

An anthropological account of the Vim text editor: features and tweaks after 10 years of usage

Renato Fabbri
`renato.fabbri@gmail.com`
University of São Paulo,
Institute of Mathematical and Computer Sciences
São Carlos, SP, Brazil

December 18, 2017

Abstract

The Vim text editor is very rich in capabilities and thus complex. This article is a description of Vim and a set of considerations about its usage and design. It results from more than ten years of experience in using Vim for writing and editing various types of documents, e.g. Python, C++, JavaScript, Chuck programs; \LaTeX , Markdown, HTML, RDF, Make and other markup files; binary files. It is commonplace, in the Vim users and developers communities, to say that it takes about ten years to master (or start mastering) this text editor, and I find that other experienced users have a different view of Vim and that they use a different set of features. Therefore, this document exposes my understandings in order to confront my usage with that of other Vim users. Another goal is to make available a reference document with which new users can grasp a sound overview by reading it and the discussions that it might generate. Also, it should be useful for users of any degree of experience, including me, as a compendium of commands, namespaces and tweaks. Upon feedback, and maturing of my Vim usage, this document might be enhanced and expanded.

1 Introduction

Vim is a very complex editor, most often considered to be matched only by Emacs. They both are the standard advanced text editors of the free software and open source communities and have been developed for decades. Vim is very useful because:

- it is meant to be a plain text (e.g. ASCII, UTF-8) editor and does not

(by standard) insert special characters (e.g. for formatting or with binary instructions).

- It has a powerful architecture and set of commands.
- It is highly configurable and most often the users hold a set of commands for standard settings and routines kept in the vimrc [5] and other configuration files.
- It has been used for more than 25 years, is based on established technologies, and is an established text editor. Thus, it has a considerable and seasoned user base, high quality (both official and unofficial) documentations, and countless publicly available scripts, most often in the form of plugins.

This document describes the Vim text editor and proposes a set of sharpening of the user experience through simple tweaks and utilization strategies. The contents herein presented is a report on the overall understandings I have of Vim after a bit more than ten years using it, resorting to the (very mature) official documentation whenever possible. The purposes of this document are:

- to help new users grasp Vim essentials and convenient practices.
- To attain a sound overall description of the editor.
- To record the comprehension about the editor that a user (me) has after 10 years of usage.
- To confront my usage with that of other experienced users. This is helpful for me, but also for the other users as they might benefit from this content and from discussions that might arise from it.
- To propose some enhancements to Vim through simple tweaks and potential plugins.

Advanced users might just skim through Section 2, where standard capabilities of the editor should become clear, and consider more carefully Section 2.3. The concluding remarks and proposed enhancements in Section 3 may also deserve some attention because Vim is constantly evolving and there are many possible enhancements (often made available e.g. as plugins).

1.1 Further remarks about this document

This document is written in a DRY KISS (Don't Repeat Yourself, Keep It Simple Stupid) style. Complex is to master the use of Vim and one finds sound references in help files and a nice vimrc. Therefore the following content is kept as uncomplicated and original as possible. Also, because of Vim's complexity and entailed bond of this document to my usage, there is an anthropological component which is evident in the occasional use of the first person. This

can be understood as anthropological computer science [1, 2] and considered to help in the technological groundwork of the civil society. Accordingly, here are some notes about my experience: I’ve had experience with other editors, e.g. Kate, gedit, and Notepad2. I used Vim for writing and editing computer code (Python, Javascript, C++, ChucK, bash, etc), markup languages (HTML, CSS, RDF, Markdown, L^AT_EX, etc) and binary files. Eventually, I edited database files and other types of files. Within Vim, I mostly write (web and scientific) software, scientific articles, music, poems and short stories.

1.2 Historical note

Vim was first released publicly in 1991. It is a cross-platform GNU licensed free and open source extended clone of Bill Joy’s vi text editor. Vi was written in 1976 as a hard link to ex: a shorthand to start ex in visual mode, i.e. vi is ex. Vim’s development is coordinated (and performed) since the beginning mainly by Bram Moolenaar. Today, current bleeding-edge version is 8.0.1401. I found no explicit stable, alpha or beta versions. I found no scientific article on Vim (this might be the first one), although there are books, software and third-party documentation on the web. [3]

2 Basics

Vim’s interface is text-based. In the GUI (gVim), there are convenient menus and toolbars but all functionalities are still available through the command line mode. Vimscript is the internal language of Vim, and is often used for scripting by users although other languages might be used (e.g. Python, Perl, Lua, Racker, Ruby and Tcl). Each line of a Vimscript is a command on the command-line mode. This section may be considered a tutorial that focuses on the namespaces, i.e. sets of tokens that carry values or triggers procedures. One should see Appendix ?? to understand the notation Vim uses.

2.1 The bare minimum

You open a file at Vim startup by executing the command: `vim <filename>`. Inside Vim, you start in the normal mode, and might want to move around using `h-j-k-l` for left-down-up-right. To insert characters, move your cursor to the desired location and press `i`, which puts Vim in the insert mode. Go back to normal mode by pressing `<ESC>`, `<C-]>` or `<C-C>`. You save the file by typing `:w<CR>`, and exit Vim by typing `:q<CR>`. You can save and quit with `:wq<CR>` or `:x<CR>` or `ZZ`.

