that Bill had taken his ex as a starting point and had added a lot of features to it. He called it **en** initially, which later became **ex**.

EX

Bill Joy took inspiration from several other ed clones as well, and their own tweaks to ed, although the primary inspiration was **em**. Bill and Chuck Haley built upon em to make en, which later became ex.

Figure 1.7: Bill Joy

This editor had a lot of improvements over em, such as adding the ability to add abbreviations (using the `ab` command), and adding keybindings (maps).

It also added the ability to mark some line using the `k` key followed by any letter, and then jump to that line from any arbitrary line using the `'` key followed by the letter.

Slowly, with time, the modern systems were able able to handle the raw mode, and real time editing more and more. This led to the natural progression, What if we could see the entire file at once, and not just one line at a time?

VI

Bill added the **visual mode** to ex in 1977. [10] This was not a separate editor, but rather just another mode of the **ex** editor. You could open ex in visual mode using the `-v` flag to ex.

10: This visual mode is not the same as the visual mode in vim.

```
$ ex -v filename
```

This visual mode was the first time a text editor was **modal**. This means that the editor had different modes for different tasks. When you want to edit text, you would go to the insert mode, and type the text. When you want to navigate, you would go to the normal mode, and use the navigation keys and other motions defined in vi.

Slowly, as the visual mode became more and more popular, Bill added a hardlink to ex called **vi**. [11]

The modal version of vi was also inspired from another editor called **bravo**, which was developed at Xerox PARC. [12]

If you use vi/vim, you may notice that the key to exit the insert mode is `Esc`. This may seem inconveniently placed at the top left corner, but this was because the original vi was developed on a ADM-3A terminal, which had the `Esc` key to the left of the **Q** key, where modern keyboards have the `Tab` key. [13]

Also, the choice of h,j,k,l for navigation was because the ADM-3A terminal did not have arrow keys, rather, it had h,j,k,l keys for navigation.

This can be seen in Figure 1.8.

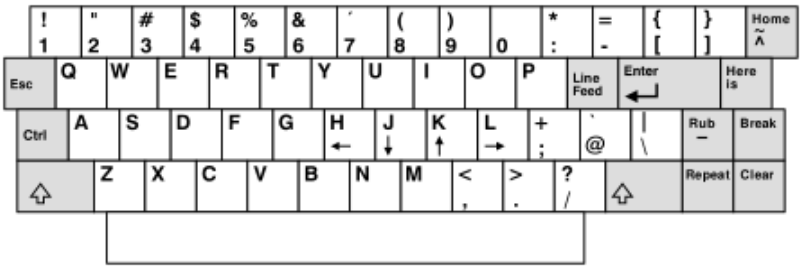

Figure 1.8: The Keyboard layout of the ADM-3A terminal

Bill Joy was also one of the people working on the **Berkeley Software Distribution (BSD) of Unix**. Thus he bundled vi with the first BSD distribution of UNIX released in 1978. The pre-installed nature of vi in the BSD Distribution made it very popular.

However, since both the source code of ed was restricted by Bell Labs - AT&T, and the source code of vi was restricted by the University of California, Berkeley, they could not be modified by the users or distributed freely.

This gave birth to a lot of clones of vi.

Vi Clones

The vi clones were written because the source code for the original version was not freely available until recently. This made it impossible to extend the functionality of vi. It also precluded porting vi to other operating systems, including Linux.

- **calvin**: a freeware "partial clone of vi" for use on MS-DOS. It has the advantages of small size (the .exe file is only 46.1KB!) and fast execution but the disadvantage that it lacks many of the ex commands, such as search and replace.

Figure 1.9: Stevie Editor

11: This means that it did not take up additional space on the disk, but was just another entry in the directory entry that pointed to the same inode which stored the ex binary path. Upon execution, **ex** would detect if it was called as **vi** and would start in visual mode by default. We have covered hardlinks in Chapter **??**.

12: Xerox PARC has always been ahead of its time. The first graphical user interface was developed at Xerox PARC. The bravo editor used bitmapped graphics to display the text, and had extensive mouse support. The overdependence on the mouse in such an early time was one of the reasons that the bravo editor was not as popular as vi.

13: Since the placement of the Escape key is inconvenient in modern keyboard layouts, many people remap the Escape key to the `Caps Lock` key either in vim or in the operating system itself.

- **lemmy**: a shareware version of vi implemented for the Microsoft Windows platforms which combines the interface of vi with the look and feel of a Windows application.
- **nvi**: It is a re-implementation of the classic Berkeley vi editor, derived from the original 4.4BSD version of vi. It is the "official" Berkeley clone of vi, and it is included in FreeBSD and the other BSD variants.
- **stevie**: 'ST Editor for VI Enthusiasts' was developed by Tim Thompson for the Atari ST. It is a clone of vi that runs on the Atari ST. Tim Thompson wrote the code from scratch (not based on vi) and posted its source code as a free software to comp.sys.atari.st on June 1987. Later it was ported to UNIX, OS/2, and Amiga. Because of this independence from vi and ed's closed source license, most vi clones would base their work off of stevie to keep it free and open source.
- **elvis**: Elvis creator, Steve Kirkendall, started thinking of writing his own editor after Stevie crashed on him, causing him to lose hours of work and damaging his confidence in the editor. Stevie stored the edit buffer in RAM, which Kirkendall believed to be impractical on the MINIX operating system. One of Kirkendall's main motivation for writing his own vi clone was that his new editor stored the edit buffer in a file instead of storing it in RAM. Therefore, even if his editor crashed, the edited text could still be retrieved from that external file. Elvis was one of the first vi clones to offer support for GUI and syntax highlighting.

The clones add numerous new features which make them significantly easier to use than the original vi, especially for neophytes. A particularly useful feature in many of them is the ability to edit files in multiple windows. This facilitates working on more than one file at the same time, including cutting and pasting text among them.

Many of the clones also offer GUI versions of vi that operate under the X Windows system and can take advantage of bit-mapped (high resolution) displays and the mouse.

Vim

Bram Moolenaar, a Dutch programmer, was impressed by STeVIe, a vi clone for the Atari ST. But he was working with the Commodore Amiga at that time, and there was no vi clone for the Amiga. So Bram began working on the stevie clone for the AmigaOS in 1988.

Figure 1.10: Bram Moolenaar

He released the first public release (v 1.14) in 1991 as visible in Figure 1.11.

Since Vim was based off of Stevie, and not ed or vi so it could be freely distributed. It was licensed under a charityware license, named as Vim License. The license stated that if you liked the software, you should consider making a donation to a charity of your choice.

