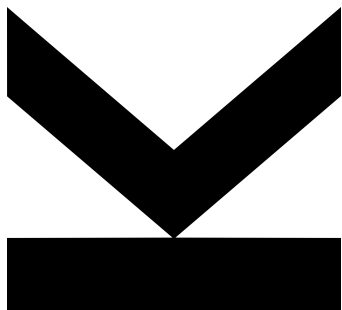


Author / Eingereicht von
Xaver Haider
k12043507

17. September 2024

Latex und vim



Einrichtung und Nutzung

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 17. September 2024

Inhaltsverzeichnis

1	Einleitung	2
2	Einrichtung der Entwicklungsumgebung	3
2.1	Voraussetzungen und Software	3
2.2	Lua und Nvim	4
2.3	Plugins	5
2.4	Syntaxparser	7
2.5	Languageserver und Compiler	7
2.6	Snippets und Autocomplete	9
2.7	PDF, Vor- und Rückwärtssuche	11
2.8	Rechtschreibkorrektur und Thesaurus	12
3	Nutzung und spezifische Arbeitsweise	14
3.1	vim-keybindings	14
3.2	search and replace, macros	16
3.3	Snippets	21
3.4	buffers	22
3.5	Sonstiges	23
3.6	Anwendungsbeispiele	24
4	Anhang	26

Abbildungsverzeichnis

3.1	Einfügen von Grafik	22
4.1	Screenshot Arbeitsumgebung	26

Diese Seminararbeit entstand im Rahmen der Lehrveranstaltung *Wissenschaftliches Schreiben und Layouten anhand von L^AT_EX 2*. Ihr Schwerpunkt liegt in der Konfiguration des *neovim*-Editors und dessen Anwendung, um die Arbeit an L^AT_EX Dokumenten zu optimieren.

1 Einleitung

Die Bearbeitung von \LaTeX Dokumenten bedarf keiner aufgeblasenen Entwicklungsumgebung. Im Grunde genügt ein einfaches Textbearbeitungsprogramm und ein passender Compiler. Sämtliche Zusatzfunktionen wie *syntax-highlighting*, *autocomplete* oder Fehlerhinweise erleichtern den Arbeitsablauf, sind aber nicht erforderlich. In den meisten Fällen ist von der Anwendung eines nackten Editors dennoch abzuraten.

*neovim*¹ bietet das Beste aus beiden Welten. Das auf *vim* (drop-in) basierende ultra-leichte Terminalprogramm ist hoch konfigurierbar und ermöglicht die Einbindung von diversen Erweiterungen.

neovim ist, wie *vim* und *vi*, eine tastengesteuerte Software. Die Verwendung der Maus ist möglich - aber nicht empfohlen. Nach einer gewissen Einarbeitungsphase ermöglicht eben diese Eigenschaft dem Nutzer eine signifikante Steigerung der Arbeitsgeschwindigkeit im Vergleich zu konventionellen Methoden. Aus diesem Grund bieten zahlreiche IDEs und Editoren eine Einstellung (oder ein Plugin) zur Verwendung von *vi-keybindings*. Ein Großteil der in Folge beschriebenen Konfigurationen und Arbeitsweisen ist neben \TeX auch auf andere Dateitypen übertragbar. So kann eine einheitliche und effiziente Entwicklungsumgebung für die Arbeit mit verschiedensten Skript- und Programmiersprachen geschaffen werden.

¹<https://neovim.io/>

2 Einrichtung der Entwicklungsumgebung

Es besteht die Möglichkeit einen Großteil der in diesem Kapitel angeführten Schritte zu umgehen. Pakete wie `lsp-zero` erleichtern die Einrichtung und verknüpfen eine Sammlung an hilfreichen Plugins. Noch einfacher ist die Installation von fertigen Setups wie `lazyvim`.

Zudem sei anzumerken, dass viele der weiters beschriebenen Funktionen auch unter dem Paket `vimtex` implementiert wurden. Online finden Sie zahlreiche Tutorials zu diesem, speziell für \LaTeX entwickelten, Plugin. Unter anderen ist der exzellente Leitfaden von Elijan Mastnak [**mastnak**] zu empfehlen. Um die Einrichtung modular und ihr Anwendungsgebiet möglichst breit zu halten, wird hier auf `vimtex` verzichtet.

2.1 Voraussetzungen und Software

Die Einrichtung sollte in einem breiten Spektrum an Softwareversionen unter diversen GNU-Linux basierenden Operationssystemen (und Windows) stabil laufen. Erforderliche Pakete (getestet mit angeführter Versionen):

- Editor: `neovim` 0.10.1
- Compiler(\TeX): `texlive-*` 2024.2
- Compiler(config): `lua` 5.4.7
- Terminal: `alacritty` 0.13.2
- PDF-Viewer: `zathura` 0.5.8
- Languageserver: `texlab` 4.3.2 (cargo oder npm)
- Für Plugins: `node.js` 22.7.0
- Compiler(C): `GCC` 14.2.1

2.2 Lua und Nvim

neovim verfügt wie *vim* über eine ausgezeichnete Dokumentation, welche über den Befehl `:help` im Editor geöffnet werden kann. Die Konfigurationsdateien sind im internen *vim-script* oder mit der Skriptsprache *lua* zu verfassen. Diese bietet, neben einem eleganten Syntax, die einfache Einbindung komplexer Funktionen in die Entwicklungsumgebung. Details können in der offiziellen Onlinedokumentation *Lua in Nvim*¹ nachgelesen werden.

neovim lädt die Datei `~/.config/nvim/init.lua` (oder `init.vim`) und führt das darin enthaltene Skript in seiner Startprozedur aus. Der Anwendungsname kann auch explizit angegeben werden:

```
alias vi-latex='NVIM_APPNAME=nvim-latex nvim'
```

`vi-latex` öffnet nun `~/.config/nvim/init.lua`.

