SKRIPT FÜR MATLAB

ALBERTO CATTANEO

Inhaltsverzeichnis

1. Einführung

In diesem Kapitel werden die Grundbegriffe zum Gebrauch von MATLAB erklärt. Starten Sie MATLAB am Anfang jedes Abschnittes, und beenden Sie es am Schluss mit `exit@exit`. All die in einem Abschnitt gezeigten Befehle sollten in der gleichen Reihenfolge gegeben werden (andernfalls sind manchmal verschiedene Antworten zu erwarten als die gezeigten). Um das Lesen zu erleichtern, sind Abschnitte in Paragraphen unterteilt.

1.1. **Elementare Berechnungen und Gebrauch von Variablen.**

1.1.1. *Einmaleins.* Die gewöhnlichsten Zahlenoperationen werden in MATLAB mit den üblichen Symbolen bezeichnet:

@+@*@-@/	`+`	Addition
	`*`	Multiplikation
	`-`	Subtraktion
	`/`	Division

Zum Beispiel schreibt man `3*2` bzw. `3/2` um 3mal 2 bzw. 3 durch 2 zu berechnen. Zu bemerken ist, dass die Division von ganzen Zahlen nicht ganz sein kann; in diesem Falle wird die Antwort mit vier Ziffern nach dem Komma gegeben.

```
>> 3*2

ans =

     6
```

aber

```
>> 3/2

ans =

    1.5000
```

Die Antwort wird in der Variablen `ans`@`ans` (aus dem englischem Wort *answer* = Antwort) gespeichert. Man kann `ans` in eine neue Berechnung einsetzen. Z.B.: Will man die Antwort der oberen Berechnung, d.h. 1.5, mit 4 multiplizieren, so schreibt man

```
>> ans*4

ans =

     6
```

Und jetzt hat die Variable `ans` den neuen Wert 6.

1.1.2. *Variable.* Man kann Variable definieren. Man wähle dazu einfach einen Namen, der aus Buchstaben oder Zahlen (als einziges Wort[1]) bestehen kann, und ordnet ihm einen Wert durch Verwenden vom Symbol `=` zu. Z.B.:

```
>> a=3

a =

     3
```

[1]Es gibt zusätzliche Syntaxregeln (z.B., ein Name muss nicht mit einer Zahl beginnen). Man verwende das `Help` von MATLAB, um sie anzuschauen.

```
>> a7=4

a7 =

     4

>> b2003q=-8

b2003q =

    -8

>> a+a7+2*b2003q-1

ans =

   -10
```

All die definierten Variablen werden im (normalerweise oberen linken) Fenster „Workspace“ angezeigt. Unter „Size“ wird die Dimension der von MATLAB als Matrix aufgefassten Variablen verstanden.
Die Variablen waren in den bisherigen Beispielen Zahlen (also 1×1-Matrizen), also ist ihr „Size“ `1x1`.

1.1.3. *Zum Schluss.* Uralte Befehle aus der Zeit der nichtgraphischen Versionen stehen noch zur Verfügung. Man kann zum Beispiel alle Variablen mit dem Befehl `who`@`who` auflisten:

```
>> who

Your variables are:

a       a7      ans     b2003q
```

Um eine wie in „Workspace“ angezeigte Tabelle zu kriegen, verwendet man den Befehl `whos`@`whos`:

```
>> whos
  Name         Size                    Bytes  Class

  a            1x1                         8  double array
  a7           1x1                         8  double array
  ans          1x1                         8  double array
  b2003q       1x1                         8  double array

Grand total is 4 elements using 32 bytes
```

Ob eine Variable existiert, erfährt man durch Verwenden des Befehls `exist('variablenname')`@`exist`. Die Antwort (`ans`) ist 1, wenn die Variable mit Namen `variablenname` existiert, 0 andernfalls:

```
>> exist('a7')

ans =

     1

>> exist('zwerg')

ans =

     0
```

Zum Löschen einer Variablen, z.B. `variablenname`, schreibt man @`clear`

```
clear('variablenname')
```

Um alle Variablen zu löschen, schreibt man einfach

```
clear
```

Bevor man diesen Befehl eintippt, ist es empfehlenswert alles in einer Datei (z.B. `datei.mat`) zu speichern. Dazu schreibt man @`save`

```
save datei.mat
```

Um die in `datei.mat` gespeicherten Variablen zurückzukriegen, schreibt man @`load`

```
load datei.mat
```

Um MATLAB zu beenden, schreibt man @`exit`

```
exit
```

1.2. **Zusätzliche Berechnungen mit Zahlen und vordefinierte Variable.**

1.2.1. *Notationen.* Wir haben gesehen, dass nichtganze Zahlen in MATLAB mit vier Ziffern nach dem Komma gezeigt werden. Man kann mehr Ziffern ansehen, indem man den Befehl `format long`@`format` gibt:

```
>> a=1/3

a =

    0.3333

>> format long
>> a

a =

   0.33333333333333
```

Die wissenschaftliche Notation[2] wird durch `format short e` aktiviert.

```
>> format short e
>> a

a =

   3.3333e-01
```

Man kann rationale Zahlen als Brüche durch `format rat` darstellen.

```
>> format rat
>> a

a =

     1/3

>> a+.5

ans =

     5/6
```

Da aber alle reellen Zahlen in MATLAB durch rationelle approximiert werden, so gilt das rationale Format im Allgemeinen. Und z.B. wird $\sqrt{2}$ durch `1393/985` gegeben.

Man geht zur normalen Darstellung zurück, indem man den Befehl `format short` verwendet:

```
>> format short
>> a

a =

    0.3333
```

Andere Formate sind in MATLABs Help beschrieben.

1.2.2. *Unzahlen.* Man sollte nie durch Null dividieren; MATLAB ist aber grosszügig und erlaubt es, nur mit kleinem Brummen:

```
>> 1/0
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)
```

[2]Als wissenschaftliche Notation wird die Schreibweise von Zahlen als Produkt einer rationalen Zahl zwischen 1 und 10 (dabei 1, aber nicht 10 eingeschlossen) und einer Zehnerpotenz bezeichnet; z.B.

$$7,823 \cdot 10^5 = 782300,$$
$$1,2 \cdot 10^{-4} = 0,00012.$$

```
ans =

   Inf
```

Das Ergebnis wird in MATLAB mit `Inf@Inf` bezeichnet, welches dem mathematischen Symbol ∞ entspricht. Man kann einer Variablen `Inf` zuordnen und damit arbeiten.

```
>> Riese=1/0
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

Riese =

   Inf

>> Riese - 100000

ans =

   Inf

>> Riese*-2

ans =

  -Inf
```

Manchmal ist aber das Ergebnis nicht klar, z.B. im Falle $0 \cdot \infty$ (MATLAB erinnert sich nicht daran, wie das Unendliche erzeugt wurde, und hat deshalb nur eine Art vom Unendlichen). Die unklare Antwort wird mit `NaN@NaN` (*Not a Number* = keine Zahl) bezeichnet.

```
>> Riesli=Riese*0

Riesli =

   NaN
```

Man kann mit `NaN` weiter arbeiten, aber man erhält nichts Neues.

```
>> Riesli*100

ans =

   NaN

>> Riesli+Inf

ans =
```

```
   NaN

>> Riesli/0
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

ans =

   NaN
```

Bemerkung 1. Es ist wichtig zu wissen, dass alle Variablen in MATLAB verändert werden können. So kann man z.B. schreiben

```
>> Inf=5

Inf =

     5
```

und ab diesem Punkt ist `Inf` gleich 5 statt ∞. Also jetzt:

```
>> 0*Inf

ans =

     0

>> 1*Inf

ans =

     5
```

Zu unterscheiden ist es aber zwischen der Variablen `Inf`, die für uns jetzt gleich 5 ist, und dem Symbol `Inf`, das in Antworten vorkommen kann, welches noch immer gleich ∞ ist.

```
>> Inf

Inf =

     5

>> 1/0
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

ans =

   Inf
```

Ähnlicherweise kann man `NaN` verändern.

Am besten bringen wir aber jetzt `Inf` zu ∞ zurück:

```
>> Inf=1/0
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

Inf =

   Inf
```

1.2.3. *Weitere Operationen und komplexe Zahlen.* Potenzerhebung wird in MATLAB durch `^` ausgedrückt.

```
>> 5^2

ans =

    25
```

Die Quadratwurzel wird mit dem Befehl `sqrt`@`sqrt` gegeben:

```
>> sqrt(ans)

ans =

     5
```

Freilich kann man dasselbe durch Erhebung zu $\frac{1}{2}$ erhalten:

```
>> 25^.5

ans =

     5
```

Versucht man die Quadratwurzel einer negativen Zahl zu berechnen, so landet man in die Welt der imaginären Zahlen.

```
>> sqrt(-1)

ans =

        0 + 1.0000i
```

Die imaginäre Zahl ist in der Variablen `i`@`i` gespeichert.

```
>> i^2

ans =

    -1
```

Natürlich kann man, ähnlich zur Bemerkung 1, der Variablen `i` irgendeinen anderen Wert zuordnen.

```
>> i=8*8

i =

    64

>> i^2

ans =

        4096
```

In ausgedrückten Antworten wird aber mit `i` noch die imaginäre Einheit bezeichnet.

```
>> sqrt(-4)

ans =

        0 + 2.0000i
```

Jetzt bringen wir aber `i` zum ursprünglichen Wert $\sqrt{-1}$ zurück, um mögliche Fehler zu vermeiden.

```
>> i=sqrt(-1)

i =

        0 + 1.0000i
```

1.2.4. *Funktionen.* Weitere Funktionen sind in MATLAB definiert und haben normalerweise die üblichen Namen, die man in Taschenrechnern findet. Wir listen hier die wichtigsten auf:

@abs@sign@exp@log@log10@sqrt@sin@cos@tan@asin@acos@atan

`abs`	Absolutbetrag
`sign`	Signumfunktion
`exp`	Exponentialfunktion
`log`	natürlicher Logarithmus
`log10`	Logarithmus zur Basis 10
`sqrt`	Quadratwurzel
`sin`	Sinus
`cos`	Kosinus
`tan`	Tangens
`asin`	Arkussinus
`acos`	Arkuskosinus
`atan`	Arkustangens

Beispiele:

```
>> abs(-5)
```

```
ans =

     5

>> abs(1+i)

ans =

    1.4142

>> sign(-3)

ans =

    -1

>> sign(0)

ans =

     0

>> cos(pi)

ans =

    -1
```

Im letzten Beispiel haben wir die vordefinierte Variable `pi`@`pi`$= \pi$ verwendet:

```
>> pi

ans =

    3.1416
```

Zusätzliche Funktionen zum Runden von Zahlen sind in MATLAB definiert: Die Funktion `fix`@`fix` rundet gegen Null, während die Funktion `round`@`round` rundet zur nächsten ganzen Zahl.

