Wissenschaftliches Programmieren

Bálint Aradi

http://www.bccms.uni-bremen.de/cms/people/b-aradi/

5. Versionsverwaltung für Einzelentwickler

Verwendete Literatur:

- **Scott Chacon: Pro Git** (Kapitel 1 und 2)
- GitHowto

Versionsvewaltung

Aufgaben eines Versionsvewaltungsystems:

- Entwicklungsgeschichte (Schnappschnüsse über den Zustand des Codes zu verschiedenen Zeitpunkten) festhalten
- Entwicklungskoordination bei mehreren Entwicklern erleichtern
- Parallele entwicklung von mehreren Versionen erleichtern

Zentralisierte Versionsverwaltung (CVS, Subversion, ...):

- Zentraler Server enthält Versionsdatenbank (Repository)
- Programmier müssen jedesmal Verbindung zum Server aufbauen, wenn Sie eine neue Version des Codes festhalten wollen.

Dezentralisierte Versionsverwaltung (Git, Mercurial, Bazaar, ...):

- Jeder Entwickler hat die gesammte Versionsgeschichte lokal auf seinem Rechner
- Neue Versionen des Codes werden lokal festgehalten und später zusammengeführt (keine ständige Netzwerkverbindung nötig)

Versionsverwaltung für Einzelentwicklung – Problemstellung

Szenario:

- Ein neues Programm wird entwickelt (z.B. einfache Gausselimination)
- Programm wird getestet, es funktioniert.
- Programm wird erweitert/umgeschrieben (z.B. Pivotverfahren eingebaut)
- Programm wird erweitert/umgeschrieben (z.B. Daten aus Datei eingelesen)
- :
- *Hoppala! Eine Funktionalität, die früher mit Sicherheit in Ordnung war, funktioniert nicht mehr (z.B. für gewisse Matrizen kommt das falsche Ergebnis raus.)*

Problemlösung mit Versionsverwaltung:

- Man sucht in der Versionsgeschichte so lange rückwärts (evtl. mit Bisektion), bis man die letzte funktionierende Version gefunden hat.
- Man untersucht die Veränderung zur nächsten Version (wo es das erste mal nicht funktioniert), um festzustellen, was die richtige Funktion zerstört hat und behebt den Fehler.

Sich der Versionsverwaltung vorzustellen

- Name und Emailadresse eintragen, damit diese beim Loggen verwendet werden:

```
git config --global user.name "Bálint Aradi"
git config --global user.email "aradi@uni-bremen.de"
```

Befehl Unterbefehl Option

- Standardtools (Editor und Codevergleicher) festlegen:

```
git config --global core.editor emacs
git config --global diff.tool kdiff3
git config --global merge.tool kdiff3
```

- Wenn die Option „--global“ angegeben wird, gelten die Einstellungen für jedes git-Projekt unter dem **aktuellen** Unixaccount automatisch.
- Die globalen Optionen sind in der Datei **~/.gitconfig** gespeichert.
- Alle aktuellen Optionen auflisten (systemweite, globale und projektspezifische)

```
git config --list
```

→

```
user.name=Bálint Aradi
user.email=aradi@bccms.uni-bremen.de
core.editor=emacs
:
```

Repository einrichten

- Projektrepository erzeugen:

```
mkdir ~/gauss_elimination
cd ~/gauss_elimination
git init
```

Erzeugt leere Versionsdatenbank in ~/gauss_elimination/.git

- Danach können Dateien im Verzeichnis ~/gauss_elimination und die Dateien in dessen Unterverzeichnissen unter Versionsverwaltung gestellt werden.
- Dateien im .git Verzeichniss dürfen nicht manuell verändert werden.
- Wenn Projektverzeichnis als ganzes (inkl. .git-Verzeichnis) kopiert wird, geht auch die Verionsgeschichte mit.