Um eine gewisse Struktur zu wahren wird empfohlen, das Konfigurationsskript auf verschiedene Dateien aufzuteilen und von der Wurzel `init.lua` zu laden:

```
--init.lua
require("set")
require("lsp")
require("cmp")
require("treesitter")
require("plugins")
```

Struktur und Namen der Dateien sind frei wählbar. `set.lua` beinhaltet in diesem Beispiel das Setzen wichtiger Parameter und personalisierter Tastenkürzel:

```
--set.lua
--relative Zeilennummerierung
vim.wo.relativenumber = true
vim.wo.number = true
--theme
vim.o.background = "dark"
vim.cmd([[colorscheme gruvbox]])
```

¹<https://neovim.io/doc/user/lua.html>


```
--zsh in terminal
vim.opt.shell = "/bin/zsh"

vim.g.mapleader= "_"
--spellcheck
vim.keymap.set("n", "<leader>sp", '<cmd>setlocal spell _
spelllang=de_at<CR>')

vim.keymap.set("n", "<leader>bb", vim.cmd.bn)
vim.keymap.set("n", "<leader>bd", vim.cmd.bd)
vim.keymap.set("n", "<leader>ex", vim.cmd.Ex)

vim.keymap.set("t", "<Esc>", '<C-\\><C-n>')

vim.keymap.set("n", "<leader>fs", '<cmd>_TexlabForward_<CR>')
--anonyme Funktion um PDF-Viewer zu starten
vim.keymap.set("n", "<leader>pd", function()
    local current_bufnr = vim.api.nvim_get_current_buf()
    local file_name = vim.api.nvim_buf_get_name(
        current_bufnr)
    io.popen("nohup _zathura_ .. file_name:gsub("%. %w%w%w$
        ", ".pdf") .. "_&")
end
)
```

2.3 Plugins

neovim ermöglicht die Einbindung von *vim-script* und *lua*-Plugins. Die Erweiterungen können über einen Pluginmanager wie packer oder direkt installiert werden. Eine automatisierte Verwaltung spart Zeit und Energie. Nur der Manager selbst ist manuell einzurichten:


```
$ git clone --depth 1 https://github.com/wbthomason/packer.
  nvim\
~/.local/share/nvim/site/pack/packer/start/packer.nvim
```

Die gewünschten Erweiterungen werden im Lua-Skript angegeben:

```
--plugins.lua
vim.cmd [[packadd packer.nvim]]

require('packer').startup(function(use)
  --Manager selbst
  use 'wbthomason/packer.nvim'
  --LSP
  use 'neovim/nvim-lspconfig'
  --autocomplete
  use 'hrsh7th/nvim-cmp'
  use 'hrsh7th/cmp-nvim-lsp'
  use 'hrsh7th/cmp-buffer'
  use 'hrsh7th/cmp-path'
  use 'hrsh7th/cmp-cmdline'
  --snippets
  use 'L3MON4D3/LuaSnip'
  use 'saadparwaiz1/cmp_luasnip'
  use 'rafamadriz/friendly-snippets'
  --parser
  use {
    'nvim-treesitter/nvim-treesitter',
    run = ':TSUpdate'
  }
  --theme
  use { "ellisonleao/gruvbox.nvim" }
end)
```

Wurde das Skript geladen (Befehl: `:source` oder Neustart), können die gelisteten Pakete mit der Anweisung `:PackerSync` installiert werden. Näheres und mögliche Optionen sind im Github-Repository nachzulesen.

 [wbthomason/packer](https://github.com/wbthomason/packer)

github.com/wbthomason/packer

2.4 Syntaxparser

*tree-sitter*² ist eine bewährte Parser-Bibliothek. Sie baut konkrete Syntax-Bäume einer Quelldatei und updated diese inkrementell. Implementiert in C ist der Parser blitzschnell und frei von lokalen Dependencies. Zudem ist er akkurater und vielseitiger als der in *nvim* integrierte Standard.

Die Einbindung von *tree-sitter* verbessert Syntax-Highlights, Navigation und Umstrukturierung (refactoring) von Code (oder Skripten).

```
--treesitter.lua
require 'nvim-treesitter.configs'.setup {
  ensure_installed = { "latex", "bash", "lua", "python" }, --
    Install parsers for these languages
  highlight = {
    enable = true, -- false will disable the
      whole extension
    additional_vim_regex_highlighting = false,
  },
}
```

 [nvim-treesitter/nvim-treesitter](https://github.com/nvim-treesitter/nvim-treesitter) github.com/nvim-treesitter/nvim-treesitter

2.5 Languageserver und Compiler

In einfachen Worten ist ein Languageserver ein Hintergrundprozess, welcher dem Editor oder der IDE eine Vielzahl an spezifischen Funktionen zur Verfügung stellt. Die Kommunikation läuft über ein definiertes Protokoll (LSP). Das Plugin *nvim-lspconfig* automatisiert einen Großteil der erforderlichen Konfiguration.

```
$ cargo install texlab
$ sudo pacman -Syu texlive-langgerman
```

²<https://github.com/tree-sitter/tree-sitter>

Der \LaTeX -Compiler Texlive ist in zahlreichen Paketmanager Bibliotheken enthalten. Zudem besteht die Möglichkeit einer direkten Installation. Hierzu sei erneut auf die Dokumentation³ der Entwickler verwiesen. Die Anwendung von MiKTeX⁴ oder anderen Alternativen ist auch möglich. Für LSP-Support ist ein Compile-Skript auf dem Pfad und die notwendigen Argumente anzugeben (hier `latexmk <file> -pdf ...` (Texlive)).

