Workflow Notes

May 17th 2020:

Hwan Goh

Abstract

Notes detailing my workflow using Vim and i3wm. Also details emulation on Windows.

Contents

1 Vim

1.1 Vim Tips and Tricks

Some good references are [**?**, **?**]. A more advanced tutorial that I haven't watched yet is [**?**]. A document on effective text editing written by the creator of Vim can be found at [**?**].

1. In Linux based systems, put 'setxkbmap -option "caps:escape' into /.bashrc to map the caps lock key to escape.
2. To set Vim as your default text editor, use 'sudo update-alternatives –config editor'.
3. Suppose the cursor is in the middle of a word. Whilst 'cw' and 'dw' will change/delete until the end of the word, 'ciw' and 'diw' to change or delete the whole word, thereby not requiring you to move the cursor to the beginning of the word first.
4. To reformat a paragraph, use 'gq'. Reformatting includes enforcing the wrap limit which can be set in your vimrc with 'set tw=<number>'. Other options can be se such as indentation. Note that this command requires a minimum of two lines, so you'll need to at least use 'gqj' and at most use 'gjG' for the rest of the document under the line currently under the cursor.
5. Use 'm<key>to set a mark to <key>then <key>to jump to it. Note that if you set it to the capitalized version of <key>then the jumping can occur between buffers. To see your list of marks, use ':mark'.
6. Use '"<key>' then an action like 'y' or 'd' to store text in the *register* <key>. Then use '"<key>p' to paste it. Note that using 'd' will automatically store it in the register at register x and 'p' will automatically paste whatever is in register 'x'. Use ':reg' to see the register list.
7. Use 'f<key>' to forward search for <key>. To backwards search, use 'F<key>'. To repeat the search, use ';'.
8. The 't' stands for 'til'. For example, 'dt=' will delete up to and not including an = sign.
9. The '*' can be used to search for the word under cursor.
10. 'J' is used to join the line below to the line currently under the cursor. This is useful for reformatting improperly wrapped lines but in general 'gq' is more useful for this purpose.
11. 'zt' or 'z<CR>' will put the line under the cursor to the to of the window. 'z.' will put the line to the center of the window and 'z-' will put it to the bottom of the window.
12. 'z=' will give spelling suggestions for the word under the cursor.
13. '[s' and ']s' to cycle backwards and forwards through misspelled words.

14. Use ‘g$’ to go to the end of an unwrapped line.

15. If you type a long line of text in insert mode, this counts as one action. Therefore, if you press undo, the whole line will be deleted. To break the undo chain, use <c-g>u while in insert mode. Alternatively, you could map space to <c-g>u so that the undo chain is broken whenever a space is added in insert mode:
inoremap <Space><Space><c-g>u

16. Use the accent ‘<shift+6>’ in insert mode to move to the first non-white space character in the line.

17. Use ‘%’ while your cursor is over a parenthesis, square or curly bracket to move to the corresponding open/closing parenthesis or bracket.

18. Use ‘:ls’ to see the list of buffers then ‘:b’ and the number to select one. A useful mapping for this process is:
nnoremap <leader>b:ls<cr>:b<space>

19. Use ‘<number>+<c-6>’ to switch to numbered buffer.

20. Use ‘<c-f>’ in command line mode to view the command history in a buffer.

21. In general, all yank, change or delete actions such as ‘y, c, d, x’ etc will register the text object into the ”” register. However, any yank action will also register the text object into the ”0 register and any delete or change action will register the text object into the ”- register.

22. Use ‘s’ or ‘S’ to substitute. This is useful when you want to replace one letter with multiple letters.

23. When using search ‘/<word>’, you can use <c-g >and <c-t>to cycle through them without confirming your search with <CR>.

24. Use ‘<c-r>’ in command line mode to paste from the register.

25. Use ‘<c-r>’ in insert mode to paste from the register.

26. Use <c-w>N in a Vim terminal to switch to normal mode, which allows you to navigate as if you were in Vim.

27. Use ‘<c-z>’ to suspend Vim and return to shell. Use ‘fg’ in shell to resume the suspended program. You can also use

```
stty susp undef
bind '"\C-z":"fg\015"'
```

in your bashrc to, overall, set ‘<c-z>’ to toggle a suspended progran.

28. Using ‘ctags -R’ in terminal creates a tag file. Then, in vim, use ‘<c-] >’ while the cursor is over a function name to jump to the code defining the function. You can also add the following shortcut to your vimrc:

```
command! MakeTags !ctags -R .
```

so that tags can be created from the command line within Vim. Note that you need to add

```
set tags=tags;/
```

to your vimrc so that ctags will check the current folder for tags file and keep going one directory all the way to root folder. See [**?**] for more details.

1.2 Plugin Management

The first part of this section follows [**?**]. First, we work towards turning /.vim into a git repository:

1. Move ∼/.vimrc into ∼/vim.
2. When vim boots, it's still going to look for .vimrc in the home directory. To ensure that it looks for vimrc in the ∼/.vim directory, we can create a symbolic link to that file using 'ln -s ∼/.vim/vimrc ∼/.vimrc'.
3. Make ∼/.vim into a git repository.

An issue now is if you install a plugin that itself is a git repository, you lose the version-control capabilities of that plugin. To circumvent this issue, we use a plugin manager; in this case *Pathogen*.

1.2.1 Pathogen

The pathogen plugin makes it possible to cleanly install plugins as a bundle. Rather than having to place all of your plugins side by side in the same directory, you can keep all of the files for each individual plugin together in one directory (see video from first link for example). This makes installation more straightforward, and also simplifies the tasks of upgrading and even removing a plugin if you decide you no longer need it since they are carefully segregated from each other. For a good tutorial on Pathogen, see [**?**].

Following the readme on the repo at [**?**], to install Pathogen do the following:

1. Run in terminal:
   mkdir -p ∼/.vim/autoload ∼/.vim/bundle &&
   curl -LSso ∼/.vim/autoload/pathogen.vim https://tpo.pe/pathogen.vim
2. Add the following to your vimrc: execute pathogen#infect()

Now any plugins you wish to install can be extracted to a subdirectory under /.vim/bundle, and they will be added to the 'runtimepath'. For example, to install "sensible.vim", simply run: "cd /.vim/bundle && git clone https://github.com/tpope/vim-sensible.git".

1.2.2 Submodules: Installing Git Repositories Within Git Repositories

This section follows [**?**]. Now that we can install plugins via Pathogen, let's see how we preserve the version control capabilities of our plugins. As a worked example, let us install the *Vimtex* plugin:

1. cd ∼/.vim
2. Now to clone a git repository into the bundle directory, use:
   git submodule add https://github.com/lervag/vimtex.git bundle/vimtex

Now, to upgrade this plugin, use:

1. cd ∼/.vim/bundle/vimtex
2. git pull origin master

To update ALL of your plugins, use:

1. cd ∼/.vim
2. git submodule foreach git pull origin master

1.2.3 Importing Your Vim Configuration and Plugins To a New Machines

One of the main benefits of version controlling your Vim configuration and plugins is the ease of which they can be imported into a new machine. To do so, use the following:

1. cd ∼
2. git clone <git repo url>∼/.vim
3. ln -s ∼/.vim/vimrc ∼/.vimrc
4. cd ∼/.vim
5. git submodule init
6. git submodule update

1.2.4 Vim-plug

Whilst Pathogen is the most basic plugin manager, there are limitations when porting your setup to a new machine:

1. When you run git submodule update, you'll pull the latest versions of the plugins from their respective repositories. So unless you've noted down somewhere all the commit IDs for your favourite version of each plugin, your overall plugin collection cannot be preserved when porting to a new machine.
2. Suppose one of the plugins is no longer being maintained by the owner and suppose also that you've made your own changes to the plugin. If you were to pull your configuration on a new machine, you will pull the latest version of the plugin; that is your changes will not be ported. Further, on your own repository for your configuration, the directories containing the plugins will be treated as repositories. Unless you manually install the plugins, there is no way to preserve your changes onto Github.

The plugin manager Vim-plug [**?**] has a solution to the first problem. This plugin manager is extremely simple to use:

1. Run in terminal:
   curl -fLo ∼/.vim/autoload/plug.vim –create-dirs https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
2. In your vimrc, add the line:
   call plug#begin('∼/.vim/plugged')
3. To include a plugin you wish to install under the line above, in your vimrc add the following line:
   Plug 'https://github.com/lervag/vimtex.git'
4. Under 'Plug <url >' of your last plugin, in your vimrc add the following line:
   call plug#end()
5. Reload your vimrc and use the command ':PlugInstall' while in vim. Now all your plugins are installed.

Now to preserve the versions of your plugin:

1. In Vim, run the command ':PlugSnapshot! <filename >.vim'.
2. To restore the state of your plugins, in vim run the command:
   :source ∼/.vim/<filename >.vim
   or, in terminal:
   vim -S ∼/.vim/<filename >.vim

See the bradagy's answer in the reddit post [**?**]. To port your configuration to a new machine:

1. cd ∼
2. git clone <git repo url >∼/.vim
3. cd ∼/.vim/vimrc
4. :PlugInstall
5. :source ∼/.vim/<filename >.vim

To uninstall plugins:

1. Delete 'Plug <url >' line from your vimrc
2. :PlugClean

1.3 Native Plugin Management

Vim-plug provides a solution to the first of the two issues mentioned in the previous section, but not the second. To address the second issue, we can use Vim's native plugin management system to install a plugin and delete the .git repository. That way, we can track the files and any changes in our own git repository. This section follows [**?**]. For a more detailed explanation, see [**?**] or [**?**].

The package feature of Vim 8 follows a pathogen-like model and adds the plugins found inside a custom-path ∼/.vim/pack/ to Vim's runtime path. You can check the version of Vim installed using 'vim –version'. To install using this native feature, we use the following steps:

1. mkdir -p ∼/.vim/pack/plugins/start/
2. cd ∼/.vim/pack/plugins/start/
3. git clone <url>or git submodule add <url >

and that's it! To remove a plugin, simply remove its directory: rm -r ∼/.vim/pack/plugins/start/<plugin_directory_name >if the plugin was cloned. Or, if it was added as a submodule, use:

1. git submodule deinit vim/pack/shapeshed/start/<plugin_directory_name >
2. git rm vim/pack/shapeshed/start/<plugin_directory_name >
3. rm -Rf .git/modules/vim/pack/shapeshed/start/<plugin_directory_name >

To import your vim configuation and plugins to your new machine, simply use the same steps as required for Pathogen:

1. cd ∼
2. git clone <git repo url>∼/.vim
3. ln -s ∼/.vim/vimrc ∼/.vimrc
4. cd ∼/.vim
5. git submodule init

6. git submodule update

and also, to update all your plugins, you again use

1. cd ∼/.vim
2. git submodule foreach git pull origin master

If you have your own plugins that are not cloned from external repositories (i.e. you've written your own .vim files), you can simply create a directory /.vim/plugin and copy your plugins into there.

