Automatisierung mit `make`

PeP et al. Toolbox Workshop

PeP et al. e.V.

Physikstudierende und
ehemalige Physikstudierende
der TU Dortmund

2024

Automatisierte, reproduzierbare Prozesse

Problem:

Kurz vor Abgabe noch neue Korrekturen einpflegen

1. Tippfehler korrigieren, Plots bearbeiten
2. TEX ausführen, ausdrucken

→ vergessen, Plots neu zu erstellen

→ zurück zu Schritt 1 ...

Lösung: Make

- → prüft, welche Dateien geändert wurden
- → berechnet nötige Operationen um Abhängigkeiten zu erfüllen
- → führt Befehle aus
  - → Python-Skripte
  - → TEX
  - → etc ...

Warum?

- → **Automatisierung** verhindert Fehler
- → Dient als **Dokumentation**
- → **Reproduzierbarkeit**: unverzichtbar in der Wissenschaft
- → **Spart Zeit**: nur notwendige Operationen werden ausgeführt

Ziel: Eingabe von `make` erstellt komplettes Protokoll/Paper aus Daten

Makefile

- Von `make` benutzte Datei heißt `Makefile` (keine Endung)
  - bei Windows Dateiendungen einschalten, siehe `http://techmixx.de/windows-10-dateiendungen-anzeigen-oder-ausblenden/`
- `Makefile` besteht aus Regeln (Rules):

Rule

```
target: prerequisites
  recipe
```

`target` Datei(en), die von dieser Rule erzeugt werden

`prerequisites` Dateien, von denen diese Rule abhängt

`recipe` Befehle, um vom `prerequisites` zu `target` zu kommen

- wird mit Tab unter `target: prerequisites` eingerückt

Einfachstes Beispiel

```
plot.pdf: plot.py data.txt
        python plot.py
```

- Wir wollen `plot.pdf` erzeugen (`target`)
  - `plot.pdf` hängt von `plot.py` und `data.txt` ab (`prerequisites`)
  - Der Befehl, um `plot.pdf` aus den `prerequisites` zu erhalten ist `python plot.py`

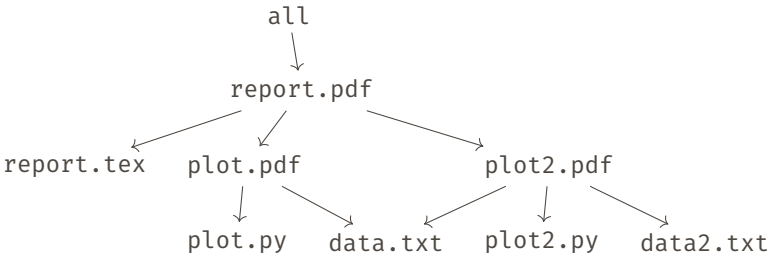
Beispiel

```
all: report.pdf  # convention

plot.pdf: plot.py data.txt
        python plot.py

report.pdf: report.tex
        lualatex report.tex

report.pdf: plot.pdf  # add prerequisite
```

`make` eingeben:

- → `all` braucht `report.pdf`
  - → `report.pdf` braucht `plot.pdf`
    - → `python plot.py`
  - → `lualatex report.tex`

Funktionsweise



- → Abhängigkeiten bilden einen DAG (directed acyclic graph/gerichteter azyklischer Graph)
- → Dateien werden neu erstellt, falls sie nicht existieren oder älter als ihre Prerequisites sind
- → Prerequisites werden zuerst erstellt
- → top-down Vorgehen

Argumente für `make`

`make` *`target`*	statt des ersten in der `Makefile` genannten Targets (meist `all`) nur `target` erstellen
`make -n`	dry run: Befehle anzeigen aber nicht ausführen
`make -B`	Force: ausführen aller Schritte, ignorieren des Alters aller Dateien
`make -p`	Datenbank aller Abhängigkeiten ausgeben