Die Languageserver Executables können ebenfalls über verschiedenste Paketmanager oder direkt installiert werden. Zu beachten ist erneut die Einbindung der Binärdateien in den Pfad.

```
--lsp
local nvim_lsp = require('lspconfig')

nvim_lsp.texlab.setup{
  settings = {
    texlab = {
      build = {
        executable = "latexmk", --Compile-Script
        args = { "-pdf", "-interaction=nonstopmode", "-synctex=1", "%f" },
        onSave = true,
      },
      forwardSearch = {
        executable = "zathura",
        args = { "--synctex-forward", "%l:1:%f", "%p" },
      },
      chktex = {
        onOpenAndSave = true,
        onEdit = false,
      },
    },
  }
}
```

³<https://www.tug.org/texlive/>

⁴<https://miktex.org/>

2.6 Snippets und Autocomplete

An der Wurzel von Funktionen wie Snippets, automatischer Vervollständigung oder Vorschlägen sitzt eine, im Plugin nvim-cmp verpackte, Completion-Engine. Die Konfiguration ist einfach und nahtlos (Engine in Lua). Angeführt ist eine einfache Konfiguration. Bibliotheken werden eingebunden und die Tastenbelegung definiert:

```
-- cmp
local cmp = require 'cmp'

require("luasnip.loaders.from_lua").load({paths = "~/ .config/
nvim/snippets/"})
require('luasnip.loaders.from_vscode').lazy_load()

cmp.setup({
  snippet = {
    expand = function(args)
      require('luasnip').lsp_expand(args.body)
    end,
  },
  mapping = {
    ['<C-b>'] = cmp.mapping.scroll_docs(-4),
    ['<Space><Space>'] = cmp.mapping.scroll_docs(4),
    ['<C-Space>'] = cmp.mapping.complete(),
    ['<C-e>'] = cmp.mapping.close(),
    ['<CR>'] = cmp.mapping.confirm({ select = true }),
    ['<Tab>'] = cmp.mapping(function(fallback)
      if require('luasnip').expand_or_jumpable() then
        require('luasnip').expand_or_jump()
      else
        fallback()
      end
    end)
```

```
end, { 'i', 's' })),
['<S-Tab>'] = cmp.mapping(function(fallback)
    if require'luasnip'.jumpable(-1) then
        require'luasnip'.jump(-1)
    else
        fallback()
    end
end, { 'i', 's' })),
},
sources = cmp.config.sources({
    { name = 'nvim_lsp' },
    { name = 'luasnip' },
}, {
    { name = 'buffer' },
})
})
```

Neben den Inhalten der Bibliotheken bietet LuaSnip die Möglichkeit benutzerdefinierte Snippets einzubinden.


```
$ cd ~/.config/nvim-latex/snippets
$ ls -al
total 24
drwxr-xr-x 2 xaver xaver 4096 Jun 21 02:57 .
drwxr-xr-x 5 xaver xaver 4096 Aug 31 18:29 ..
-rw-r--r-- 1 xaver xaver 275 Jun 21 02:57 all.lua
-rw-r--r-- 1 xaver xaver 137 Jun 21 02:57 lua.lua
-rw-r--r-- 1 xaver xaver 128 Jun 21 02:57 python.lua
-rw-r--r-- 1 xaver xaver 285 Jun 21 02:57 tex.lua
```

Wird eine \LaTeX -Datei bearbeitet lädt das Plugin entsprechend dem Dateitypen `tex.lua` und `all.lua`, welche etwa folgendes Skript enthalten:

```
return{
    s("figure",{t("\\begin{figure}[",i(1,"where"),t("{") "
        , "\\centering", "\\includegraphics[width="}),i(2,"
        width"),t("cm]{"),i(3,"path"),t("{")", "\\vspace{-0
```

```
.3cm}","\\caption{"}),i(4,"caption"),t({"},"\\  
label{fig:"}),i(5,"label"),t({"},"\\vspace{-0.1cm  
},"\\end{figure}")))),  
}
```

Eine detaillierte Beschreibung des Syntax würde den Rahmen dieser Arbeit sprengen. Es sei auf das Github-Repositoryum LuaSnip verwiesen.

 hrsh7th/nvim-cmp

github.com/hrsh7th/nvim-cmp

 hrsh7th/cmp-nvim-lsp

github.com/hrsh7th/cmp-nvim-lsp

 hrsh7th/cmp-buffer

github.com/hrsh7th/cmp-buffer

 hrsh7th/cmd-path


github.com/hrsh7th/cmd-path

 hrsh7th/cmp-cmdline

github.com/hrsh7th/cmp-cmdline

 L3MON4D3/LuaSnip

github.com/L3MON4D3/LuaSnip

 saadparwaiz1/cmp-luasnip

github.com/saadparwaiz1/cmp-luasnip


 rafamadriz/friendly-snippets github.com/rafamadriz/friendly-snippets

2.7 PDF, Vor- und Rückwärtssuche

Zathura ist ein schlanker und hoch konfigurierbarer Dokumenten-Viewer. Für PDF Unterstützung bedarf es dem Render-Library-Plugin poppler. Vorwärtssuche funktioniert mit der Konfiguration des LSP (Kapitel 2.5) mit dem Befehl :TexlabForward. Rückwärtssuche steht nach der Anfügung:

```
$ echo "set␣synctex-editor-command␣'nvr␣--remote-silent␣+{%  
line}␣%{input}␣'" >> ~/.config/zathura/zathurarc
```

mit STRG+Click in Zathura zur Verfügung.

 pwmt/zathura-pdf-poppler

github.com/pwmt/zathura-pdf-poppler

Für eine vollständige Dokumentation sei auf die Webseite⁵ von Zathura verwiesen.

