Wissenschaftliches Programmieren

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

9. Dokumentation, Aspekte der numerischen Programmierung, Programmtests

Quellen:

- Doxygen-Handbuch
- Metcalf et al., Fortran 95/2003 explained
- Oliveira et al., Writing Scientific Software
- Stoer, Numerische Mathematik 1

Quellcodedokumentation

Externe Dokumentation der Schnittstellen

- **Schnittstellen** (Module, Unterprogramme etc.) müssen dokumentiert werden, damit Zweck der Komponenten klar ist, und zueinander passende Komponenten geschrieben werden können.
- Schnittstellendokumentation sollte unabhängig vom Quellcode lesbar sein (z.B. stehen manche Bibliotheken nur in binärer Form, ohne Quellcode zur Verfügung.)
- Dokumentation sollte soweit wie möglich automatisch vom Quellcode erzeugt werden, damit sie auch nach Veränderungen im Code aktuell bleibt.

Interne Dokumentation des Programmcodes:

- Außer der Schnittstellendokumentation sollen auch der Programmcodestellen dokumentiert werden, die nicht trivial sind.
- Diese Dokumentation sollte möglichst kompakt sein, damit sie die Lesbarkeit der Programmcodes nicht beeinflußt.
- Diese Information soll **nicht in externe Dokumentation** extrahiert werden, sollte also **nicht in Doxygen-Format** geschrieben werden.

Doxygen

- Code-Dokumentationssystem, ursprünglich für C++ Quellcode geschrieben, später für Java, Python, PHP etc. erweitert
- Anwendung auf Fortran hat Einschränkungen
- Kein Dokumentationssystem für Fortran 95/2003 mit vergleichbarer Funktionalität.
- Alternative: Ford https://github.com/cmacmackin/ford

---

1. Erzeugung einer Konfigurationsdatei (Doxyfile)

```
doxygen -g
```

2. Anpassen der Konfigurationsdatei

```
PROJECT_NAME = GaussElimination
OUTPUT_DIRECTORY = doc
MULTILINE_CPP_IS_BRIEF = YES
OPTIMIZE_FOR_FORTAN = YES
```

3. (HTML und LaTeX) Dokumention aus Kommentaren im Quellcode erstellen:

```
doxygen
```

Doxygen (#2)

- Doxygen erfordert **spezielle Kommentare** für Schnittstellendokumentation.

```
!> Transforms a square matrix to upper triangle form together with a vector.
!!
!! The provided matrix is transformed to upper triangle for using
!! Gauss elimination technique. The subroutine stops if linear dependency is
!! detected.
!!
subroutine transformToUpperTriangle(aa, bb)

  !> Matrix to transform. Contains the upper triangle form on exit.
  real(dp), intent(inout) :: aa(:,:)

  !> Vector to transform. Contains the transformed vector on exit.
  real(dp), intent(inout) :: bb(:,:)
```


- Kommentar **vor** dem Block, der dokumentiert werden soll.
- Unterprogramname, Variablenattribute etc. werden bei der Erstellung der Dokumentation automatisch aus dem Quellcodetext entnommen.

Schnittstellendokumentation (#1)

Folgende Objekte sollten (mit Doxygen) dokumentiert werden:

- **Modul als ganzes** (Ziel des Modules)

```
!>  Contains input related routines.
!!
!! Eventual some longer description, if needed.
!!
module Input
```

- **Exportierte Objekte** in Modulen
  (Variablen, Typdeklarationen, Interface, etc.)

```
!> Contains accuracy related settings.
module Accuracy
  implicit none

  !> Kind for double precision.
  integer, parameter :: dp = selected_real_kind(15, 99)
  :
```

Schnittstellendokumentation (#2)

- (Öffentliche) **Unterprogramme in Modulen**
  - Beschreibung
  - Parameterdokumentation
  - Erwartungen gegenüber eingehende Parameter Voraussetzungen
  - Zurückgegebene Werte
  - Hinweise auf nicht abgefangene Fehler, abnormales Verhalten

```
!> Transforms a square matrix to upper triangle form together with a vector.
!!
!! The provided matrix is transformed to upper triangle form using
!! Gauss elimination technique. The applied row exchanges and linear
!! combination of rows are also applied to the provided vector, to keep the
!! linear system of equations invariant. The subroutine stops if linear
!! dependency is detected.
!!
subroutine transformtouppertriangle(aa, bb)

  !> Matrix to transform. Contains the upper triangle form on exit.
  real(dp), intent(inout) :: aa(:,:)

  !> Vector to transform. Contains the transformed vector on exit.
  real(dp), intent(inout) :: aa(:,:)
```