Bemerkung 2 (Komplexe Zahlen). Es gibt auch Funktionen, die nur für komplexe Zahlen interessant sind:

	`real`	Realteil
@`real`@`imag`@`conj`	`imag`	Imaginärteil
	`conj`	komplexe Konjugation

Beispiele:

```
>> z=7+3i

z =
```

```
   7.0000 + 3.0000i

>> real(z)

ans =

     7

>> imag(z)

ans =

     3

>> conj(z)

ans =

   7.0000 - 3.0000i
```

1.2.5. *Zufall.* Zufallszahlen werden in MATLAB mit `rand`@`rand` erzeugt:

```
>> rand

ans =

    0.9501
```

Es ist jetzt sehr unwahrscheinlich, dass der Leser die gleiche Antwort bekommt. Sicher hat er aber eine Zahl zwischen 0 und 1 erhalten. Alle Zahlen im Intervall $[0, 1]$ sind gleichwahrscheinlich.[3]

1.2.6. *Interpunktion.* Man muss nicht verschiedene Berechnungen auf verschiedenen Zeilen eingeben. Berechnungen in der selben Zeile müssen aber mit einem Komma getrennt werden.@`,`

```
>> 5+2, pi^2

ans =

     7


ans =

    9.8696
```

[3]Man kann reelle Zufallszahlen nach der Gaussschen Verteilung mit `randn`@`randn` erzeugen.

Man kann auch einen Ausdruck, z.B. wenn er zu lang ist, auf einer weiteren Zeile durch Verwenden von vier Auslassungspunkten fortsetzen.@....

```
>> (2003-5)*Riese-Riesli+....
8, Inf*i

ans =

   NaN


ans =

      NaN +    Infi
```

Braucht eine Antwort nicht angezeigt zu werden, so beendet man den entsprechenden Befehl mit einem Strichpunkt.@;

```
>> Zwerg=1/Riese; Zwerg+5

ans =

     5
```

1.3. **Matrizen und Vektoren.**

1.3.1. *Eingeben von Matrizen und Vektoren.* Zeilenvektoren definiert man in MATLAB, indem man die Komponenten in eckigen Klammern schreibt. Um den Zeilenvektor $\mathbf{v} = (1\ 2\ 3)$ zu definieren, schreibt man in MATLAB

```
>> v=[1 2 3]

v =

     1     2     3
```

Matrizen definiert man ähnlicherweise, indem man alle Einträge in eckigen Klammern auflistet und Zeilen mit einem Strichpunkt trennt. Z.B.: Die Matrix

$$\mathsf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

wird in MATLAB folgendermassen definiert:

```
>> A=[1 2 3;4 5 6]

A =

     1     2     3
     4     5     6
```

Spaltenvektoren können als Matrizen mit einer Spalte betrachtet werden. Also kann man

$$\mathbf{w} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$$

definieren:

```
>> w=[4;5;6]

w =

     4
     5
     6
```

Es ist aber viel angenehmer, einen Spaltenvektor zunächst als Zeilenvektor zu definieren und ihn dann zu einem Spaltenvektor mit dem Strich-Befehl umzuformen:@'

```
>> w=[4 5 6]'

w =

     4
     5
     6
```

Im Allgemeinen bedeutet der Strich in MATLAB die Transposition. Ist $\mathsf{B} = (b_{ij})$ eine $m \times n$-Matrix, so ist die transponierte Matrix ${}^{t}\mathsf{B}$ als die $n \times m$-Matrix mit Komponenten $\tilde{b}_{ij}$ definiert, wobei $\tilde{b}_{ij} := b_{ji}$. Wir kriegen z.B. ${}^{t}\mathsf{A}$ wie folgt:

```
>> A'

ans =

     1     4
     2     5
     3     6
```

1.3.2. *Vordefinierte spezielle Matrizen.* Es gibt Matrizen spezieller Form, die man oft braucht. MATLAB hat spezielle Befehle, um sie zu erzeugen. All diese Befehle können mit zwei ganzzahligen Argumenten (z.B. `m,n`) eingesetz werden, welche die Dimension der Matrix (im Beispiel $m \times n$) angeben. Man kann auch nur ein einziges Argument (z.B. `m`) eingeben: In diesem Fall versteht es sich, dass man eine Quadratmatrix der gegebenen Dimension (im Beispiel $m \times m$) definieren will. Ist ein Argument negativ, so wird es als Null betrachtet. Es besteht die Möglichkeit, leere Matrizen mit keinen Zeilen oder keinen Spalten zu definieren.

Wir beginnen mit den Befehlen `zeros@zeros` bzw. `ones@ones`, die Matrizen definieren, all deren Einträge gleich Null bzw. Eins sind.

```
>> Null2mal3=zeros(2,3), Null2mal2=zeros(2)
```

```
Null2mal3 =

     0     0     0
     0     0     0


Null2mal2 =

     0     0
     0     0

>> Eins1mal3=ones(1,3)

Eins1mal3 =

     1     1     1

>> leer3mal0=zeros(3,0), leer0mal4=ones(-58,4)

leer3mal0 =

   Empty matrix: 3-by-0


leer0mal4 =

   Empty matrix: 0-by-4

>> nichts=zeros(0)

nichts =

     []
```

Mit dem Befehl `eye@eye` wird eine Matrix (a_{ij}) erzeugt, wobei a_{ij} gleich Eins ist, falls $i = j$, und gleich Null andernfalls.

```
>> I4mal3=eye(4,3), E3=eye(3)

I4mal3 =

     1     0     0
     0     1     0
     0     0     1
     0     0     0


E3 =
```

```
     1     0     0
     0     1     0
     0     0     1
```

Im zweiten Beispiel (und i.A. mit dem Befehl `eye(n)`) haben wir eine Einheitsmatrix definiert.[4]

Ähnlicherweise kann man den bereits ohne Argumente eingeführten Befehl `rand@rand` verwenden:

```
>> rand(2,5)

ans =

    0.9501    0.6068    0.8913    0.4565    0.8214
    0.2311    0.4860    0.7621    0.0185    0.4447
```

Alle Einträge sind in diesem Beispiel Zufallszahlen zwischen 0 und 1.

1.3.3. *Ausgabe und Verarbeitung von Einträgen.* Um den Eintrag an der i-ten Zeile und j-ten Spalte einer Matrix anzuzeigen, schreibt man den von `(i,j)` befolgten Namen der Matrix. Z.B.: Wir definieren zunächst die Matrix

```
>> B=[1 1 2 3; 5 8 13 21; 34 55 89 144]

B =

     1     1     2     3
     5     8    13    21
    34    55    89   144
```

Ausgewählte Einträge können wie folgt angezeigt werden:

```
>> B(2,3), B(3,1)

ans =

    13


ans =

    34

>> B(4,2)
??? Index exceeds matrix dimensions.
```

Im letzten Beispiel haben wir eine Fehlermeldung bekommen, denn die Matrix ist nur 3×4.

[4]Der Name des Befehls kommt eigentlich aus der englischen Aussprache der Buchstabe I, die üblicherweise die Einheitsmatrix (auf englisch *Identity matrix*) bezeichnet.

Um mehrere Einträge zusammen anzuzeigen (z.B. eine ganze Zeile oder eine ganze Spalte) müssen wir den Gebrauch des Doppelpunkts lernen: Zwei durch Doppelpunkt `:@:` getrennte ganze Zahlen werden von MATLAB als die ganze Folge der dazwischen-liegenden Zahlen verstanden.

```
>> 3:6

ans =

     3     4     5     6
```

Also hat der Doppelpunkt in MATLAB die gleiche Bedeutung wie die Auslassungspunkte in der üblichen mathematischen Schreibweise (im obigen Beispiel: $3, \dots, 6$).

Falls die zweite Zahl kleiner als die erste ist, ist das Ergebnis leer:

```
>> 6:3

ans =

   Empty matrix: 1-by-0
```

Man kann die Doppelpunkte auch mit nichtganzen Zahlen verwenden. In diesem Falle gibt `a:b` die Folge $a, a+1, a+2, \dots, a+n$ an, wobei $a+n \leq b$ aber $a+n+1 > b$ ist.

```
>> 3.5:8.2

ans =

    3.5000    4.5000    5.5000    6.5000    7.5000
```

Es ist auch möglich, drei Zahlen in der Form `a:s:b@:` anzugeben, wobei s die „Schrittlänge“ ist: damit wird die Folge $a, a+s, a+2s, \dots, a+ns$ erzeugt, wobei $a+ns \leq b$ aber $a+(n+1)s > b$ ist.

```
>> 3.1:2.3:12.5

ans =

    3.1000    5.4000    7.7000   10.0000   12.3000
```

Wollen wir jetzt die drei ersten Einträge der zweiten Zeile der Matrix `B` sehen (d.h. $b_{2j},\ j = 1, \dots 3$), so schreiben wir

```
>> B(2,1:3)

ans =

     5     8    13
```

Wollen wir die ganze zweite Zeile (d.h. $b_{2j}\ \forall j$), so haben wir auch eine vereinfachte Notation zur Verfügung, indem wir nur den Doppelpunkt benutzen:

```
>> B(2,:)

ans =

     5     8    13    21
```

Um Zeilen bzw. Spalten zu permutieren, schreiben wir die entsprechende Permutation als Zeilen- bzw. Spaltenargument. Eine Permutation ϕ der Zahlen $1, \ldots, n$ wird in MATLAB durch den „Bildvektor“ $(\phi(1), \phi(2), \ldots, \phi(n))$ angegeben. Z.B.: Die Permutation von $\{1, 2, 3, 4\}$, die 2 mit 3 vertauscht, bezeichnet man in MATLAB mit `[1 3 2 4]`. Also erhalten wir die Vertauschung der zweiten mit der dritten Spalte der Matrix `B` wie folgt:

```
>> C=B(:,[1 3 2 4])

C =

     1     2     1     3
     5    13     8    21
    34    89    55   144
```

Man kann eine ganze Zeile oder eine ganze Spalte löschen, indem man sie gleich dem leeren Vektor `[]@[]` setzt.

```
>> C(2,:)=[]

C =

     1     2     1     3
    34    89    55   144
```

Schliesslich kann man Matrizen mit gleicher Anzahl Zeilen bzw. Spalten verketten. Z.B.: Uns stehen jetzt die Matrizen `A` und `C` zur Verfügung, die zwei Zeilen haben. Also können wir eine neue Matrix mit zwei Zeilen so definieren:

```
>> D=[A C]

D =

     1     2     3     1     2     1     3
     4     5     6    34    89    55   144
```

Haben die Matrizen die gleiche Anzahl Spalten, so benutzt man den Strichpunkt, um sie zu verketten:

```
>> E=[B;C]

E =

     1     1     2     3
     5     8    13    21
    34    55    89   144
```

```
     1     2     1     3
    34    89    55   144
```

Hier ist die Gefahr gross, Fehler zu machen. Vor allem wenn man nicht richtig zählt:

```
>> U=[D E]
??? Error using ==> horzcat
All matrices on a row in the bracketed expression must have the
 same number of rows.
```

1.3.4. *Berechnungen mit Matrizen.* Matrizen addiert man mit dem üblichen Symbol `+`:

```
>> S=A+ones(2,3)

S =

     2     3     4
     5     6     7
```