Moolenaar was an advocate of a NGO based in Kibaale, Uganda, which he founded to support children whose parents have died of AIDS. In 1994, he volunteered as a water and sanitation engineer for the Kibaale Children's Centre and made several return trips over the following twenty-five years.

Later Vim was re-branded as 'Vi IMproved' as seen in Figure 1.12.

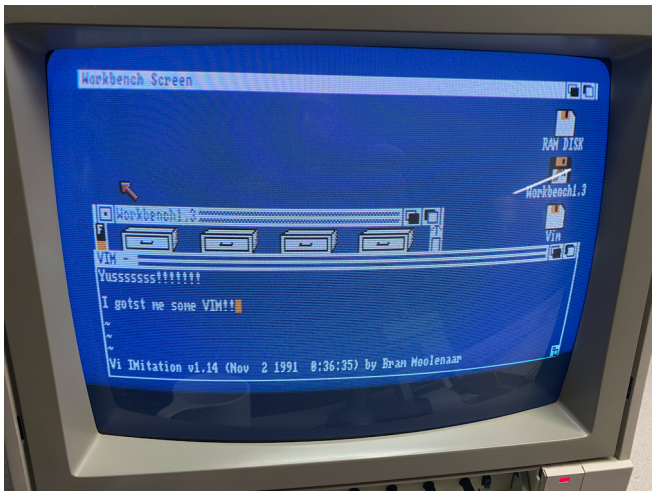

Figure 1.11: The inital version of Vim, when it was called Vi IMitation

Figure 1.12: Vim 9.0 Start screen

Vim has been in development for over 30 years now, and is still actively maintained. It has added a lot of features over the years, such as syntax highlighting, plugins, completion, PCRE support, mouse support, etc.

neovim

Recently there have been efforts to modernize the vim codebase. Since it is more than 30 years old, it has a lot of legacy code. The scripting language of vim is also not a standard programming language, but rather a custom language called vimscript.

To counter this, a new project called **neovim** has been started. It uses **lua** as the scripting language, and has a lot of modern features like out of the box support for LSP, [14] better mouse integration, etc.

In this course, we will be learning only about basic vi commands and we will be using vim as the editor.

14: LSP stands for Language Server Protocol. It is a protocol that allows the editor to communicate with a language server to provide features like autocompletion, go to definition, etc. This makes vim more like an IDE.

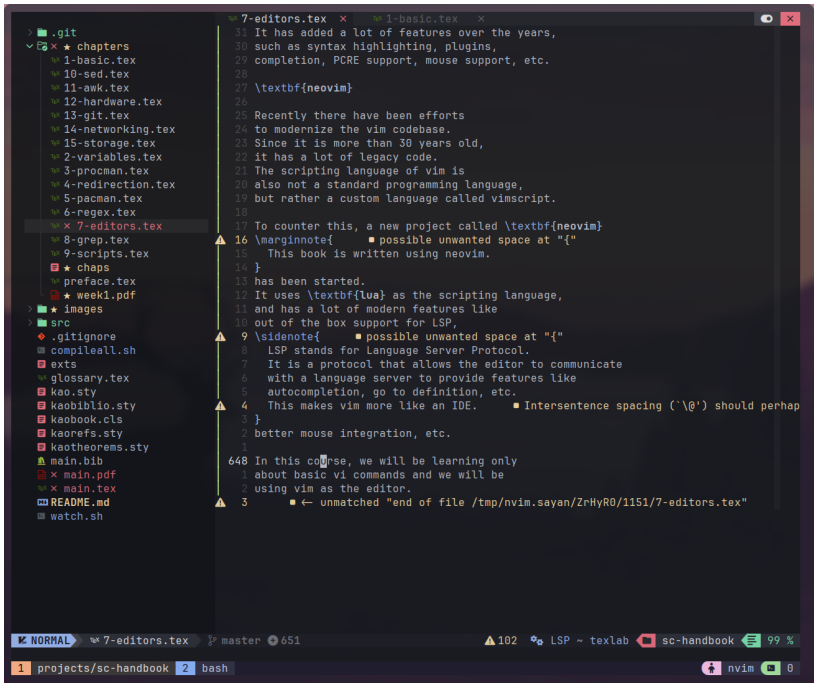

Figure 1.13: Neo Vim Window editing this book

This book is written using neovim.

1.2.2 Ed Commands

Before we move on to Vi Commands, let us first learn about the basic ed commands. These will also be useful in vim, since the ex mode of vim is based on ed/ex where we can directly use ed commands on our file in the buffer.

Table 1.2: Ed Commands

Description	Commands
Show the **P**rompt	`P`
Command Format	`[addr[,addr]]cmd[params]`
Commands for location	`1 . $ % + - , ; /RE/`
Commands for editing	`f p a c d i j s m u`
Execute a shell *command*	`!command`
edit a file	`e filename`
read file contents into buffer	`r filename`
read *command* output into buffer	`r !command`
write buffer to filename	`w filename`
quit	`q`

Commands for location

Table 1.3: Commands for location

Commands	Description
a number like `2`	refers to second line of file
`.`	refers to current line
`$`	refers to last line
`%`	refers to all the lines
`+`	line after the cursor (current line)
`-`	line before the cursor (current line)
`,`	refers to buffer holding the file or last line in buffer
`;`	refers to current position to end of the file
`/RE/`	refers line matched by pattern specified by 'RE'

Commands for Editing

Table 1.4: Commands for Editing

Commands	Description
`f`	show name of **f**ile being edited
`p`	**p**rint the current line
`a`	**a**ppend at the current line
`c`	**c**hange the current line
`d`	**d**elete the current line
`i`	**i**nsert line at the current position
`j`	**j**oin lines
`s`	**s**earch for regex pattern
`m`	**m**ove current line to position
`u`	**u**ndo latest change

Let us try out some of these commands in the ed editor.

Lets start with creating a file, which we will then open in the ed editor.

```
$ echo "line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file" > test.txt
```

This creates a file in the current working directory with the name `test.txt` and the contents as given above.

We invoke ed by using the executable `ed` and providing the filename as an argument.

```
$ ed test.txt
117
```

As soon as we run it, you will see a number, which is the number of characters in the file. The terminal may seem hung, since there is no prompt, of either the bash shell, or of the ed editor. This is because ed is a line editor, and it does not print the contents of the file on the screen.