Doxygen und Fortran 2003

- Module werden in „namespaces“ verwandelt (in älteren Doxygenversionen)
- Doxygen kann nicht richtig zwischen privaten und öffentlichen Modulvariablen, Modulunterprogrammen, etc. unterscheiden
- Manche Variablendeklarationen (z.B. character(len=*)) werden nicht richtig interpretiert.
- Doxygen kann viel-viel mehr: Formeln in der Dokumentation, zusätzliche Dokumentationsseiten (z.B. für Handbuch), automatisches Verlinken von Begriffen, etc. (Siehe Handbuch).

Prioritäten

Prioritäten bei der Entwicklung numerischer Software:

- **Korrektheit**
  (Wahl des richtigen numerischen Algorithmus, korrekte Implementierung)
- **Numerische Stabilität**
  (z.B. Numerischer Fehler schaukelt sich nicht hoch)
- **Genaue Diskretisierung**
  (Diskretisierung muss entsprechend der maximalen Fehlertoleranz gewählt werden)
- **Flexibilität**
  (das Programm und auch Teile des Programms sind wiederverwendbar – dazu gehört auch richtige Dokumentation!)
- **Effizienz**
  Sowohl Zeit- als auch Speicherbedarf sollte nach Möglichkeit minimiert werden (meistens Kompromiss nötig, da Zeit- und Speicherbedarf sich oft reziprokal verhalten)

Zahlendarstellung (reelle Zahlen)

- Zahlen werden mit endlicher Genauigkeit dargestellt
- Intern wird immer das binäre Zahlensystem verwendet

$$x = \pm\, \underbrace{a_m 2^m + a_{m-1} 2^{m-1} + \ldots +}_{n_1} \underbrace{a_0 2^0 + a_{-1} 2^{-1} + a_{-2} 2^{-2}}_{n_2} + \ldots \qquad a_i = 0, 1$$

$$n = n_1 + n_2$$

- Grundsätzlich zwei Darstellungsarten für reelle Zahlen:
  - **Fixkomma-** bzw. Festpunktdarstellung
  - **Fließkomma-** bzw. Gleitpunktdarstellung

Fixkomma-/Festpunktdarstellung

- n_1 und n_2 werden fixiert (in der Regel $n_1 = n$ und $n_2 = 0$)

  $p = c * z$

  z ganze Zahl (integer)

  c festgelegte Konstante (für jede Zahl gleich)

  c = 0,00001
  z = 123456789 → p = 1234,56789

- Darstellbarer Bereich hängt vom Integerbereich ab:

  z.B. 4 Byte Integers, c = 0,00001 $-2^{31} = -2147483648 <= z <= 2^{31} = 2147483648$

  $-21474{,}83648 <= p <= 21474{,}83648$

- Vorteil: Rechenoperationen sind Integer-Rechenoperationen: sehr schnell

- Nachteil: Unkontrollierter **Überlauf** mit fatalen Folgen.

  21474,50000 + 0,40000 = **-21474,77296**

  Siehe Steuerungssystem von Ariane 5:
  „pre-flight“: Fixkomma, „flight“: Fließkomma

- Nur von sehr wenigen Programmiersprachen nativ unterstützt (z.B: ADA)

Fließkomma-/Gleitpunktdarstellung

- Ort des Dezimalkommas ist nicht festgelegt

$p = x * b^e$

Mantisse:	$x = 0{,}x_1x_2...x_t$
Exponent:	e
Basis:	b

- Normalisierte Darstellung:

$|x| >= b^{-1}$

- In den meisten Architekturen:

Eingabe/Ausgabe: b = 10
Intern: **b = 2**

- Bei gegebener Anzahl von Bytes für die Darstellung der Mantisse und des Exponenten Anzahl der darstellbaren Zahlen endlich.

- Es gibt nicht darstellbare Zahlen

$0{,}1 \rightarrow 1{,}0 * 10^{-1}$ **aber binär:** $0{,}1 \rightarrow 1{,}100110011001... * 2^{-4}$

Rundung

- Ergebnis einer arithmetischen Operation kann außerhalb des darstellbaren Bereiches liegen, sogar dann wenn die Operanden darstellbar sind!