2.2 Vim help

Help on using Vim is found in various places. The standard resource is the Vim help files. They are accessed by typing `:h <anything><CR>` in normal mode.

Examples of such `<anything>` are: `color`, `navigation`, `:vs`, `vimtutor`. Type `:h usr_toc<CR>` to access the official User Manual, which is considerably lengthy and complex and is usually not read by users before a few years of experience. In learning Vim, one might want to run the `vimtutor` command (outside Vim) to start the Vim Tutor.

There are good resources on the Web for learning and tweaking Vim:

- “Vim Adventures” is an online RPG game for practicing and memorizing Vim commands. This game is quite famous among Vim users.
- There are official and semi-official Vim sites e.g.: www.vim.org, <https://www.vi-improved.org> and <http://vim.wikia.com/>.
- Many hacks, understandings and general issues (e.g. how to make such a move) are asked and answered in online platforms (e.g. Quora, Stack Overflow, Stack Exchange, Reddit, Email list, IRC Channel). One often finds these links through a search engine.
- Many videos about Vim are publicly available. One traditional site is <http://vimcasts.org>, but you might find them using a search engine (e.g. <http://derekwyatt.org/vim/tutorials/>) or in Youtube and Vimeo.

2.3 Namespaces

Vim is a text editor ouroboros [4] because text and writing alters text and writing. What you end with is a collection of namespaces where tokens have scalar or arbitrarily complex values.

2.3.1 Commands and mappings

There are commands, i.e. typing sequences which trigger automated actions, for each mode:

- in Normal mode all keys are mapped to commands. There is redundancy and additional commands using Ctrl and Shift keys. Some keys expect a second key, and have combinations not used (thus available for new mappings), specially the `z` and `g`. See Section 2.4.1 and Appendix C for more insights into the commands available in the normal mode.
- In the other modes, the sequences available for mappings are more obvious and abundant. One should look at `:h index` to know about all the standard mappings and use `:map` to list the user-defined mappings.

`<C-\>` is often reserved for extensions, which makes it a safe namespace to use (while there are no such extensions).

A colon command can be written as a string and executed by the `:execute` colon command. E.g. `:execute 'vs afile.txt'`. As there are colon commands that execute commands in other modes, e.g. `:normal ?~def`, the `:execute` is a way to build commands in any mode, e.g.

```
:execute 'normal i' . string(atan(bufnr('%'))).
```

2.3.2 Variables

There are some types of variables in Vim:

- Environment variables: names start with `$` and hold system variables, such as `$PATH` and `$PWD`.
- Option variables: names start with `&` and are meant to control the behavior of the editor. One might change a value through `set` or `let`, e.g. `:set bg=light` or `:let &bg=light`.
- Registers: start with `@` and are meant for automation and transfer of texts (copy and paste). More on register in Section 2.3.3 and scattered in this document.
- Internal variables are created with `let` and preferably have a prefix: `b:`, `w:`, `t:`, `l:`, `s:`, are local to the buffer, window, tab page, function, and sourced Vim file, respectively. `v:`, `g:` are global, the first are predefined by Vim. `a:` is for function arguments. If there is no prefix, the variable is global or internal to a function if occurring inside a function. More about internal variables in `:h internal-variables`.
- The value of a variable can be a scalar, string, list, dictionary, function reference, etc (see `:h eval`).

You can echo any of such variables or use them in expressions. Notice that you will only be able to echo a `b:` variable inside the buffer where it is defined. For all the Vimscript capabilities, including loops, conditionals, and builtin functions, refer to `:h vim-script` and Section 2.9. Classes are possible only in rudimentary forms, e.g. through dictionaries, but the language is otherwise overall quite powerful, specially in dealing with text and editor behavior, as expected.

2.3.3 State lists

Vim keeps a number of useful lists which expresses the state of the editor:

- All the entered commands are accessed through `:hist a`. The tokens “`a / e : i d`” may be used for specific types of commands, such as search and colon commands.
- File buffers are kept with numeric ids. See buffers with `:ls` and load a buffer to the window with `:b <num or token in file name>`.
- The windows open are listed in `:ls` with a character `a` in the second column, and are listed with `:tabs`.
- Tabs list can be reached through `:tabs`. It is usual both to show and hide the tabs bar (mapping in [5], discussion in Section 2.3.4).