Off the bat, we can observe the terseness of the ed editor since it does not even print a prompt. To turn it on, we can use the `P` command. The default prompt is `*`.

```
ed test.txt
117
P
*
```

Now we can see the prompt `*` is always present, whenever the ed editor expects a command from the user.

Lets go to the first line of the file using the `1` command. We can also go to the last line of the file using the `$` command.

```
*1
line-1 hello world
*$
line-4 end of file
*
```

To print out all the lines of the file, we can use the `,` or `%` with `p` command.

```
*,p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

```
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

However, if we use the `,` command without the `p` command, it will not print all the lines. Rather, it will just move the cursor to the last line and print the last line.

```
*,
line-4 end of file
```

We can also print any arbitrary line range using the line numbers separated by a comma and followed by the `p` command.

```
*2,3p
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
```

One of the pioneering features of ed was the ability to search for a pattern in the file. Let us quickly explain the syntax of the search command. [15]

15: The details of regular expressions will be covered in a later chapter.

```
*/hello/
line-1 hello world
```

We may or may not include the `p` command after the last `/` in the search command.

We can advance to the next line using the `+` command.

```
*p
line-1 hello world
*+
line-2 welcome to line editor
```

And go to the previous line using the `-` command.

```
*p
line-2 welcome to line editor
*-
line-1 hello world
```

We can also print all the lines from the current line to the end of the file using the `;p` command.

```
*.
line-2 welcome to line editor
*;p
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

We can also run arbitrary shell commands using the `!` command.

```
*! date
Mon Jun 10 11:36:34 PM IST 2024
!
```

The output of the command is shown to the screen, however, it is not saved in the buffer.

To read the output of a command into the buffer, we can use the `r` command.

```
*r !date
32
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
Mon Jun 10 11:37:42 PM IST 2024
```

The output after running the `r !date` command is the number of characters read into the buffer. We can then print the entire buffer using the `%p` command.

The read data is appended to the end of the file.

We can write the buffer [16] to the disk using the `w` command.

16: Remember that the buffer is the in-memory copy of the file and any changes made to the buffer are not saved to the file until we write the buffer to the file.

```
*w
149
```

The output of the `w` command is the number of characters written to the file.

To exit ed, we can use the `q` command.

```
*q
```

To delete a line, we can use the `d` command. Lets say we do not want the date output in the file. We can re-open the file in ed and remove the last line.

```
$ ed test.txt
149
P
*$
Mon Jun 10 11:38:49 PM IST 2024
*d
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
*wq
117
```

We can add lines to the file using the `a` command. This appends the line after the current line. On entering this mode, the editor will keep on taking input for as many lines as we want to add. To end the input, we can use the `.` command on a new line.

```
$ ed test.txt
117
P
*3
line-3 ed is perhaps the oldest editor out there
*a
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
.
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
line-4 end of file
*
```

We can also utilize the regular expression support in ed to perform search and replace operations. This lets us either search for a fixed string and replace with another fixed string, or search for a pattern and replace it with a fixed string.

Let us change hello world to hello universe.

```
*1
line-1 hello world
*s/world/universe/
line-1 hello universe
```

```
*%p
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
line-4 end of file
*
```

We can print the name of the currently opened file using the `f` command.

```
*f
test.txt
```

If we wish to join two lines, we can use the `j` command. Let us join lines 4,5, and 6.

```
*4
perhaps not, since we know it was inspired from QED
*5
which was made multiple times by thompson and ritchie
*6
before ed was made.
*4,5j
*4
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchie
*5
before ed was made.
*4,5j
*4
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
*
```

Here we can see that we do the joining in two steps, first the lines 4 and 5 are joined, and then the newly modified line 4 and 5 are joined.

We can move a line from its current position to another line using the `m` command.

Lets insert a line-0 at the end of the file and then move it to the beginning of the file.

```
*7
line-4 end of file
*a
line-0 in the beginning, there was light
.
*8
line-0 in the beginning, there was light
*m0
*1,4p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
*
```

We can also undo the last change using the `u` command. [17]

17: The undo command in ed is not as powerful as the undo command in vim. In vim, we can undo multiple changes using the `u` command. In ed, we can only undo the last change. If we run the `u` command multiple times, it will undo the last change of undoing the last change, basically redoing the last change.

```
*1
line-0 in the beginning, there was light
*s/light/darkness
line-0 in the beginning, there was darkness
*u
*.
line-0 in the beginning, there was light
*
```

If search and replace is not exactly what we want, and we want to totally change the line, we can use the `c` command. It will let us type a new line, which will replace the current line.

```
*%p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
line-4 end of file
*4
line-3 ed is perhaps the oldest editor out there
*c
line-4 ed is the standard editor for UNIX
.
*4
line-4 ed is the standard editor for UNIX
*
```

Just like the `a` command, we can also use the `i` command to insert a line at the current position. This will move the current line to the next line.

```
*6
line-4 end of file
*i
before end of file
.
*6,$p
before end of file
line-4 end of file
*
```

Finally, we can also number the lines using the `n` command.

```
*%p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-4 ed is the standard editor for UNIX
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
before end of file
line-4 end of file
*%n
1       line-0 in the beginning, there was light
2       line-1 hello universe
3       line-2 welcome to line editor
```

```
4       line-4 ed is the standard editor for UNIX
5       perhaps not, since we know it was inspired from QEDwhich
    was made multiple times by thompson and ritchiebefore ed was
    made.
6       before end of file
7       line-4 end of file
*
```

1.2.3 Exploring Vim

There are a plethora of commands in vim. We wont be able to cover all of them in this course. Only the basic commands required to get started with using vim as your primary editor would be covered. A detailed tutorial on vim can be found by running the command `vimtutor` in your terminal.

```
$ vimtutor
```

This opens a temporary files that goes through a lot of sections of vim, explaining the commands in detail. This opens the text file in vim itself, so you can actually try out each exercise as and when you read it. Many exercises are present in this file to help you remember and master commands. Feel free to modify the file since it is a temporary file and any changes made is lost if the command is re-run.

To open a file in vim, we provide the filename as an argument to the vim executable.