In MATLAB kann man zu einer Matrix `A` eine Zahl a addieren: das bedeutet eigentlich, dass die zu addierende Zahl a als eine Matrix dergleichen Grösse wie `A` und mit allen Einträgen gleich a aufgefasst wird:

```
>> A+2

ans =

     3     4     5
     6     7     8
```

Die Multiplikation wird mit `*` bezeichnet.

```
>> P=A*B

P =

   113   182   295   477
   233   374   607   981
```

Ein zweiter üblicher Fehler:

```
>> U=A*C
??? Error using ==> *
Inner matrix dimensions must agree.

>> V=A+C
??? Error using ==> +
Matrix dimensions must agree.
```

Die Multiplikation eines Zeilenvektors mit einem Spaltenvektor der gleichen Länge ist ein Speziallfall der Matrixmultiplikation und wird mit `*` bezeichnet.

```
>> v,w

v =

     1     2     3


w =

     4
     5
     6

>> v*w

ans =

    32
```

Zu bemerken ist, dass die mit * bezeichneter Multiplikation eines Spaltenvektors mit einem Zeilenvektor (möglicherweise von verschiedenen Längen) in MATLAB erlaubt ist.

```
>> Q=w*v

Q =

     4     8    12
     5    10    15
     6    12    18
```

Natürlich ist diese Multiplikation kein Speziallfall der Matrixmultiplikation (die Dimensionen der als Matrizen betrachteten Vektoren passen nicht zueinander). In diesem Falle bezeichnet stattdessen * eine andere Multiplikation, die sog. Tensormultiplikation: aus dem m-Zeilenvektor $\mathbf{w}$ und aus dem n-Spaltenvektor $\mathbf{v}$ wird eine $m \times n$-Matrix $\mathsf{A} = (a_{ij})$ erzeugt, deren Einträge folgendermassen definiert sind:

$$a_{ij} := w_i v_j.$$

Eine Quadratmatrix kann man mit sich selbst multiplizieren.

```
>> Q2=Q*Q

Q2 =

   128   256   384
   160   320   480
   192   384   576
```

Das führt natürlich zur Definition der Potenzerhebung, die durch ^ bezeichnet wird.

```
>> Q^2
```

```
ans =

   128   256   384
   160   320   480
   192   384   576

>> Q2^.5

ans =

    4.0000    8.0000   12.0000
    5.0000   10.0000   15.0000
    6.0000   12.0000   18.0000
```

Man kann auch Funktionen auf Matrizen anwenden. Die Regel in MATLAB ist wie folgt: Ist f eine Funktion und $\mathsf{A} = (a_{ij})$ eine Matrix, so ist $f(\mathsf{A})$ die Matrix, die aus den Einträgen $f(a_{ij})$ besteht.

```
>> A, log(A)

A =

     1     2     3
     4     5     6


ans =

         0    0.6931    1.0986
    1.3863    1.6094    1.7918
```

Das entspricht nicht der üblichen mathematischen Notation[5] und kann zu Missverständnissen führen; z.B., die in MATLAB so definierte Quadratwurzel von `Q2` ist nicht die Matrix `Q`, die man erwarten könnte und die mit `Q2^.5` erhalten wird.

[5]Die in MATLAB verwendete Vereinbarung eignet sich aber für graphische Anwendungen. Will man z.B. einen approximierten Graphen der Funktion f zeichnen, so wählt man normalerweise einige Punkte $(x_1, \dots, x_n)$ auf der x-Achse. Man berechnet dann die Werte $f(x_1), \dots, f(x_n)$ und verbindet die Punkte $(x_1, f(x_1)), \dots, (x_n, f(x_n))$ mit geraden Strecken.

In MATLAB definiert man einfach den Vektor `x` mit Komponenten $(x_1, \dots, x_n)$, und man erhält den Vektor mit Komponenten $f(x_1), \dots, f(x_n)$ durch den Befehl `y=f(x)`. Die durch die Punkte $(x_1, y_1), \dots, (x_n, y_n)$ laufende gebrochene Gerade zeichnet man mit dem Befehl `plot(x,y)`.@`plot` Z.B.: man kann die Sinusfunktion approximiert zeichnen wie folgt:

```
>> x=0:pi/100:2*pi;
>> y=sin(x);
>> plot(x,y)
```

Oder in kompakterer Form:

```
>> plot(0:pi/100:2*pi,sin(0:pi/100:2*pi))
```

schreiben.

MATLABs im Falle von Matrizen verwendete Vereinbarung ist auch dann nützlich, wenn man Oberflächen anstatt Kurven zeichnen will, wo ein Gitter von Punkten benötigt ist.

```
>> sqrt(Q2)

ans =

   11.3137   16.0000   19.5959
   12.6491   17.8885   21.9089
   13.8564   19.5959   24.0000
```

Doch gibt es manchmal Varianten der Funktionennamen, die die mit den üblichen mathematischen Vereinbarungen erwarteten Ergebnisse geben. Oft ist der Name bei der Hinzufügung von `m` (für Matrix) erhalten:@`sqrtm`

```
>> sqrtm(Q2)
Warning: Matrix is singular and may not have a square root.
(Type "warning off MATLAB:sqrtm:SingularMatrix" to suppress this warning.)
> In /usr/local/matlab6p5/toolbox/matlab/matfun/sqrtm.m at line 65

ans =

    4.0000    8.0000   12.0000
    5.0000   10.0000   15.0000
    6.0000   12.0000   18.0000
```

Wie bereits gesehen, stimmt im Gegensatz die durch `*` erklärte Multiplikation mit der üblichen Matrixmultiplikation überein. Man kann aber auch die Einträge von Matrizen multiplizieren, indem man `.*` statt `*` verwendet. D.h., sind $\mathsf{A} = (a_{ij})$ und $\mathsf{B} = (b_{ij})$ Matrizen mit gleichen Anzahlen Zeilen und Spalten, so definiert man die kommutative Multiplikation $\mathsf{C} = \mathsf{A}.*\mathsf{B} = (c_{ij})$ durch $c_{ij} := a_{ij}b_{ij}\ \forall i, j$. Z.B.:

```
>> A

A =

     1     2     3
     4     5     6

>> B=[3 1 0; -1 -1 1]

B =

     3     1     0
    -1    -1     1

>> A.*B

ans =

     3     2     0
    -4    -5     6
```

Bemerkung 3 (Komplexe Matrizen). Matrizen können auch komplexe Einträge haben. Man verwendet dann die entsprechenden Rechenregeln. Man kann natürlich die in Bemerkung 2 auf Seite 10 auf komplexen Zahlen definierten Funktionen zu komplexen Matrizen erweitern.@`real`@`imag`@`conj`

```
>> Z=[1 i 3+i;-2 8+9i 0]

Z =

   1.0000                  0 + 1.0000i   3.0000 + 1.0000i
  -2.0000             8.0000 + 9.0000i        0

>> real(Z), imag(Z), conj(Z)

ans =

     1     0     3
    -2     8     0


ans =

     0     1     1
     0     9     0


ans =

   1.0000                  0 - 1.0000i   3.0000 - 1.0000i
  -2.0000             8.0000 - 9.0000i        0
```

1.3.5. *Spezielle Funktionen.* Es gibt einige sehr interessante auf Vektoren oder Matrizen definierte Funktionen (in der dritten Spalte geben wir die Auswertung der Funktion auf dem Vektor $(x_1, \dots, x_n)$:

@`max`@`min`@`sum`@`prod`@`mean`@`sort`@`size`

`max`	Maximum	$\min\{x_1, \dots, x_n\}$
`min`	Minimum	$\max\{x_1, \dots, x_n\}$
`sum`	Summe	$\sum_{i=1}^n x_i$
`prod`	Produkt	$\prod_{i=1}^n x_i$
`mean`	Mittelung	$\frac{1}{n}\sum_{i=1}^n x_i$
`sort`	Sortierung	$(x_{i_1}, x_{i_2}, \dots, x_{i_n})$ s.d. $x_{i_1} \le x_{i_2} \le \dots \le x_{i_n}$
`size`	Grösse	$(1, n)$

Beispiele:

```
>> w =[7, -3, 5, 14, 2]

w =

     7    -3     5    14     2
```

```
>> min(w), max(w)

ans =

    -3


ans =

    14

>> sum(w), mean(w)

ans =

    25


ans =

     5

>> sort(w)

ans =

    -3     2     5     7    14

>> size(w)

ans =

     1     5
```

Bemerkung 4. Mit den Befehlen `max`@`max` und `min`@`min` ist es auch möglich die Stelle des Maximums bzw. des Minimums zu bestimmen. Dazu definiert man die Antwort als Vektor, dessen erste Komponente das Maximum bzw. das Minimum ist und dessen zweite Komponente die entsprechende Stelle ist (im Falle von mehreren Maximum- bzw. Minimumstellen wird das kleinste Stellenindex betrachtet). Z.B.:

```
>> [M MI]=max(w), [m mi]=min(w)

M =

    14


MI =

     4
```

```
m =

    -3


mi =

     2
```

Auf Matrizen werden die obigen Funktionen, ausser `size`, spaltenweise berechnet:

```
>> A, max(A), sum(A)

A =

     1     2     3
     4     5     6


ans =

     4     5     6


ans =

     5     7     9
```

Die Funktion `size`@`size` ergibt die Grösse der Matrix als einen Spaltenvektor mit Komponenten die Anzahl Zeilen und die Anzahl Spalten: ist A eine $m \times n$-Matrix, so ist ihre Grösse der Vektor (m, n).

```
>> size(A)

ans =

     2     3
```

1.3.6. *Zum Schluss.* Man kann die Einträge auf der Diagonale (d.h. die der Form a_{ii}) als Spaltenvektor mit dem Befehl `diag`@`diag` auffassen:

```
>> A, d=diag(A)

A =

     1     2     3
     4     5     6
```

```
d =

     1
     5
```

Man kann auch die Diagonale verschieben, indem man ein zusätzliches ganzes Argument einsetzt:

```
>> d1=diag(A,1)

d1 =

     2
     6
```

Ist das Argument von `diag`@`diag` ein Vektor anstatt einer Matrix, so wird eine Diagonalmatrix erzeugt, deren Einträge auf der Diagonale die Komponenten des gegebenen Vektors sind.

```
>> diag(d)

ans =

     1     0
     0     5
```

Die iterierte Anwendung von `diag` auf eine Quadratmatrix löscht alle Einträge, die nicht auf der Diagonale liegen.

```
>> Q, diag(diag(Q))

Q =

     4     8    12
     5    10    15
     6    12    18


ans =

     4     0     0
     0    10     0
     0     0    18
```

Die Befehle `triu`@`triu` bzw. `tril`@`tril` (*upper triangular* bzw. *lower triangular*) erzeugen obere bzw. untere Dreiecksmatrizen, indem sie die anderen Einträge löschen.

```
>> E, triu(E), tril(E)

E =
```

```
     1     1     2     3
     5     8    13    21
    34    55    89   144
     1     2     1     3
    34    89    55   144


ans =

     1     1     2     3
     0     8    13    21
     0     0    89   144
     0     0     0     3
     0     0     0     0


ans =

     1     0     0     0
     5     8     0     0
    34    55    89     0
     1     2     1     3
    34    89    55   144
```

Man kann dann auch die oben erklärten Befehle zusammenknüpfen:

```
>> tril(E,-2)

ans =

     0     0     0     0
     0     0     0     0
    34     0     0     0
     1     2     0     0
    34    89    55     0
```

2. Lineare Gleichungssysteme

In diesem Kapitel werden lineare Gleichungssysteme behandelt. Im Falle von zwei Unbestimmten beschreiben wir kurz, wie man sie graphisch „auflösen“ kann. Dann werden wir die bereits gelernten Befehle verwenden, um explizit lineare Gleichungssysteme aufzulösen. Schliesslich diskutieren wir vordefinierte Funktionen, die erlauben, Lösungsmengen zu bestimmen.