- Datei(en) im Projekt erzeugen, editieren (z.B. gauss_elim.f90)

```
emacs gauss_elim.f90 &
```

Editor im Hintergrund starten

Versionsverwaltungsstatus ändern

- Status der Dateien (bezüglich Versionsverwaltung) abfragen

```
git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include ...
#
#       gauss_elimination.f90
```

sog. **master branch** (Hauptzweig der Entwicklung)

Bis jetzt noch nichts unter VV gestellt

Dateien im Projektverzeichnis, die noch nicht unter VV stehen (nicht verfolgte Dateien)

- Nicht verfolgte Dateien müssen explizit unter VV gestellt werden:

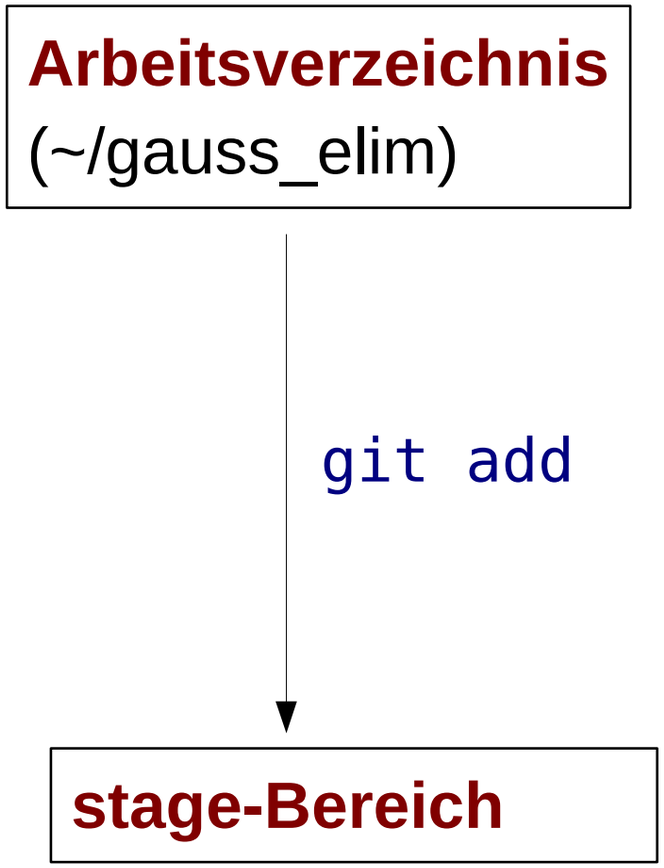
```
git add gauss_elimination.f90
git status
# :
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to ...
#
#       new file:   gauss_elimination.f90
```

Liste der Dateien, die beim nächsten Schnappschuss (commit) festgehalten werden. (stage-Bereich)

Dateien im stage-Bereich

- Beim „git add“ werden die Dateien in den sog. stage-Bereich gestellt
- Dateien im stage-Bereich werden beim nächsten Schnappschuss (commit) festgehalten.
- Dateien im stage-Bereich haben den **Zustand, den sie beim Ausführen des entsprechend „git add“ Befehls gehabt haben**. Werden sie nachträglich geändert, muss (wenn gewünscht) wieder „git add“ ausgeführt werden, damit ihr Status im stage-Bereich aktualisiert wird.

```
# wenn gauss_elimination.f90 verändert wurde:
git status
:
# Changes to be committed:
:
#       new file:   gauss_elimination.f90
#
# Changed but not updated:
#
#       modified:   gauss_elimination.f90
git add gauss_elimination.f90
```



Bühnenbild erstellen (commit)

- Schnappschuss über den Stand der Dateien im Stage-Bereich erstellen:

```
git status
 :
# Changes to be committed:
 :
#       new file:   gauss_elimination.f90
#
git commit
[master (root-commit) d910d55] Initial checkin of the working Gauss-
elimination algorithm.
 1 files changed, 93 insertions(+), 0 deletions(-)
 create mode 100644 gauss_elimination.f90
git status
# On branch master
nothing to commit (working directory clean)
```