```
$ vim test.txt
```

Sometimes the normal mode is called command mode or escape mode, since we can run commands in this mode and we press the `Esc` key to go to this mode. However, the ex mode is also called command mode, since we can run ex commands in this mode. To avoid confusions, we will refer to the navigational(default) mode as normal mode, since vim internally also refers to it as normal mode, and we will refer to the ex mode as ex mode.

Modal Editor

Vim is a modal editor, which means that it has different modes that it operates in. The primary modes are:

- Normal/Command Mode - The default mode where we can navigate around the file, and run vim commands.
- Insert Mode - The mode where we can type text into the file.
- Visual Mode - The mode where we can select text to copy, cut, or delete.
- Ex Mode - The mode where we can run ex commands.

Pressing the `Esc` key takes you to the normal mode from any other mode.

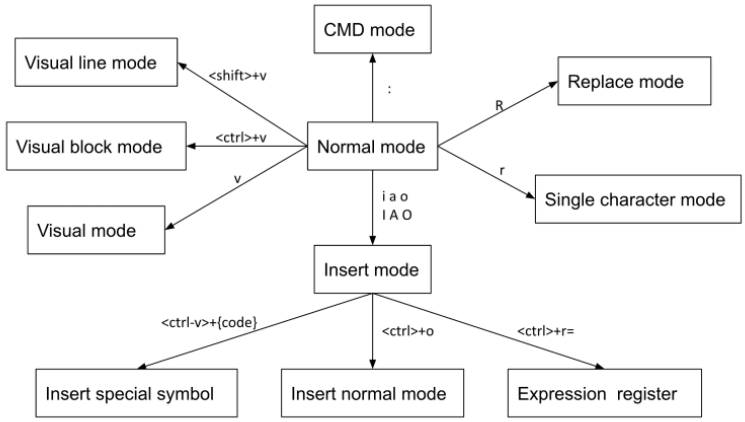

Figure 1.14: Simplified Modes in Vim

18: This is a simplified version of the modes in vim. There are other interim modes and other keystrokes that toggle the modes. This is shown in detail in Figure 1.15.

The figure Figure 1.14 demonstrates how to switch between the different modes in vim. [18]

Commands in Ex mode

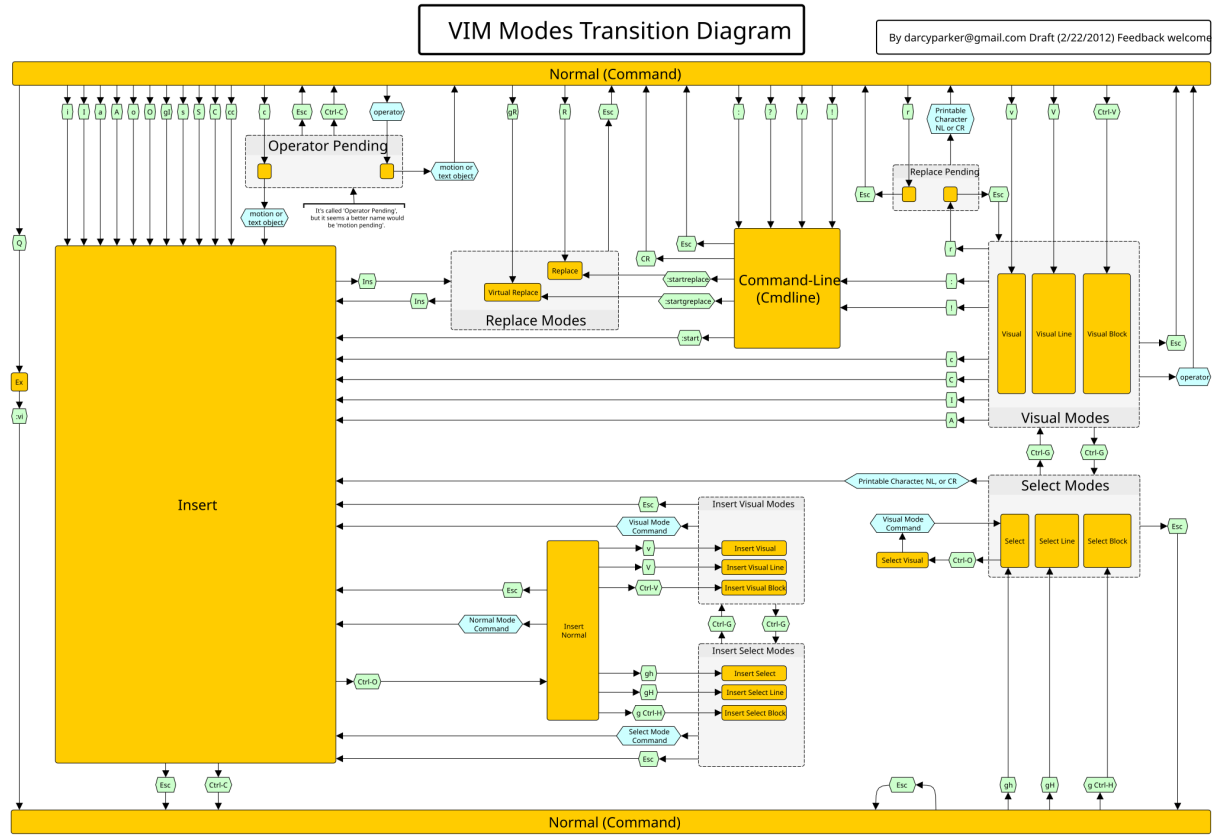

Figure 1.15: Detailed Modes in Vim

Since we are already familar with commands in **ed**, most of the commands are same/similar in **ex** mode of vim.

Table 1.5: Ex Commands in Vim

Key	Description
`:f`	show name of file
`:p`	print current line
`:a`	append at current line
`:c`	change current line
`:d`	delete current line
`:i`	insert line at current position
`:j`	join lines
`:s`	search and replace regex pattern in current line
`:m`	move current line to position
`:u`	undo latest change
`:w [filename]`	write buffer to filename
`:q`	quit if no change
`:wq`	write buffer to filename and quit
`:x`	write buffer to filename and quit
`:q!`	quit without saving
`:r filename`	read file contents into buffer
`:r !command`	read command output into buffer
`:e filename`	edit a file
`:sp [filename]`	split the screen and open another file
`:vsp [filename]`	vertical split the screen and open another file

There are many more commands in the ex mode of vim. Along with implementing the original ed commands, it also has a lot of additional

commands to make it more integrated with the vim editor, such as the ability to open split windows, new tabs, buffers, and perform normal mode commands in ex mode.

Basic Navigation

The basic keys for moving around in a text file in vim are the `h,j,k,l` keys. They move the cursor one character to the left, down, up, and right respectively. These keys are chosen because they are present on the home row of the keyboard, and do not require the user to move their hands from the home row to navigate. [19]

19: These were historically chosen because the ADM-3A terminal had these keys for navigation as seen in Figure 1.8.

Along with these, we have keys to navigate word by word, or to move to the next pattern match, or the next paragraph, spelling error, etc.

Table 1.6: Navigation Commands in Vim

Key	Description
`h`	move cursor left
`j`	move cursor down
`k`	move cursor up
`l`	move cursor right
`w`	move to the beginning of the next word
`e`	move to the end of the current word
`b`	move to the beginning of the previous word
`%`	move to the matching parenthesis, bracket, or brace
`0`	move to the beginning of the current line
`$`	move to the end of the current line
`/`	search forward for a pattern
`?`	search backward for a pattern
`n`	repeat the last search in the same direction
`N`	repeat the last search in the opposite direction
`gg`	move to the first line of the file
`G`	move to the last line of the file
`1G`	move to the first line of the file
`1gg`	move to the first line of the file
`:1`	move to the first line of the file
`{`	move to the beginning of the current paragraph
`}`	move to the end of the current paragraph
`fg`	move cursor to next occurrence of 'g' in the line
`Fg`	move cursor to previous occurrence of 'g' in the line

Wait you forgot your cursor behind!

All of the above commands move the cursor to the mentioned location. However, if you want to move the entire screen, and keep the cursor at the its current position, you can use the `z` command along with `t` to top the line, `b` to bottom the line, and `z` to center the line.

There are other commands using the `Ctrl` key that moves the screen, and not the cursor.

Table 1.7: Moving the Screen Commands in Vim

Key	Description
`Ctrl+F`	move forward one full screen
`Ctrl+B`	move backward one full screen
`Ctrl+D`	move forward half a screen
`Ctrl+U`	move backward half a screen
`Ctrl+E`	move screen up one line
`Ctrl+Y`	move screen down one line

Replacing Text

Usually in other text editors, if you have a word, phrase, or line which you want to replace with another, you would either press the backspace or delete key to remove the text, and then type the new text. However, in vim, there is a more efficient way to replace text.

Key	Description
`r`	replace the character under the cursor
`R`	replace the character from the cursor till escape is pressed
`cw`	change the word under the cursor
`c4w`	change the next 4 words
`C`	delete from cursor till end of line and enter insert mode
`cc`	delete entire line and enter insert mode
`5cc`	delete next 5 lines and enter insert mode
`S`	delete entire line and enter insert mode
`s`	delete character under cursor and enter insert mode

Table 1.8: Replacing Text Commands in Vim

Toggling Case

You can toggle the case of a character, word, line, or any arbitrary chunk of the file using the or the `g` command.

Key	Description
`~`	toggle the case of the character under the cursor
`g w`	toggle the case of the word under the cursor
`g 0`	toggle the case from cursor till beginning of line
`g $`	toggle the case from cursor till end of line
`g {`	toggle the case from cursor till previous empty line
`g {`	toggle the case from cursor till next empty line
`g %`	toggle the case from the bracket, brace, or parenthesis till its pair

Table 1.9: Toggling Case Commands in Vim

You might start to see a pattern emerging here. Many commands in vim do a particular command, and on which text it operates is determined by the character followed by it. Such as `c` to change, `d` to delete, `y` to yank, etc. The text on which the command operates is mentioned using `w` for the word, `0` for till beginning of line, etc.

This is not a coincidence, but rather a design of vim to make it more efficient to use. The first command is called the operator command, and the second command is called the motion command.

Vim follows a `operator-count-motion` pattern. For example: `d2w` deletes the next 2 words. This makes it very easy to learn and remember commands, since you are literally typing out what you want to do.

Deleting or Cutting Text

In Vim, the delete command is used to cut text from the file.

Motion - till, in, around

By now you should notice that `dw` doesn't always delete the word under the cursor. Technically `dw` means delete till the beginning of the next word. So if you press `dw` at the beginning of a word, it will delete the word under the cursor. But if your cursor is in the middle of a word and you type `dw`, it will only delete the part of the word till the beginning of the next word from the cursor position.

Table 1.10: Deleting Text Commands in Vim

Key	Description
`x`	delete the character under the cursor
`X`	delete the character before the cursor
`5x`	delete the next 5 characters
`dw`	delete the word under the cursor
`d4w`	delete the next 4 words
`D`	delete from cursor till end of line
`dd`	delete entire line
`6dd`	delete next 6 lines

To delete the entire word under the cursor, regardless of where the cursor is in the word, you can use the `diw` this means **d**elete **i**nside **w**ord.

However, now you may notice that `diw` doesn't delete the space after the word. This results in two consequtive spaces, one from the end of the word, and one from the beginning of the word being left behind. To delete the space as well, you can use `daw` which means **d**elete **a**round **w**ord.

This works not just with `w` but with any other motion such as **d**elete **i**nside **p**aragraph, which will delete the entire paragraph under the cursor, resulting in two empty lines being left behind, and **d**elete **a**round **p**aragraph, which will delete the entire paragraph under the cursor, and only one empty line being left behind.

Try out the same with deleting inside other items, such as brackets, parenthesis, braces, quotes, etc. The syntax remains the same, `di{`, `di[`, `di(`, `di"`, `di'`, etc.

Yanking and Pasting Text

Yes, copying is called yanking in vim. The command to yank is `y` and to paste is `p`. You can combine `y` with all the motions and **in** and **around** motions as earlier. You can also add the count to yank multiple lines or words.

Table 1.11: Deleting Text Commands in Vim

Key	Description
`yy`	yank the entire line
`yw`	yank the word under the cursor
⋮	⋮
`p`	paste the yanked text after the cursor
`P`	paste the yanked text before the cursor

Remark 1.2.2 Important to note that the commands