4 Stellen für Mantisse: $a = 1{,}0000 = 0{,}1000\ 10^{-1}$

$b = 0{,}00001 = 0{,}1000\ 10^{-5}$ → $a + b = 1{,}00001$

darstellbar — nicht darstellbar!

- Nicht darstellbare Zahlen werden gerundet:

Dezimal:

$$rd(x) := \begin{cases} 0{,}x_1x_2x_3..x_t & 0 <= x_{t+1} < 5 \\ 0{,}x_1x_2x_3..x_t + 10^{-t} & x_{t+1} >= 5 \end{cases}$$

Binär:

$$rd(x) := \begin{cases} 0{,}x_1x_2...x_t & x_{t+1} = 0 \\ 0{,}x_1x_2...x_t + 2^{-t} & x_{t+1} = 1 \end{cases}$$

- Relativer Fehler wegen Rundung:

$$\left|\frac{rd(x)-x}{x}\right| \leqslant \frac{5\cdot 10^{-(t+1)}}{|x|} \leqslant 5\cdot 10^{-t}$$

Rundungseinheit: u

⇒ $rd(x) = x(1+\delta)$

$|\delta| <= u$

Fundamentale Parameter

Rundungseinheit (*unit roundoff*) $\boldsymbol{u}$:

Kleinste positive Zahl, für die 1+**u** nicht 1 ist

Maschinengenauigkeit (*machine epsilon*) $\boldsymbol{\varepsilon}$:

Kleinster Wert für *a* - 1, wobei *a* die kleinste darstellbare Zahl ist, die größer als 1 ist.

- In binärer Arithmetik gilt meistens: $\varepsilon = 2u$
- Beispiel: dezimaler Arithmetik mit 4 Mantissenstellen:

(π wäre in diesem System als $3.141 \times 10^0 = 0.3141 \times 10^1$ dargestellt)

rd(1.000 + 0.0001) = rd(1.0001) = 1.000 → Unterscheidet sich von 1 nicht

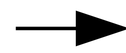
rd(1.000 + 0.0005) = rd(1.0005) = 1.001 → $u = 0.0005 = 5 \times 10^{-4}$

Kleinste darstellbare Zahl, größer 1: $a = 1.001$ → $\varepsilon = 10^{-3} = 2u$

Maximaler Exponent $\boldsymbol{E_{max}}$: Maximaler Absolutwert des Exponenten E

$-E_{max} <= E <= E_{max}$

$1/B <= X < 1$



kleinste darstellbare Zahl: B^{Emax-1}

größte darstellbare Zahl: $(1-\varepsilon)\, B^{Emax}$

Gleitkommaarithmetik

- Bei den meisten Fließkommaoperationen muss gerundet (oder gekürzt) werden:

Bsp: Arithmetik mit 5 Dezimalstellen:

$0.12345 + 0.31415 \times 10^{-1} = 0.326495 \times 10^{-1}$ $\xrightarrow{\text{gerundet}}$ 0.32650×10^{-1}

$\xrightarrow{\text{gekürzt}}$ 0.32649×10^{-1}

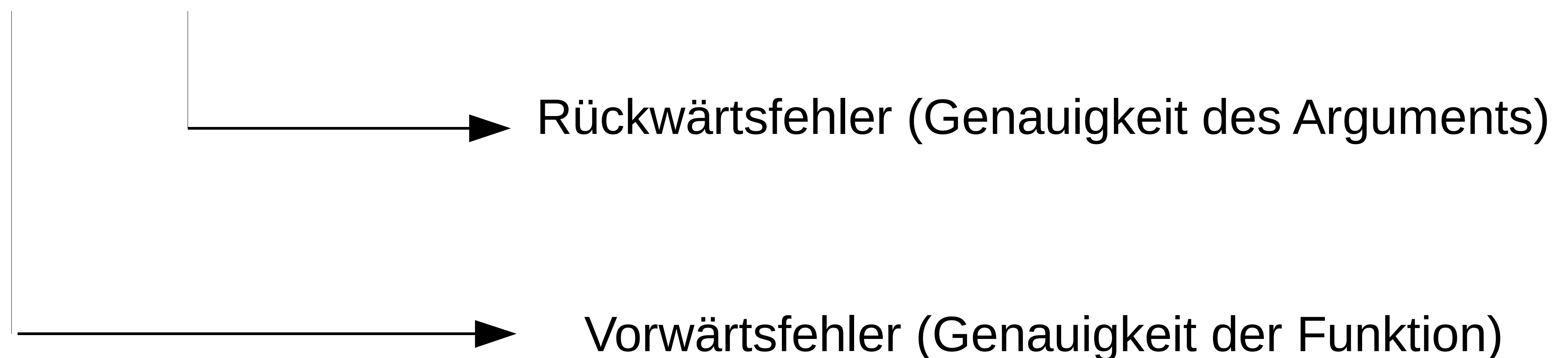
- Fließkommaergebnis *fl* einer arithmetischen Operation <*op*>:

$fl(x \text{ <op> } y) = (x <op> y)(1 + \delta) \qquad |\delta| <= u$

Gilt für IEEE-Arithmetik, solange kein Überlauf/Unterlauf auftritt

- Bestmögliches Fließkommaergebnis einer Funktionsauswertung:

$fl(f(x)) = (1 + \delta_1)\, f((1+\delta_2)x) \qquad |\delta_1|, |\delta_2| <= u$



Fehlerfortpflanzung

- Wegen Fehlerfortpflanzung ist die Ausführungsreihenfolge von arithmetischen Operationen wichtig!

Beispiel:

```
a =  0.23371258E-4
b =  0.33678429E+2
c = -0.33677811E+2
```

Addition mit 10-Stelliger Mantisse

a + b + c = = **0.64137126E-3**

a +* (b +* c) = 0.23371258E-4 + 0.61800000 = **0.64137126E-3**

(a +* b) +* c = 0.33678452E-2 + (-0.33677811E-2) = **0.64100000E-3**

Berechnung in 2 Schritten:

$z^* := (b + c)(1 + \varepsilon_1)$ $\quad y^* = (a + z^*)(1 + \varepsilon_2)$ $\quad \frac{y^* - y}{y} \approx \frac{b+c}{a+b+c} \cdot \epsilon_1 + 1 \cdot \epsilon_2$ $\quad \frac{b+c}{a+b+c} \approx 1$

$z^* := (a + b)(1 + \varepsilon_1)$ $\quad y^* = (z^* + c)(1 + \varepsilon_2)$ $\quad \frac{y^* - y}{y} \approx \frac{a+b}{a+b+c} \cdot \epsilon_1 + 1 \cdot \epsilon_2$ $\quad \frac{a+b}{a+b+c} \approx 5 \cdot 10^4$

Katastrophale Fehlerverstärkung

IEEE 754 Fließkommazahlen

	V	M	E	Tot.	Max.	*u*
• einfache Genauigkeit (*single precision*)	1	23	8	32	10^{38}	6×10^{-8}
• doppelte Genauigkeit (*double precision*)	1	52	11	64	10^{308}	2×10^{-16}
• erweiterte Genauigkeit (*extended precision*)	1	64	15	80	10^{4932}	5×10^{-20}
• vierfache Genauigkeit (*quadruple precision*)	1	112	15	128	10^{4932}	1×10^{-34}

- Fließkommaeinheit der x86 Prozessoren hat 80 bit Registers:
  - Eingehende Zahlen nach erweiterter Genauigkeit konvertiert
  - Arithmetische Operation wird ausgeführt
  - Zahl wird zurückkonvertiert

- Viele Programmiersprachen unterstützen erweiterte Genauigkeit nicht

> Es sollten nach Möglichkeit Fließkommazahlen mit **doppelter Genauigkeit** verwendet werden (außer Speicherbedarf ist kritisch oder Speicher-Prozessor-Kommunikation zu langsam).