In conclusion, although Pathogen seems to be the most cumbersome, it could be the case that the servers do not have Vim 8 installed and so the native plugin manager will not work nor do they allow Vim-plug to install plugins. In such a case, Pathogen is your best bet.

1.4 Vimtex

This section discusses the Vimtex [**?**] plugin. See also the vimways article [**?**] for a good overview. To install MikTex into Linux, run

```
sudo apt-get install texlive-full
```

1.4.1 Installing and Running Vimtex

Assuming you are using the Pathogen plugin manager and version controlling your /.vim directory, follow the steps used in Section **??**. Then use ':Helptags' which is Pathogen's method for generating help tags. With this, we can use ':h vimtex' to see the manual for Vimtex. To confirm that the plugin works, type ":VimtexInfo" to see a summary of the tex file.

1.4.2 Compiling a Tex File

The following commands are useful:

- :VimtexCompile # this is a continuous compiler meaning that everytime you save with ":w" it will automatically compile
- :VimtexStop # this stops the continuous compiler
- :VimtexCompileSS # this is a single shot compiler. Note that you have to save your file first
- :VimtexClean # Cleans auxiliary files generated in compilation process

I set the following mappings in my vimrc:

```
autocmd FileType tex nnoremap <F5> :VimtexView<CR>
autocmd FileType tex inoremap <F5> <Esc> :VimtexView<CR>
autocmd FileType tex nnoremap <F6> :w! <bar> :VimtexCompileSS<CR>
autocmd FileType tex inoremap <F6> <Esc> :w! <bar> :VimtexCompileSS<CR>
```

Here are some useful commands I use in my vimrc:

```
" Avoids opening an empty .tx file only to have vimtex recognize it as plain Tex rather than Latex
    let g:tex_flavor = 'latex'

" Use folding. Use zx to unfold and zX to fold all
    let g:vimtex_fold_enabled = 1
```

```
" Toggle Error Window On and Off
    autocmd FileType tex map <F4> \le

" Shortcut for Compiling and Viewing PDF
    autocmd FileType tex nnoremap <F5> :VimtexView<CR>
    autocmd FileType tex inoremap <F5> <Esc> :VimtexView<CR>
    autocmd FileType tex nnoremap <F6> :w! <bar> :VimtexCompileSS<CR>
    autocmd FileType tex inoremap <F6> <Esc> :w! <bar> :VimtexCompileSS<CR>

" VimtexClean on exit
  augroup vimtex_config
    au!
    au User VimtexEventQuit call vimtex#compiler#clean(0)
  augroup END
```

And here are some useful mappings for creating common math related LaTeX tex objects:

```
" Teleportation!
    autocmd FileType tex inoremap <Space><Space> <Esc>/<++><CR>"_c4l

" Inline Stuff
    autocmd FileType tex inoremap ;mm $$<++><Esc>5ha
    autocmd FileType tex inoremap ;bf \textbf{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;it \textit{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;bfs \boldsymbol{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;ct \cite{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;lb \label{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;rf \ref{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;erf (\ref{})<++><Esc>7hf}i

" Environments
    autocmd FileType tex inoremap ;itm \begin{itemize}<CR><CR>\end{itemize}<CR><++><Esc>2kA\item<Space>
    autocmd FileType tex inoremap ;enu \begin{enumerate}<CR><CR>\end{enumerate}<CR><++><Esc>2kA\item<Space>
    autocmd FileType tex inoremap ;aln \begin{align}<CR><CR>\end{align}<CR><++><Esc>2kA

" Math Environments
    autocmd Filetype tex inoremap ;def \begin{definition}<CR><CR>\end{definition}<CR><++><Esc>2kA
    autocmd Filetype tex inoremap ;thm \begin{theorem}<CR><CR>\end{theorem}<CR><++><Esc>2kA
    autocmd Filetype tex inoremap ;lem \begin{lemma}<CR><CR>\end{lemma}<CR><++><Esc>2kA
    autocmd Filetype tex inoremap ;cor \begin{corollary}<CR><CR>\end{corollary}<CR><++><Esc>2kA
    autocmd Filetype tex inoremap ;prp \begin{proposition}<CR><CR>\end{proposition}<CR><++><Esc>2kA
    autocmd Filetype tex inoremap ;prf \begin{proof}<CR><CR>\end{proof}<Esc>1kA

" Math Stuff
    autocmd Filetype tex inoremap ;T ^\mathrm{T}
    autocmd FileType tex inoremap ;sd _{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;su ^{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;frc \frac{}{<++>}<++><Esc>12hf}i
    autocmd FileType tex inoremap ;mbb \mathbb{}<++><Esc>6hf}i
    autocmd FileType tex inoremap ;lrp \left(\right)<++><Esc>12hf(a
    autocmd FileType tex inoremap ;lrs \left[\right]<++><Esc>12hf[a
    autocmd FileType tex inoremap ;lrn \left\lVert\right\rVert<++><Esc>15hi<Space>
```

1.4.3 Forward and Backwards Searching with Synctex

NOTE: As of 22/04/2025, this seems to clash with YouCompleteMe on WSL2. I haven't checked if this is the same for Linux.

In my dotfiles repository [**?**], you can run **install_vimtex_packages.sh** to install the necessary packages for forward and backwards navigation.

This section follows [**?**]. For this, you will need a Vimtex server which allows you to do forward and backward to navigate between corresponding sections of the tex file and the pdf. You will also need the pdf viewer *Zathura*. To install, simply use

```
sudo apt-get install zathura
```

Then, in your vimrc, add the following line

```
let g:vimtex\_view\_method = 'zathura'
```

Finally, following [**?**], you will also need to install 'vim-gtk3' using:

```
sudo apt-get install vim-gtk3
```

To edit a tex file with navigation capabilities:

1. In the directory containing the tex file, use:
   vim –servername <servername><texfile>.tex
2. While in tex file, to jump to the text on the pdf corresponding to the line under your cursor, use:
   <leader>lv
3. Move your mouse cursor over some text on the pdf. Then, to jump to corresponding text on the tex file, use:
   Ctrl + left-click

Note that <leader>lv forward search works without the Vim server; it's the backwards search that requires the Vim server.

Note that if you are using WSL2 in Windows 11, after installing vim-gtk, anything you open in Vim will freeze until you press 'ctrl+c' and it will reappear. You will first need to install and activate VcXsrv; the details of which can be found in Section **??**.

1.4.4 Other Useful References

- Good article overviewing Vimtex [**?**]
- Nice compilation of Vimtex commands [**?**]
- Quick guide on the basics of Vimtex [**?**]
- For a comparison with other Vim LaTeX plugins [**?**]
- Some useful mappings such as the teleportation trick [**?**, **?**].
- Note-taking using Vim and LaTeX for math lectures [**?**]
- Vim and LaTeX on MacOS [**?**]

1.5 IPython

The following mapping may prove useful for starting an IPython terminal in Vim:

```
nnoremap <leader>P :botright vertical terminal ipython --no-autoindent<CR><C-w><left>
```

1.5.1 sendtowindow Plugin

When coding in Python with a Vim terminal on the side running IPython, I use the *sendtowindow* plugin [**?**] to send lines of code to the terminal. See [**?**] for a Reddit post from the author discussing the plugin. It's a very simple plugin that allows the use of vim motions to move lines of text to terminals left, right, above or below the Vim window. I use the following maps:

```
let g:sendtowindow_use_defaults=0
nmap ,sr <Plug> SendRight
xmap ,srv <Plug> SendRightV
nmap ,sl <Plug> SendLeft
xmap ,slv <Plug> SendLeftV
nmap ,su <Plug> SendUp
xmap ,suv <Plug> SendUpV
nmap ,sd <Plug> SendDown
xmap ,sdv <Plug> SendDownV
```

Note that these mappings must not be noremaps (mappings that are non-recursive). For an explanation on why this is the case, please see [**?**]. Any one of these mappings will specify to either send some text in normal mode or in visual mode. When in normal mode, to specify which text to send, use the usual Vim movements. For example, ',sr$' will send from the cursor to the end of the line to the window on the right. However, if you are in the middle of a line, it may become tedious having to first go to the beginning of the line and having to type '$'. Therefore, we can define line objects using the following maps:

```
onoremap <silent> <expr> - v:count==0 ? ":<c-u>normal! 0vg_<CR>" : ":<c-u>normal! V" . (v:count) . "jk<CR>"
onoremap <silent> <expr> i- v:count==0 ? ":<c-u>normal! ^vg_<CR>" : ":<c-u>normal! ^v" . (v:count) . "jkg_<CR>"
```

With '-' alone the indentation is left intact and 'i-' is without indentation. So, for example, ',sr-' will send the line under the cursor to the window on the right with indentation and ',sr8-' will send four lines under the cursor to the window on the right with indentation. Note that this is important for Python as if you try send a for loop with 'i-' and leave out the indentation, then it may not run.

I have extended the plugin with SendTextToTerminalRight command and SendVariableRight() and SendMarkedSectionRight() functions (I've also coded for the other three directions as well). Using these, we can send clear, %reset and run code commands as well as variable and marked sections of code to the IPython terminal.

```
" sendtowindow for IPython in Vim Terminal: clear, reset, run code, run variable, run marked section
    noremap ,sL :SendTextToTerminalRight clear<CR>
    noremap ,sD :SendTextToTerminalRight %reset -f<CR>
    noremap ,sR :w!<CR>:SendTextToTerminalRight run <c-r>%<CR>
    nmap ,sV <Plug>SendVariableRight
    nmap ,sM <Plug>SendMarkedSectionRight
```

For the map that runs the Python code, we leverage the fact that '<c-r>' in command mode will prime pasting from the register and % in the register stores the file name.

The SendMarkedSectionRight() function has been coded such that the section must be marked with 'x' on the top of the section and 'z' on the bottom. The plugin vim-signature [**?**] may come in handy for manipulating the marks. In particular, the plugin displays marks and the command 'dm<mark name>' deletes the mark; so 'dmq' above deletes the mark called 'q'.