```
yy
```

and

```
0y$
```

are not the same. The first command yanks the entire line, including the newline character at the end of the line. The second one yanks the entire line, but does not include the newline character at the end of the line. Thus if you directly press `p` after the first command, it will paste the line below the current line, and if you press `p` after the

second command, it will paste the line at the end of the current line.

Undo and Redo

The undo command in vim is `u` and the redo command is `Ctrl+R`. You can undo multiple changes, unlike ed.

Remark 1.2.3 If you want to use vim as your primary editor, it is highly recommended to install the **mbbill/undotree** plugin. This plugin will show you a tree of all the changes you have made in the current buffer, and you can go to any point in the tree and undo or redo changes. This becomes very useful if you undo too many changes and by mistake make a new change, this changes your branch in undo tree, and you cannot redo the changes you undid. With the undotree plugin, you can switch branches of the undo tree and redo the changes.

Searching and Replacing

The search command in vim is `/` for forward search and `?` for backward search. You can use the `n` command to repeat the last search in the same direction, and the `N` command to repeat the last search in the opposite direction. For example, if you perform forward search then using the `n` command will search for the next occurrence of the pattern in the forward direction, and using the `N` command will search for the previous occurrence of the pattern in the backward direction. However if you perform a backward search using the `?` command, then using the `n` command will search for the previous occurrence of the pattern in the backward direction, and using the `N` command will search for the next occurrence of the pattern in the forward direction.

You can also use the $*$ command to search for the word under the cursor, and the `#` command to search for the previous occurrence of the word under the cursor.

You can perform search and replace using the `:s` command. The command takes a line address on which to perform the search and replace. Usually you can use the `%` address to search in the entire file, or the `.,$` address to search from cursor till the end of the file.

You can also use any line number to specify the address range, similar to the ed editor.