2.1. **Graphische Lösungen.** Eine nichtentartete lineare Gleichung in zwei Unbestimmten beschreibt eine Gerade in der Ebene. Um eine Gerade zu zeichnen, braucht man zwei ihrer Punkte. Diese finden wir, indem wir der einen Unbestimmten einen Wert zuordnen und die Gleichung bez. der anderen auflösen. Sei

$$ax + by = c$$

unsere Gleichung, so können wir x für jeden Wert von y als $(c - by)/a$ bestimmen, wenn $a \neq 0$ ist. Sonst haben wir eine waagerechte Gerade, die durch die Punkte $(x_1, c/b)$ und $(x_2, c/b)$ geht, wobei x_1 und x_2 beliebig sind.

Nachdem wir ein Paar Lösungen (x_1, y_1) und (x_2, y_2) bestimmt haben, können wir die entsprechende Gerade mit dem Befehl `plot@plot` zeichnen. Für gegebene n-Vektoren `x` und `y` zeichnet `plot(x,y)` die durch die Punkte $(x_1, y_1), \ldots, (x_n, y_n)$ laufende gebrochene Gerade (s. auch Fussnote 5). Wir sind jetzt am Falle $n = 2$ interessiert, wo `plot(x,y)` die gesuchte Gerade zeichnet.

Z.B.: Wir wollen die Gleichung $2x + 3y = 5$ studieren. Die Punkte $(-2, 3)$ und $(4, -1)$ sind Lösungen. Also haben wir

$$x_1 = -2, \qquad x_2 = 4,$$
$$y_1 = 3, \qquad y_2 = -1,$$

und die Gerade wird wie folgt gezeichnet:

```
>> plot([-2,4],[3,-1])
```

Haben wir jetzt zwei nichtentartete Gleichungen, so können wir approximierte Lösung(en) des Gleichungssystem erhalten, indem wir die zwei entsprechenden Geraden gleichzeitig zeichnen. Im Allgemeinen können wir die r durch Vektorpaare $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_r, \mathbf{y}_r)$ definierten Graphen mit `plot(x1,y1,x2,y2,...,xr,yr)` gleichzeitig zeichnen.

Z.B.: Sei $3x - y = 2$ die zweite Gleichung unsres Systems. Da $(-2, -8)$ und $(4, 10)$ Lösungen sind, können wir die entsprechende Gerade mit `plot([-2,4],[-8,10])` zeichnen. Wir wollen aber die zwei Geraden gleichzeitig zeichnen, also schreiben wir

```
>> plot([-2,4],[3,-1],[-2,4],[-8,10])
```

Im graphischen Fenster sollten wir jetzt zwei in unterschiedlichen Farben sich schneidende Geraden sehen.

Stehen Farben nicht zur Verfügung, so können wir die Geraden unterscheiden, indem wir sie auf unterschiedliche Weise zeichnen. Die "Zeichnungsart" wird mit einem zusätzlichen Argument angegeben, das in Strichen geschrieben wird. Eine gebrochen Linie wird zum Beispiel mit `'--'` angegeben. So verbessern wir unser Bild mit dem Befehl:

```
>> plot([-2,4],[3,-1],[-2,4],[-8,10],'--')
```

Das erhaltene Bild, s. Abbildung 1, zeigt den Schnittpunkt $(1, 1)$, welcher Lösung des Gleichungssystems ist.

Graphische Methoden können natürlich auch zur Auflösung nichtlinearer Gleichungssysteme verwendet werden. Man kann z.B. Lösungen des Gleichungssystems $\begin{cases} y = \sin x \\ y = \exp(-x) \end{cases}$ folgendermassen sehen:

```
>> x=0:pi/100:2*pi; plot(x,sin(x),x,exp(-x),'--')
```

2.2. **Explizites Auflösen linearer Gleichungssysteme.** Mit dem Eliminationsverfahren von Gauss kann man jedes lineare Gleichungssystem auflösen. Wir

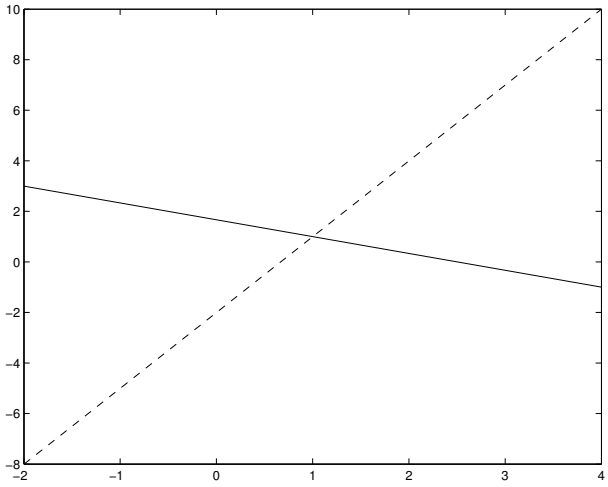

Abbildung 1. Graphische Lösung

diskutieren zunächst seine Anwendung in einem einfachen Beispiel, und dann definieren wir einen allgemeinen MATLAB-Algorithmus, um Lösungen zu erhalten, im Falle von oberen Dreiecksmatrizen.

2.2.1. *Ein Beispiel.* Wir wollen das System

$$\mathsf{A}\mathbf{x} = \mathbf{b}$$

auflösen, wobei

$$\mathsf{A} = \begin{pmatrix} 5 & 3 & 7 \\ -2 & 4 & 9 \\ 6 & 2 & 0 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 26 \\ 8 \\ 44 \end{pmatrix}. \tag{2.1}$$

Zunächst definieren wir es in MATLAB:

```
>> A=[5 3 7; -2 4 9; 6 2 0]

A =

     5     3     7
    -2     4     9
     6     2     0

>> b=[26 8 44]'

b =

    26
     8
    44
```

Wir betrachten dann die erweiterte Koeffizientenmatrix $\mathsf{AUG} = (\mathsf{A} \,|\, \mathbf{b})$, die wir in MATLAB, z.B., `AUG` nennen:

```
>> AUG=[A b]

AUG =

     5     3     7    26
    -2     4     9     8
     6     2     0    44
```

Wir wollen jetzt diese Matrix durch Zeilenumformungen in Zeilenstufenform überführen. Wir können z.B. zur zweiten bzw. dritten Zeile die mit $\frac{2}{5}$ bzw. $-\frac{6}{5}$ multiplizierte erste Zeilen addieren. In MATLAB:

```
>> AUG(2,:)=AUG(2,:)+(2/5)*AUG(1,:)

AUG =

    5.0000    3.0000    7.0000   26.0000
         0    5.2000   11.8000   18.4000
    6.0000    2.0000         0   44.0000

>> AUG(3,:)=AUG(3,:)-(6/5)*AUG(1,:)

AUG =

    5.0000    3.0000    7.0000   26.0000
         0    5.2000   11.8000   18.4000
         0   -1.6000   -8.4000   12.8000
```

Oder besser in Brüchen:

```
>> format rat, AUG

AUG =

      5            3            7           26
      0          26/5         59/5         92/5
      0          -8/5        -42/5         64/5
```

Jetzt addieren wir zur dritten Zeile die mit $\frac{4}{13}$ multiplizierte zweite Zeile:

```
>> AUG(3,:)=AUG(3,:)+(4/13)*AUG(2,:)

AUG =

      5            3            7           26
      0          26/5         59/5         92/5
      0            *        -62/13       240/13
```

Das Sternchen * bezeichnet eine sehr kleine Zahl, die MATLAB mit Brüchen nicht schreiben kann. In unserem Beispiel sollte das Sternchen genau Null sein, und es ist nur wegen Rundungsfehler davon verschieden. Wir haben jetzt das zu (2.1) äquivalente System:

$$\begin{aligned} 5x_1 \quad +3x_2 \quad\quad +7x_3 \quad &= \quad 26 \\ +\tfrac{26}{5}x_2 \quad +\tfrac{59}{5}x_3 \quad &= \quad \tfrac{92}{5} \\ -\tfrac{62}{13}x_3 \quad &= \quad \tfrac{240}{13} \end{aligned}$$

Schliesslich können wir das System auflösen. Die dritte Gleichung bestimmt unmittelbar die dritte Komponente x_3 des Lösungsvektors $\mathbf{x}$:

```
>> x3=AUG(3,4)/AUG(3,3)

x3 =

  -120/31
```

Die Kenntnis von x_3 erlaubt jetzt x_2 aus der zweiten Gleichung zu bestimmen:

```
>> x2=(AUG(2,4)-AUG(2,3)*x3)/AUG(2,2)

x2 =

   382/31
```

Schliesslich bestimmen wir x_1:

```
>> x1=(AUG(1,4)-AUG(1,3)*x3-AUG(1,2)*x2)/AUG(1,1)

x1 =

   100/31
```

So haben wir die Lösung

```
>> x=[x1 x2 x3]'

x =

   100/31
   382/31
  -120/31
```

wie man leicht verifizieren kann:

```
>> A*x

ans =

     26
      8
```

```
44
```

2.2.2. *Obere Dreiecksmatrizen.* Wir wollen hier Lösungen explizit finden, unter der Annahme, dass die Koeffizientenmatrix bereits in Zeilenstufenform ist. Der Einfachheit halber betrachten wir quadratische Matrizen maximales Ranges (d.h. invertierbar), sodass die entsprechende lineare Gleichungssysteme genau eine Lösung haben.

Abstrakte Formulation. Wir wiederholen zunächst das abstrakte Verfahren. Sei

$$\mathsf{A} = \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots \\ 0 & a_{22} & \dots & \dots & \dots \\ 0 & 0 & a_{33} & \dots & \dots \\ 0 & \dots & 0 & \ddots & \dots \\ 0 & \dots & \dots & 0 & a_{nn} \end{pmatrix}$$

die Koeffizientenmatrix. Genauer gesagt: $\mathsf{A} \in M(n \times n; \mathbb{K})$ ($\mathbb{K}$ ein Körper), s.d.

$$a_{ij} = 0 \ \forall i < j, \qquad a_{ii} \neq 0 \ \forall i.$$

Wir wollen die Lösung von $\mathsf{A}\mathbf{x} = \mathbf{b}$, für beliebiges $\mathbf{b}$, finden. Die letzte Gleichung ist

$$a_{nn} x_n = b_n.$$

Da $a_{nn} \neq 0$ ist, können wir sie auflösen:

$$x_n = \frac{b_n}{a_{nn}}.$$

Die $(n-1)$-te Gleichung,

$$a_{n-1,n-1} x_{n-1} + a_{n-1,n} x_n = b_{n-1},$$

kann auch wie folgt geschrieben werden:

$$a_{n-1,n-1} x_{n-1} = b_{n-1} - a_{n-1,n} x_n.$$

Jetzt sind aber alle Grösse auf der rechten Seite bereits bekannt und wir können x_{n-1} finden, indem wir durch $a_{n-1,n-1}$ dividieren.