→ Startet einen Editor für eine temporäre Datei, in die man eine kurze Beschreibung eingeben soll. Logmeldung wir mit dem Schnappschuss über die Dateien festgehalten.
(Datei speichern, Editor verlassen)

Eingegebene Beschreibung

Versionsgeschichte

- Festgehaltene Versionen (zusammen mit den Logmeldungen) anzeigen:

```
git log
commit d910d55495eb9ce34637d305bd5afcffbe731268
Author: Bálint Aradi <balint.aradi@bccms.uni-bremen.de>
Date:   Mon Apr 29 11:42:36 2013 +0200

    Initial checkin of the working Gauss-elimination algorithm.
```

Schnappschuss-Prüfsumme

Logmeldung

- Einzelne Schnappschüsse durch ihre Prüfsummen identifiziert
- Man kann auch verkürzte Prüfsummen verwenden, solange eindeutig
- Mit Option --oneline werden verkürzte Prüfsummen ausgegeben.
- Wenn mehrere Versionen festgehalten wurden, werden im Log immer die neueren Versionen als erstes angezeigt.

```
git log --oneline
10cb29e Adding README file.
d910d55 Initial checkin of the working Gauss-elimination algorithm.
```

letztes checkin

vorletztes checkin

Git-Workflow

0. Git global für den Unix-Account einrichten

   `git config --global ...`

1. Repository für ein Projekt erstellen

   `git init`

2. Dateien zum Projekt geben, Dateien verändern

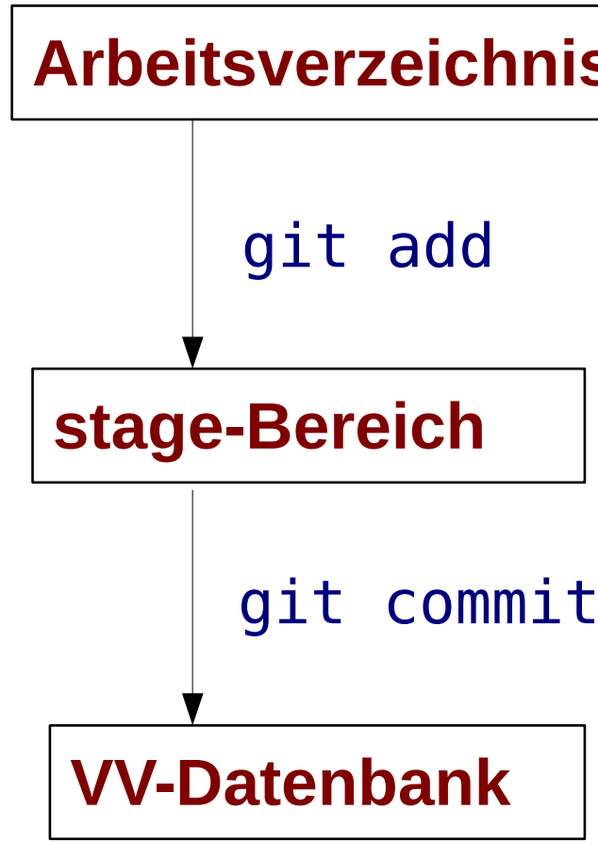
3. Dateien für den Schnappschuss vorbereiten
   (Dateien mit aktuellem Stand in stage-Bereich kopieren)

   `git add ...`

4. Zustand des Projektes festhalten
   (Dateien im stage-Bereich fotographieren/einchecken)

   `git commit`

5. Weiterentwickeln



Bemerkungen

- Es sollte in der Regel dann eingecheckt werden, wenn die Implementation eines neuen Features abgeschlossen wurde.
- Anhand der Logmeldungen sollte Entwicklungsverlauf nachvollziehbar sein.
- Die Menge der Veränderungen am Code zwischen zwei Schnappschüssen sollte überschaubar bleiben.
- Kürzere Logmeldungen können auch direkt in der Kommandozeile mit der Option „-m“ angegeben werden:

```
git commit -m "Initial checkin with working ..."
```

- Man kann den mit einem Befehl alles auf Stage setzen was sich geändert hat:

```
git add -u
```

Fügt alle Dateien dem Stage-Bereich zu, die unter Versionsverwaltung stehen und seit dem letzten Checkin geändert wurden.