1.5.2 Slimux plugin

There are many plugins that allow interaction between Vim and tmux. For example, there is Vim-Slime [**?**] and Vimux [**?**]. The one I use is Slimux [**?**]. There is a blog post about it here [**?**]. As discussed in the blog post, Slimux differs from Vimux in that it is more disjoint from tmux. In particular, Vimux will create a pane on which you

must run your commands whereas Slimux allows you to select the pane you wish to use. Also, unlike Vim-Slime, you do not need to manually type in the pane to select it; in Slimux an interactive prompt is given. It's also worth noting that Vim-Slime supports many terminal multiplexers such as GNU Screen, kitty and Vim's native terminal.

However, with Slimux, there is an issue with sending commands to IPython where the indentation is constantly carried over [**?**]. There are three proposed fixes [**?**, **?**, **?**]. The first modifies python.vim, the second modifies both slimux.vim and python.vim by implementing IPython's cpaste function and the third modifies a single line of slimux.vim. Since the repo owner hasn't pulled any of these fixes, I opted for the third fix which requires changing line 328 of slimux.vim in the SlimuxSendCode() function:

```
let b:code_packet["text"] = a:text
```

to:

```
b:code_packet["text"] = "\e[200~" . a:text . "\e[201~\r\r.
```

There is also an issue with the SlimuxSendKeys() function. This function sends keys to the terminal using the 'tmux send-keys' syntax. A simple fix is to change line 228 of slimux.vim from

```
call system(g:slimux_tmux_path . ' send-keys -t " . target . " " . keys)
```

to

```
call system(g:slimux_tmux_path . " send-keys -t " . target . " " . keys)
```

It's surprising that this small typo went unnoticed. Note that this repository has not been updated in years and so all pull requests have not been fulfilled. Therefore, if you wish to track the fixes you've made, you may want to install the plugin manually so that it does not follow the original repository. This was disussed in Section **??**.

I use the following mappings:

```
" Primer for Vim motions to select text to be sent
    nmap ,t <Plug>SlimuxREPLSendWithMotion

" Send text selected in visual mode
    vnoremap ,t :SlimuxREPLSendSelection<CR>

" Send terminal commands
    nnoremap ,tX :SlimuxShellPrompt<CR>
    nnoremap ,tT :SlimuxShellRun<Space>
    nnoremap ,tE :SlimuxShellRun exit<CR>

" Send keys using the 'tmux send-keys' syntax
    nnoremap ,tK :SlimuxSendKeys<Space>
    nnoremap ,tC :SlimuxSendKeys<Space>c-c<CR>

" Slimux for IPython in tmux terminal : IPython, clear, reset, run code, run variable, run marked section
    nnoremap ,tP :SlimuxShellRun ipython<CR>
    nnoremap ,tL :SlimuxShellRun clear<CR>
    nnoremap ,tD :SlimuxShellRun %reset -f<CR>
    nnoremap ,tR :w!<CR>:SlimuxShellRun run <c-r>%<CR>
    nmap ,tV <Plug>SlimuxREPLSendVariable
    nmap ,tM <Plug>SlimuxREPLSendMarkedSection
```

where SlimexShellRun awaits a command to send to shell and SLimuxSendKeys sends key inputs using the 'tmux send-keys' syntax. The SlimuxREPLSendWithMotion was not part of the original slimux plugin. My code is as follows:

```
function! s:GetTextWithMotion(type)
  let s:saved_registert = @t

  " Obtain wanted text
```

```
  if a:type ==# "char"
    keepjumps normal! `[v`]"ty
    call setpos(".", getpos("']"))
  elseif a:type ==# "line"
    keepjumps normal! `[V`]$"ty
    call setpos(".", getpos("']"))
  endif
  let text = @t

  " Restore register
  let @t = s:saved_registert

  return text
endfunction

function! s:SlimuxREPLSendWithMotion(type)
   let text = s:GetTextWithMotion(a:type)
   call SlimuxSendCode(text)
endfunction

nnoremap <silent> <Plug>SlimuxREPLSendWithMotion :<C-U> set operatorfunc=<SID>SlimuxREPLSendWithMotion<CR>g@
```

The 'g@' sets the mark '[' at the beginning of the motion and ']' at the end of the motion. This can then be leveraged in functions. The '@' in the context of functions refers to the register.

1.5.3 Other Useful References

- A good Reddit thread on workflow with Vim and IPython can be found at [**?**]. Note that the majority of the responses indicate that they use vim-slime and tmux.
- The vim-tmux-runner plugin [**?**] may be worth checking out.
- The vim-ipython-cell plugin [**?**] may be worth checking out. There is a reddit post [**?**] by the author on this plugin. Note that this plugin is leverages vim-slime. According to the author, the main contribution of this plugin is that it provides many ways to define and run cells, even using Vim marks.
- Another lightweight workflow can be found in [**?**]. For this, you will need three plugins: Vimux [**?**], vim-pyShell [**?**] and vim-cellmode [**?**]. However, I have trouble getting this to work. Starting and stopping a pyShell session works fine as well as sending one line of code, but I have issues sending over cells.

1.6 Autocompletion with YCM

When deciding between *You Complete Me (YCM)* [**?**] and *Conquer of Completion (CoC)* [**?**], I found the following the discussion in [**?**] where a maintainer of YCM details the design philosophy of YCM in the comments. Specifically, one of the maintainers (puremourning) states that YCM prioritizes consistency, testing, stability and performance over a range of functions such as snippet support of insertion of parentheses. This appeals to me because honestly all I want is to be able to jump to definition. Finally, as of writing this, I prefer to out-source error checking to the compiler and linting to some external linting program. Basically, I want my text-editor to just edit text and help me navigate my code. With this in mind, YCM seems ideal as you can easily configure and disable all these extra functions.

Another helpful resource was the video [**?**] where the speaker uses both as YCM seems to perform better with TypeScript whilst CoC performs better with C++. I'm not super sold on my choice, it was a bit of a coin flip in the end, but I decided to go with YCM. Another possible option was converting to Neovim and using it's native LSP, but I decided to hold out on the conversion until Vim 9 is released.

1.6.1 Installation

I'm not entirely sure why the repository [?] suggests to use Vundle accompanied by some complicated instructions. I instead found on [?] that you can simply include

```
Plug 'ycm-core/YouCompleteMe', { 'do': './install.py' }
```

into your vimrc. If you wish to use autocompletion for C, you'll need to instead use

```
Plug 'ycm-core/YouCompleteMe', { 'do': './install.py --clang-completer'}
```

and requires a few extra steps which is outlined in Section **??**.

The installation process requires you to have 'make' and 'cmake' which can be installed using the commands:

```
sudo apt-get -y install build-essential
sudo apt install -y make
sudo apt-get -y install cmake
```

and also, even if you already have Python 3.8 installed, you may need:

```
sudo apt-get install python-dev-is-python3
```

If you repeatedly see the error:

```
YouCompleteMe ERROR: Python headers are missing in /usr/include/python3.9.
```

there may be an issue with Python. You can fix issues by running

```
sudo apt install --reinstall python3 python python3-minimal --fix-broken
```

If this all doesn't work, you may need to simply use

```
sudo apt-get -y update
```

and reboot your terminal.

1.6.2 Mappings

Here are a few additions I made to the vimrc, beginning with the jump commands:

```
nnoremap <Leader>gd :YcmCompleter GoTo<CR>
nnoremap <Leader>gr :YcmCompleter GoToReferences<CR>
nnoremap <Leader>gi :YcmCompleter GoToImplementation<CR>
nnoremap <Leader>gy :YcmCompleter GoToType<CR>
nnoremap <Leader>rr :YcmCompleter RefactorRename<space>
```

The repository README provides a fantastic description of each of these commands.

Additionally, I found configuring the behaviour to be a very elegant procedure:

```
" Turn off automatic summoning of completion suggestions
    let g:ycm_auto_trigger = 0

" Completion Pop-Up Settings
    set pumheight=10
    set completeopt+=popup
    set previewpopup=height:10,width:60,highlight:PMenuSbar
    set completepopup=height:15,width:60,border:off,highlight:PMenuSbar
    let g:ycm_max_num_candidates_to_detail = 5
    let g:ycm_max_num_identifier_candidates = 5

" Disable preview at bottom of terminal when using completion
    set completeopt-=preview
    let g:ycm_add_preview_to_completeopt = 0
```

```
" If using preview, close window after using a selection
    let g:ycm_autoclose_preview_window_after_insertion = 1
    let g:ycm_autoclose_preview_window_after_completion = 1

" Disable summoning of documentation when hovering
    let g:ycm_auto_hover = ''

" Diagnostics UI settings
    let g:ycm_show_diagnostics_ui = 0
    let g:ycm_enable_diagnostic_signs = 0
    let g:ycm_enable_diagnostic_highlighting = 0
    let g:ycm_echo_current_diagnostic = 0
    let g:ycm_update_diagnostics_in_insert_mode = 0

" Commands to control completion pop up
    let g:ycm_key_invoke_completion = '<C-Space>'
    let g:ycm_key_list_stop_completion = ['<C-c>']

" Open Search
    nmap <Leader>F <plug>(YCMFindSymbolInWorkspace)

" Instead of triggering after hovering, summon documentation using a command
    nmap <Leader>D <plug>(YCMHover)

" Jumps
    nnoremap <Leader>G :YcmCompleter GoTo<CR>
    autocmd FileType python nnoremap <Leader>G :YcmCompleter GoToType<CR>
    nnoremap <Leader>gd :YcmCompleter GoToDefinition<CR>
    nnoremap <Leader>gy :YcmCompleter GoToType<CR>
    nnoremap <Leader>gi :YcmCompleter GoToImplementation<CR>
    nnoremap <Leader>gs :YcmCompleter GoToSymbol<CR>
    nnoremap <Leader>gr :YcmCompleter GoToReferences<CR>
    nnoremap <Leader>rr :YcmCompleter RefactorRename<space>
```

Some notes on this configuration. Firstly, for Python, both **GoTo** and **GoToDefinition** just sends you to the import at the top of the file. Intead, **GoToType** sends you to the file in which it was created. Therefore, I instead specified a special map for when in Python files.