I.A. kann die j-te Gleichung wie folgt geschrieben werden:

$$a_{jj} x_j = b_j - \sum_{k=j+1}^{n} a_{jk} x_k. \tag{2.2}$$

Haben wir bereits alle Gleichungen für $k > j$ aufgelöst, so ist die rechte Seite bestimmt, und wir können x_j finden, denn es gilt $a_{jj} \neq 0$.

MATLABs Kodierung. Zunächst sollen wir den Vektor `x` definieren, der zum Schluss die Lösung darstellen wird. Aus (2.2) ist es klar, dass man die Summe für $k = 1, \dots, n$ (statt $k = j+1, \dots, n$) schreiben könnte, wenn die Komponenten x_k, $k \leq j$, gleich Null wären. So ist es vernünftig mit dem Nullvektor zu beginnen. Das wird mit dem Befehl `x=zeros(n,1)` erzielt, wobei `n` eine bereits definierte Variable ist (um mit undefinierten Variablen zu arbeiten, muss man MATLABs Programme schreiben). Die Lösung zu (2.2) kann jetzt in MATLAB wie folgt geschrieben werden:

```
x(j) = (b(j) -A(j,:)*x)/A(j,j)
```

Das müssen wir für alle `j` in der Reihenfolge `j=n`, `j=n-1`, `j=n-2`, ..., `j=2`, `j=1` wiederholen. Das wird mit der Struktur `for`...`end`@`for`@`end` geschaffen. Die Syntax ist wie folgt:

```
for j=a:s:b, Befehle; end
```

Die Befehle werden für `j=a`, `j=a+s`, `j=a+2s` usw. ausgeführt. Das Verfahren stoppt, wenn `j` grösser als `b` wird. (Ist die Schrittlänge `s` gleich Eins, so kann man auch `for j=a:b` statt `for j=a:1:b` schreiben.) Der Strichpunkt vor `end` kann auch mit einem Komma ersetzt werden: in diesem Falle wird aber der Output der Befehle für jedes `j` angezeigt, was normalerweise nicht erwünscht ist.

In unsrem Beispiel wird dann das Verfahren kodiert wie folgt:

```
x=zeros(n,1);....
for j=n:-1:1, x(j) = (b(j) -A(j,:)*x)/A(j,j); end, x
```

wobei wir die Definition von `x` am Anfang und den Befehl, `x` zu zeigen, am Ende hinzugefügt haben.

Ein konkretes Beispiel. Sei jetzt $n = 3$,

$$\mathsf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \\ 0 & 0 & 8 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

In MATLAB:

```
>> A=[1 2 3; 0 3 4; 0 0 8]; b=[1 2 3]'; n=3;
```

Die Anwendung des oben erklärten Verfahrens ergibt die Lösung:

```
>> x=zeros(n,1);....
for j=n:-1:1, x(j) = (b(j) -A(j,:)*x)/A(j,j); end, x

x =

   -0.4583
    0.1667
    0.3750
```

Das kann man einfach verifizieren:

```
>> A*x

ans =

     1
     2
     3
```

2.3. **Vordefinierte Befehle zur Lösung linearer Gleichungssysteme.** In diesem Abschnitt erklären wir, wie man mit MATLAB ein lineares Gleichungssystem auflösen kann. Wir beschreiben drei Methoden:

(1) Man kann die erweiterte Koeffizientenmatrix in Zeilenstufenform überführen (s. Unterabschnitt 2.3.1).
(2) Falls die Koeffizientenmatrix invertierbar ist, kann man ihre Inverse berechnen (s. Unterabschnitt 2.3.2).
(3) Man kann geeignete MATLAB-Befehle verwenden, um spezielle oder allgemeine Lösungen zu bestimmen (s. Unterabschnitt 2.3.3).

Im Prinzip kann man sich zwar mit dem Inhalt von Unterabschnitt 2.3.3 begnügen. Die Diskussion in den anderen Unterabschnitten ist aber didaktisch nützlich: wir können eine Verbindung mit dem Stoff aus der Vorlesung aufbauen, um die hinter einigen Befehlen liegenden Verfahren zu beschreiben, und schliesslich um einige Gefahren zu vermeiden, die sich hinter einem dummen Gebrauch der MATLAB-Befehle verstecken.

2.3.1. *Zeilenstufenform.* Aus der Vorlesung ist bekannt, dass die Überführung in Zeilenstufenform der erweiterten Koeffizientenmatrix die Hauptmethode ist, ein System aufzulösen. Das kann man immer erreichen, indem man endlich viele Zeilenumformungen verwendet. Diese lauten wie folgt (vgl. [F, 0.4.6]):

(1) Vertauschung von zwei Zeilen.
(2) Addition der λ-fachen i-ten Zeile zur k-ten Zeile, wobei $\lambda \in \mathbb{K}^*$ und $i \neq k$.

Man kann etwas schöneres kriegen, wenn man *elementare Zeilenumformungen* einführt; nämlich (vgl. [F, 1.5.7]):

I: Multiplikation der i-ten Zeile mit $\lambda \in \mathbb{K}^*$.
II: Addition der j-ten Zeile zur i-ten Zeile.

Man kann den folgenden Satz zeigen:

Satz 1. *Es gelten folgende Aussagen:*

(1) *Die elementaren Zeilenumformungen I und II erzeugen die Zeilenumformungen 1) und 2).*
(2) *Ist* $(\tilde{\mathsf{A}}, \tilde{\mathbf{b}})$ *aus* $(\mathsf{A}, \mathbf{b})$ *durch endlich viele elementare Zeilenumformungen erhalten, so gilt* $\text{Lös}(\mathsf{A}, \mathbf{b}) = \text{Lös}(\tilde{\mathsf{A}}, \tilde{\mathbf{b}})$.
(3) *Jede Matrix kann durch endlich viele elementare Zeilenumformungen in reduzierte Zeilenstufenform überführt werden.*

Zur Erinnerung:

Definition 1. Eine Matrix heisst in reduzierter Zeilenstufenform (auf englisch reduced row echelon form), wenn sie in Zeilenstufenform ist und folgende Bedingungen erfüllt sind:

(1) Alle Pivots sind gleich Eins.
(2) Alle Einträge oberhalb eines Pivots sind gleich Null.

In MATLAB überführt man eine (reelle oder komplexe) Matrix in reduzierte Zeilenstufenform mit dem Befehl `rref`.@`rref` Z.B.:

```
>> A=[1 2 3;4 5 6;7 8 9]

A =
```

```
     1     2     3
     4     5     6
     7     8     9

>> rref(A)

ans =

     1     0    -1
     0     1     2
     0     0     0
```

Aus der Vorlesung ist bekannt, dass man den Rang einer Matrix berechnen kann, indem man die von Null verschiedenen Zeilen der entsprechenden Matrix in Zeilenstufenform zählt. In unsrem Beispiel ist dann $\operatorname{Rang} \mathsf{A} = 2$. Man kann aber den Rang direkt mit dem Befehl `rank@rank` bekommen:

```
>> rank(A)

ans =

     2
```

Der Rang ist sehr wichtig, um die Lösbarkeit eines linearen Gleichungssystems festzulegen. Nämlich (vgl. [F, 2.3.2]):

Satz 2. $\text{Lös}(\mathsf{A}, \mathbf{b}) \neq \emptyset \Leftrightarrow \text{Rang}(\mathsf{A}) = \text{Rang}(\mathsf{A}, \mathbf{b})$

Also

```
>> b=[1 2 3]'; c=[1 2 4]'; rank([A b]), rank([A c])

ans =

     2


ans =

     3
```

zeigt, dass das System $\mathsf{A}\mathbf{x} = \mathbf{b}$ lösbar ist, während das System $\mathsf{A}\mathbf{x} = \mathbf{c}$ keine Lösung hat. Das sieht man auch expliziter, wenn man die entsprechenden Zeilenstufenformen betrachtet:

```
>> Eb=rref([A b]), Ec=rref([A c])

Eb =

    1.0000         0   -1.0000   -0.3333
         0    1.0000    2.0000    0.6667
         0         0         0         0
```

```
Ec =

     1     0    -1     0
     0     1     2     0
     0     0     0     1
```

Ist das System lösbar, so kriegt man sehr einfach die allgemeine Lösung, indem man die reduzierte Zeilenstufenform betrachtet. Es gilt nämlich:

Satz 3. *Seien* $\mathsf{A} \in M(m \times n; \mathbb{K})$, $\mathbf{b} \in \mathbb{K}^m$ *und* $\mathrm{Rang}(\mathsf{A}) = \mathrm{Rang}(\mathsf{A}, \mathbf{b}) =: r$. *Sei* $(\tilde{\mathsf{A}}, \tilde{\mathbf{b}})$ *die in reduzierter Zeilenstufenform aus* $(\mathsf{A}, \mathbf{b})$ *erhaltene Matrix. So gelten folgende Aussagen:*

(1) *Die letzte Spalte von* $(\tilde{\mathsf{A}}, \tilde{\mathbf{b}})$ *ist eine spezielle Lösung von* $\mathsf{A}x = \mathbf{b}$.
(2) *Seien* $\mathbf{w}'_1, \ldots, \mathbf{w}'_k \in \mathbb{K}^m$, $k = n - r$, *die Spalten von* $\tilde{\mathsf{A}}$, *die keine Pivots enthalten. Sei* $\mathbf{w}_i$, $i = 1, \ldots, k$, *der* r*-Vektor, der aus den ersten* r *Komponenten von* $\mathbf{w}'_i$ *besteht. Sei* $\mathbf{v}_i := \binom{-\mathbf{w}_i}{\mathbf{e}_i} \in \mathbb{K}^n$, $i = 1, \ldots, k$, *wobei* $(\mathbf{e}_i)_{i=1,\ldots,k}$ *die kanonische Basis von* $\mathbb{K}^k$ *ist. So ist* $(\mathbf{v}_i)_{i=1,\ldots,k}$ *eine Basis von* $\mathrm{Lös}(\mathsf{A}, 0)$.