```
:[addr]s/pattern/replace/[flags]
```

The flags at the end of the search and replace command can be **g** to replace all occurrences in the line, and **c** to confirm each replacement.

The address can be a single line number, a range of line numbers, or a pattern to search for. The pattern can be a simple string, or a regular expression.

Some examples of addresses are shown in Table 1.12.

Insert Mode

You can enter insert mode from escape mode using the keys listed in Table 1.13. In insert mode, if you want to insert any non-graphical character, you can do that by pressing `Ctrl+V` followed by the key combination for

Table 1.12: Address Types in Search and Replace

Key	Description
`m,n`	from line m to line n
`m`	line m
`m,$`	from line m to end of file
`.,$`	from current line to end of file
`1,n`	from line 1 to line n
`/regex/,n`	from line containing regex to line n
`m,/regex/`	from line m to line containing regex
`.,/regex/`	from current line to line containing regex
`/regex/,.`	from line containing regex to current line
`1,/regex/`	from the first line to line containing regex
`/regex/,$`	from line containing regex to the last line
`/regex1/;/regex2/`	from line containing regex1 to line containing regex2
`%`	entire file

Table 1.13: Keys to enter Insert Mode

Key	Description
`i`	enter insert mode before the cursor
`a`	enter insert mode after the cursor
`I`	enter insert mode at the beginning of the line
`A`	enter insert mode at the end of the line
`o`	add new line below the current line and enter insert mode
`O`	add new line above the current line and enter insert mode

the character. For example, to insert a newline character, you can press `Ctrl+V` followed by `Enter`.

These are just the basic commands to get you started with vim. You can refer to vim cheat sheets present online to get more familiar with the commands.

- `https://vim.rtorr.com/` is a good text based HTML cheat sheet for vim.
- `https://vimcheatsheet.com/` is a paid graphical cheat sheet for vim. [20]

20: A free version of the graphical cheat sheet is shown in Figure 1.16.

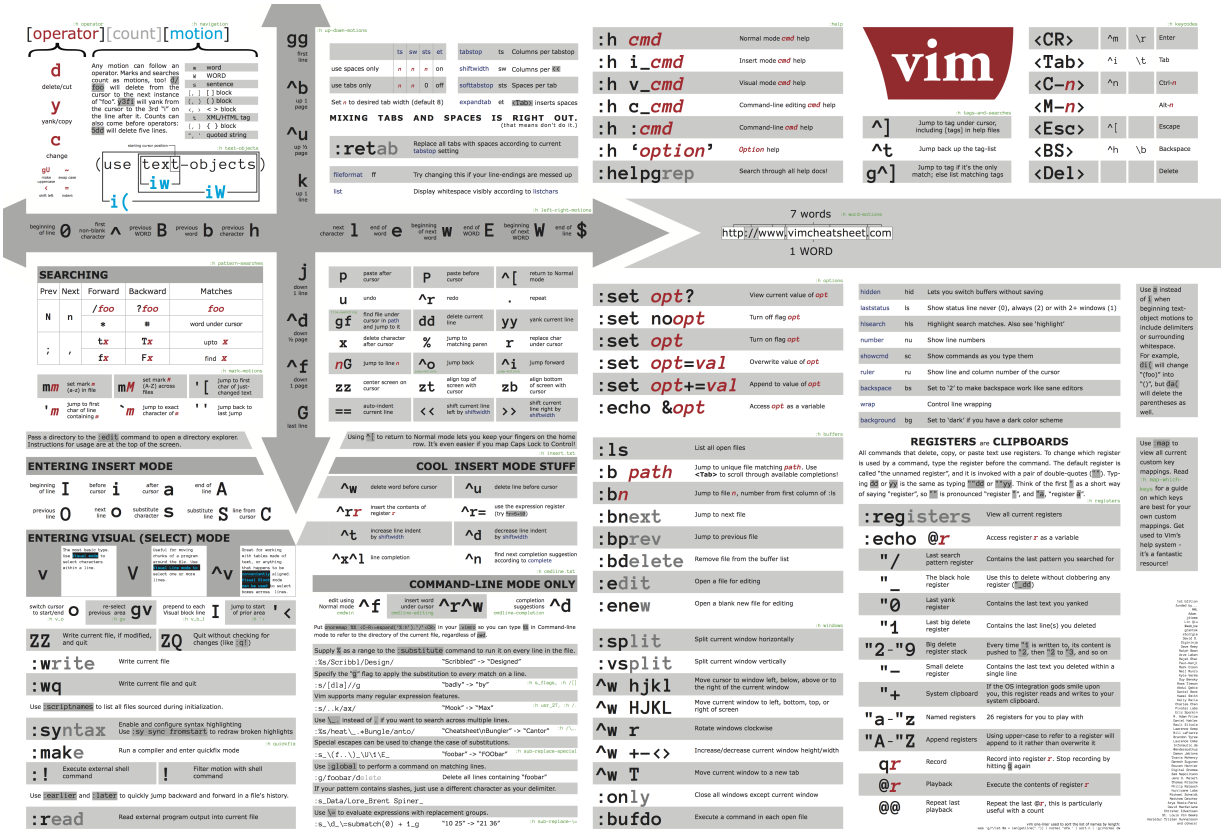

Figure 1.16: Vim Cheat Sheet

1.3 Emacs

1.3.1 History

Emacs was mostly developed by Richard Stallman and Guy Steele.

Table 1.14: History of Emacs

1962	TECO (Tape Editor and Corrector) was developed at MIT.
1976	Richard Stallman visits Stanford AI Lab and sees Fred Wright's E editor.
1978	Guy Steele accumulates a collection of TECO macros into EMACS.
1979	EMACS becomes MIT's standard text editor.
1981	James Gosling writes Gosling Emacs that runs on UNIX.
1984	Richard Stallman starts GNU Emacs - a free software alternative to Gosling Emacs.

Figure 1.17: Emacs Logo

Figure 1.18: Richard Stallman - founder of GNU and FSF projects

TECO

TECO was developed at MIT in 1962. It was a text editor used to correct the output of the PDP-1 computer. It is short for Tape Editor and Corrector. Unlike most modern text editors, TECO used separate modes in which the user would either add text, edit existing text, or display the document One could not place characters directly into a document by typing them into TECO, but would instead enter a character ('i') in the TECO command language telling it to switch to input mode, enter the required characters, during which time the edited text was not displayed on the screen, and finally enter a character (<esc>) to switch the editor back to command mode. This is very similar to how vi works.

Stallman's Visit to Stanford

In 1976, Richard Stallman visited the Stanford AI Lab where he saw Fred Wright's E editor. He was impressed by E's WYSIWYG [21] interface where you do not need to tackle multiple modes to edit a text file. This is the default behaviour of most modern editors now. He then returned to MIT where he found that Carl Mikkelsen had added to TECO a combined display/editing mode called Control-R that allowed the screen display to be updated each time the user entered a keystroke. Stallman reimplemented this mode to run efficiently and added a macro feature to the TECO display-editing mode that allowed the user to redefine any keystroke to run a TECO program.

Initially TECO was able to only edit the file sequentially, page by page. This was due to earlier memory restrictions of the PDP-1. Stallman

21: What You See Is What You Get

modified TECO to read the entire file into the buffer, and then edit the buffer in memory allowing for random access to the file.

Too Many Macros!

The new version of TECO quickly became popular at the AI Lab and soon accumulated a large collection of custom macros whose names often ended in MAC or MACS, which stood for macro. This quickly got out of hand as there were many divergent macros, and a user would be totally lost when using a co-worker's terminal.

In 1979, Guy Steele combined many of the popular macros into a single file, which he called EMACS, which stood for Editing MACroS, or E with MACroS.

To prevent thousands of forks of EMACS, Stallman declared that 'EMACS was distributed on a basis of communal sharing, which means all improvements must be given back to me to be incorporated and distributed.'

Till now, the EMACS, like TECO, ran on the PDP-10 which ran the ITS operating system and not UNIX.

Figure 1.19: Guy L. Steele Jr. combined many divergent TECO with macros to create EMACS

EINE ZWEI SINE and other clones

No, that is not German. These are some of the popular clones of EMACS made for other operating systems.

EINE [22] was a text editor developed in the late 1970s. In terms of features, its goal was to 'do what Stallman's PDP-10 (original) Emacs does'. Unlike the original TECO-based Emacs, but like Multics Emacs, EINE was written in Lisp. It used Lisp Machine Lisp.

In the 1980s, EINE was developed into ZWEI [23]. Innovations included programmability in Lisp Machine Lisp, and a new and more flexible doubly linked list method of internally representing buffers.

SINE [24] was written by Owen Theodore Anderson in 1981.

In 1978, Bernard Greenberg wrote a version of EMACS for the Multics operating system called Multics EMACS. This used Multics Lisp.

22: EINE stands for Eine Is Not EMACS

23: ZWEI stands for ZWEI Was Eine Initially

These kinds of recursive acronyms are common in the nix world. For example, GNU stands for GNU's Not Unix, WINE (A compatibility layer to run Windows applications) is short for WINE Is Not an Emulator.

24: SINE stands for SINE Is Not EINE

Gosling Emacs

In 1981, James Gosling wrote Gosling Emacs for UNIX. It was written in C and used Mocklisp, a language with lisp-like syntax, but not a lisp. It was not free software.

Figure 1.20: James Gosling - creator of Gosling Emacs and later Java

GNU Emacs

In 1983, Stallman started the GNU project to create a free software alternatives to proprietary softwares and ultimately to create a free [25] operating system.

In 1984, Stallman started GNU Emacs, a free software alternative to Gosling Emacs. It was written in C and used a true Lisp dialect, Emacs Lisp as the extension language. Emacs Lisp was also implemented in C. This is the version of Emacs that is most popular today and available on most operating systems repositories.

25: Recall from the previous chapter that free software does not mean a software provided gratis, but a software which respects the user's freedom to run, copy, distribute, and modify the software. It is like free speech, not free beer.

How the developer's keyboard influences the editors they make

Remember that ed was made while using **ADM-3A** which looked like Figure 1.21.

Figure 1.21: ADM-3A terminal

Figure 1.22: Space Cadet Keyboard

Whereas emacs was made while the **Knight keyboard** and the **Space Cadet keyboard** were in use, which can be seen in Figure 1.22.

Notice how the ADM-3A has very limited modifier keys, and does not even have arrow keys. Instead it uses `h,j,k,l` keys as arrow keys with a modifier. This is why vi uses mostly key combinations and a modal interface. Vi also uses the `Esc` key to switch between modes, which is present conveniently in place of the Caps Lock or Tab key in modern keyboard layouts.

The Space Cadet keyboard has many modifier keys, and even a key for the Meta key. This is why emacs uses many key modifier combinations, and has a lot of keybindings.

1.3.2 Exploring Emacs

This is not a complete overview of Emacs, or even its keybindings. A more detailed reference card can be found on their website.

Opening a File

We can open a file in emacs by providing its filename as an argument to the emacs executable.