Dateien umbenennen/löschen

- Datei umbenennen (wird auch im Arbeitsverzeichnis umbenannt):

```
git mv README README.txt
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README -> README.txt
```

- Datei vom Arbeitsverzeichnis löschen und von **künftigen** Schnappschüssen entfernen:

```
git rm README
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    README
```

← Datei wird ab nächsten commit nicht mehr verfolgt

Veränderungen untersuchen

- Unterschiede zwischen **aktueller Version im Arbeitsverzeichnis** und **Version im Repository/Stage** Bereich:

```
git diff gauss_elimination.f90
```

```
diff --git a/gauss_elimination.f90 b/gauss_elimination.f90
index ea7aefe..c35a48f 100644
--- a/gauss_elimination.f90
+++ b/gauss_elimination.f90
@@ -8,7 +8,7 @@ program GaussElimination
   implicit none
   !! precision
-  integer, parameter :: dp = selected_real_kind(15, 300)
+  integer, parameter :: dp = selected_real_kind(14, 300)
   !! Minimal leading element
   real(dp), parameter :: minpivot = 1e-10_dp
```

Zeile entfernt

Zeile dazugefügt

- Ohne Dateinamen werden die Änderungen für alle geänderten Dateien angezeigt:

```
git diff
```

Veränderungen untersuchen

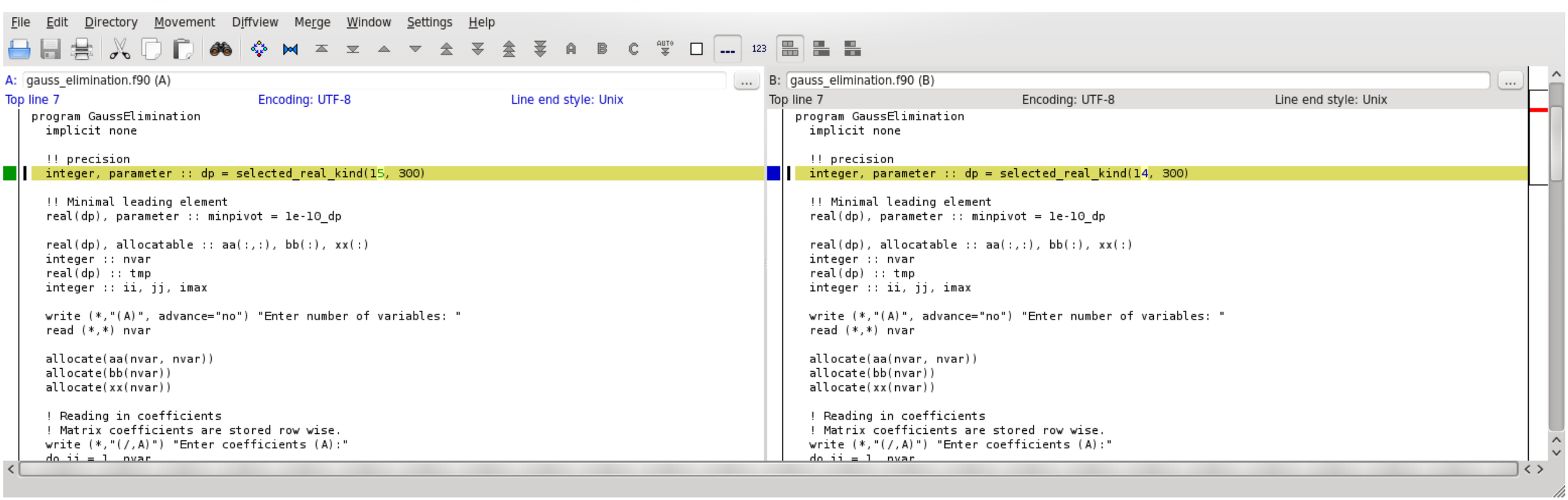
- Veränderungen zwischen zwei eingechekten Versionen werden durch die Angabe der entsprechenden Versionsprüfummen angezeigt:

```
git diff 10cb29e d910d55 -- gaus_elimination.f90
```