Beweis. Der Einfachheit halber nehmen wir an, dass die r Pivots von $\tilde{\mathsf{A}}$ in den ersten r Spalten sind. Das kann man durch eine Umordnung der Spalten immer erhalten (und das entspricht einfach eine Umnumerierung der Unbestimmten). Also ist

$$\tilde{\mathsf{A}} = \begin{pmatrix} \mathsf{E} & \mathsf{B} \\ 0 & 0 \end{pmatrix},$$

wobei E die r-dimensionale Einheitsmatrix, 0 Nullmatrizen und $\mathsf{B} \in M(r \times k; \mathbb{K})$ sind. Da das System nach Voraussetzung lösbar ist, ist $\tilde{\mathbf{b}} = \binom{\mathbf{b}'}{0}$, mit $\mathbf{b}' \in \mathbb{K}^r$ und 0 der $(m-r)$-dimensionale Nullvektor. Also

$$(\tilde{\mathsf{A}}, \tilde{\mathbf{b}}) = \begin{pmatrix} \mathsf{E} & \mathsf{B} & \mathbf{b}' \\ 0 & 0 & 0 \end{pmatrix},$$

Schreiben wir

$$\mathbf{x} = \begin{pmatrix} \mathbf{y} \\ \boldsymbol{\lambda} \end{pmatrix}, \ \mathbf{y} \in \mathbb{K}^r, \ \boldsymbol{\lambda} \in \mathbb{K}^k,$$

so ist das Gleichungssystem $\tilde{\mathsf{A}}\mathbf{x} = \tilde{\mathbf{b}}$ äquivalent zu

$$\mathbf{y} + \mathsf{B}\boldsymbol{\lambda} = \mathbf{b}'.$$

Hier können wir die Komponenten $\lambda_1, \ldots, \lambda_k$ von $\boldsymbol{\lambda}$ als Parameter betrachten. Also kriegen wir für $\boldsymbol{\lambda} = 0$ die spezielle Lösung $\mathbf{y} = \mathbf{b}'$. Das ergibt die spezielle Lösung

$$\mathbf{x} = \begin{pmatrix} \mathbf{b}' \\ 0 \end{pmatrix},$$

die mit der letzten Spalte von $(\tilde{\mathsf{A}}, \tilde{\mathbf{b}})$ übereinstimmt.

Setzen wir jetzt $\mathbf{b}' = 0$, so kriegen wir die allgemeine Lösung des homogenen Systems: $\mathbf{y} = -\mathsf{B}\boldsymbol{\lambda}$; d.h.:

$$\mathbf{x} = \begin{pmatrix} -\mathsf{B}\boldsymbol{\lambda} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -\mathsf{B} \\ \mathsf{E} \end{pmatrix} \boldsymbol{\lambda}.$$

Entwickeln wir $\boldsymbol{\lambda}$ bzgl. der kanonischen Basis, $\boldsymbol{\lambda} = \sum_{i=1}^{k} \lambda_i \mathbf{e}_i$, so haben wir $\mathbf{x} = \sum_{i=1}^{k} \lambda_i \mathbf{v}_i$, wobei $\mathbf{v}_i$ die i-te Spalte von $\begin{pmatrix} -\mathsf{B} \\ \mathsf{E} \end{pmatrix}$ ist. Dem Leser überlassen wir den einfachen Beweis, dass die $\mathbf{v}_i$ linear unabhängig und gleich den im Satz definierten Vektoren sind. □

Mit der Hilfe dieses Satzes kriegt man dann eine explizite Beschreibung von Lös$(\mathsf{A}, \mathbf{b})$. In unsrem Beispiel haben wir die spezielle Lösung

```
>> u=Eb(:,4), A*u

u =

   -0.3333
    0.6667
         0


ans =

    1.0000
    2.0000
    3.0000
```

und die Basis des eindimensionalen Lösungsraums des assoziierten homogenen Systems

```
>> v=[-Eb(1:2,3);1], A*v

v =

     1
    -2
     1


ans =

     0
     0
     0
```

So ist $\mathtt{u}+\mathbb{R}\mathtt{v}$ der Lösungsraum von $\mathsf{A}\mathbf{x} = \mathbf{b}$.

Bemerkung 5. Um Lös$(\mathsf{A}, 0)$ zu bestimmen, kann man direkt die reduzierte Zeilenstufenform von A betrachten.

Bemerkung 6. Soll man mehrere lineare Gleichungssysteme

$$\mathsf{A}\mathbf{x} = \mathbf{b}_1, \ldots, \mathsf{A}\mathbf{x} = \mathbf{b}_l,$$

mit derselben Koeffizientenmatrix $\mathsf{A} \in M(m \times n; \mathbb{K})$ auflösen, so kann man die erweiterte Koeffizientenmatrix $(\mathsf{A}, \mathbf{b}_1, \ldots, \mathbf{b}_l)$ betrachten und sie in reduzierte Zeilenstufenform $(\tilde{\mathsf{A}}, \tilde{\mathbf{b}}_1, \ldots, \tilde{\mathbf{b}}_l)$ überführen. Aus $\tilde{\mathsf{A}}$ liest man Lös$(\mathsf{A}, 0)$, während

$\tilde{\mathbf{b}}_j$, $j = 1, \ldots, l$, eine spezielle Lösung des j-ten Systems ist, falls seine letzten $m - \operatorname{Rang} \mathsf{A}$ Komponenten gleich Null sind.

Bemerkung 7. Es gibt spezielle MATLAB-Befehle um Lös$(\mathsf{A}, 0)$ und spezielle Lösungen von $\mathsf{A}\mathbf{x} = \mathbf{b}$ zu bestimmen; nämlich `null(A)`@`null`, `A\b`, `inv(A)*b`@`inv` und `pinv(A)*b`@`pinv`. Wir werden diese Befehle in 2.3.3 diskutieren. Zu bemerken ist, dass der Gebrauch dieser Befehle eine gewisse Vorsicht verlangt. Probieren Sie `inv(A)*b` in unsrem Beispiel einzugeben: Sie werden keine spezielle Lösung kriegen!

2.3.2. *Invertierbare Matrizen.* Eine Quadratmatrix heisst invertierbar, falls sie von maximalem Rang ist. Eine äquivalente Definition ist, dass sie eine inverse Matrix besitzt. Ist $\mathsf{A} \in M(n \times n; \mathbb{K})$ invertierbar, so ist ihre inverse Matrix $\mathsf{A}^{-1} \in M(n \times n; \mathbb{K})$ durch die Gleichung $\mathsf{A}^{-1}\mathsf{A} = \mathsf{E}$ oder, äquivalent, $\mathsf{A}\mathsf{A}^{-1} = \mathsf{E}$ definiert, wobei E die n-dimensionale Einheitsmatrix ist. Nennen wir $\mathbf{x}_j$, $j = 1, \ldots, n$, die j-te Spalte von A^{-1} und bemerken wir, dass die j-te Spalte von E das j-te Element $\mathbf{e}_j$ der kanonischen Basis von $\mathbb{K}^n$ ist, so sehen wir, dass die Bestimmung von A^{-1} äquivalent zur Auflösung folgender n linearer Gleichungssysteme ist:

$$\mathsf{A}\mathbf{x}_1 = \mathbf{e}_1, \ldots, \mathsf{A}\mathbf{x}_n = \mathbf{e}_n.$$

Gemäss Bemerkung 6 lösen wir sie auf, indem wir die erweiterte Koeffizientenmatrix

$$\mathsf{K} := (\mathsf{A}, \mathbf{e}_1, \ldots, \mathbf{e}_n) = (\mathsf{A}, \mathsf{E})$$

betrachten und sie in reduzierte Zeilenstufenform überführen:

$$\tilde{\mathsf{K}} = (\tilde{\mathsf{A}}, \tilde{\mathsf{E}}).$$

Hier ist $\tilde{\mathsf{A}}$ die zu A entsprechende Matrix in reduzierter Zeilenstufenform. Hat A maximalen Rang, so ist notwendigerweise $\tilde{\mathsf{A}} = \mathsf{E}$ (die Pivots müssen alle Stellen auf der Diagonale besetzen). Die Spalten der Matrix $\tilde{\mathsf{E}}$ sind in diesem Fall spezielle Lösungen der entsprechenden Gleichungssysteme (eigentlich die einzigen Lösungen). Also $\tilde{\mathsf{E}} = \mathsf{A}^{-1}$. Wir haben dann den folgenden Satz bewiesen (vgl. auch [F, 2.7.4]):

Satz 4. *Sei* A *eine* n*-dimensionale Quadratmatrix. Sei* $(\tilde{\mathsf{A}}, \tilde{\mathsf{E}})$ *die zu* (A, E) *entsprechende Matrix in reduzierter Zeilenstufenform. So ist* A *genau dann invertierbar, wenn* $\mathsf{A} = \mathsf{E}$ *gilt; in diesem Falle ist* $\tilde{\mathsf{E}}$ *die inverse Matrix zu* A*.*

Wir verwenden jetzt diese Methode in MATLAB. Sei `A` die bereits definierte Matrix. Wir brauchen noch die 3-dimensionale Einheitsmatrix:

```
>> A, E=eye(3)

A =

     1     2     3
     4     5     6
     7     8     9


E =

     1     0     0
     0     1     0
     0     0     1
```

Die Überführung in Zeilenstufenform ergibt:@`rref`

```
>> rref([A E])

ans =

    1.0000         0   -1.0000         0   -2.6667    1.6667
         0    1.0000    2.0000         0    2.3333   -1.3333
         0         0         0    1.0000   -2.0000    1.0000
```

und wir stellen fest, dass `A` nicht invertierbar ist (wie zu erwarten war, denn $\mathrm{Rang(A)} = 2 < 3$).

Modifizieren wir denn `A`, z.B. wie folgt:

```
>> A(3,3)=10

A =

     1     2     3
     4     5     6
     7     8    10

>> rref([A E])

ans =

    1.0000         0         0   -0.6667   -1.3333    1.0000
         0    1.0000         0   -0.6667    3.6667   -2.0000
         0         0    1.0000    1.0000   -2.0000    1.0000
```

Die neue Matrix `A` ist invertierbar. Ihre Inverse kriegen wir als rechtes Kästchen wie folgt:

```
>> B=ans(:,4:6)

B =

   -0.6667   -1.3333    1.0000
   -0.6667    3.6667   -2.0000
    1.0000   -2.0000    1.0000

>> A*B, B*A

ans =

     1     0     0
     0     1     0
     0     0     1
```

```
ans =

    1.0000    0.0000         0
         0    1.0000         0
         0         0    1.0000
```

Äquivalente Matrizen. In der Vorlesung (vgl. [F, 2.6.7]) haben wir zwei $m \times n$ Matrizen A und $\tilde{\mathsf{A}}$ als äquivalent definiert, falls es $\mathsf{S} \in GL(m, \mathbb{K})$ und $\mathsf{T} \in GL(n, \mathbb{K})$ gibt, so dass

$$\tilde{\mathsf{A}} = \mathsf{S}\mathsf{A}\mathsf{T}^{-1}.$$

Wir haben auch gesehen, dass jede Matrix von Rang r äquivalent zur Normalform $\left(\begin{smallmatrix} \mathsf{E}_r & 0 \\ 0 & 0 \end{smallmatrix}\right)$ ist, wobei E_r die r-dimensionale Einheitsmatrix ist. Das Problem ist dann, invertierbare Matrizen S und T zu bestimmen, die A in Normalform überführen.