- Jumps are available through `:jumps`. One positions the cursor at each jump through `<C-O>` and `<C-I>`.
- Registers are available through the `:reg` command, as variables and through shortcuts in different modes. They also keep track of your copy, edition and deletion and are promptly defined by recording a typing sequence with the `q` normal command. Vim keeps only the last edition, in register `"`. An autocommand to keep the four latest inserts is in [5]. A hack to keep the latest deletions and copies in the standard register might follow the same pattern, but use another `:h event` and monitor register `"` (maybe also monitor `"0`).
- An undo list can be accessed through `:changes`.
- A list with all the sourced scripts in a Vim instance is displayed through `:scriptnames`.
- The markers defined are listed with the `:marks` command. These are set by `mX` in normal mode, where `X` is the marker identifier. Uppercase letters are cross buffers.
- Quickfix and Location lists which are populated through `:vim` and `:make` and variations, such as `:grep`. One might run `:vim /section/ %` and then `:copen` to open the Quickfix window, where the lines of occurrence are in sequence and one can `<CR>` one of them to have the cursor in the main window active at the first character of the match. One might run `:lvim /section/ %` and then `:lopen` to use the location-list window instead of the Quickfix, which is very similar, but one per window instead of one per buffer. More information in `:h quickfix`.
- A tags list have to be made so one can use tags. Most often one will generate the tags list using the exuberant `ctags`, which supports dozens of languages. E.g. `!ctags-exuberant functions.py` or `!ctags-exuberant -R ./`, and then using `<C-]>` to go to the position of tag under cursor, and `:vs tags` to open the tags file.
- The argument list holds a list of files to be edited or browsed. The list can be input at Vim startup (e.g. `$ vim file1.txt file2.py`) or using commands (e.g. `:ar ./*`). The file being edited is changed by `:n` and `:p` commands, one might perform actions on each file in argument list using `:argdo`. All files in argument list are also in the buffers list. For further information, see `:h arglist` and Section 2.5.
- All the autocommands might be accessed through `:au`. They are event-triggered actions Vim performs.

A file with information about the state of the editor can be achieved through: `:source $VIMRUNTIME/bugreport.vim`. Also, this script might be examined

because it has a collection of commands to access various settings of Vim. Another good list of commands to know about Vim's state is kept on http://vim.wikia.com/wiki/Displaying_the_current_Vim_environment. I would specially highlight the `:syntax` command because it displays token groups and their meanings when run inside e.g. a `.py`, `.vim` or help file.

2.3.4 On the persistence of visual cues about the editor state

You can keep track of the editor state through commands, as stated above. Also, one might rely on persistent visual cues, specially the tabs bar, the status line, and the line reserved for the command-line. A good strategy I find is to have selective visual cues of the state to make persistent or hide and a mapping to toggle each of them. Currently, I toggle byobu/screen/tmux bar with `<F5>`, status line and tabs bar with `<localleader-T or B>` according to script [5]. I am mostly using the cleanest setting, toggling on the tabs bar and status line sometimes. Numbering is always there, I rarely turn them off but keep the mappings `<leader-n or N>` to toggle just in case. Instead of keeping the status bar, I use `<C-G>` to know about the file and `<gC-G>` to know more and rarely. It seems not possible to remove the statusbar between horizontal splits. After asking in online forums and experimenting, I realized that it seems reasonable to keep at least one line dividing the windows, so if it comes to it, I just `set statusline=-`. Unfortunately, as far as I could dig, one will need to alter the Vim's source code to enable a horizontal split without losing a line. For me, it would be ideal for this feature to have a visual cue of the first and/or last line of the windows in the lines-number column, or complete the spaces and empty chars with `$$$$` or so.

2.3.5 On the persistence of the editor state

For state persistence, one might keep an undo file for each file as in [5]. Sessions are easy to manage, enabling one to save and load the editor's state, with the opened windows, tabs, buffers, etc. The mappings in [5] keep the sessions in a reasonable directory and makes it easier to remember and tweak the standard commands to deal with sessions. More information in `:h sessions`. One might use `:h views` to keep the state of one window, but sessions keep all the states from all windows. This entails a strategy to deal with Vim that is similar to the use of Byobu/Tmux/Screen¹, because one can rely on restoring the state of the windows. The main limitation I found to this approach is that Vim is not keeping track of the terminals opened. If you open a terminal inside Vim with the `:term` command, you will save the session as usual, but when loading you get dummy empty windows for them and an error message. More about Terminal-Job mode in Section 2.4.6.

Autocommands are the standard way to define event-triggered routines in Vim. These are often related to particular file types, but are also often in

¹[https://en.wikipedia.org/wiki/Byobu_\(software\)](https://en.wikipedia.org/wiki/Byobu_(software)), <https://github.com/tmux/tmux/wiki>, <https://www.gnu.org/software/screen/>

defining the automated behavior of Vim. If the autocommands are placed inside a configuration file (e.g. the `vimrc`), the automated behavior is persistent across Vim instances. E.g. in [5] is found an autocommand for keeping track of the last inserted texts in the `\".lkjh` registers (`@.lkjh` variables).

2.4 Using Vim's modes

These are the basic and fully implemented modes of usage in Vim:

- Normal mode: used for changing the position of the cursor or the text displayed at the window. A core goal of the normal mode is to support fast navigation of the document while allowing the typist to maintain the fingers on the home row (i.e. on the center of the keyboard). The mode is also used for manipulating text (e.g. copy, paste, delete, change case) and changing to other modes. More in Section 2.4.1.
- Insert mode: for inserting text. More in Section 2.4.2.
- Command-line mode: for entering Ex commands. More in Section 2.4.4.
- Ex mode: similar to command-line mode, but more specialized for running various Ex commands. More in Section 2.4.5.
- Visual mode: for making, manipulating and navigating selections of texts.
- Select mode: similar to visual mode but favors CUA².