```
1 $ emacs test.txt
```

Most of emacs keybindings use modifier keys such as the `Ctrl` key, and the `Meta` key. The `Meta` key is usually the `Alt` key in modern keyboards. In the reference manual and here, we will be representing the `Meta` key as `M-` and the `Ctrl` key as `C-`.

Basic Navigation

These keys are used to move around in the file. Like vim, emacs also focusses on keeping the hands free from the mouse, and on the keyboard. All the navigation can be done through the keyboard.

Table 1.15: Navigation Commands in Emacs

Key	Description
`C-p`	move up one line
`C-b`	move left one char
`C-f`	move right one char
`C-n`	move down one line
`C-a`	goto beginning of current line
`C-e`	goto end of current line
`C-v`	move forward one screen
`M-<`	move to first line of the file
`M-b`	move left to previous word
`M-f`	move right to next word
`M->`	move to last line of the file
`M-a`	move to beginning of current sentence
`M-e`	move to end of current sentence
`M-v`	move back one screen

Exiting Emacs

We can exit emacs either with or without saving the file. We can also suspend emacs and return to the shell. This is a keymapping of the shell, and not of emacs.

Key	Description
`C-x C-s`	save buffer to file
`C-z`	suspend emacs
`C-x C-c`	exit emacs and stop it

Table 1.16: Exiting Emacs Commands

Searching Text

Emacs can search for a fixed string, or a regular expression and replace it with another string.

Key	Description
`C-s`	search forward
`C-r`	search backward
`M-x`	replace string

Table 1.17: Searching Text Commands in Emacs

Copying and Pasting

Copying can done by marking the region, and then copying it.

Key	Description
`M-backspace`	cut the word before cursor
`M-d`	cut the word after cursor
`M-w`	copy the region
`C-w`	cut the region
`C-y`	paste the region
`C-k`	cut from cursor to end of line
`M-k`	cut from cursor to end of sentence

Table 1.18: Copying and Pasting Commands in Emacs

1.4 Nano

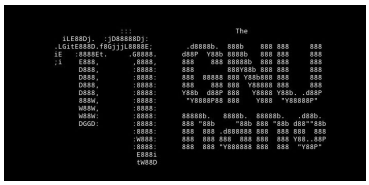

Figure 1.23: Nano Text Editor

Although vim and emacs are the most popular command line text editors, nano is also a very useful text editor for beginners. It is very simple and does not have a steep learning curve.

It is a non-modal text editor, which means that it does not have different modes for different actions. You can directly start typing text as soon as you open **nano**.

Although it uses modifier keys to invoke commands, it does not have a lot of commands as vim or emacs.

1.4.1 History

Pine [26] was a text-based email client developed at the University of Washington. It was created in 1989. The email client also had a text editor built in called Pico.

Although the license of Pine and Pico may seem open source, it was not. The license was restrictive and did not allow for modification or redistribution. [27]

Due to this, many people created clones of Pico with free software licenses. One of the most popular clones was TIP (TIP isn't Pico) which was created by Chris Allegretta in 1999. Later in 2000 the name was changed to Nano. [28] In 2001, nano became part of the GNU project.

GNU nano implements several features that Pico lacks, including syntax highlighting, line numbers, regular expression search and replace, line-by-line scrolling, multiple buffers, indenting groups of lines, rebindable key support, and the undoing and redoing of edit changes.

In most modern linux systems, the **nano** binary is present along with the **pico** binary, which is actually a symbolic link to the **nano** binary.

You can explore this by finding the path of the executable using the `which` command and long-listing the executable.