Überlauf und Unterlauf (overflow, underflow)

Überlauf (overflow): Ergebnis einer Operation größer als die größte Dartellbare Zahl

Unterlauf (underflow): Absolutwert des Ergebnisses einer Operation kleiner als der kleinstmögliche noch darstellbare Absolutwert

Spezielle Zahlenwerte:

Infinity|Inf	Unendlich
NaN	Keine Zahl (not a number)

```
1.0 / 0.0                 +Infinity
-1.0 / 0.0                -Infinity
1.0 / 0.0 – 1.0 / 0.0     NaN
sqrt(-1.0)                NaN
```

Abfrage Fortran 95:

```
real(dp) :: xx
:
# Check for NaN
if (xx /= xx) then
  write(*,*) “Not a number”
end if

# Check for Infinity
if (abs(xx) > huge(xx)) then
  write(*,*) “Infinity”
end if
```

Abfrage Fortran 2003:

```
use, intrinsic :: ieee_arithmetic
real(dp) :: xx
:
# Check for NaN
if (ieee_is_nan(xx)) then
  write(*,*) “Not a number”
end if

# Check for Infinity
if (.not. ieee_is_finite(xx)) then
  write(*,*) “Infinity”
end if
```

Wichtige Regel

- Fließkommavariablen sollten **nie** auf exakte Gleichheit geprüft werden, sondern auf absoluten oder relativen Fehler.

Falsch!

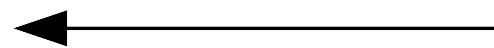
```
if (xx == x0) then
  :
end if
```

Richtig!

```
if (abs(xx - x0) <= tol) then
  :
end if

if (abs(xx - x0) <= relTol * x0) then
  :
end if
```

```
real(dp) :: xx, yy
:
yy = xx
:
if (xx == yy) then
  :
end if
```



Sogar das kann schief gehen, wenn *xx* im Prozessoregister (80 bit) gespeichert ist und *yy* im Speicher (64 bit)

Wichtige Regel (#2)

- Fließkommazahlen sollten möglichst nicht sukzessiv addiert oder subtrahiert werden

Falsch!

```
xx = 1.0_dp / 9.0_dp
yy = 0.0_dp
do ii = 1, 9
  yy = yy + xx
  :
end do
write (*,*) yy - 1.0_dp
```

→ 2.22044604925031 3E-16

Richtig!

```
xx = 1.0_dp / 9.0_dp

do ii = 1, 9
  yy = real(ii, dp) * xx
  :
end do
write (*,*) yy - 1.0_dp
```

→ 0.0

- Man sollte möglichst nicht Zahlen in der selben Größenordnung subtrahieren und dann mit einer kleinen Zahl dividieren:

```
do ii = 2, 8
  xx = 10.0**(-ii)
  write(*,*) (1.0_dp - cos(xx)) / (xx * xx)
end do
```

→

```
0.499995833347912
0.499999958345600
0.500000000018929
0.500000066632313
0.500044452816152
0.499600349404564
0.000000000000000E+000
```

Katastrophale Kompensation

Wird auf 1.0 gerundet (cos(xx)) – ergibt 0.0

Testen von Programmen

Programm schreiben

Funktionalität anhand ausgewählter Beispiele überprüfen

Progamm erweitern

Neue Funktionalität anhand ausgewählter Beispiele überprüfen

Funktionieren die alten Sachen noch?!

Programm schreiben

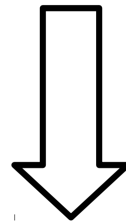
Funktionalität überprüfen

Programm verteilen:
- anderer Compiler
- anderes Betriebssystem
- andere Machinenarchitektur

Funktionieren die Sachen genauso, wie sie bei den Entwicklern funktioniert haben?

Testen von Programmen

- Nach jeder (größeren) Änderung muss möglichst die gesammte Funktionalität neu getestet werden.
- Nach jedem Compilieren in neuer Umgebung (andere Architektur, anderer Compiler, andere externe Bibliotheken, etc.) muss getestet werden.



Automatisiertes Testen (mit Protokoll)
essentieller Teil der Entwicklung

Testen von Programmen (#2)

Komponententests/Modultests (unit tests – white box testing):

- Jeder Programmteil (jede Subroutine) wird getestet, ob bei verschiedenen Eingangsvariablen die erwarteten Ausgangsvariablen erzeugt werden.
- Subroutinen werden unabhängig voneinander getestet.
- Testgetriebene Entwicklung (z.B. agiles Programmieren)
  - Zuerst wird der Modultest für eine Funktionalität geschrieben
  - Danach wird erst die Funktionalität implementiert

Gesammtest (black box testing):

- Es wird das Verhalten des Gesammtprogrammes getestet
- Es sollen verschiedene Eingaben getestet werden, die unterschiedliche Funktionalitäten (verschiedene Ausführungspfade) aktivieren.
- Es sollen für jede Funktionalität mehrere Tests angelegt werden
- Kritische Fälle (z.B. fast singuläre Fälle, etc.) sollten auf jeden Fall geprüft werden
- Es sollten möglichst verschiedene Systemgrößen getestet werden.
- Im Idealfall sollte auch das Abfangen von Fehlern (z.B. singuläre Matrix) getestet werden.