→ Nützlich, wenn man einen Plot bearbeitet: `make plot.pdf`

Projekte „sauber“ halten

make clean

(Nützliche) Konvention: `make clean` löscht alle vom `Makefile` erstellten Dateien/Ordner.

```
clean:
  rm plot.pdf report.pdf
```

Das Projekt sollte dann so aussehen, wie vor dem ersten Ausführen von `make`.

build-Ordner

build-Ordner: Projekt sauber halten

```
all: build/report.pdf

build/plot.pdf: plot.py data.txt | build
  python plot.py  # savefig('build/plot.pdf')

build/report.pdf: report.tex build/plot.pdf | build
  lualatex --output-directory=build report.tex

build:
  mkdir -p build

clean:
  rm -rf build

.PHONY: all clean
```

- → `| build` ist ein order-only Prerequisite: Alter wird ignoriert
- → Targets, die bei `.PHONY` genannt werden, erzeugen keine Dateien (guter Stil). Bsp: **`clean`** löscht Dateien, wird versehentlich eine Datei clean erstellt, soll trotzdem **`clean`** ausgeführt werden. Nennung hier hebt die Verwirrung von `make` auf, beugt vor.

build-Ordner

LaTeX und `biber` bieten Optionen an, um einen `build`-Ordner zu benutzen.

Aufrufe

```
lualatex --output-directory=build file.tex
biber build/file.bcf
```

Um Dateien aus dem `build`-Ordner zu finden (Plots, Tabellen):

Aufrufe

```
TEXINPUTS=build: lualatex --output-directory=build file.tex
biber build/file.bcf
```

- → `TEXINPUTS`, `BIBINPUTS`: Suchpfade für TeX- und `.bib`-Dateien
- → Elemente getrennt mit `:`, der erste Treffer wird genommen (wie `PATH`)
- → Hilfreich um z. B. den Header nur einmal für alle Protokolle abzuspeichern. (Siehe latex-template)
- → `TEXINPUTS` auch für `\includegraphics`
- → `:` am Ende der Liste: Standardsuchpfade anhängen (wichtig!)
- → `.` (der aktuelle Ordner) ist am Anfang der Standardliste, braucht man also nicht selbst angeben
- → Endet ein Element mit `//`, werden auch alle Unterordner durchsucht

LaTeX noch besser integrieren

nonstopmode

In Makefiles will man keine Interaktion.

Keine Interaktion

```
lualatex --interaction=nonstopmode file.tex
```

Beim ersten Fehler abbrechen

```
lualatex --interaction=nonstopmode --halt-on-error file.tex
```

Neben `nonstopmode` gibt es auch `batchmode`, was die Ausgabe nur in der `.log`-Datei speichert, aber nicht ausgibt.

Log schöner machen

```
max_print_line=1048576 lualatex file.tex
```

- Problem: Mehrfaches Kompilieren von Dokumenten ist aufwändig und fehleranfällig
- `latexmk` ist ein Kommandozeilenwerkzeug, das automatisch `tex` (und andere Programme wie `biber`) oft genug aufruft
- Bei TeXLive mitgeliefert
- Auswahl von LuaLaTeX durch Parameter `--lualatex`
- Versteht auch viele `tex`-Argumente wie `--interaction` und `--halt-on-error`

Aufruf auf der Kommandozeile

```
latexmk --lualatex --output-directory=build --interaction=nonstopmode
 ↪  --halt-on-error file.tex
```

- Noch mehr Kontrolle durch Konfigurationsdatei `latexmkrc`
- Siehe dazu Dokumentation

latexmk im Makefile

Im Makefile

```
build/file.pdf: FORCE plots... tabellen...
  TEXINPUTS=build: \
  max_print_line=1048576 \
  latexmk \
    --lualatex \
    --output-directory=build \
    --interaction=nonstopmode \
    --halt-on-error \
  file.tex

FORCE:

.PHONY: FORCE all clean
```