There is another basic mode, but it is not fully implemented: the Terminal-Job mode (more in Section 2.4.6). There are seven additional modes which are mostly subordinate to the basic modes and that will be described when convenient. The manual page for Vim modes can be accessed by typing `:h vim-mode`. Some of the modes are now further considered for the achievement of an overview of the Vim usage possibilities.

2.4.1 Normal mode

Sometimes also called navigation or command mode, the normal mode is most powerful for navigating, manipulating texts and changing to other modes. The simplest of these three is changing to other modes: type any of these letters to change to insert mode: `iIaAoOsScC`. More on the transition between normal and insert mode on Section 2.4.3. Type any of these characters to change to command-line mode: `:/?`. Type `Q` to enter Ex mode. Type `v`, `V`, or `CTRL+V` to enter visual mode.

For very basic and naive navigation, one should check Section 2.1. There are many facilities to navigate Vim as explained in `:h navigation`. Most often, one uses:

²IBM Common User Access: https://en.wikipedia.org/wiki/IBM_Common_User_Access.

- `Ctrl+(d,u,f,b)` for half-page down and up and whole page down and up, respectively, although these commands might be set to scroll a different number of lines.
- `Ctrl+(e,y)` to move the window one line down or up.
- `(w,b,e)` to move to the next and previous word, and next end-of-word. There are motions to iterate over sequences of characters separated by special characters (e.g. punctuation and parenthesis) as specified by the output of `:se iskeyword`. To iterate over space-separated tokens, use `W,B,E`. To move to the end of last word, one might use `be` or `ge`. The `)`, `(` commands iterate through sentences, `}`, `{` through text blocks separated by empty lines.
- `(fX,tX,FX, TX)` to move to or just before any `X` character, `;` and `,` for next and previous found character.
- Search with `/` or `?`, although these are in truth command-line commands.
- `CTRL+(o,i)` to move to an older or newer position in jump list.
- `'X`, `'X` to move the cursor to a mark bind to the alphanumeric character `X`: `'X` moves to the exact position while `'X` moves to the first non-blank character of the line. A mark is registered by the user in any cursor position by typing `mX`, where `X` is any letter. If `X` is lowercase, the mark is local to the buffer (the file), if it is uppercase or numeric, it is global to the Vim session (cross buffers).

For changing the text, usual commands include:

- `d{motion}` to delete the characters involved in the motion command.
- `dd,D` to delete a line or from the cursor to the end of the line.
- `x,X` to delete the character under or before the cursor.
- `~` to swap the case of a character or selection.
- `gu{motion}`, `gU{motion}`, `g~{motion}` to make lowercase, uppercase or switch the case of the characters involved in the motion.

There are way more commands to change the texts. Some of them are discussed in Section 2.4.3 because they involve a transition to the insert mode. A thorough consideration of the commands in the normal mode is found by executing `:h navigation`, `:h change.txt`, `:h index`.

2.4.2 Insert mode

Once in the insert mode, the character keys input the characters at the cursor position at the current buffer. One can exit insert mode by pressing `<Esc>` (or `<C-[>` or `<C-C>`), and Vim will be put in the normal mode. Most useful commands in insert mode include:

- `<C-O>` to execute one and only command in normal mode. This enters a secondary mode (see Section 2.4).
- `<C-R>` to paste a register (a variable starting with `'@'`, defined, copied or recorded through a `q` command in normal mode and as covered in Section 2.3.3).
- `<C-T>` to indent current line.
- `<C-U,W>` to delete all chars from cursor to the beginning of the line or to the previous word.
- `<C-N,P>` to find next and previous keywords that match the prefix at hand.
- `<C-X>` commands for scrolling the window with multiple `<C-E>` and `<C-Y>` strokes and for some completion facilities.

2.4.3 Normal → insert modes

Many commands bridge from Normal to Insert modes, e.g. `iws` or any of these letters: `cscSR`. These make convenient the replacement of text and populates registers. The absence of a short command to insert one char is a known issue in Vim. Reasonable mappings to insert or append a char to and around another char are in [5]. Vim couples operator and motion commands by design. There are many operator commands that take the editor from the normal mode to the insert mode, most of them favoring deletion or change, as detailed in `:h operator`. Motion commands are described in `:h motion`.

2.4.4 Command-line mode

This mode is dedicated to writing colon, search and filter commands, entered through typing `:`, `?`, `/` and `!` in normal or visual modes. Most useful commands in this mode include:

- `<C-B>` and `<C-E>` to move cursor to the beginning and end of the line.
- `<C-W>` and `<C-U>` to delete last word or everything until the cursor.
- `<C-R>` to paste a register (as in insert mode).

2.4.5 Ex mode

One might use the normal commands `q(:,/?)` to have a window with the colon or search command history to be edited normally, and the chosen command can be run with `<CR>`. Vimscript was largely based on the ex editor [6], and a more advanced user might use it for prototyping by defining mappings and settings and managing scripts probably in a `plugin/` folder of a directory in `:echo &runtimepath`. In the default interface started with the `Q` command in normal mode, each command is input without entering `:` again. Use `:vi` to exit Ex mode, follow documentation from `:h Ex-mode` for further information. I might be loosing something, but for tweaking I use the command-line window accessed through `q:`. I find it comfortable to browse the history in a normal-like mode, edit them also using the insert mode as usual, and having auto-completion when pressing `<TAB>`. None of such features are available in Ex mode by default and customization by mappings have to be performed through autocommands as in [5].