optional, wenn fehlt für alle Dateien

- Veränderungen können auch graphisch angezeigt werden:

```
git difftool
Viewing: 'gauss_elimination.f90'
Hit return to launch 'kdiff3':
```



Änderungen im Arbeitsverzeichnis rückgängig machen

- Datei im Arbeitsverzeichnis auf letzte eingechekte Version zurücksetzen:

```
git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   gauss_elimination.f90
#
no changes added to commit (use "git add" and/or "git commit -a")
```

```
git checkout -- gauss_elimination.f90
```

```
git status
# On branch master
nothing to commit (working directory clean)
```

Vorsicht, checkout überschreibt Dateien im Arbeitverzeichnis!

Änderungen im Stage-Bereich rückgängig machen

- Eine Datei kann aus dem Stage-Bereich wieder entfernt werden. Der Inhalt der Datei im Arbeitsverzeichnis wird dabei nicht verändert.

```
git st
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  modified:   README
#
git reset HEAD README
Unstaged changes after reset:
M  README
git st
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#  modified:   README
```

Frühere Versionen untersuchen

- Frühere Version via Prüfsumme auswählen und Dateien im Arbeitsverzeichnis auf den entprechenden Stand ändern:

```
git log --oneline
10cb29e Adding README file.
d910d55 Initial checkin of the working Gauss-elimination algorithm.

git checkout d910d55
Note: checking out 'd910d55'.
:
HEAD is now at d910d55... Initial checkin of the working Gauss-
elimination algorithm.

git status
# Not currently on any branch.
nothing to commit (working directory clean)
```

- In diesem Zustand kann man den Code untersuchen, ob der Fehler in einer gegebenen Version schon da war.
- Es sollten **keine Änderungen vorgenommen und eingecheckt** werden, da sie beim Zurückwechseln zum letzten Stand verlorengehen würden.

Zurückwechseln auf den aktuellen Stand

- Wenn der Fehler gefunden wurde, sollte er im aktuellen Stand der Entwicklung behoben werden.
- Zurückwechseln zum aktuellen Projektstand:

```
git checkout master
Previous HEAD position was d910d55... Initial checkin of the working..
Switched to branch 'master'

git status
# On branch master
nothing to commit (working directory clean)
```

- Nach dem Zurückwechseln zum aktuellen Stand sollte der Fehler behoben werden und die Änderungen eingecheckt werden.

git Aliases

- Häufig benutzte git-Befehle können mit aliases abgekürzt werden:

```
git config --global alias.ci commit                    →    git ci
git config --global alias.co checkout                       git co ...
git config --global alias.st status                         git st
git config --global alias.gdiff difftool                    git gdiff ...
git config --global alias.slog "log                         git slog
--pretty=format:\"%h | %ad | %s%d [%an]\"
--graph --date=short --all"
```

Versionen taggen

- Wenn eine Version eine spezielle Bedeutung hat (z.B. veröffentlichte Version des Projektes), kann diese mit speziellen Namen versehen werden.
- Wir verwenden die sogenannten „annotated tags“.
- Per default wird die letzte eingechekte Version mit dem Tag versehen

```
git slog
* ba8a4bd | 2013-04-29 | Test (HEAD, master) [Bálint Aradi]
* 10cb29e | 2013-04-29 | Adding README file. [Bálint Aradi]
* d910d55 | 2013-04-29 | Initial checkin of the working ...
```

```
git tag -a v1.0
git slog
 * ba8a4bd | 2013-04-29 | Test (HEAD, v1.0, master) [Bálint Aradi]
```