2.8 Rechtschreibkorrektur und Thesaurus

Die Einrichtung der Rechtschreibhinweise ist in *nvim* mit dem eingebauten Befehl `:setlocal` möglich. Das erforderliche Wörterbuch wird automatisch oder nach Bedarf manuell eingebunden. (Siehe `:help spell`)

```
vim.keymap.set("n", "<leader>sg", '<cmd>setlocal␣spell␣  
spelllang=de_at<CR>')
```

Thesaurus ist für viele ein unumgängliches Werkzeug im Verfassen von Texten. Zunächst ein einfaches Bash-Skript, das Synonyme aus der Onlinequelle [openthesaurus.de](https://www.openthesaurus.de) bezieht: (Dependencies: Das JSON-Parsertool `jq`)

```
#!/bin/bash  
word="$1"  
curl -s "https://www.openthesaurus.de/synonyme/search?q=$word  
&format=application/json" | jq '.synsets[].terms[].term'
```

Liegt das Skript im Pfad und ist ausführbar (`chmod +x`), erleichtert es bereits die Suche nach Synonymen über das interne Terminal(`:term`) oder die Kommandozeile (z.B. `!thesaurus.sh apfel`).

Noch praktischer wäre es, mit einer Tastenkombination das Wort unter dem Cursor als Argument zu wählen und die Ergebnisse in einem temporären Buffer aufzulisten (vgl. Kapitel 3.4 "Buffer"):

⁵<https://pwmt.org/projects/zathura/>


```
--set.lua called by init.lua config file
function ShowGermanThesaurus()
    local current_word = vim.fn.expand("<word>")
    local handle = io.popen("german_thesaurus.sh" ..
        current_word)
    local result = handle:read("*a")
    handle:close()

    vim.api.nvim_command("new")
    vim.api.nvim_buf_set_option(0, 'buftype', 'nofile')
    vim.api.nvim_buf_set_option(0, 'bufhidden', 'wipe')
    vim.api.nvim_buf_set_option(0, 'swapfile', false)

    vim.api.nvim_buf_set_lines(0, 0, -1, false, vim.split(
        result, "\n"))
end

vim.api.nvim_set_keymap('n', '<leader>th', ':lua
    ShowGermanThesaurus()<CR>', { noremap = true, silent =
    true })
```

Dieses Beispiel zeigt erneut die Möglichkeiten und Vielseitigkeit der Konfiguration durch die Skriptsprache *Lua*.

3 Nutzung und spezifische Arbeitsweise

vi-Tutorials gibt es wie Sand am Meer. Dieser Leitfaden beschränkt sich auf die absoluten Grundlagen und einige ausgewählte Funktionen, welche sich im Kontext von \LaTeX als nützlich erwiesen haben. Wie in den meisten Bereichen des Lebens gilt auch hier: Übung macht den Meister! Ähnlich dem Erlernen einer Sprache braucht es Zeit, um einen flüssigen Abfolge an Befehlen zu meistern. Für Anfänger ist ein "Schummelzettel" der grundlegenden Kommandos zu empfehlen. So genügt ein Blick, um Vergessenes zurück in die Erinnerung zu holen.

3.1 vim-keybindings

In Kapitel 2.2 sind einige Beispiele von benutzerspezifischen Tastenkombinationen aufgelistet. Es ist *good-practice* solche Konfigurationen auf längliche Befehle zu beschränken, und im Fall kurze Standardfunktionen auf Defaultwerte zurückzugreifen. Auf diesem Weg lässt sich der *Workflow* einfacher auf neue Umgebungen (fremder Rechner, Server, Minimalinstallation mit vim, IDE mit vim-keybindings) übertragen, da die vi-Standards tief im Gedächtniss verankert wurden.

In vim gibt es drei Modi: INSERT, COMMAND und VISUAL (+VISUAL-BLOCK), wobei in Folge vorwiegend auf die ersten beiden eingegangen wird. Der Wechsel vom Kommando- in den Einfügemodus erfolgt mit `i` (`v` für VISUAL). Zurück in den Einfügemodus gelangt man mit ESC.

Der Einfügemodus erscheint dem Neuling intuitiv. Die Funktionen der Tasten gleichen jenen aus anderer Entwicklungsumgebungen. Zahlen, Buchstaben und Sonderzeichen entsprechen ihrem Symbol und auch Sondertasten erfüllen ihre typischen Aufgaben.

Im Kommandomodus liegt die Stärke des Editors. Wie der Name bereits andeutet, sind den Tasten nützliche Kommandos zugeordnet. Spezielle und längere Befehle werden mit einem Doppelpunkt `:` eingeleitet. Es folgen die grundlegendsten Anweisungen

kategorisiert nach Funktion:

:help cmdline
:w speichert die Datei (write)
:q schließt den Editor (quit)
:q! schließt den Editor ohne Speichern (quit and discard changes)
:wq speichert und schließt die Datei
:e <file> öffnet Datei in neuem Buffer

:help movement
h j k l bewegt den Cursor nach links unten oben rechts
^ 0 weicher und harter SOL¹
\$ hartes EOL
w W springt zum Anfang des nächsten Wortes
b B springt zum Anfang des vorherigen Wortes
f<char> springt zum nächsten char
gg springt zum Anfang des Dokuments
G springt ans Ende des Dokuments

Für die Anwendung an L^AT_EX-Dokumenten können die Befehle {, }, (und) von Nutzen sein, um an den Anfang (das Ende) eines Paragraphen oder Satzes zu springen.