2.4.6 Terminal-Job mode

This mode is reported as not having reached a stable usage design (see `:h terminal`). I find that it works exceptionally well and have used it to run scripts in an IPython shell, compile latex files, and even open PDFs and images. Vim browsing of windows and text manipulation is well developed, so the Terminal-Job mode enables a very convenient integration of files being edited and bash terminals, more traditionally achieved through Byobu/Tmux/Screen terminal multiplexers. Most useful commands in the terminal-Job mode include:

- `<C-W>N` and `<C-W>_:` for entering the Terminal-Normal and command-line modes. Terminal-Normal mode is very similar to Normal mode, but one cannot change the text, cannot enter insert mode, and the status line reports if the job is finished or not.
- `<C-W>_"` to paste a register.
- `i` for entering the Terminal-Job mode from the normal mode.

It is useful to define the same mappings for navigating splits and tabs for both Normal and Terminal-Job modes, as in [5].

Because browsing the interface in Vim is fast, and it favors copy and paste of text, it is often handy to keep (e.g. a tab with) some terminals: one with an IPython shell, another two for compiling \LaTeX and opening PDF files (e.g. with `$ evince <filename.pdf>`). It is very comfortable to copy and browse the terminal lines in the Terminal-Normal mode. I found the mappings on [5] very helpful for directing editor focus to splits (`<C-hjkl>`) and tabs (`gr, gt`), which I make available across Terminal-Job and Normal modes. But I've been thinking on using `<C->` commands also for tabs (not only for splits), both to avoid mixing typing sequences with commands, and to allow the use of the commands in all Normal, Insert and Terminal modes.

2.5 Netrw

The standard interface of Vim for browsing file trees is Netrw. It starts when you open a directory, such as with `:e .<CR>`. It has solid support for editing remote files (such as over ssh or ftp) and handy e.g. mappings to open the files as splits and tabs (specially `pot`). Most useful commands in netrw include:

- `d` and `%` for creating directories and files. `<Delete>` removes both.
- `mf` and `mb` for marking files and bookmarking directories.
- `gb` and `uU` are used to load directories while marked files might be copied, moved, edited, grep-ed, tagged and migrated to and from the arglist as in `:h netrw-mf`.

Last notes: there is no insert mode in netrw interface; the commands in `:h netrw-explore` are convenient for opening the directory of the file being edited; further information in `:h netrw`.

2.6 Standard configuration files and directories and my .vim/vimrc

You can check the scripts Vim loads by using the debug script mentioned in Section 2.3.3. By default, `~/.vimrc` and `~/.vim/vimrc` files are run by Vim at the beginning of the startup. One might edit the vimrc file with `:e $MYVIMRC` and reload it with `:source $MYVIMRC`, but I use the mappings in [5] to encourage a continuous enhancement of my settings. Any other file might be included to run at startup by adding a line `:source <afilename>` in vimrc. In fact, it is on vimrc that one usually specifies the plugins and plugin managers they use. Use an `after/` folder of a directory in `:se runtimepath` or follow some patterns described on the next section to change the scripts and sequence of them to be loaded.

2.7 Plugins and packages

One can see the list of standard plugins with `:h standard-plugin-list` command. Any `plugin/**/*.vim` file inside a directory listed by `:se runtimepath` will be loaded (e.g. `.vim/plugin/something/ascript.vim`). There are various ways to automate the installation and enhance the management of plugins. By default, one has the GETSCRIPT interface (see `:h getscrip`), that downloads latest scripts from sourceforge as specified in `:h getscrip-data`, and the Vimball interface, which creates and loads a Vimball for a plugin with `:[range]MkVimball <filename> path`, where range specifies lines that hold paths to files to be included in the `<filename>.vba` Vimball. The Vimball can be installed in a system by `:source <filename>.vba` or loading it at Vim startup with `$ vim <filename>.vba`.

A Vim package is a directory that contains plugins. It should be located inside a `pack/` directory somewhere in the directories listed with `:se runtimepath`.

The plugins found in `pack/<packName>/*/start/` are loaded at startup, the plugins found at `pack/<packName>/opt/**` are loaded with `:packdd <script_or_directory_name>`.

All directories Vim looks for scripts are described in `:h vimfiles` and are basically set by `:se runtimepath` and conventions inside each path therein, such as to look for `vimrc` files and `plugin/` or `pack/` directories. Vim scripts can be loaded conditionally, e.g. only if a function is used as in `:h autoload-functions`. Example of such are filetype plugins (enabled by in a `ftplugin/<filetype>.vim` file, e.g. inside a plugin directory). There is a number of plugin managers for Vim. Pathogen and Vundle seem to be the most popular, one because of its minimalism, the other because of advanced features, e.g. for searching and installing plugins with colon commands.