Interestingly enough, YCM does not offer the ability to limit the number of suggestions displayed [**?**]. To be honest, I'm not entirely sure what the options:

```
let g:ycm_max_num_candidates_to_detail = 5
let g:ycm_max_num_identifier_candidates = 5
```

do. Instead, you have to use a vanilla Vim configuration and decrease the size of the window with

```
set pumheight=10
```

Now regarding the verbosity of the suggestions and the redundancy of its appearance in the popup, there seems to be lengthy discourse over this in [**?**]. It doesn't seem like puremourning is in favour of truncating the signatures following the idea that, without them, you can only see details of the currently selected suggestion.

1.6.3 Python

You can find some detailed information about YCM with Python in the 'Python Semantic Completion' section of [**?**]. One idiosyncracy of Python YCM must deal with is the use of virtual environments. There are two ways offered by YCM:

- A local solution. Create a **.ycm_extra_conf.py** file at the root directory of every project containing

```
def Settings( **kwargs ):
return {
    'interpreter_path': '/path/to/virtual/environment/python'
}
```

- A global solution. In your vimrc, add:

```
let g:ycm_python_interpreter_path = ''
let g:ycm_python_sys_path = []
let g:ycm_extra_conf_vim_data = [
    \  'g:ycm_python_interpreter_path',
    \  'g:ycm_python_sys_path'
    \]
let g:ycm_global_ycm_extra_conf = '~/global_extra_conf.py'
```

and create the ∼/**global_extra_conf.py** file containing:

```
def Settings( **kwargs ):
    client_data = kwargs[ 'client_data' ]
    return {
        'interpreter_path': client_data[ 'g:ycm_python_interpreter_path' ],
        'sys_path': client_data[ 'g:ycm_python_sys_path' ]
    }
```

What this does is it sets all the paths you need for code jumps. Indeed, with your file open in Vim, if you use **:YcmDebugInfo**, you'll be able to see something like:

Figure 1.1: YCM debug information

Notice in particular the 'Extra configuration path' which I placed within ∼/**.vim/plugins/YouCompleteMe**.

One thing to take note of is that the code jumps may not work if you modify your **sys.path** within the code. For example:

```
import os
import sys
sys.path.insert(0, os.path.realpath('../../../src'))
sys.path.insert(0, os.path.realpath('..'))

# Import src code
from utils_io.filepaths_vae import FilePathsTraining

# Import Project Utilities
from utils_project.filepaths_project import FilePathsProject
```

Here, I have modified the path such that I can abbreviate the import of **../../../src/utils_io/filepaths_vae** to simply **utils_io/filepaths_vae**. The issue here is, in Figure **??**, notice that **/home/hwangoh/codes/uq-vae/codes/src** is not included in the 'Python path' segment. Therefore, unless one manually adds to their vimrc:

```
:let $PYTHONPATH="/path/to/project/directory"
```

or runs this in Vim and and then runs **:YcmRestartServer**, the code jumps will not work. Maybe one can add

```
let $PYTHONPATH .= getcwd()
```

into their Vimrc and somehow jank their way into automating the inclusion of the path. But ultimately, YCM works fine and it is the programmer's responsibility to properly manage how they import their paths.

1.6.4 C

As mentioned in Section **??**, if you wish to use autocompletion for C, you'll need to include

```
Plug 'ycm-core/YouCompleteMe', { 'do': './install.py --clang-completer'}
```

into your vimrc. There are a few more steps required to set up autocompletion; section 'C-family Semantic Completion' of the README in [**?**] has the details required. For one, at the root directory of every C project, you will need to include a **.ycm_extra_conf.py** file. A template for this can be found in

```
~/.vim/plugged/YouCompleteMe/third_party/ycmd/.ycm_extra_conf.py
```

Once this is in your root directory, if you open Vim then a prompt will appear asking if you wish to load it.

If this does still not work, ChatGPT suggests:

1. In the root directory of your C project, create a **CMakeLists.txt** file that includes at least the following

```
cmake_minimum_required(VERSION 3.10)
project(myproject VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add compile options
add_compile_options(-Wall -Wextra -Werror)

# generate compile_commands.json
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# specify the source files
file(GLOB_RECURSE SRC_FILES "src/*.c" "src/*.cpp")
add_executable(myproject ${SRC_FILES})

# specify include directories
include_directories(include)

# specify any libraries to link against
target_link_libraries(myproject -lm)
```

2. If it already does not exist, create a **build** directory in your root directory and navigate into it:

```
mkdir build
cd build
```

3. In the **build** directory, run the following command:

```
cmake .. -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

which creates a **compile_commands.json** file in the **build** directory.

After all these steps, YCM should allow for autocompletion and code jumps.

1.7 Autocompletion with CoC

Another option for autocompletion is *Conquer of Completion (CoC)* [**?**]. If you're using VimPlug, to install, simply add:

```
Plug 'neoclide/coc.nvim', {'branch': 'release'}
```

to your vimrc. This doesn't work immediately out of the box. You need to install language specific extensions. For example, add this to your vimrc:

```
let g:coc_global_extensions = ['coc-json', 'coc-tsserver', 'coc-python']
```

to install the json, TypeScript and Python extensions. See [**?**] for more information about extensions.

The repo [**?**] provides an example configuration. I don't exactly know what it all does so, I simply added the ones that I understand. These include:

- For code navigation:

```
" GoTo code navigation.
nmap <silent> gd <Plug>(coc-definition)
nmap <silent> gy <Plug>(coc-type-definition)
nmap <silent> gi <Plug>(coc-implementation)
nmap <silent> gr <Plug>(coc-references)
```

- To show an inline preview of the definitions:

```
" Use K to show documentation in preview window.
nnoremap <silent> K :call <SID>show_documentation()<CR>

function! s:show_documentation()
    if (index(['vim','help'], &filetype) >= 0)
        execute 'h '.expand('<cword>')
    elseif (coc#rpc#ready())
        call CocActionAsync('doHover')
    else
        execute '!' . &keywordprg . " " . expand('<cword>')
    endif
endfunction
```

2
i3wm

i3wm is a tiling windows manager for Linux. The benefits of using tiling windows managers in general is maximization of your screen real estate. Additionally, i3wm uses Vim-like keybindings to enable you to navigate your machine almost entirely using your keyboard. Here are some basic tips:

- The official user guide can be found at [**?**]
- Good Youtube Video on Configuring i3: [**?**]
- To install: sudo apt-get install i3
- Use mod+d (where mod is set by default to be your Windows key) to open a search bar for your programs. For example, to run file explorer: mod+d then type "nautilus"
- To open Ubuntu settings, use "env XDG_CURRENT_DESKTOP=GNOME gnome-control-center". You'll find that you may need to Google many basic commands for navigating Ubuntu as many of the default icons are not available.

2.1 Standard Configuration

2.1.1 Generate Default Config File and Symlink

1. You should be prompted to do so immediately upon first run of i3. However, you can also do so by using "i3-config-wizard" in the terminal.
2. The config file is located in ∼/.config/i3/config
3. To create a symlink, CUT, paste and rename ∼/.config/i3/config to your desired folder (say ∼/.vim/i3_config) then use "ln -s ∼/.vim/i3_config ∼/.config/i3/config"
4. After making edits, use "mod+shift+r" to restart i3

2.1.2 Configure i3 for Dual Monitors

1. In terminal, use "xrandr" to see your connected monitors
2. Then use "xrandr –output HDMI-1 –auto –left-of eDP-1" to set your HDMI-1 monitor to the left of your eDP-1 monitor
3. To set it so that it starts up in this way, put "exec –no-startup-id xrandr –output HDMI-1 –left-of eDP-1 –auto" in the config file

2.1.3 Wallpaper

1. sudo apt-get install feh
2. Add to config: "exec –no-startup-id feh –bg-fill ∼/.vim/Wallpapers/my_wallpaper.jpg"

2.1.4 Remove Title and Borders

Add the following to config file:

1. new_window pixel 0
2. new_float pixel 0

2.1.5 Volume and Brightness Controls for Laptop

From [**?**]:

1. sudo apt-get install xbacklight alsa-utils pulseaudio
2. Add the following to config:

```
# Pulse Audio Controls
bindsym XF86AudioRaiseVolume exec --no-startup-id pactl set-sink-volume 0 +5% # increase volume
bindsym XF86AudioLowerVolume exec --no-startup-id pactl set-sink-volume 0 -5% # decrease volume
bindsym XF86AudioMute exec --no-startup-id pactl set-sink-mute 0 toggle # mute volume

# Sreen brightness controls
bindsym XF86MonBrightnessUp exec xbacklight -inc 20 # increase screen brightness
bindsym XF86MonBrightnessDown exec xbacklight -dec 20 # decrease screen brightness
```

2.1.6 Screenshots

From [**?**]:

1. sudo apt-get install scrot
2. sudo apt-get install xclip
3. Add this to config file:
   - bindsym –release Print exec scrot 'screenshot_%Y%m%d_%H%M%S.png' -e 'mkdir -p /Downloads && mv $f ∼/Downloads && xclip -selection clipboard -t image/png -i ∼/Downloads/`ls -1 -t ∼/Downloads — head -1`'
   - bindsym –release Shift+Print exec scrot -s 'screenshot_%Y%m%d_%H%M%S.png' -e 'mkdir -p ∼/Downloads && mv $f ∼/Downloads && xclip -selection clipboard -t image/png -i ∼/Downloads/`ls -1 -t ∼/Downloads/screenshots — head -1`'

Note that there is some strange issue on my machine where xclip takes up 99% CPU usage. Therefore, after each screenshot, you'll need to terminate xclip via the gnome-system-manager.

2.1.7 Shutdown Options:

From [**?**]:

1. Add the following to your config file:

```
# Set shut down, restart and locking features
bindsym $mod+0 mode "$mode_system"
set $mode_system (l)ock, (e)xit, switch_(u)ser, (s)uspend, (h)ibernate, (r)eboot, (Shift+s)hutdown
mode "$mode_system" {
bindsym l exec --no-startup-id ~/.vim/i3_exit.sh lock, mode "default"
bindsym s exec --no-startup-id ~/.vim/i3_exit.sh suspend, mode "default"
bindsym u exec --no-startup-id ~/.vim/i3_exit.sh switch_user, mode "default"
bindsym e exec --no-startup-id ~/.vim/i3_exit.sh logout, mode "default"
bindsym h exec --no-startup-id ~/.vim/i3_exit.sh hibernate, mode "default"
bindsym r exec --no-startup-id ~/.vim/i3_exit.sh reboot, mode "default"
bindsym Shift+s exec --no-startup-id ~/.vim/i3_exit.sh shutdown, mode "default"
# exit system mode: "Enter" or "Escape"
bindsym Return mode "default"
bindsym Escape mode "default"
```

2. Create and make executable ∼/.vim/i3_exit.sh with the following contents:

```
#!/bin/sh
lock() {
    i3lock
}
case "$1" in
    lock)
        lock
        ;;
    logout)
        i3-msg exit
        ;;
    suspend)
        lock && systemctl suspend
        ;;
    hibernate)
        lock && systemctl hibernate
        ;;
    reboot)
        systemctl reboot
        ;;
    shutdown)
        systemctl poweroff
        ;;
    *)
        echo "Usage: $0                                                {lock|logout|suspend|hibernate|
     reboot|shutdown}"
        exit 2
esac

exit 0
```

2.1.8 Assigning Programs to Specific Windows

From [**?**] at around 25:27. To check the class of an application:

1. Open terminal

2. Use "xprop"
3. With cursor now as a cross-hair, click on application
4. WM_CLASS should display on the terminal as the second value
5. Add to config: assign [class="google-chrome"] $workspace1

2.1.9 i3 Status Bar

I think by default, installing i3 on Ubuntu gives version 2.11. To check version, use "i3status –version" in terminal. If not installed at all, use "sudo apt-get install i3status".

Note that the sample config file in the man pages uses stuff that doesn't exist in version 2.11. I followed the config file [**?**] from [**?**] which also uses some options that do not exist in version 2.11. I couldn't figure out how to upgrade but since I only want the bare minimum, I didn't bother and just wrote a very minimal config file.

Following [**?**]:

- create the directory /.config/i3status
- in your i3 config file add:

```
bar {
        status_command i3status --config ~/.config/i3status/config
}
```

- create i3status.conf somewhere (suppose in .vim folder)
- ln -s ∼/.vim/i3status.conf ∼/.config/i3status/config
- Edit the config then refresh i3 using mod+SHIFT+r
- If there is an error, type "i3status -c" in terminal and the error will be displayed

2.1.10 Setting Mouse Speed

Setting Mouse speed:

1. "xinput list" in terminal to get list of connected devices. Find the id for mouse
2. "xinput list-props <id>" to get list of settings for the device.
3. Find the bracketed number for "libinput Accel Speed (<number>)"
4. "xinput –set-prop <id ><number ><value >" for value in [-1.0, 1.0]. This reduces the mouse speed

2.1.11 Floats

To ensure that matplotlib opens as float, to your i3_config, add:

for_window [class="matplotlib"] floating enable

2.2 i3 in Windows

There are quite a few AutoHotkey repos for binding keys to switch between desktops. However, many of them do not seem to work due to perhaps updates to the OS. The one I found to work best can be found here: [**?**].

However, it doesn't seem to be the best idea to emulate i3 by switching between desktops. The key difference I found is that you cannot assign separate monitors a separate desktop; each desktop takes into account all monitors. For that reason, I have simply opted to keep the default Win keybindings which correspond to the task bar order. Instead, I have elected to implement the other i3wm bindings such as Win-f to toggle maximize and Win-m to toggle moving windows between screens.

The two scripts I use to emulate i3 in Windows are:

1. **.vim/config_files/i3wm-windows-emulator/i3_config.ahk**
2. **.vim/config_files/i3wm-windows-emulator/i3wm_windows_emulator.ahk**

Note that the **i3wm_windows_emulator.ahk** script contains alot of backend functions. The config itself is contained in **i3_config.ahk**, below is a snippet of the contents:

```
; === Disable Windows Visual Effects ===
RegWrite, REG_DWORD, HKEY_CURRENT_USER, SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\VisualEffects,
    AnimateMinMax, 2

; === Remapping Capslock to Esc ===
SetCapsLockState, alwaysoff
Capslock::Esc

; ====================
; === Key Bindings ===
; ====================
; 1. Any lines starting with ; are ignored
; 2. After changing this config file run script file "i3wm_windows_emulator.ahk"
; 3. Every line is in the format HOTKEY::ACTION

; === SYMBOLS ===
; !   <- Alt
; +   <- Shift
; ^   <- Ctrl
; #   <- Win
; For more, visit https://autohotkey.com/docs/Hotkeys.htm

; === Window Manipulation ===
#+q::quitWindow()
#+f::toggleMaximize()
#O::#x
#+m::#+Left
; The following requires more complex code as shift is registered when Arrow is sent
#+j::
Send {Blind}{Shift up}{Down}{Shift down}
return
#+k::
Send {Blind}{Shift up}{Up}{Shift down}
return
#+h::
Send {Blind}{Shift up}{Left}{Shift down}
return
#+l::
Send {Blind}{Shift up}{Right}{Shift down}
return
```

```
; === Open Programs ===
#Enter::openAndPositionTerminal()
#PrintScreen::Run C:\Program Files\ShareX\ShareX.exe

; === File Explorer ===
#z::openAndPositionExplorer("\\wsl$\Ubuntu\home\hwangoh")
#+z::openAndPositionExplorer("C:\")
```

For more details about the code, see Section **??**.

To enable these settings to take action on your machine, you want to open the **i3wm_windows_emulator.ahk** script and not the **i3_config.ahk** script.

2.2.1 Modifier Subtleties

Modifier keys refer to the control, alt, shift and windows keys. The effect of these modifiers persist through remappings. For example, the remap a::b would be imply typing the capital A by either holding shift or having capslock on would print the capital B. Further, holding control and pressing the 'a' key would output holding control and pressing the 'b' key [**?**]. Under the hood, the "::" remapping command uses the *Blind* mode which avoids releasing the modifier keys [**?**]. That is, if the user continues to hold down the modified key, the remapping program will not logically register it as released. For example, the following remapping

```
+s::Send {Blind}abc
```

would send ABC rather than abc because the user is holding down Shift symbolized by the + symbol. Another example is the remapping

```
^space::Send {Ctrl up}
```

automatically pushes control back down if the user is sill physically holding control whereas

```
^space::Send {Blind}{Ctrl up}
```

allows control to be logically up even though it is physically down.

Returning to the emulation of i3, suppose we want to map Win+Shift+h to Win+Left which moves a window left. Since shift is held down, it will instead logically register as move the window to the monitor to the left. Therefore, we require the following mapping:

```
#+h::
Send {Blind}{Shift up}{Left}{Shift down}
return
```

where the # symbolizes the Win key and the + symbolizes the shift key. The last {Shift down } allows for the use of Win+Shift+h,..h,h,h to continuously move the windows without needed to repress Win+Shift each time.

2.2.2 Cursor to Follow Active Window

One annoying thing in Windows when you have multiple screens is that if you use a hotkey to focus on a window, the mouse cursor doesn't follow. The script **.vim/config_files/i3wm-windows-emulator/mouse_cursor_follows_focus.ahk** implements the desired behaviour where the mouse cursor centers on the active window. The contents are displayed below:

```
Gui +LastFound

lastMouseClickTime := 0
hWnd := WinExist()
```

```
DllCall("RegisterShellHookWindow", UInt, hWnd)
msgNum := DllCall("RegisterWindowMessage", Str, "SHELLHOOK")
OnMessage(msgNum, "OnShellMessage")
OnMessage(WM_MOUSEMOVE:=0x0201, "OnMouseDown")
Return

OnShellMessage( wParam, lParam )
{
  global
  ; HSHELL_WINDOWACTIVATED | HSHELL_RUDEAPPACTIVATED
  If (wParam = 4 or wParam = 32772) {
    ; ignore when dragging
    GetKeyState, mouseDown, LButton
    if (mouseDown <> "D" and A_TickCount - lastMouseClickTime > 500) {
      ; delay a tiny bit to ignore taskbar focus on Win+Number switching
      Sleep, 10
      CoordMode, Mouse, Screen
      WinGetPos, wx, wy, width, height, A

      ; puts the cursor in the upper right corner of the active window, tweak to your needs
      mx := Round(wx + width * 0.5)
      my := Round(wy + height * 0.5)

      DllCall("SetCursorPos", int, mx, int, my)
    }
  }
}

*~LButton::
  lastMouseClickTime := A_TickCount
Return

*~RButton::
  lastMouseClickTime := A_TickCount
Return

*~MButton::
  lastMouseClickTime := A_TickCount
Return
```

2.2.3 Autostart AutoHotKey Scripts

There are two methods to setting your configuration to open on start up. I found that on different machines, at least one of the two methods will work.

The first method is the simplest. Simply create a shortcut to your .ahk file and place it in the "C:\Users\Username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Start-up" directory.

The second method is more complicated but, in general, more versatile. This requires the use of the Task Scheduler [**?**]:

1. Open the start menu and search "Task Scheduler"
2. On the right panel, select "Create Basic Task" option
3. Name the task anything you like and click next
4. Select "When I log on" option and click next

5. Select “Start a program” option and click next
6. In the “Program/script” text box, add the path to your .ahk file without the quotation marks
   i.e. C:\Users\Hwan\i3wm-windows-emulator\i3wm_windows_emulator.ahk.
   Note that you may want to place your .ahk files in your Windows directories as oppose to your WSL if you want them to be started following log on. This is due to the delay in starting WSL following logon; which may cause the .ahk files to not be executed.
7. Click Next then Finish
8. To check if it works, find your newly added task in the “Task Scheduler Library” and click “Run”
9. If you’re on a laptop, check the “Conditions” tab in the “Power” section and uncheck the box “Start the task only if the computer is on AC power”.

Note that you can start up multiple scripts with this method. For example, I also have the ‘windows_follows_mouse_cursor.ahk’ script which makes the mouse cursor center on whatever window is the focus. This is expecially useful when you have multiple monitors. In this case, you can run both scripts simultaneously (for some reason, including the windows_follows_mouse_cursor.ahk code into the i3wm_windows_emulator.ahk code didn’t work) and both hotkey configurations seem to take effect just fine.

2.2.4 Disabling Win+l

Since this is not exactly a windows tiling manager, I have also opted to use the Win+arrow key functionality to organize my windows. However, I would like to do so using the sacred Vim keys. A major issue is that Win+l locks the computer. It’s a bit of a hassle to disable this as you have to disable the whole computer lock function, but it can be done using the following steps [**?**]:

1. Use Win+r to open the run app
2. Type ”regedit” and open
3. Navigate to ”HKEY_CURRENT_USER/SOFTWARE/Microsoft/Windows/CurrentVersion/Policies/”
4. Right-click the ”Policies” key and choose ”New >Key”. Name the new key ”System”. Note that this may already be there. If so, this step can be skipped Right-click the ”System” key and choose ”New >DWORD (32-bit) Value”. Name this ”DisableLockWorkstation”
5. Double-click DisableLockWorkstation. Change the value from 0 to 1

These changes should take place immediately without needing to restart. Now anything bound to Win+l can be used. I should mention that in general this is inadvisable as you are not only disabling the Win+l binding but your computer’s ability to lock itself (even after waking sleep mode).

3 WSL2

In this section, we utilize the Windows Subsystem for Linux. This creates an extremely lightweight virtual machine in within your Windows OS that emulates the Linux kernel with a home drive.

3.1 Installing WSL2

Note, as of today (27th October 2021), the site [**?**] states that you can simply run