:help editing
i I Einfügemodus vor Curser oder SOL
a A Einfügemodus nach Cursor oder in nächster Zeile
o O Einfügemodus in neuer Zeile unterhalb oder oberhalb
dd yy löscht oder kopiert Zeile
dw yw löscht, kopiert nächstes Wort
cw löscht nächstes Wort und wechselt in Einfügemodus
u undo
Ctrl + r redo

¹Start of Line: mit oder ohne Rücksicht auf Leerzeichen

Nach Kommandos wie `c` (change), `d` (delete) oder `y` (yank) folgt immer eine Spezifikation eines Bereiches auf den der Befehl wirken soll (`w W b B f<char> ...`). Doppelte Eingabe (`yy, dd`) wählt die ganze Zeile-, Großbuchstaben (`C, D`) meist von der Cursorposition bis zu EOL aus.

Gelöschte oder kopierte Strings werden in einem Standardregister zwischengespeichert. `p P` (paste) fügt den Inhalt des Registers ein. Es können auch andere Registerplätze mit `"<char><Bereich>` belegt werden (z.B.: `"ayy` speichert die aktuelle Zeile in Register `a`). Der Platz `+` ist für das System-Clipboard reserviert.

Befehle können durch das Vorstellen eines Integers beliebig oft wiederholt werden. Beispiele: `5dd` löscht die nächsten 5 Zeilen, `3u` revidiert 3 Schritte.

`:Ex` öffnet einen integrierten Filebrowser im aktuellen Arbeitsverzeichnis. Terminal Befehle können über `:exec <shell>`, `:! <shell>` oder dem internen Emulator `:term` ausgeführt werden.

3.2 search and replace, macros

Verfügen Sie nicht über das Gedächtnis eines Elefanten ist die Navigation durch große Dateien über die Zeilennummer (absolut: `<nummer>G` oder relativ: `<nummer>j|k`) äußerst mühselig und meist nicht zielführend. Einfacher ist die Suche nach bestimmten Mustern über `/<string>` (`?<string>` für Rückwärtssuche). Mit `n` und `N` kann zwischen den Treffern navigiert werden.

Systematische Anpassungen können nach dem *search and replace*-Muster durchgeführt werden (`:substitute`). In Dokumenten ist dies ein effizientes Hilfsmittel zur Korrektur von Rechtschreibfehlern. Der allgemeine Syntax lautet:

```
:<range>s/<string1>/<string2>/<optionen>
```

```
:help range
```

Standardwert des optionalen Parameters ist die aktuelle Zeile, `%` wendet den Befehl auf die gesamte Datei an, `.,.+<Zeilen>` gibt den Bereich von der aktuellen Zeile (`.`) bis zur

Zeile (.+<Zeilen>) an. Die Zeilennummern können auch absolut angegeben werden (<start>,<stop>).

`optionen`

`g` wenn der Befehl auf alle Instanzen anzuwenden ist (Standard: erste der Zeile), `i` um Groß- und Kleinschreibung zu ignorieren, `c` um jede Instanz zu prüfen (y or n prompt).

`string`

Das zu ersetzende Muster wird zuerst angeführt.

Der Syntax umfasst eine Vielzahl an Möglichkeiten komplexere Muster zu erfassen. Eine Auswahl an für die T_EX-Anwendung nützliche Beispiele sind im nächsten Absatz aufgelistet. Näheres in der integrierten Dokumentation `:help (:substitute)`.

0. Minimalsyntax

`:s/foo/bar`

Ersetzt die erste Instanz des Strings `foo` in der aktuellen Zeile mit `bar`.

1. Ganze Wörtern:

`:.+.5s/\bwort\b/neu/g`

Ersetzt das Wort `wort` mit `neu` in der aktuellen und den folgenden 5 Zeilen. Partielle Treffer wie `wortlaut` werden nicht berücksichtigt.

2. Wortgruppen:

`:%s/\(wort1\) and \(wort2\)\/\2 and \1/g`

Die Klammern fassen die Teilmuster `wort1`, `wort2`. Die Muster werden temporär (Variablen: 1 und 2) gespeichert und können im Ersatztext eingefügt werden.

3. Ziffern und chars:

`:%s/\d\+/Zahl/g`

Das Muster `\d\+` umfasst Zahlen mit einer oder mehreren (`\+`) Ziffern. Hier werden diese durch den String `Zahl` ersetzt. Das Pendant dazu lautet `\D` für alle chars mit Ausnahme

der Ziffern.

4. SOL und EOL:

`:%s/^wort/neu/g`

`:%s/wort$/neu/g`

`^` referenziert den Beginn einer Zeile `$` das Ende. Das Wort wird in diesem Beispiel nur ersetzt, wenn es sich an der angegebenen Position befindet. Diese Sonderzeichen können auch alleine angegeben werden. `:s/^/%` kommentiert eine Zeile in \LaTeX -Dokumenten aus.

7. Wildcard:

`:%s/w.r.t/neu/g`

Der Punkt `.` symbolisiert eine Wildcard. `wort`, `wirt` und `w0rt` werden hier erfasst.