2.8 Spell and spelllang (en and pt_br)

One might set the spelling language with `:se spelllang=en_us` or `:se spelllang=pt_br` and toggle spell checking with `:setl spell!`. These are used so often that one might use shortcuts as in [5]. Currently, Vim will download the files for a specific language if not found in system.

2.9 Scripting, Functions, Vimscript and other languages (e.g. Python)

In Vimscript, the colon commands (also Ex or command-line commands) are related through spaces, punctuation and keywords (see `:h script`). Scripting the Vim editor can also be accomplished using other languages, as well documented e.g. `:h python`. Functions are defined through colon commands and are called inside colon commands e.g. `:call MyFunction()` or `:echo MyFunction(4)`. Notice that functions are not commands but might be bind to them through colon commands e.g. `nnoremap gF :call MyFunction()<CR>`. Executing source files is very straightforward with `:so <filename>.vim`, and one can always use the Ex mode for rapid scripting.

At the same time, the `:term` and terminal-job mode make scripting other software more convenient, as output is promptly navigated and copied. The shell works well for generic use, such as opening images and PDFs through `$ eog` and `$ evince`. See Section 2.4.6 for further directions about the terminal.

2.10 Color

Newer versions of Vim support 24 bit true color (aka 16 million colors) in terminal Vim (not only in GVim). The terminal must support true color, and tests are available e.g. in <https://gist.github.com/XVilka/8346728>. Then Vim needs to be set to use true colors with `:set termguicolors`. If using 8 or 16 bit colors, Vim uses the color pallete from the terminal, if using true colors each color is defined directly. Settings for using true colors inside Byobu/Tmux

involve tweakings and are available in [5]. Good color schemes to use with true colors are Gruvbox and Solarize. One might source syntax files at any time to change syntax, usually though linking tokens to syntax groups with `:syntax keyword <group_name> <token1> <token2> <token3> ...` and then relating the group to another group `:highlight link <group_name> <group_name2>` or by specifying the group characteristics directly: `highlight <group_name> guifg=#ffffff`. If you change a syntax file, reloading a file with `:e<CR>` updates the highlighting on the window with the corresponding filetype. A complete syntax highlighting support typically involves at least three files:

- `ftdetect/<filetype>.vim`, where the file type is detected with e.g. `:autocmd BufNewFile,BufRead *.<file_extension> setfiletype <filetype>`
- `ftplugin/<filetype>.vim` with general settings for the file type, such as: `:set tabstop=2 softtabstop=2 shiftwidth=2 textwidth=70 expandtab autoindent`.
- A `syntax/<filetype>.vim` file, with bindings between tokens and highlighting groups; and highlighting group definitions.

This very scheme is implemented very straightforward in this plugin [7] for highlighting text in the Toki-Pona language [8]. Syntax highlighting plugins are as filetype plugins, but also have a `syntax/<filetype>.vim` file relating keywords to highlighting groups. One might see every highlighting group, and their final visual results, with the command `:so $VIMRUNTIME/syntax/hitest.vim`. In my system, the `ftdetect/` and `ftplugin/` folders load as expected in the `plugin/` directory, but `syntax/` files had to be moved to `~/.vim/syntax/`.

2.11 Fonts

The fonts are defined by your terminal or inside GVim. `<C-+>` and `<C-->` can be used to change font size. Some settings for fonts, such as boldface, might be set using the syntax highlighting facilities described in the previous section.

3 Conclusions and further work

This document seems reasonable as an overall reference of the Vim editor, at least for my usage and proficiency. Given the folkloric milestone of using Vim for 10 years, this article might serve as a benchmark for one to relate it's current use and understandings. As a pedagogical material, it seems to be unique in the emphasis on namespaces, understood as commands, variables, state-related lists, etc, especially in Section 2.3, and the reference to the standard Vim documentation to achieve the DRY KISS style described in Section 1.

Potential enhancements to this document are:

- The inclusion of facilities such as reading emails and connecting over ssh. There is a working hack in [5] for browsing over the WWW, but such aspect

of Vim usage might receive more attention given that it is comfortable to use the navigation and editing facilities and the resulting integrated environment.

- Updating of the information I can find about the issues discussed, such as about status lines in Section 2.3.4.
- Include a discussion about Neovim. I have never used it, but it seems to be reaching a considerable user base and it might be feasible to give an account of Vim and Neovim after some tests and researching the official and user base documentation.
- Better cite documentation, plugins and Vim authors. I preferred to keep the references inline through `:h` commands and URLs, more in accordance with the style of Vim documentation, but bibliographical items constitute a valuable asset for academic literature, and some authors might find their work more pretiged if more thoroughly cited.
- An analysis of my usage, e.g. according to <http://www.drbusen.org/vim-croquet/>. This undertake might benefit from data from many users, which favors the potential plugin for usage analytics described in the next bullet list.

Potential next steps in using Vim:

- Measure the performance of text mining routines in Vim against those implemented in C or Python.
- Enhance the HTTP browsing capabilities of [5].
- Better integrate Python and Vimscript, especially for data visualization and syntax highlighting management. In accordance with the visualization issues described in Sections 2.3.4 and 2.10.