```
$ which pico
/usr/bin/pico
$ ls -l /usr/bin/pico
lrwxrwxrwx 1 root root 22 Sep  6  2023 /usr/bin/pico -> /etc/
    alternatives/pico
$ ls -l /etc/alternatives/pico
lrwxrwxrwx 1 root root 9 Sep  6  2023 /etc/alternatives/pico -> /
    bin/nano
```

Remark 1.4.1 Note that here we have a symlink to another symlink. Theoretically, you can extend to as many levels of chained symlinks as you want. Thus, to find the final sink of the symlink chain, you can use the `readlink -f` command or the `realpath` command.

```
$ realpath $(which pico)
/usr/bin/nano
```

26: It is believed that pine stands for Pine is Not Elm, Elm being another text-based email client. However, the author clarifies that it was not named with that in mind. Although if a backronym was to be made, he prefered 'Pine is Nearly Elm' or 'Pine is No-longer Elm'

27: Up to version 3.91, the Pine license was similar to BSD, and it stated that 'Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee to the University of Washington is hereby granted ...' The university registered a trademark for the Pine name with respect to 'computer programs used in communication and electronic mail applications' in March 1995. From version 3.92, the holder of the copyright, the University of Washington, changed the license so that even if the source code was still available, they did not allow modifications and changes to Pine to be distributed by anyone other than themselves. They also claimed that even the old license never allowed distribution of modified versions.

28: Mathematically, nano is 10^{-9} or one billionth. and pico is 10^{-12} or one trillionth. or put relatively, nano is 1000 times bigger than pico, although the size of nano binary is smaller than pico.

1.4.2 Exploring Nano

In nano, the Control key is represented by the ^symbol. The Meta or Alt key is represented by the `M-`.

File Handling

You can open a file in nano by providing the filename as an argument to the nano executable.

```
$ nano test.txt
```

Key	Description
`^S`	save the file
`^O`	save the file with a new name
`^X`	exit nano

Table 1.19: File Handling Commands in Nano

Editing

Nano is a simple editor, and you can do without learning any more commands than the ones listed above, but here are some more basic commands for editing text.

Key	Description
`^K`	cut current line and save in cutbuffer
`M-6`	copy current line and save in cutbuffer
`^U`	paste contents of cutbuffer
`M-T`	cut until end of buffer
`^]`	complete current word
`M-U`	undo last action
`M-E`	redo last undone action
`^J`	justify the current paragraph
`M-J`	justify the entire file
`M-:`	start/stop recording a macro
`M-;`	run the last recorded macro
`F12`	invoke the spell checker, if available

Table 1.20: Editing Commands in Nano

There are many more commands in nano, but they are omitted from here for brevity. You can find the complete list of keybindings by pressing `^G` key in nano, or by running `info nano`.

You can also find third-party cheat sheets online.

1.4.3 Editing A Script in Nano

Since learning nano is mostly to be able to edit a text file even if you are not familiar with either vim or emacs, let us try to edit a simple script file to confirm that you can use nano.

```
$ touch myscript.sh
$ chmod u+x myscript.sh
$ nano myscript.sh
```

Now try to write a simple script in the file. An example script is shown below.

```
#!/bin/bash
read -rp 'What is your name? ' name
echo "Hello $name"
date=$(date "+%H:%M on a %A")
echo "Currently it is $date"
```

If you do not understand how the script works, do not worry. It will be covered in depth in later chapters.

Now save the file by pressing `^S`

Remark 1.4.2 In some systems, the `^S` key will freeze the terminal. Any key you press after this will seem to not have any effect. This is because it is interpreted as the `XOFF` and is used to lock the scrolling of the terminal. To unfreeze the terminal, press `^Q`. In such a system, you can save the file by pressing `^O` and then typing out the name of the file if not present already, and pressing Enter. To disable this behaviour, you can add the line

```
  set -ixon
```

to your `.bashrc` file.

and exit nano by pressing `^X`.

Now you can run the script by typing

```
$ ./myscript.sh
What is your name? Sayan
Hello Sayan
Currently it is 21:50 on a Tuesday
```

Now that we are able to edit a text file using text editors, we are ready to write scripts to solve problems.