8. Einrückungen:

`:%s/\t/ /g`

`:s/^s\+/space`

`\t` ist das Sonderzeichen für Tabulatoren. In Dokumenten hilfreich um Bereiche zu formatieren. Leerzeichen und Tabulatoren werden von vim unterschieden, doch unter dem Symbol `\s` zusammengefasst. Das zweite Beispiel zeigt wie alle Einrückungen zu Beginn einer Zeile ersetzt werden können.

11. Wortwiederholung:

`:%s/(\w\+)\ \1/\1/g`

Hier werden einige, bereits beschriebene Konzepte kombiniert. Neu ist die Wildcard für Buchstaben `\w`. `(\w\+)` erfasst Buchstaben zu einer Gruppe (Wort). `\1` referenziert diese. Zwei idente aufeinanderfolgende Worte führen zu einem Treffer und werden hier auf ein Wort reduziert.

12. Zeilenumbrüche:

`:%s/\n\{2,\}/\r/g`

`\n` ist erwartungsgemäß das Sonderzeichen für newline. (`\n\{2,\}`) sucht nach mindestens zwei aufeinanderfolgenden Umbrüchen und ersetzt diese durch einen einzigen (`\r` (carriage return)). Ein weiteres nützliches Werkzeug für die Arbeit an langen Dokumenten.

Viele Wege führen ans Ziel. Neben dem `:substitute` Befehl gibt es weitere Möglichkeiten die *vim*-Umgebung zu nutzen, um repetitive und monotone Abläufe in Befehlen zusammenzufassen. Unter diesen sticht besonders die Aufzeichnung von Arbeitsabläufen über Macro-Funktionen hervor. Diese ermöglicht die Erfassung von Befehlsabläufen, um diese später automatisch wiederzugeben. Der Syntax lautet:

`q<register><Befehlskette>q`

Ähnlich den Speicherregistern stehen sämtliche char-Symbole der Adressierung zur Verfügung. Häufig wird `q` genutzt. Nach der Auswahl des Registers startet die Aufnahme der Anweisungsfolge. Diese wird durch das erneute Betätigen der Funktionstaste beendet. Demzufolge lassen sich Macros nicht verschachteln. Die Wiedergabe erfolgt durch die Referenzierung `@<register>`. Meist wird dieser Befehl wiederholt ausgeführt (z.B.: `10@a`).

Für neue Nutzer sei anzumerken, dass innerhalb der Funktionskette auch ein Wechsel in den Eingabe- oder Visualmodus möglich ist. Der Umfang des Editors wird nicht eingeschränkt. Dies macht Macros zu einem mächtigen Werkzeug, das ein breites Spektrum an Aufgaben erfüllt. Einige auf \LaTeX bezogene Anwendungsbeispiele sollen dies demonstrieren:

Ein wesentlicher Vorteil gegenüber der Substitutionsmethode liegt in der einfachen Handhabung. Die aufzunehmende Bearbeitung erfolgt direkt (im Vergleich zu `preview`), zudem ist kein neuer Syntax zu erlernen.

A. Einrücken oder Kommentieren

Ein typischer Anwendungsfall für `:s` lässt sich ebenso mit einem Macro lösen:

`:. ,.+10s/^/\t` um die nächsten 10 Zeilen einzurücken, kann durch die Folge `qqI<Tab><Esc>jq` und `9@q` ersetzt werden.

Ähnlich Beispiele:

Kommentar löschen: `qq0f%Djq`

Leerzeichen durch Tabulator ersetzen: `qq0d~i<Tab><Esc>jq`

Solche Zeilenmanipulationen starten häufig mit `0`, um einen Ausgangspunkt unabhängig der Cursorposition zu schaffen und enden mit `j`, um in die nächste Zeile zu wechseln, in welcher der Befehl erneut ausgeführt werden kann.

B. Wortmanipulationen

Auch kleine wiederkehrende Arbeitsabläufe sollten in Macros gespeichert werden. `qkbi"<Esc>ei"<Esc>q` kann genutzt werden, um Wörter einzuklammern. \LaTeX spezifische Anwendungen:

`qbi\textbf{<Esc>ei}<Esc>q`

oder `qqbi\cite{<Esc>ea}<Esc>jq`. `b` steht für back (zurück an den Wortanfang).

Abzuwiegen ist, ab welchem Umfang der Umweg über ein Macro Arbeit einspart. Besonders häufig genutzte Operationen können auch in die Konfigurationsdatei eingebunden werden.

C. Beispiele aus \LaTeX

Block in Umgebung einwickeln:

`qq0\begin{Umgebung}Esc>5j0\end{Umgebung}<Esc>jq`

Ein Snippet wäre hier zielführender.

Nummerierung der Überschriften entfernen:

`qq0f\cw\section*<Esc>jq`

Springt zum Anfang der Zeile `0`, sucht erstes `\` und ändert Wort zu `\section*`

Wechsel von itemize zu enumerate Umgebung:

qq/itemize<Enter>cwenumerate<Esc>yiwnp<Esc>q
oder qqcwenumerate<Esc>q in Kombination mit /itemize
(yank-in-word wird gefolgt von next und paste)

\$\$ expandieren zu equation-Umgebung:

qq0f\$ct\begin{equation}<Esc>A\end{equation}<Esc>jq
A steht für append (in nächster Zeile).