Many of the next steps might entail plugins. This is a selection of the issues that are more prominent for me:

- listing all the mappings available in each mode and the typing combinations which are available for new mappings.
- Sessions, as described in Section 2.3.4.
- AA messages (shouts): to keep track, document and share of working sessions as in [9, 10], with capabilities to manage AA sessions, send visual or sonic cues for temporal marks, use Vim state to build AA shouts, relate AA sessions to other media, such as software repositories, screencasts and images, interact with IRC channels and other social platforms.
- Slick Vim: a collection of the settings, mappings and plugins I use. Enhancements such as using `<C->` commands also to browse tabs, shortcuts

to join a window into a tab, and dummy minimal plugins as simplest, file type and syntax highlighting, Some more elaborate tweaks should also be present, such as breaking lines in sensible places for natural language texts while respecting e.g. `:se textwidth`.

- Dealing with `.swo` and `.swp` temporary files. In summary, if the restored `.swo` file has the same content and the file being opened, the restoration phase can be omitted. If the contents differ, Vim should open a tab with each file in a vertical split and run `:windo :diffthis`.
- Rendering images and equations. These are useful for using Vim in presentations or achieving a textual representation when it is mandatory, such as to comply with the limitations of a platform (e.g. Vim editor). but also hold stylistic merits as ascii art is often very appreciated. One can both obtain an ascii representation of a binary image (e.g. JPG, PNG), and can directly render ascii charts from data using cues such are shape, position and color.
- Redirecting the commands that usually show the results in a 'more' interface, which cannot be searched nor copied, to a parsed and linked quickfix or location window. The basic idea is to use `:redir` command to redirect the output of such commands with `:se nomore`. Reasonable functions (and convenient commands) for having the output of such commands in a standard Vim window are in [5].
- Slide presentations. I've been using some automation for browsing slides and opening figures and some Vim users asked for the settings and commands. They are very elementary use of registers being executed over arbitrary but consistent textual patterns.
- To facilitate the tweaking of syntax highlighting. This should include ways to easily access and change the syntax highlighting scheme and dump it to a file (dumping current highlighting scheme is currently not supported). Should also include changing the color scheme an highlighting scheme incrementally and selectively using the features described in Section 2.10.
- For bringing back all the splits after an `:only`. Also for bringing back the tabs after a `:tabo`.
- Run Python excepts from file in an IPython shell. The buffer number of the terminal window should be stored, and than any selected lines should be run on that instance. Mappings should make available all movements to catch the excepts, execution of registers, remain on script or in the IPython shell. Also, terminal mode can use some improvements, such as `<C-O>` for one normal mode command.
- For keeping track of the usage and making analysis for optimizing the usage, as described in [?]. Usage analytics.

- Colorschemes for true color (gui). The standard colors (e.g. blue, elflord) loose some of the distinctions, e.g. SpellBad tokens are not highlighted on these colorschemes if you have a functioning `se termguicolors` (standard GUI mode and available in terminals since recent versions of Vim 8, see Section 2.10). In [5] are some lines that make such colorschemes after over `:colorscheme blue`. This design of color schemes over the standard color schemes might be a very simple and effective way to make new color schemes.
- Opening the editor status listings such as in Section ?? in a quickfix or location window where they might be searched, edited and used as wanted. These windows open in a more-like interface that is very restricted in what they allow (e.g. no search, no persistence).

Acknowledgments

FAPESP (project 2017/05838-3); Vim developers and documentation maintainers; Vim user community.

A Example of usage session

I usually begin by opening a file or directory with `$ vim <filename>`. The color scheme is alternated between blue and GruvBox with `:colo gruvbox` and `:colo blue`. I open an vertical split and then move the window to a new tab using `:vs` and `<C-W,T>`. I then search tokens related to the enhancements I want to make or the knowledge I want to acquire. I go back to the previous tab with `gr` and make a global replace with `:%s/<this>/<that>/g`. On adding dots to sentences, I record in the "q register the sequence `jA.jj`, using it as a macro 10 times by `10@q`. I move to the other tab with `gt` and open a terminal window with `:term` for compiling latex files. I start another terminal with `<C-W> :term` for opening the resulting PDFs with evince. If any new idea comes to mind and I have time, `\\s` opens my vimrc [5] for editing and `\s` sources it. If there is e.g. code or notes in other projects I am working on, I reach them through `<Alt-arrows>` as other Vim instances in Byobu/Tmux sessions and windows.

B My vimrc file and usage

In this Appendix is my vimrc file. Although it has comments, one should look for the options that (s)he does not understand, as a thorough explanation of the file is tedious and out of the scope of this document.

In using Vim with my vimrc file [5], I mostly toggle the status line with `\\B` and the tab line with `\\T`. Save and close windows with `\w` and `\q`. The mappings for transitioning through splits and tabs are also used constantly.

C Example notes on mappings

`:h index` shows all the default mappings while `:map` shows the user-defined mappings. By considering such information, one can make useful observations exemplified in this Appendix.