- Beliebige Versionen können mit Angabe der Versionsprüfsumme getagt werden:

```
git tag -a v0.1 d910d55
```

- Statt Revisionsprüfsummen kann überall auch der Tagname genutzt werden:

```
git diff v0.1 v1.0
```

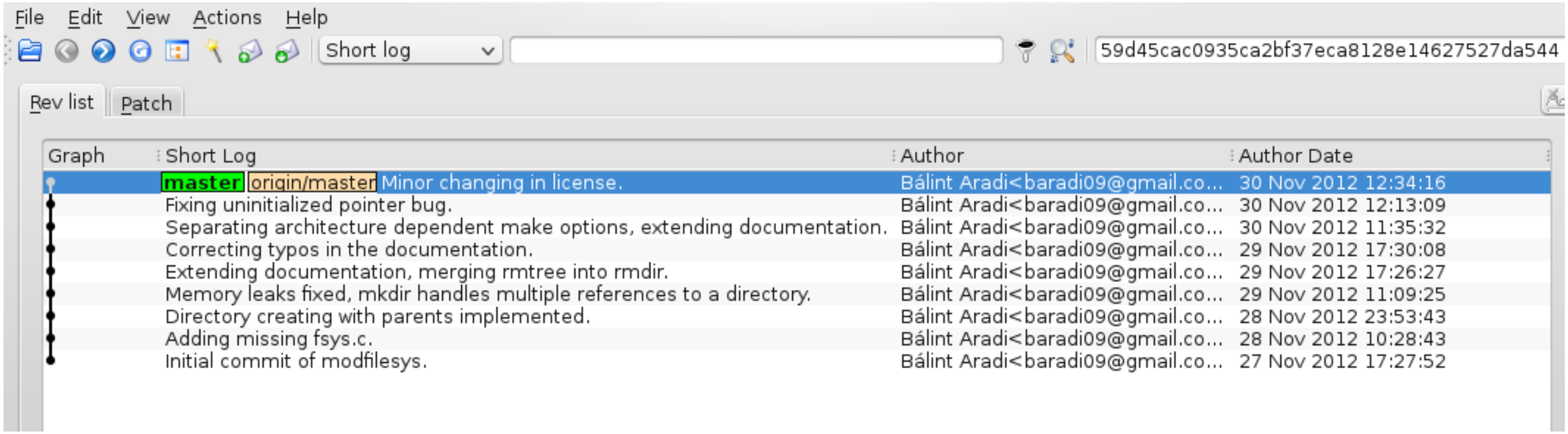
git-Hilfe, graphische Oberflächen

- Zu jedem git-Befehl ist eine Hilfeseite mit „git help“ verfügbar:

  ```
  git help commit
  ```
  ← Zeigt die Hifeseite für das commit-Befehl an

- Es gibt diverse graphische Oberflächen (gitk, qgit, …) um Revisionsgeschichte anzuzeigen bzw. Git-Befehle auszuführen:

  ```
  qgit &
  ```



Repository klonen

- Bestehende Projektrepositories (inkl. Geschichte) klonen:

```
git clone http://www.bccms.uni-bremen.de/fileadmin/BCCMS/CMS
/personen/aradi/wissprog/gausselim.git
git slog
```

```
* df9c53d | 2013-06-17 | Changed to LAPACK-solver...
* 0cc10ed | 2013-05-25 | Efficient storage for L, U and P implemented...
* 69cb0fe | 2013-05-25 | Program for automatic input generation added...
* f3fc43d | 2013-05-24 | Autotesting for linearly dependent equations...
```

- Im geklonten Repository kann man wie in einem eigenem arbeiten, neue Versionen einchecken, etc.