→ `latexmk` bestimmt Abhängigkeiten selbst

→ Sollte also immer ausgeführt werden

  → `FORCE:` definiert eine niemals erfüllte Abhängigkeit, sodass immer ausgeführt wird

Kontinuierliche Updates

```
latexmk -pvc --interaction=nonstopmode … document.tex
```

- → `latexmk` merkt, wenn ihr eure Dateien ändert
- → Kompiliert automatisch neu
- → Öffnet den Standard-PDF-Betrachter
- → Einfach im Hintergrund laufen lassen

Is the cake a lie?

Vergleich: Kuchen backen

```
Kuchen: Teig Backofen
  Ofen auf 140°C vorheizen
  Teig in Backform geben und in den Ofen schieben
  Kuchen nach 40 min herausnehmen

Teig: Eier Mehl Zucker Milch Rumrosinen | Schüssel
  Eier schlagen
  Mehl, Zucker und Milch hinzugeben
  Rumrosinen unterrühren

Rumrosinen: Rum Rosinen
  Rosinen in Rum einlegen
  Vier Wochen stehen lassen

Schüssel:
  Rührschüssel auf den Tisch stellen, wenn nicht vorhanden

clean:
  Kuchen essen
  Küche sauber machen und aufräumen
```

Expert

Können mehrere unabhängige Auswertungen parallel ausgeführt werden?
→ Ja: `make -j4` (nutzt 4 Prozesse (`j`: jobs) gleichzeitig, Anzahl beliebig)

Problem: Manchmal führt `make` Skripte gleichzeitig zweimal aus (hier `plot.py`)

```
all: report.txt

report.txt: plot1.pdf plot2.pdf
  touch report.txt

plot1.pdf plot2.pdf: plot.py data.txt
  python plot.py  # plot.py produziert sowohl plot1.pdf als auch plot2.pdf
```

Lösung: Ein `make`-Feature (`v4.3`): *grouped targets*

```
all: report.txt

report.txt: plot1.pdf plot2.pdf
  touch report.txt

plot2.pdf plot1.pdf &: plot.py data.txt  # das &: definiert die targets als group
    python plot.py
```

→ Alle `targets` werden durch eine einzigen Durchlauf der `recipe` (gebündelt) erzeugt.

Expert

Wenn ein Skript sehr viele Dateien erzeugt kann die Liste der `targets` unübersichtlich lang werden. Diese `targets` werden außerdem in der Regel als `prerequisites` verwendet, z. B. für die `recipe` der Berichtdatei → Die Liste befindet sich sogar zweimal im `Makefile`

```
all: report.txt

report.txt: plot1.pdf plot2.pdf plot3.pdf plot4.pdf plot5.pdf
  touch report.txt

plot1.pdf plot2.pdf plot3.pdf plot4.pdf plot5.pdf & : plot.py data.txt
  python plot.py  # plot.py produziert alle plot{i}.pdf
```

Lösung: Ein weiteres `make`-Feature: *variables*

```
all: report.txt

script_targets = plot1.pdf plot2.pdf plot3.pdf plot4.pdf plot5.pdf  # Variablen Definition

report.txt: $(script_targets)  # Variablen Verwendung
  touch report.txt

$(script_targets) & : plot.py data.txt  # Variablen Verwendung
  python plot.py
```

Die Variablenliste kann auch weiter bearbeitet werden. Mit `addprefix build/` wird in diesem Beispiel der `build` Ordner an den Anfang jedes Dateipfades geschrieben. `addsuffix .pdf` hängt an jeden Dateinamen die Endung `.pdf` an.

```
all: report.txt

plots = $(addprefix build/, $(addsuffix .pdf, plot1 plot2 plot3 plot4))  # Definition

report.txt: $(plots)  # Verwendung
  touch report.txt

$(plots) & : plot.py data.txt  # Verwendung
  python plot.py
```