C.1 Normal mode

Every letter and character in the keyboard is used. In Normal mode: `<TAB>` is the same as `<C-I>`, `<BS>` and `<C-H>` is the same as `h`. `<C-J>` and `<C-N>` is the same as `j`. `<C-P>` is the same as `k`. Space is same as `l`. `<C-[>` and `<Esc>` are not used, `<C-\>` `a-z` is reserved for extensions, `<C->` not used. `+` is the same as `<CR>` and they are both not very useful. `Del` is same as `x`.

Many `]` `[` combinations are not used, e.g. with `abhjklfg`. The `_` command might be used as a more powerful `^`, leaving it free for mappings.

Directions, home, end, page up and down, insert, all have mappings in more centrally located keys. unbounded: `g`, `z`, `[]`

The `<C-(HJKL)>` commands are redundant, with the exception of `<C-L>` which redraws the screen, so it is a reasonable choice to use them to move focus of the editor to splits in the `hjkl` directions.

Many key combinations are available for new mappings through the `g` and `z`, and `v` commands. They have typical uses, e.g. `z` for folds, spell checking and some movements (mainly when wrap is set).

C.2 Insert mode

All standard char keys are used for entering chars on text. `<C-G>`(`j,k`) can be achieved by `<C-O>`(`j,k`) which is more powerful in moving through multiple lines. `<C-[>` is the same as `<ESC>`. `<C-J>` and `<C-M>` are `<CR>`. `<C-\>` `a-z` is reserved for extensions, other combinations with `<C->` are not used.

D My :version

I should keep the output of `:version` executed on the system in which I wrote this document in this link: . It was compiled with this Makefile [11] in a 16.04 LTS Ubuntu Linux, and is tagged as 8.0, Included Patches 1-1173.

E Key notation and meaning for Vim

The tokens `<C-X>`, `<S-X>` and `<M-X>` mean `Ctrl+x`, `Shift+x` and `Alt+x` (or `Meta+x`). `<A-X>` is the same as `<M-X>` and refer to Alt or Meta keys. `<C-X>`, `<C-S-X>`, `<C-S-x>` and `<C-x>` are the same, i.e. Vim does not distinguish lower and uppercase letters in `<C->` commands. The `<M-x>` and `<M-X>` (or `<M-S-x>` or `<M-S-X>`) are different commands, and they are not used by Vim's builtin keys. Such `<M->` key combinations are potentially conflicting with shortcuts of

other programs (e.g. `<M-F10>` for a terminal menu), but are otherwise a large and safe set of combinations for one to use.

In summary, one might choose for mappings to overwrite defaults key combinations, use combinations not used by Vim (e.g. some of the `g` and `z` normal commands, `<C->` and `<S->` commands, or any `<M->` and `<M-S->` command), or create new mappings using `<leader>` and `<localleader>`. More information about key notation in `:h key-notation`. Default mappings are in `:h index` and user-defined (and plugin-defined) mappings in `:map`.

References

- [1] Anthropological physics and social psychology in the critical research of networks. Complex Networks Digital Campus (CS-DC'15). Available at <https://youtu.be/oe0KYc3-nbM>
- [2] Fabbri, R. What are you and I? [anthropological physics fundamentals], 2015. Available at https://www.academia.edu/10356773/What_are_you_and_I_anthropological_physics_fundamentals_
- [3] Vim (text editor). (2017, December 4). In Wikipedia, The Free Encyclopedia. Retrieved 07:08, December 16, 2017, from [https://en.wikipedia.org/w/index.php?title=Vim_\(text_editor\)&oldid=813716105](https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=813716105)
- [4] Ouroboros. (2017, November 2). In Wikipedia, The Free Encyclopedia. Retrieved 22:19, November 9, 2017, from <https://en.wikipedia.org/w/index.php?title=Ouroboros&oldid=808392809>
- [5] Fabbri, R. (2017). A reasonable vimrc file. Available at <https://raw.githubusercontent.com/ttm/vim/master/vimrc>
- [6] Ex (text editor). (2017, March 22). In Wikipedia, The Free Encyclopedia. Retrieved 22:22, November 9, 2017, from [https://en.wikipedia.org/w/index.php?title=Ex_\(text_editor\)&oldid=771621020](https://en.wikipedia.org/w/index.php?title=Ex_(text_editor)&oldid=771621020)
- [7] Fabbri, R. (2017). A Toki Pona Python Package and Vim Syntax Highlighting. Available at <https://github.com/ttm/tokipona>
- [8] Lang, S. (2014). Toki Pona: the language of good. Tawhid Publishing. ISBN-10: 0978292308, ISBN-13: 978-0978292300.
- [9] Fabbri, R., Fabbri, R., Vieira, V., Penalva, D., Shiga, D., Mendonça, M., Negrao, A., Zambianchi, L., & Thumé, G. (2013). AA: The Algorithmic Autoregulation (Distributed Software Development) Methodology. RESI. From <https://arxiv.org/abs/1604.08255>
- [10] Fabbri, R. (2017). The Algorithmic-Autoregulation (AA) Methodology and Software: a collective focus on self-transparency. ENMC2017. From <https://github.com/ttm/ensaio/raw/master/emc/article.pdf>

- [11] Fabbri, R. (2017). The Makefile with which I compiled Vim for this article. Available at <https://raw.githubusercontent.com/ttm/vim/master/Makefile>