Wie erklärt in [F, 2.7.6], kann man S und T durch Zeilen- und Spaltenumformungen bestimmen. Überführt man nämlich $(\mathsf{A}, \mathsf{E}_m)$ in reduzierte Zeilenstufenform $(\tilde{\mathsf{A}}, \tilde{\mathsf{E}}_m)$, so ist $\mathsf{S} = \tilde{\mathsf{E}}_m$. Überführt man dann $\left(\begin{smallmatrix} \tilde{\mathsf{A}} \\ \mathsf{E}_n \end{smallmatrix}\right)$ in reduzierte Spaltenstufenform $\left(\begin{smallmatrix} \hat{\tilde{\mathsf{A}}} \\ \hat{\mathsf{E}}_n \end{smallmatrix}\right)$ (d.h. die Transposition der zu $({}^t\tilde{\mathsf{A}}, \mathsf{E}_m)$ entsprechenden Matrix in reduzierter Zeilenstufenform), so ist $\hat{\tilde{\mathsf{A}}} = \left(\begin{smallmatrix} \mathsf{E}_r & 0 \\ 0 & 0 \end{smallmatrix}\right)$ und $\mathsf{T}^{-1} = \hat{\mathsf{E}}_n$.

Es folgt ein Beispiel in MATLAB. Wir überlassen dem Leser seine Überprüfung:

```
>> A=[1 2 0; 2 2 1]

A =

     1     2     0
     2     2     1

>> Er=eye(2), Ed=eye(3)

Er =

     1     0
     0     1


Ed =

     1     0     0
     0     1     0
     0     0     1

>> B=rref([A Er])

B =

    1.0000         0    1.0000   -1.0000    1.0000
         0    1.0000   -0.5000    1.0000   -0.5000

>> S=B(:,4:5)

S =
```

```
   -1.0000    1.0000
    1.0000   -0.5000

>> rref([B(:,1:3); Ed]')'

ans =

    1.0000         0         0
         0    1.0000         0
         0         0    1.0000
    0.5000    1.0000   -0.5000
    1.0000         0   -1.0000

>> T1=ans(3:5,:)

T1 =

         0         0    1.0000
    0.5000    1.0000   -0.5000
    1.0000         0   -1.0000


>> S*A*T1

ans =

     1     0     0
     0     1     0
```

2.3.3. *Lösungsräume durch MATLAB-Befehle.* Man hat u.a. folgende Befehle zur Verfügung:

@null@inv@pinv		
	`null(A)`	eine Basis von Lös$(\mathsf{A}, 0)$
	`A\b`	eine spezielle Lösung von $\mathsf{A}\mathbf{x} = \mathbf{b}$
	`inv(A)`	die inverse Matrix zu A
	`pinv(A)`	die pseudoinverse Matrix zu A

Nullraum. Wir beginnen mit der Beschreibung von `null@null`. Es gibt zwei Varianten dieses Befehls: `null(A)` und `null(A,'r')`. Beide Befehle ergeben eine Basis des Nullraums von A, d.h. Lös$(\mathsf{A}, 0)$; die Basisvektoren werden als Spalten einer Matrix gegeben. Nur die Methoden sind verschieden. Mit der Option `'r'`, die die sog. rationale Basis ergibt, wird die Basis durch die Überführung der Matrix in reduzierte Zeilenstufenform (s. Satz 3 auf Seite 35) berechnet. Es kann aber passieren, dass eine Spalte der Matrix in Zeilenstufenform nur wegen Rundungsfehler gleich Null gesetzt wird; die so erhaltene Basis ist in diesem Falle falsch. Eine schwierige, langsamere aber sicherere Methode wird im Befehl `null(A)` (ohne Optionen) verwendet.[6]

Ist Lös$(\mathsf{A}, 0) = \{0\}$, so erhält man mit beiden Befehlen eine leere Matrix.

[6]Die so erhaltene Basis ist ferner normiert (d.h., ${}^t v_i \cdot v_j = \delta_{ij}$ für alle Vektoren aus der Basis), was oft vorteilhaft ist.

```
>> A=[1 2; 4 5; 7 8]

A =

     1     2
     4     5
     7     8

>> null(A), null(A,'r')

ans =

   Empty matrix: 2-by-0


ans =

   Empty matrix: 2-by-0
```

Im Falle der in 2.3.1 diskutierten Matrix sehen wir den Unterschied zwischen die zwei Methoden:

```
>> A(:,3)=[3 6 9]'

A =

     1     2     3
     4     5     6
     7     8     9

>> null(A), null(A,'r')

ans =

   -0.4082
    0.8165
   -0.4082


ans =

     1
    -2
     1
```

Betrachten wir ein Beispiel mit zweidimensionalen Lösungsraum.

```
>> A(:,4)=[1 4 7]'

A =
```

```
     1     2     3     1
     4     5     6     4
     7     8     9     7

>> null(A), null(A,'r')

ans =

   -0.3726   -0.6377
    0.8292   -0.1993
   -0.4146    0.0997
   -0.0420    0.7374


ans =

     1    -1
    -2     0
     1     0
     0     1
```

Man stelle die rationale Basis in der reduzierten Zeilenstufenformfest:

```
>> rref(A)

ans =

     1     0    -1     1
     0     1     2     0
     0     0     0     0
```

Division von links. Das Symbol `\` beschreibt die Division von links. Ist `A` eine invertierbare Matrix, so ist `A\b` äquivalent zu $\texttt{A}^{-1}\texttt{b}$. Sonst ergibt `A\b` eine spezielle Lösung des Gleichungssystems `Ax=b`, die man durch das Eliminationsverfahren von Gauss erhält (s. Satz 3).

```
>> A(:,4)=[], b=[1 2 3]', A\b

A =

     1     2     3
     4     5     6
     7     8     9


b =

     1
     2
```

```
     3

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-18.
(Type "warning off MATLAB:nearlySingularMatrix" to suppress this warning.)

ans =

   -0.3333
    0.6667
         0
```

Inverse und pseudoinverse Matrizen. Ist eine Matrix `B` invertierbar, so können wir durch ihre Inverse (die in MATLAB mit dem Befehl `inv@inv` berechnet wird) alle Gleichungssysteme auflösen, für die sie die Koeffizientenmatrix ist.

```
>> B=[1 2 3; 4 5 6; 7 8 10], inv(B), x=inv(B)*b

B =

     1     2     3
     4     5     6
     7     8    10


ans =

   -0.6667   -1.3333    1.0000
   -0.6667    3.6667   -2.0000
    1.0000   -2.0000    1.0000


x =

   -0.3333
    0.6667
   -0.0000

>> B*x

ans =

    1.0000
    2.0000
    3.0000
```

Normalerweise ist es aber sicherer und schneller die mit `B\b` erhaltene Lösung zu betrachten.

Ist die Matrix aber nicht invertierbar, so ergibt `inv(A)` nicht ihre Inverse, und `inv(A)*b` ist i.A. keine Lösung von `Ax=b`.

```
>> inv(A), x=inv(A)*b
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-18.

ans =

   1.0e+16 *

   -0.4504    0.9007   -0.4504
    0.9007   -1.8014    0.9007
   -0.4504    0.9007   -0.4504

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-18.

x =

    -2
     4
     0

>> A*x

ans =

     6
    12
    18
```

Es ist also immer empfehlenswert `A\b` einzugeben, um eine korrekte Lösung zu finden.

Eine noch bessere Methode verwendet die sog. pseudoinverse Matrix: die pseudoinverse Matrix B zu A erfüllt die Bedingungen $\mathsf{ABA} = \mathsf{A}$ und $\mathsf{BAB} = \mathsf{B}$.[7] Es folgt, dass die inverse Matrix, falls sie existiert, auch eine pseudoinverse Matrix ist. Ferner ist $\mathsf{B}\mathbf{b}$ eine spezielle Lösung des Gleichungssystems $\mathsf{A}\mathbf{x} = \mathbf{b}$. In MATLAB wird die Pseudoinverse mit dem Befehl `pinv`@`pinv` berechnet.[8]

```
>> pinv(A), x=pinv(A)*b, y=A\b

ans =
```

[7] Ferner sind mit dem in MATLAB verwendeten Algorithmus AB und BA hermitesch.

[8] Der Unterschied zwischen `x` und `y` ist wie folgt:

(1) `x` ist die spezielle Lösung mit kleinster Norm (die Norm eines Vektors $\mathbf{v}$ ist durch $||\mathbf{v}|| := \sqrt{{}^t\mathbf{v}\,\mathbf{v}}$ definiert).

(2) `y` ist die spezielle Lösung mit kleinster Anzahl der von Null verschiedenen Komponenten.

```
   -0.6389   -0.1667    0.3056
   -0.0556    0.0000    0.0556
    0.5278    0.1667   -0.1944


x =

   -0.0556
    0.1111
    0.2778

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-18.
(Type "warning off MATLAB:nearlySingularMatrix" to suppress this warning.)

y =

   -0.3333
    0.6667
         0
```

Die Berechnung von `y` könnte wegen Rundungsfehler unter Ungenauigkeiten leiden, wie MATLABs Meldung Bescheid sagt. Also ist die Verwendung von `pinv` empfehlenswert.

2.4. **Weitere MATLAB-Befehle für Matrizen.** Ein wichtiges Problem in der linearen Algebra ist die Bestimmung von Determinanten, Eigenwerten und Eigenvektoren von Quadratmatrizen. Dafür gibt es spezielle MATLAB-Befehle.

Die Determinante wird mit `det`@`det` berechnet:

```
>> B, det(B)

B =

     1     2     3
     4     5     6
     7     8    10


ans =

    -3
```

Die Eigenwerte erhält man mit `eig`@`eig`:

```
>> eig(B)

ans =

   16.7075
```

```
   -0.9057
    0.1982
```

Will man Eigenvektoren und Eigenwerte der Matrix `B` bestimmen, so muss man den Befehl folgendermassen eingeben:

```
>> [V D]=eig(B)

V =

   -0.2235   -0.8658    0.2783
   -0.5039    0.0857   -0.8318
   -0.8343    0.4929    0.4802


D =

   16.7075         0         0
         0   -0.9057         0
         0         0    0.1982
```

Die Eigenwerte sind jetzt die Diagonaleinträge der Diagonalmatrix `D`. Die i-te Spalte von `V` ist der Eigenvektor zu dem Eigenwert, der sich auf der i-ten Diagonalstelle von `D` befindet. Z.B.:

```
>> B*V(:,1)-D(1,1)*V(:,1)

ans =

   1.0e-14 *

   -0.0444
   -0.8882
   -0.7105
```

In anderen Worten gilt `B*V=V*D`:

```
>> B*V-V*D

ans =

   1.0e-14 *

   -0.0444    0.0333   -0.0007
   -0.8882    0.0083   -0.0888
   -0.7105    0.0333   -0.0541
```

3. Programmieren

Programme schreibt man, um automatisch Operationen zu machen oder um Verfahren zu implementieren.