Aufgabe 1

- Dokumentieren Sie die Module und die Schnittstellen (Unterprogramme und exportierte Variablen in den Modulen) im Gauss-Elimination-Programm mit Hilfe von Doxygen-Kommentaren
- Erzeugen Sie eine HTML-Dokumentation mit Doxygen, vergewissern Sie sich, dass die HTML-Dokumentation alle Schnittstellen abdeckt, und alle Doxygen-Kommentare in der Dokumentation auftauchen.
- Stellen Sie die Konfigurationsdatei für Doxygen unter Versionskontrolle und checken Sie sie ein. (*Die erzeugte Dokumentation sollte nicht unter Versionskontrolle gestellt werden!*)
- (Ab jetzt sollte jedes neu geschriebene/geänderte Unterprogramm entsprechend mit Doxygen dokumentiert werden!)

Aufgabe 2 (#1)

- Modifizieren Sie die Dateiausgabe des Gausseliminationsprogrammes so, dass es zuerst die Anzahl der Variablen und die Anzahl der b-Vektoren ausschreibt und dann die einzelnen Lösungsvektoren (1 Vektor pro Zeile).

- Schreiben Sie ein Autotestprogramm mit folgender Funktionalität:
  - Das Programm soll zwei Dateien ('gauss.out' und 'gauss.out.orig') des obigen Formats einlesen.
  - Es soll den relativen Fehler der einzelnen Elemente berechnen. Für diejenigen Elemente, für die der Wert in gauss.out.orig kleiner als 10^{-12} ist, soll der absolute Fehler berechnet werden.
  - Das Programm soll als Output nur eine Zeile ausgeben, die angibt, ob der größte berechnete Fehler (für alle Elemente) über 10^{-10} liegt:

    '*Failed! Max. error = xxxxx*' oder '*Passed*'

- Das Autotestprogramm soll im selben Verzeichnis wie das Gaussprogramm erstellt werden, und wenn sinnvoll dessen Module (z.B. accuracy) verwenden.

- Makefile soll so modifizert werden, dass nach 'make autotest' das Autotestprogramm kompiliert wird.

Aufgabe 2 (#2)

Legen Sie mind 3. Testunterverzeichnisse an ('test1', 'test2', 'test3'), in denen Sie jeweils eine Datei mit Beispielinput für das Gaussprogramm ('gauss.inp') und eine Datei mit dem erwartetem Ergebnis ('gauss.out.orig') anlegen. **Sie sollten selbtverständlich unabhängig von Ihrem Programm nachrechnen, ob das Ergebnis in gauss.out.orig das richtige ist!**

- Modifizieren Sie das Makefile so, dass 'make test' das Gaussprogramm für die Inputs in den Testverzeichnissen ausführt, und das Ergebnis (das in der Datei 'gauss.out' entsteht) mit dem vorgespeichertem Ergebnis (in 'gauss.out.orig') mit Hilfe des Autotest-Programmes vergleicht.
- Stellen Sie die Testverzeichnisse, und die Dateien 'gauss.inp' und 'gauss.out.orig' in denen unter VV.
- Optional: Überlegen Sie sich, wie Sie auch bei linear-abhängigen Gleichungssystemen automatische Tests erzeugen können (z.B. dadurch, dass bei solchen Gleichungssystemen eine spezielle 'gauss.out' Datei geschrieben wird).

Hilfestellung

- Wenn die selbe Regel für mehrere Ziele angegeben werden muss, können die Ziele aufgezählt werden.
- Make geht nach jedem ausgeführten Befehl in das ursprüngliche Verzeichnis zurück. Kommandos mit Verzeichniswechsel müssen deswegen in eine Zeile geschrieben werden. (Separation: ';')

```
# Test targets
TESTS = test1 test2 test3

# Rules for the tests
.PHONY: test $(TESTS)
test: $(TESTS)

$(TESTS): autotest
        cd $@; ../gauss_elim > /dev/null; ../autotest
```

Output nach Nullgerät umleiten.