git – Bemerkungen

- Bitte für genaue Form und weitere Optionen der Befehle die Hilfeseiten lesen!
- Nach jeder größeren (nicht trivialen) Erweiterung sollte eingechekt werden. **Faustregel**: Ein anderer Programmierer sollte die Veränderungen zwischen zwei Versionen noch (mit Hilfe der eingechekten Mitteilung) überblicken können.
- Versionsgeschichte ist im Unterverzeichnis .git gespeichert. Wird das Verzeichnis (samt .git) kopiert, bleibt die Versionsinformation erhalten.
- Die git-Befehle müssen in dem Verzeichnis ausgeführt werden, in dem sich der .git Unterverzeichnis befindet (*Quellcode-Root-Verzeichnis*), oder in einem Unterverzeichnis davon.
- git-Befehle (sofern keine explizite Dateinamen angegeben) beziehen sich auf den gesammten Quellcode-Verzeichnisbaum (Quellcode-Root + Unterverzeichnisse)
- Versionsprüfsummen werden global für alle Dateien vergeben: eine Version enthält alle Dateien des Projektes. Bei den Dateien, die beim Checkin nicht im Stage-Bereich waren, wird der Stand vom letzten Checkin genommen.

Aufgabe 1

- Stellen Sie die aktuelle Version Ihrer Gauss-Elimination unter Versionskontrolle und checken Sie es ein.
- Erweitern Sie die Gauss-Elimination, damit diese die Spezifikationen vom letzten Mal erfüllt.
- Erweitern Sie das Programm, damit es die Daten von einer externen Datei lesen kann und checken Sie diese Version mit entsprechender Mitteilung ein. Stellen Sie auch eine Beispiel-Input-Datei unter Versionskontrolle (gauss.inp.example).
- Schreiben Sie eine kurze Dokumentation in eine Textdatei (README.txt) über die Bedienung des Programmes (z. B. wie die Inputdatei aufgebaut sein muss) und stellen Sie diese unter Versionskontrolle. Checken Sie diese ein.
- Versehen Sie diese Version des Codes mit dem Tag v0.1

Aufgabe 1 (Spezifikation)

- Schreiben Sie das Gauss-Eliminationsprogramm so um, dass die Eingabe von einer Datei namens „gauss.inp“ erfolgt.
- Die einzelnen Zeilen der Datei sollen folgende Daten enthalten:
  - Anzahl_der_Variablen
  - Koeffizienten der 1. Reihe der Matrix
  - Koeffizienten der 2. Reihe der Matrix
  - :
  - Koeffizienten der letzten Zeile der Matrix
  - Koeffizienten des Vektors auf der rechten Seite

Möglicher Input:

```
3
2.0 4.0 4.0
1.0 2.0 -1.0
5.0 4.0 2.0
1.0 2.0 4.0
```

Aufgabe 2

- In der Numerik wird die Gauss-Elimination meistens als LR-Zerlegung realisiert. Dabei wird die Matrix A als Produkt **P** * **A** = **L** * **R** zerlegt, wobei **P** eine Permutationsmatrix, **R** eine obere Dreiecksmatrix (gauss-eliminierte Form des Gleichungssystems) und **L** eine untere Dreiecksmatrix mit 1-en in der Diagonale ist, die die Koeffizienten für die Linearkombinationen der Zeilen enthält.

```
0.0  0.0  1.0     2.0  4.0  4.0       1.0  0.0  0.0   5.0  4.0  2.0
1.0  0.0  0.0  *  1.0  2.0 -1.0   =   0.4  1.0  0.0 * 0.0  2.4  3.2
0.0  1.0  0.0     5.0  4.0  2.0       0.2  0.5  1.0   0.0  0.0 -3.0
```

- Gleichungssystem **A** b = **b** wird mit Hilfe von **P**, **L** und **R** in 2 Schritten gelöst:

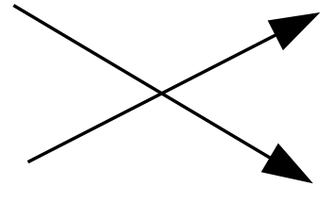
  L y = **P** **b** durch „vorwärtseinsetzen“ nach y aufgelöst.