3.3 Snippets

In einigen der Beispiele aus dem letzten Kapitel würde der Einsatz von Snippets schneller ans Ziel führen. Besonders in Fällen von häufig wiederkehrenden (auch über mehrere Arbeitseinheiten) Codesegmenten ist der Einsatz dieser Funktion ratsam.

Diese Textschnipsel werden aus dem Einfügemodus mit der Eingabe eines spezifischen Schlüsselwortes (häufig der Anfang des Segments, etwa begin für Umgebungen) und der Betätigung einer Funktionstaste (meist <Tab>) aufgerufen. Sie verfügen häufig über mehrere Eingabefelder (Abb. 3.1: where, width, path, caption und label) oder binden bereits vorhandenen Text in ihre Struktur ein. Die Navigation zwischen den Feldern erfolgt nach Spezifikation, aber meist auch mit der Tabulatortaste.

Die Konfiguration benutzerspezifischer Snippets wird in Kapitel 2.6 beschrieben. Zudem finden Sie in Kapitel 2.3 nützliche Plugins mit L^AT_EX-Bibliotheken.

In Kombination mit einer Autovervollständigung können verfügbare Snippets zusammen mit anderen Vorschlägen aufgelistet werden (vgl. Abb.3.1)

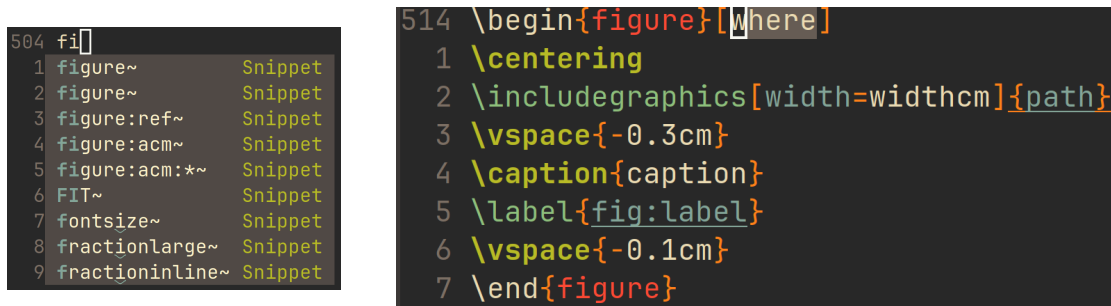


Abbildung 3.1: Einfügen von Grafik

3.4 buffers

Vim verfügt wie andere Textbearbeitungsprogramm über Fenster und Tabs. Fenster können vertikal und horizontal geteilt (`~w<v|h>`) und mit `~w<h|j|k|l>` navigiert werden. Näheres unter `:help windows` und `:help tabs`.

Erfahrene Nutzer nutzen die Vorteile der Buffer-Struktur des Editors. Laut Definition ist ein Buffer eine In-Memory-Repräsentation einer Datei oder eines Textinhalts, anders ausgedrückt beinhaltet er die Rohdaten einer Datei im Arbeitsspeicher und einige Metadaten (ID, Dateiname, Status, Typ, Flags). Mit `:w` werden diese Daten auf die Datei geschrieben. Es können mehrere Buffer von einer Datei geöffnet werden, was die Arbeit an verschiedenen Stellen eines Dokuments erleichtert. Sie sind nicht an ihre visuelle Darstellung gebunden, so kann ein Buffer in mehreren (oder keinem) Fenstern/Tabs angezeigt werden.

```
:help buffer
:new neuer Buffer (blank)
:e <Datei> neuer Buffer von bestimmter Datei
:buffers listet Buffer
:bn :bp nächster oder vorheriger Buffer
:bd schließt Buffer (+! für forced)
```

Im Kontext von $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Dokumenten kann es hilfreich sein zwei Buffer einer Datei zu öffnen, um schnell zwischen dem Dokumentenkörper und den Befehlsspezifikationen navigieren zu können. Hierfür können auch Marken gesetzt werden.

3.5 Sonstiges

Vorlagen

Das Nutzen von \LaTeX -Vorlagen spart Zeit und Energie. Online finden Sie eine Vielzahl an Skripten, geeignet für verschiedenste Anwendungen. Auch eine eigene Sammlung ist schnell aufgebaut. Häufig genutzte Strukturen können in Snippets oder Textdateien gespeichert werden.

bash befehl:

```
$ cat $TEMPLATES/latex/article2.tex >> meindokument.tex  
$ nvim meindokument.tex
```

direkt in vim:

```
:e $TEMPLATES/latex/article2.tex öffnet Template in neuem Buffer  
ggVGy kopiert gesamten Inhalt (gg: SOF, V: Visual-Block-Mode, G: EOF, y yank)  
:bpb zurück in vorherigen Buffer und einfüge n
```

oder mit Snippet.

Das Anlegen von Snippets kann mühsam sein. Alternativ können seltene Strings (<MUSTER>) als Platzhalter genutzt und mit /<MUSTER>+cw ersetzt werden.

Git

Git-Kommandos können je nach Vorzug über das integrierte Terminal, spezifische Plugins oder automatisch nach Konfigurationen im Init-Skript ausgeführt werden. *Good Practice* ist es ein externes Terminal oder ein separates Tab im Emulator der Wahl zu nutzen.