```
wsl --install
```

in PowerShell to install it. By default, it will:

- Download the latest Linux kernel
- Set WSL 2 as your default
- Install the latest Ubuntu distribution.

You can, however, choose a different distribution to install by running

```
wsl --install -d <Distribution Name>
```

To see a list of distributions available for download and installation, run:

```
wsl --list --online
```

This documentation assumes one is using Ubuntu 20.04.6 LTS which can be installed with the command

```
wsl --install -d Ubuntu-20.04
```

Below are the old instructions for installing WSL2.

To get i3 on Windows without a virtual machine (but technically a very lightweight one is used), use WSL2. On the site [**?**] , follow the "Manual Installation Steps" for when you are not using the Windows Insiders build.

1. Open PowerShell as Administrator. If you use the Windows Terminal which can be obtained from the Microsoft Store, you can pin it to taskbar then press "shift+right-click" on the icon and click "Run as administrator".
2. To enable the Windows Subsystem for Linux, in PowerShell, run

   ```
   dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
   ```

3. Update your windows version if necessary
4. To enable the Virtual Machine feature, in PowerShell, run

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

5. Restart your machine

6. To obtain WSL2, download the Linux kernel update package. The link for the download is available on step 4 of the website.

7. Before installing a new Linux distribution, set WSL2 as the default version. To do this, in PowerShell, run

```
wsl --set-default-version 2
```

8. Go to the Microsoft Store and install your favourite Linux distribution. If the "Install" button does not seem to work after you click "Get", you can go to your library and click "Install" there.

9. Open Ubuntu and select your username and password. Username need not match your Windows OS user name

10. To check that you have the correct version of WSL, in PowerShell run

```
wsl -l -v
```

11. Run

```
sudo apt update
sudo apt upgrade
sudo apt-get install build-essential'
```

    so that you have access to the standard operations.

From here, your home directory can be found at

```
\\wsl.localhost\Ubuntu-20.04\home\<username>
```

(with backslashes instead of forward slashes as its a Windows directory address)

3.1.1 Errors

Here are some errors you might encounter and how to resolve them:

- If you have an older machine, you may encounter "WslRegisterDistribution failed with error: 0x80370102 and error 0x80070002" or other similar numbers. To remedy this, you may need to do two modifications to your BIOs. To access your BIOs modification menu, first restart your computer and continuously hit 'F2' or some other button, depending on your CPU. When there, disable "Limit CPUID Maximum", then enable "Virtualization Technology". Other names may be used for "virtualization" such as "SVM".

- For the error " WslRegisterDistribution failed with error 0xc03a001a", you may need to disable compression in a specific folder. Following runner-ed's answer in https://github.com/microsoft/WSL/issues/4299: Find the package under C:/Users/<your user name>/AppData/Local/Packages and right click the folder, check advanced options and disable compression. Run the launch again.

3.1.2 Basic Operations and Config

Now to run Ubuntu in the Windows terminal. With your Windows Terminal pinned to the start bar, simply right-click and select Ubuntu. To set Ubuntu as the default, open Windows Terminal (can be opened to PowerShell or anything) and press "ctrl+,". Then simply change "Default profile" to Ubuntu.

Now that you're operating in the Linux kernel, Vim should also be ready to go! However note that, by default, you should be in your mounted Windows C drive. Simply type 'cd ∼' to change to the home directory. From here, you can git clone your Linux .vim config and continue exactly as if you're on a Linux machine!

To map the capslock key to escape, I recommend the following AutoHotKeys command:

```
SetCapsLockState, alwaysoff
Capslock::Esc
```

3.2 GWSL

NOTE: As of 22/04/2025, this seems to be no longer required. Forward and backwards search works out of the box with WSL and vim-gtk3 as mentioned in Section **??**.

GWSL is an XServer that allows you to easily run graphical Linux Apps. In particular, if you plan on using Vimtex in WSL, you'll need some method of viewing the compiled PDF file and also performing forward and backwards search as detailed in Section **??**. GWSL is obtainable from the Microsoft Store for $7 NZD. Note that Microsoft is, as of 17th of August 2023, developing WSLg but unfortunately it doesn't behave as a typical Windows window should.

There are some additional steps one needs to follow after installing GWSL to enable forward and backwards search. In particular, you may notice that forward search works fine but backwards search fails. This follows ccelik97's answer in the Reddit post [**?**]:

1. Firstly, if you are using a version of Ubuntu that has WSLg enabled, then you'll need to disable it. Otherwise, GUI apps will run in WSLg. This is done by adding the following line to your **.wslconfig** file:

   ```
   [wsl2]
   guiApplications=false
   ```

   You'll need to reboot your computer in order for this to take effect.

2. Next, in your **.bashrc** file, add:

   ```
   export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk '{print $2}'):0.0
   ```

3. You may also need 'xdotool':

   ```
   sudo apt-get install xdotool
   ```

After this you should be good to go, so long as you have followed Section **??** which covers the installation of Zathura such that forward and backward search is possible.

3.3 VcXsrv

NOTE: As of 22/04/2025, this seems to be no longer required. Forward and backwards search works out of the box with WSL and vim-gtk3 as mentioned in Section **??**.

NOTE: that as of 17th of August 2023, WSL2 Ubuntu 20.04 has updated so that the method discussed in this section no longer works! Please refer to the GWSL section **??** which is built on top of VcXsrv.

Unix systems use the X Window System as their standard GUI environment. Under X, the "server" is the program that runs on your system to draw windows while a "client' is a graphical program. The naming convention is a little confusing to most computer users but, trust me, it makes sense. While the Windows Subsystem for Linux is a very well done method for running Linux software on a Windows system, it lacks an X server so it is unable to run GUI applications by itself. VcXsrv is an X server for Windows that's written to compile as a native Windows application using Microsoft's Visual C++ (hence the name, Visual C++ X Server).

The main motivation for using VcXsrv is so that we can utilize Vimtex+Zathura+Synctex for forward and backwards searching while writing PDF documents as discussed in Section **??**. The initial idea to incorporate VcXsrv for this was obtained from [**?**]. However, the installation process is quite involved and not fully captured in the aforementioned medium post. We detail the installation process now.

3.3.1 Installation

1. Go to the VcXsrv repo at [**?**] and follow the "Releases" link.

2. Download the "vcxsrv.1.17.0.0-3.x2go.arctica.installer.exe" whichever the latest release version is

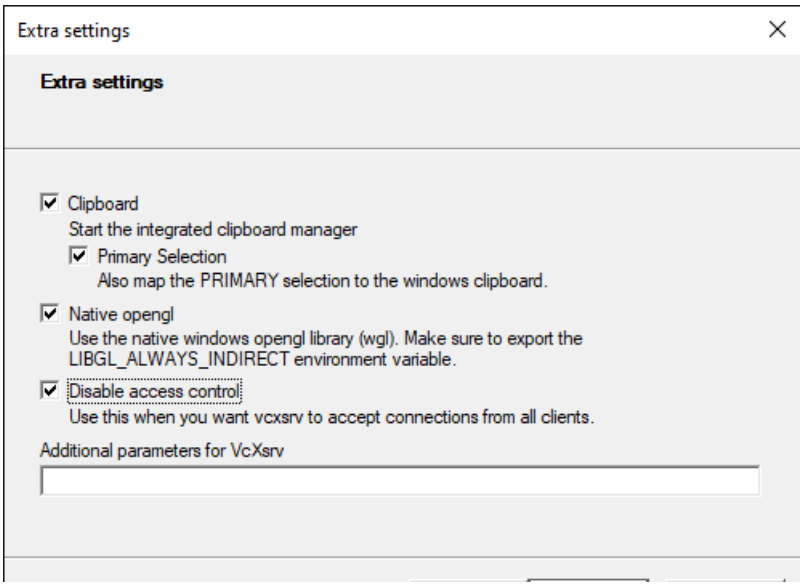
3. Run the installer

Since WSL2 is a lightweight virtual machine, you'll need to do a couple of things to ensure that your FireWall does not obstruct the connection.

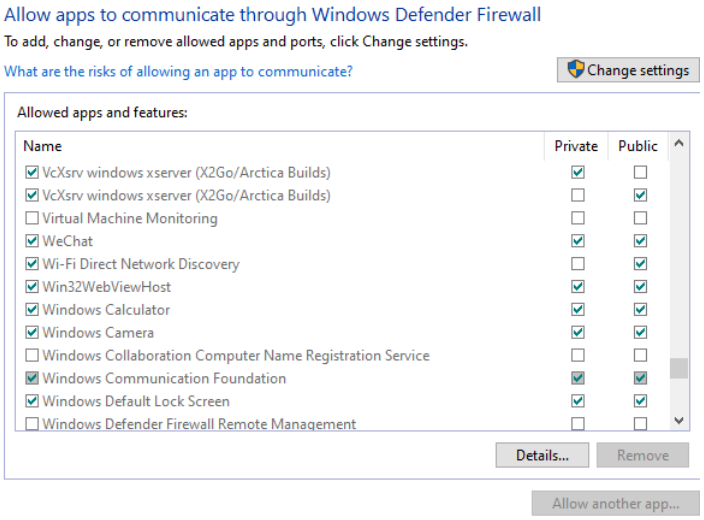
1. Following whme's TLDR answer from [**?**], in your .bashrc, add the following line:

```
export DISPLAY=$(awk '/nameserver / {print $2; exit}' /etc/resolv.conf 2>/dev/null):0
export LIBGL_ALWAYS_INDIRECT=1
```

2. Following whme's TLDR answer from [**?**], you need to disable Access Control on the Extra Settings. You can achieve this my first navigating to C:/Program Files (x86)/VcXsrv and opening "xlaunch.exe". This allows you to modify the settings. Navigate until you reach the "Extra Settings" page and check "Disable access control" as seen in the screenshot below:



3. Following [**?**], go to Settings ->Windows Defender Firewall ->"Allow an app or feature through Windows Defender Firewall" and enable BOTH "VcXsrv windows xserver (X2Go/Arctica Builds) as seen in the screenshot below:



4. To test that VcXsrv now works, with Zathura installed, run "zathura name.pdf" and see if it opens.

Note that the settings in Step 2 will revert itself. To accommodate for this, you will be prompted in the next window to save the file "config.xlaunch". From here, simply follow the instructions in Section **??** so that the configuration is set after logging in.

3.3.2 Vimtex+Zathura+Synctex

Now you're almost ready to run Vimtex+Zathura+Synctex! You'll need to also install xdotool by using

```
sudo apt-get install xdotool
```

To test whether this works with VcXsrv following the suggestion in [**?**] from lervag himself, run

```
xdotool search --class Zathura
```

and some numbers should appear. From here, you should be good to go! Simply follow the steps laid out in Section **??**.

If this fails and you've given up on Zathura, you can alternatively use the Sumatra PDF viewer. Once installed, simply add the following lines to your vimrc:

```
let g:vimtex_view_general_viewer  = '/mnt/c/Users/Hwan/AppData/Local/SumatraPDF/SumatraPDF.exe'
let g:vimtex_view_general_options = '-reuse-instance -forward-search @tex @line @pdf'
let g:vimtex_view_general_options_latexmk = '-reuse-instance'
```

However, with this, you won't have access to forward and backward search via synctex.

3.4 Windows Terminal

The default Ubuntu terminal is pretty trash. Instead, I recommend the 'Windows Terminal' which is highly customizable. Not only does it allow for multiple tabs but each tab can run a different program i.e. the Linux bash as well as PowerShell.

The customization can be done using the UI or via the **settings.json** file which can be opened by clicking the cog displayed below:

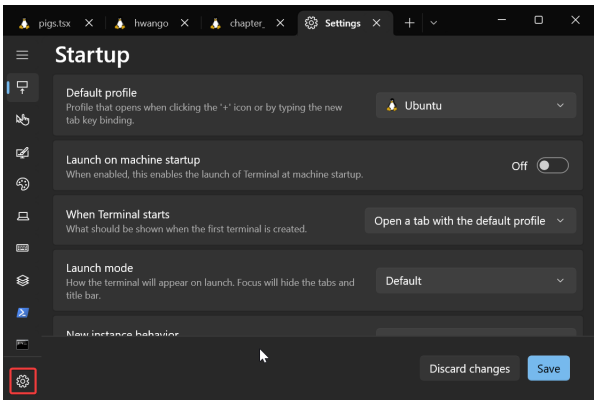


Figure 3.1: Click to open **settings.json** file.

The file path to the file is:

```
C:\Users\<Username>\AppData\Local\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\LocalState
```

3.5 Troubleshooting

3.5.1 Vim Display Issues on Startup

When using Windows Terminal, you may notice that Vim has display issues when starting up. Specifically, the columns might appear 'diagonally' and only resolves itself after playing around with the window i.e. toggling full screen on and off. This is because the following settings in your vimrc:

```
set lines=62 columns=120
```

garbles the initial startup of Vim in Windows Terminal. Simply disable this setting and the issue will disappear as suggested by waf's answer in [**?**].

3.5.2 Visual Block Mode

To enter visual block mode in Vim, you need to press 'ctrl+V'. However, in Windows, this is bound to paste. Therefore, open the Windows Terminal settings and open the 'settings.json' file. Then locate the following code:

```
{
    "command": "paste",
    "keys": "ctrl+v"
},
```

and change it to 'ctrl+shift+v' if you like to mimic Linux.

3.5.3 Clash with Anaconda

You may notice that after installing Anaconda, when using Vimtex+Zathura+Synctex, you may get the error 'vimtex cannot find zathura windows id' and forward/backwards search does not work. With some experimentation, I discovered that in your .bashrc, the following block of code that is added automatically with the installation of Anaconda:

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/home/hwangoh/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/home/hwangoh/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/home/hwangoh/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/home/hwangoh/anaconda3/bin:$PATH"
    fi
fi
# unset __conda_setup
# <<< conda initialize <<<
```

causes the issue. Specifically, I noticed that if I comment out the line:

```
__conda_setup="$('/home/hwangoh/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
```

then everything works fine with Vimtex+Zathura+Synctex. However, the terminal command 'conda' no longer works. When that line is removed, one no longer has '(base)' at the beginning of each line in the terminal, so I suspect that there is some clash with the anaconda environment. Therefore, a workaround follows from [**?**] where the answers suggest to run the following command in terminal:

```
conda config --set auto_activate_base false
```

which ensures that the Anaconda base envionment does not automatically appear by default when the terminal is started.

This generally seems to fix the problem. However, I noticed that when compiling tex documents using the LaTeX template used to write these notes, I get the error 'Compilation failed'. Despite this, you can still use 'VimtexView' to open the pdf and notice that compilation does work as the pdf does change. Additionally, forward and backward search works as well. So I just ignore the error.

3.5.4 Localhost Errors

Often in web development, one would need start a server on localhost for local development. However, with WSL2, this sometimes fails. To remedy this issue, following [**?**], we need to disable the 'Fast Start-Up' option:

1. First navigate to control panel and select 'System':

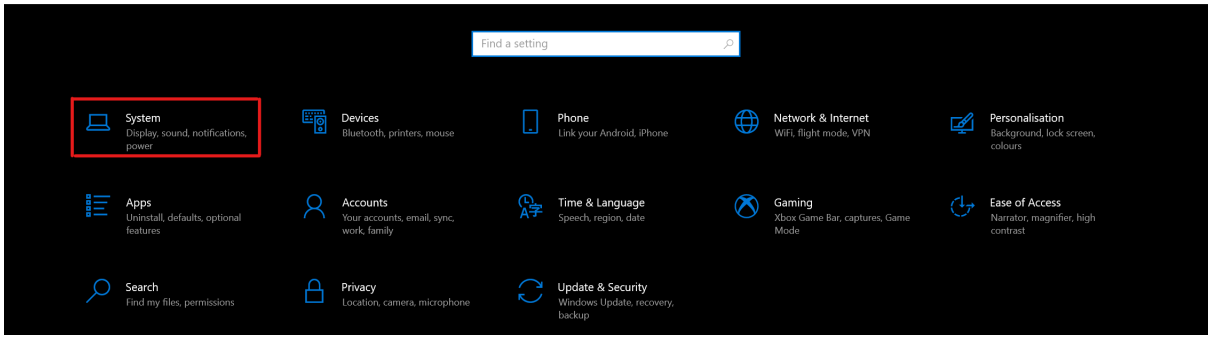


Figure 3.2: Windows settings: 'system'

2. Navigate to 'Power & sleep' on the left-blade and select 'additional power settings':

Figure 3.3: Windows settings: additional power settings

3. Select 'Choose what the power button does':

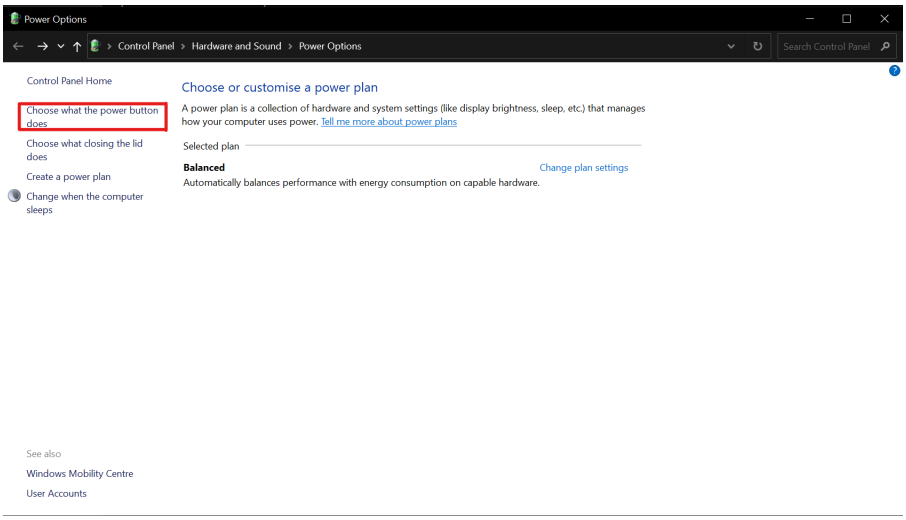


Figure 3.4: Windows settings: choose what the power button does

4. Then select 'Change settings that are currently unavailable' to allow administration privileges to alter settings. Then uncheck the checkbox for 'Turn on fast start-up':

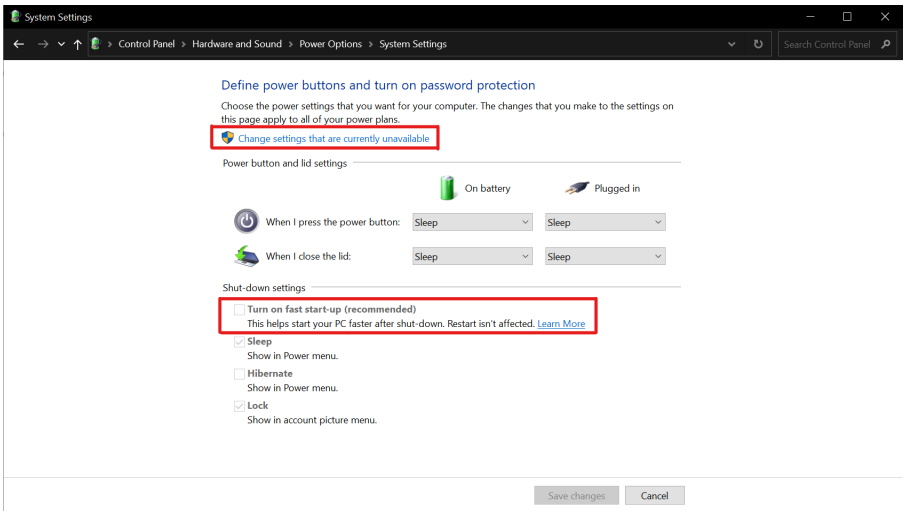


Figure 3.5: Windows settings: turn on fast startup

5. Restart your computer

3.6 Clock Sync Issue

The most unbelievable bullshit happened to me on 25/9/2022. Daylight savings occurred the night before and for some strange reason, I wake up on the morning of the 25th and all my codes that require AWS stopped working. This is because the kernel version 5.10.102.1 has a click sync issue that was fixed in 5.10.16.3 [**?**]. A possible solution is suggested by albertoflores' answer in [**?**]:

```
sudo apt-get install chrony
sudo service chrony start
```

4 Miscellaneous

4.1 .bashrc Stuff

4.1.1 Run ls after cd

Following 'frabjous' answer to the StackOverflow question [**?**], you can add the following to your .bashrc:

```
function cd {
    builtin cd "$@" && ls -F
}
```

He also mentions that he adds

```
[ -z "$PS1" ] && return
```

before this so that "everything after that line only applies to interactive sessions, so this doesn't affect how **cd** behaves in scripts." How this works is also explained by him:
"[$-z$"$PS1$"] checks if the $PS (interactive prompt variable) is 'zero length' (-z). If it is zero length, this means it has not been set, so Bash must not be running in interactive mode. The && return part exits from sourcing .bashrc at this point, under these conditions."