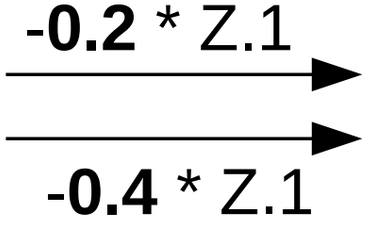
  R **x** = y durch „rückwärtseinsetzen“ nach x aufgelöst.

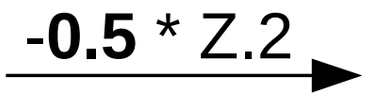
- Schreiben Sie das Gausseliminationsprogramm so um, dass es zuerst die Matrizen **P**, **L** und **R** aus **A** erzeugt, und dann durch Vorwärts- und Rückwärtseinsetzen den Lösungsvektor **x** ermittelt. (Vergessen Sie nicht, das fertige Program in das Repository einzuchecken...)

Aufgabe 2 (Beispiel für die LR-Zerlegung)

P			L			R		
1.0	0.0	0.0	1.0	0.0	0.0	2.0	4.0	4.0
0.0	1.0	0.0	0.0	1.0	0.0	1.0	2.0	-1.0
0.0	0.0	1.0	0.0	0.0	1.0	5.0	4.0	2.0
0.0	**0.0**	**1.0**	1.0	0.0	0.0	**5.0**	**4.0**	**2.0**
0.0	1.0	0.0	0.0	1.0	0.0	1.0	2.0	-1.0
1.0	**0.0**	**0.0**	0.0	0.0	1.0	**2.0**	**4.0**	**4.0**
0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
0.0	1.0	0.0	**0.2**	1.0	0.0	**0.0**	**1.2**	**-1.4**
1.0	0.0	0.0	**0.4**	0.0	1.0	**0.0**	**2.4**	**3.2**
0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
1.0	**0.0**	**0.0**	**0.4**	1.0	0.0	**0.0**	**2.4**	**3.2**
0.0	**1.0**	**0.0**	**0.2**	0.0	1.0	**0.0**	**1.2**	**-1.4**
0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
1.0	0.0	0.0	0.4	1.0	0.0	0.0	2.4	3.2
0.0	1.0	0.0	0.2	**0.5**	1.0	**0.0**	**0.0**	**-3.0**







Aufgabe 2 (Beispiel für Vorwärts- und Rückwärtseinsetzen)

b		**P * b**
1.0	→	4.0
2.0		1.0
4.0		2.0

Vorwärtseinsetzen:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.4 & 1.0 & 0.0 \\ 0.2 & \mathbf{0.5} & 1.0 \end{pmatrix} \begin{pmatrix} Y1 \\ Y2 \\ Y3 \end{pmatrix} = \begin{pmatrix} 4.0 \\ 1.0 \\ 2.0 \end{pmatrix} \rightarrow$$

$$Y1 = 4.0$$
$$Y2 = 1.0 - 0.4 * Y1 = -0.6$$
$$Y3 = 2.0 - 0.2 * Y1 - 0.5 * Y2 = 1.5$$

Rückwärtseinsetzen:

$$\begin{pmatrix} 5.0 & 4.0 & 2.0 \\ 0.0 & 2.4 & 3.2 \\ 0.0 & 0.0 & -3.0 \end{pmatrix} \begin{pmatrix} \mathbf{X1} \\ \mathbf{X2} \\ \mathbf{X3} \end{pmatrix} = \begin{pmatrix} 4.0 \\ -0.6 \\ 1.5 \end{pmatrix}$$

$$\mathbf{X3} = 1.5 / -3.0 = -0.5$$
$$\mathbf{X2} = (-0.6 - 3.2 * \mathbf{X3}) / 2.4 = 0.41666...$$
$$\mathbf{X1} = (4.0 - 4.0 * \mathbf{X2} - 2.0 * \mathbf{X3}) / 5.0 = 0.66666...$$