3.6 Anwendungsbeispiele

Bericht

Ausgangslage für einen Laborbericht seien eine Reihe an Notizen und Bildern:

```
$ls -a
apfel.txt apfel.png birne.txt birne.png ...
```

Nach wenigen Schritten ist das Fundament geschaffen:

- Laden der Vorlage und Öffnen des Editors:

```
$ cat $TEMPLATES/latex/bericht1.tex >> bericht.tex
$ nvim bericht.tex
```

- Erfassen der Dateinamen in Speicherregister a:

```
:let @a=system("ls *.txt")
```

- Einfügen in Dokument (vor `\end{document}`):

```
/doc<Enter>nnO<Esc>`ap
```

- Kapitelvorlage mit Snippet erstellen und in Register t ablegen:

```
O\sectionÖÖÖ<Enter>figure<Tab> ... ÖÖÖ.jpg<Tab> ... <Tab><Esc>9k"t9dd
```

- Start der Macroaufnahme und Einfügen von Notizen in Vorlage:

```
qq"nD"tp
```

```
:e <C-r>n<Enter>ggVGy:bd
```

```
p
```

Das Macro startet, kopiert (Register n) und löscht den Zeileninhalt (apfel.txt). Nach der Einfügung der Kapitelvorlage aus Register t wird ein neuer Buffer mit dem zuvor gespeicherten Dateinamen (Register n) geöffnet. Dessen Inhalt wird in das Standardregister kopiert und im vorherigen Buffer eingefügt.

- Kapitelname in Vorlage substituieren und Aufzeichnung beenden:

```
:%s/ÖÖÖ/<C-r>n<Return(x4)>/g
```

```
/\ .txt<Enter>:noh<Enter>O
```

Der Befehl `:noh` entfernt die Markierungen der Vorwärtssuche.

- Wiederholen für sämtliche Notizen:

```
4@q
```

Der automatisierte Ablauf schafft Zeit für die eigentliche Arbeit.

CSV

Daten einer CSV-Datei sind in einer Tabelle darzustellen:

- Snippet aktivieren und Daten einfügen:
obegin<Tab>tabular<Esc>la{ | c<Esc>hyWl6pa | }<Tab>
:e ex.csv<Enter>ggVGybdp
- Macro aufnehmen:
qq0:s/,/ & /g<Enter>A\\<Esc>jq
- Macro aufrufen:
10@q

4 Anhang

Abbildung 4.1 enthält eine Bildschirmaufnahme der für *vim*+ \LaTeX typischen Arbeitsumgebung. Das minimalistische Setup nutzt Raum- und Rechnerressourcen effizient. In diesem Beispiel wurde gerade eine Vorwärtssuche nach der Textstelle aus Zeile 43 durchgeführt. Die Passage erscheint im PDF-Dokument grün hervorgehoben.

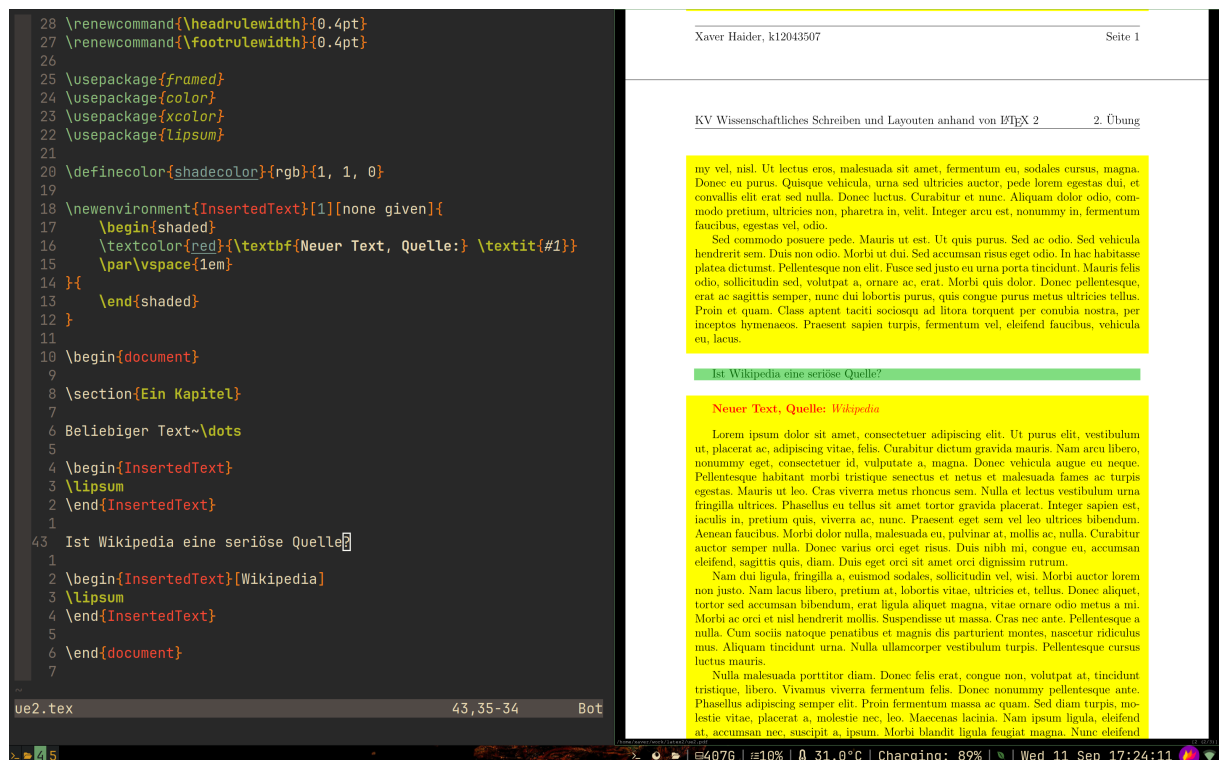


Abbildung 4.1: Screenshot Arbeitsumgebung