4.2 Ranger

Following [**?**]:

1. git clone https://github.com/hut/ranger.git
2. cd ranger
3. sudo make install

To start ranger use "ranger".

4.2.1 Configuration

After the configuration directory has been created by the Ranger, you can now copy its configuration files by running the following commands in terminal:

- "ranger –copy-config=all". Now you can run "cd /.config/ranger" to see the configuration files.

4.3 Inkdrop

Inkdrop [**?**] is an absolutely incredible minimalist and light-weight markdown-based note taking app that syncs across all devices. It is a paid service of around $5 a month or $50 a year should you chose to pay annually. It also has a convenient plugin GUI which can be accessed via the settings menu. Since the creator is a Vim user, there is of course a Vim plugin:

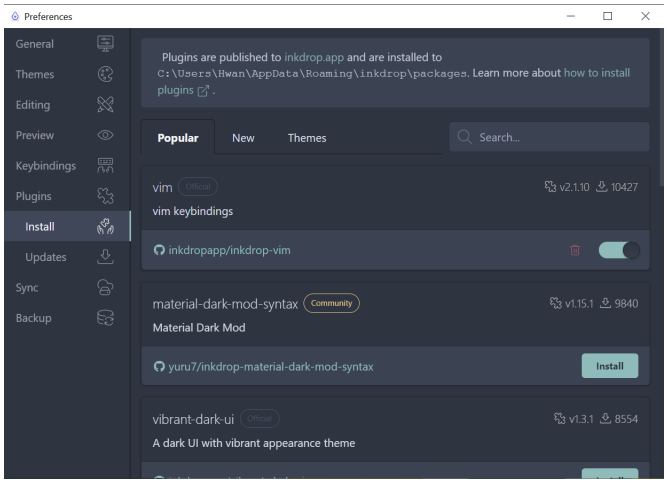


Figure 4.1: Inkdrop Vim plugin

If you cannot find the plugin in the list, you can install it using

```
ipm install vim
```

in either your Linux terminal or Powershell if you are using Windows.

To modify the key bindings, following the Inkdrop user manual [**?**] you can modify the **keymap.json** file which can be instantly directed to using the following link:

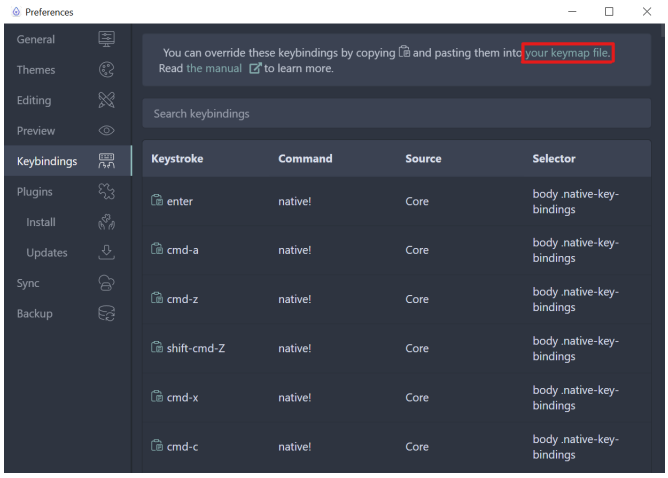


Figure 4.2: Inkdrop keymap file link

Here is an example of the **keymap.cson** file:

```
{
  ".CodeMirror.vim-mode:not(.insert-mode):not(.key-buffering) textarea": {
    "shift-j": "vim:scroll-down",
    "shift-k": "vim:scroll-up",
    "ctrl-j": "vim:join"
  }
}
```

You can modify this in Vim after navigating to it in terminal and with Inkdrop on. Any saved changes will be instantly reflected in the app and errors will result in a popup.

To enable relative line numbers, we need to instead modify the **init.js** file which can be opened from the settings menu:

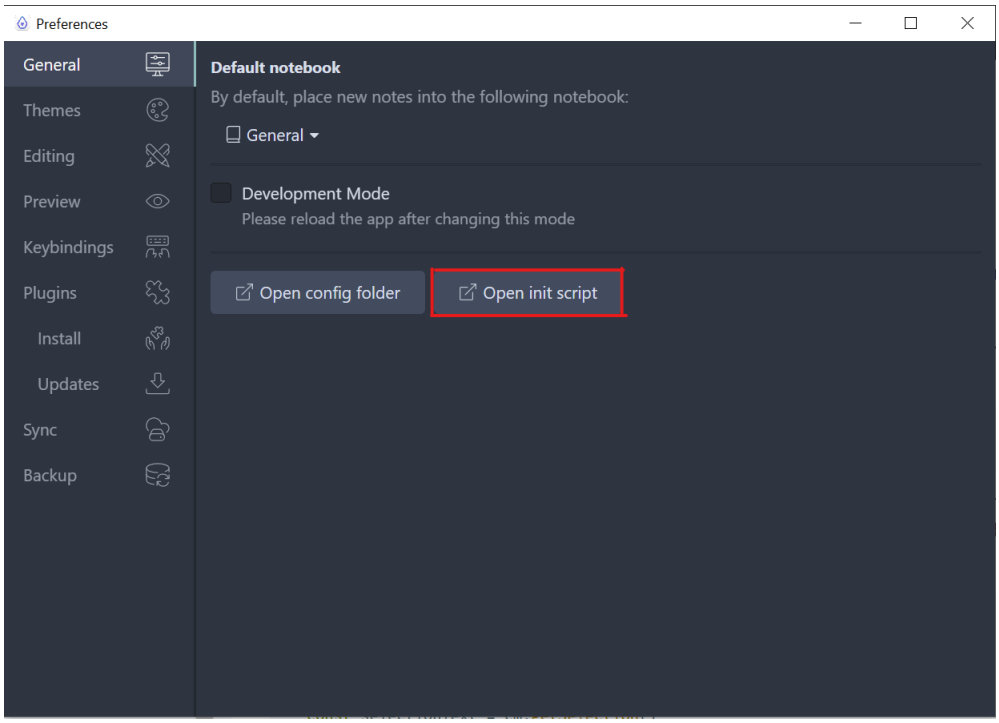


Figure 4.3: Inkdrop init.js file

and add the following:

```
function setUpRelativeLines(editor) {
  const { cm } = editor

  function showRelativeLines(cm) {
    const lineNum = cm.getCursor().line + 1;
    if (cm.state.curLineNum === lineNum) {
      return;
    }
    cm.state.curLineNum = lineNum;
    cm.setOption('lineNumberFormatter', l =>
      l === lineNum ? lineNum : Math.abs(lineNum - l));
  }
```

```
  cm.on('cursorActivity', showRelativeLines)
}

const editor = inkdrop.getActiveEditor()
if (editor) {
  setUpRelativeLines(editor)
} else {
  inkdrop.onEditorLoad(setUpRelativeLines)
}
```

4.4 ShareX

The default windows snipping tool is pretty trash. It doesn't even have the functionality to draw a simple rectangle on a screenshot! Instead, I recommend ShareX which can be obtained from [**?**]. Other than the ability to screenshot, it also has the capability to record a section of screen! This is definitely of my favourite pieces of software!

Customization can be done using the UI. You can also import settings if you wish. To do so, on the left blade, navigate to 'Application settings' then 'Settings' as displayed below:

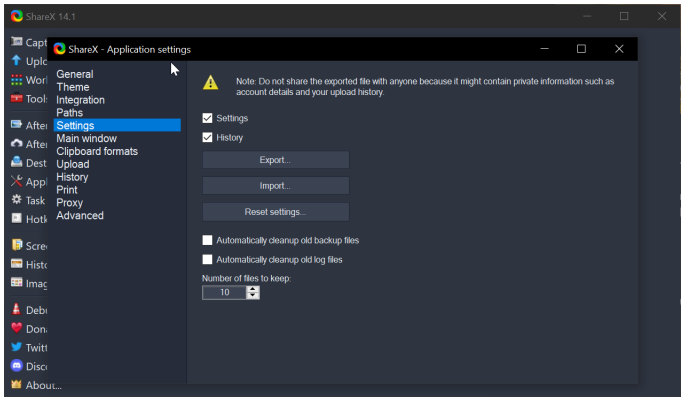


Figure 4.4: ShareX settings

4.5 Autohotkey Script for Drawing on Screen

Whilst EpicPen is fairly popular, I find that it clashes with my windows hotkeys. It's also fairly bloated and has advertisements which is unacceptable to me. Instead, there exist Autohotkey scripts that implement a simple drawing tool. I found it in the first comment by 'Hellbent' in the Autohotkey forum post [**?**]. The keys are simple:

- Hold Ctrl and left-click to draw
- Use Ctrl and right-click to erase
- Use Ctrl and mouse wheel to select colour

The script itself can be found in **.vim/config_files/i3wm-windows-emulator/draw_on_screen.ahk**.

In my config file **.vim/config_files/i3wm-windows-emulator/i3_config.ahk**, I've included the ability to toggle this program on and off as you may need to select multiple files using Ctrl+left-click. The lines of code are simply:

```
; === Draw On Screen ===
#w::
DetectHiddenWindows On ; Allows a script's hidden main window to be detected
SetTitleMatchMode 2 ; Avoids the need to specify the full path of the file below

If WinExist("draw_on_screen.ahk - AutoHotkey")
{
  PostMessage, 0x111, 65307,,, draw_on_screen.ahk - AutoHotkey  ; Exitapp
}
else
  Run, draw_on_screen.ahk
return
```

but you need to ensure that the **i3_config.ahk** and the **draw_on_screen.ahk** script must be located in the same directory.