3.1. **Beispiel: die Spur.** Wir beginnen mit einem kleinen Beispiel: die Berechnung der Spur einer quadratischen Matrix $\mathsf{A} = (a_{ij}) \in M(n \times n; \mathbb{R})$:

$$\operatorname{Sp} \mathsf{A} := \sum_{i=1}^{n} a_{ii}.$$

MATLAB stellt den Befehl `trace` (aus dem englischem Wort *trace* = Spur) für dieses Ziel zur Verfügung. Z.B.:

```
>> A=[1 2 3; 4 5 6; 7 8 9]


A =

     1     2     3
     4     5     6
     7     8     9


>> trace(A)


ans =

    15
```

Wir können auch direkt die Definition von Spur anwenden, indem wir andere, vordefinierte Befehle, wie z.B. `diag` und `sum`, gebrauchen.

```
>> sum(diag(A))


ans =

    15
```

Nehmen wir an, der Befehl `trace` stehe nicht zur Verfügung, und wir wollen dann einen Befehl `spur` definieren, um Spuren zu berechnen. Dieser Befehl soll als `sum(diag( ))` definiert werden. Diese Definition schreiben wir in einer Datei (dem Programm). Der Dateiname muss mit dem Namen des zu definierenden Befehls übereinstimmen, während die Erweiterung notwendigerweise „`.m`“ ist. In unserem Beispiel sollen wir also eine Datei

`spur.m`

erzeugen. Das kann man im Prinzip mit jedem Editor tun, aber es ist angenehmer den Editor, der bei MATLAB zur Verfügung steht, zu verwenden. Um diesen aufzurufen, gibt es den Befehl `edit`. Also schreiben wir:

```
>> edit spur.m
```

Im neuen Fenster können wir jetzt unser Programm schreiben. Wollen wir einen Befehl (welchen MATLAB als „Funktion“ betrachtet), so muss die erste Zeile das Wort `function` enthalten. In derselben Zeile geben wir die Struktur des zu definierenden Befehls an. Also in unserem Beispiel

```
function spur(A)
```

Hier ist `A` eine beliebige Variable. Unter dieser Schreibweise ist dabei gemeint, dass der Befehl `spur` nur ein Argument hat, und dass wir in der Definition vom Befehl `spur` sein Argument mit `A` bezeichnen werden. Der Befehlsname `spur` wird auch als Variable verstanden, welcher wir schliesslich dem zu berechnenden Wert zuordnen werden. Mit Blick auf diese Vereinbarungen schreiben die zweite und letzte Zeile unseres Programms wie folgt:

```
spur = sum(diag(A))
```

Das Programm ist im Prinzip fertig. Es ist aber immer nützlich, ein kurzes Kommentar in jedem Programm zur Erinnerung zu schreiben, damit man später noch weiss, was dieses Programm tut. Kommentare (das gleiche gilt auch für TEX und LATEX) schreibt man nach dem Prozentzeichen `%`. Schliesslich sieht unser Programm so aus:

```
function spur(A)
% Die Spur der Matrix A wird berechnet
spur = sum(diag(A))
```

Speichern wir die Datei, so steht der Befehl `spur` von jetzt an zur Verfügung, und wir können Spuren von Matrizen damit berechnen.

```
>> spur(A)


spur =

    15

B =

     1     2     3
     4     5     6


>> spur(B)


spur =

     6
```

Man bemerke, dass der so definierte Befehl `spur`, wie übrigens auch der vordefinierte Befehl `trace`, Spuren auch von nicht-quadratischen Matrizen berechnet.

Das Kommentar, das wir hinzugefügt haben, kann man auch mit dem Befehl `help` (= Hilfe) lesen:

```
>> help spur


  Die Spur der Matrix A wird berechnet
```

3.2. **Verzeichnisbefehle.** Man kann gespeicherte Dateien direkt mit MATLAB durch Anwenden des Befehls `dir` (oder, äquivalent, `ls`) anschauen. Wollen wir z.B. alle M-Dateien (in unserem Fall nur `spur.m`) auflisten, so schreiben wir:

```
>> ls *.m


ans =

spur.m
```

Das können wir auch mit dem Befehl `what` (= was) tun, der nur M-Dateien auflistet:

```
>> what


M-files in the current directory /home/asc

spur
```

Man kann das Verzeichnis mit dem Befehl `cd` (= *change directory* = wechsle Verzeichnis) wechseln und damit M-Dateien in anderen Verzeichnissen speichern.

3.3. **Beispiel: die Fakultät.** Die Fakultät ist die auf $\mathbb{N}$ definierte Funktion

$$n! := 1 \cdot 2 \cdot 3 \cdots n, \quad n > 0; \qquad 0! := 1.$$

Die Fakultät (auf englisch *factorial*) berechnet man in MATLAB mit dem Befehl `factorial`. Z.B.:

```
>> factorial(5)


ans =

   120


>> factorial(0)
```

```
ans =

     1


>> factorial(5.3)

??? Error using ==> factorial
N must be a positive integer.


>> factorial(-7)

??? Error using ==> factorial
N must be a positive integer.
```

Wir wollen jetzt ein Programm schreiben, um die Fakultät durch einen neuen Befehl `fak` zu berechnen. Anfangspunkt ist klarerweise:

```
>> edit fak.m
```

Um die Fakultät zu definieren, verwenden wir die bereits erklärte `for`-Schleife. Z.B., um 5! zu berechnen, gehen wir wie folgt vor:

```
>> f=1; for i=2:5, f=f*i; end, f


f =

   120
```

Also schreiben wir das Programm:

```
function fak(n)
% Fakultaet
fak=1;
for i=2:n
    fak=fak*i;
end
fak
```

Man bemerke folgende Tatsachen:

(1) Man muss zu Beginn der Schleife der Variablen `fak` den Wert 1 zuordnen.
(2) Man muss Befehle mit einem Strichpunkt beenden, falls die Antwort nicht angezeigt werden soll.

(3) Da das Ergebnis der Berechnung nicht durch die ersten sechs Zeilen angezeigt worden ist, braucht man explizit eine weitere Zeile zu schreiben, um den Wert der Variablen `fak` zu zeigen.

Man kann den letzten Schritt vermeiden, wenn man explizit eine Variable einführt, zu der das Ergebnis der Berechnung zuordnet wird. Z.B., statt `fak` können wir als Ergebnis die Variable `f` einführen, indem wir die erste Zeile wie folgt modifizieren:

```
function f = fak(n)
```

Von jetzt an sollen wir aber die Variable `f` statt der Variablen `fak` verwenden. Das Programm soll zum Schluss so aussehen:

```
function f = fak(n)
% Fakultaet
f=1;
for i=2:n
    f=f*i;
end
```

Bemerkung 8. Diese Schreibeweise ist sehr empfehlenswert. Nicht nur ermöglicht sie, eine Zeile zu sparen, sondern sie ermöglicht auch komplexere Programmierungsstrukturen zu verwenden, wie im Folgenden erklärt wird.

Nach dem Speichern können wir den Befehl `fak` verwenden:

```
>> fak(5)


fak =

   120


>> fak(0)


fak =

     1


>> fak(5.3)


fak =

   120


>> fak(-7)
```

```
fak =

     1
```

Unser Befehl `fak` erweitert also die Definition der Fakultät auch zu nicht-natürlichen Zahlen wie folgt:

$$\mathrm{fak}(n) = \begin{cases} \lfloor n \rfloor ! & n \geq 0 \\ 1 & n < 0 \end{cases}$$

wobei $\lfloor n \rfloor$ die Abrundung von n bezeichnet.

Man kann sich an die genaue Definition anpassen, indem man Bedingungen einführt. Das diskutieren wir im nächsten Unterabschnitt, nach welchem wir zurück zum Beispiel der Fakultät zurückkeheren.

Offensichtlich ist das nicht nötig, wenn man darauf achtet, dass der Befehl nur für natürliche Zahlen korrekt funktioniert. Auf diese Weise kann man das bereits erklärte einfachere und schnellere Programm verwenden.

3.4. **Logische Strukturen.** Um Algorithmen zu schreiben, braucht man Bedingungen zu betrachten. Das macht man in MATLAB durch die Befehle

`if`	= wenn
`else`	= andernfalls
`elseif`	= andernfalls: wenn

Der Befehl `if` führt eine Folge Bedingungen und Anweisungen ein, die durch den Befehl `end` beendet wird. Die möglichen Strukturen sind:

- `if` Bedingung; Anweisungen; `end`
- `if` Bedingung; Anweisungen$_1$; `else` Anweisungen$_2$; `end`
- `if` Bedingung$_1$; Anweisungen$_1$; `elseif` Bedingung$_2$; Anweisungen$_2$; `end`
- `if` Bedingung$_1$; Anweisungen$_1$; `elseif` Bedingung$_2$; Anweisungen$_2$; `else` Anweisungen$_3$; `end`

Eine Anweisung ist eine Liste aus Befehlen (die möglicherweise Schleifen oder weitere Verzweigungen enthalten kann). Eine Bedingung ist ein logischer Ausdruck (der also nur zwei Werte, richtig oder falsch, annehmen kann). Die einfachsten mathematischen Bedingungen sind Gleichungen und Ungleichungen. Diese werden in MATLAB durch die Symbole `==`, `<`, `>`, `<=`, `>=`, und `~=` ausgedrückt.

`a == b`	Ist `a` gleich `b`?
`a < b`	Ist `a` kleiner als `b`?
`a > b`	Ist `a` grösser als `b`?
`a <= b`	Ist `a` kleiner oder gleich `b`?
`a >= b`	Ist `a` grösser oder gleich `b`?
`a ~= b`	Ist `a` ungleich `b`?

Zu bemerken ist, dass `==` der logische Operator ist, die eine Gleichung ausdrückt, während `=` ein Zuweisungs-Operator ist, welcher Variablen Werte zuordnet.

Das logische NICHT wird durch `~`, das logische UND durch `&` und das logische ODER durch `|` bezeichnet. Z.B.:

`a < b | a == c`	Ist `a` kleiner als `b` oder ist `a` gleich `c`?
`a <= b & a ~= 0`	Ist `a` kleiner oder gleich `b` und ist `a` ungleich `0`?

3.5. **Wieder zum Beispiel Fakultät.** Durch `if`-Zweigungen könnnen wir jetzt das Programm `fak` verbessern. Um Verwirrungen zu vermeiden, geben wir dem neuen Programm einen neuen Namen, z.B., `vfak`. Das verbesserte Programm, das wir in der Datei `vfak.m` speichern, soll z.B. wie folgt aussehen:

```
function f = vfak(n)
% verbesserte Fakultaet
if n < 0
    disp('Fehler: keine Fakultaet von negativen Zahlen')
elseif n ~= round(n)
    disp('Fehler: keine Fakultaet von nicht-ganzen Zahlen')
else
    f=1;
    for i=2:n
        f=f*i;
    end
end
```

Der neu-eingeführte Befehl `disp` erlaubt das Anzeigen seines Arguments. Wenn ein Text anzuzeigen ist, wie in unserem Fall, schreibt man ihn zwischen einfachen Anführungszeichen.

Der neue Befehl verhält sich wie folgt:

```
>> vfak(5)


ans =

   120


>> vfak(0)


ans =

     1


>> vfak(5.3)

Fehler: keine Fakultaet von nicht-ganzen Zahlen

>> vfak(-7)

Fehler: keine Fakultaet von negativen Zahlen
```

3.5.1. *Rekursive Programmierung.* MATLAB erlaubt die rekursive Programmierung; damit ist gemeint, dass ein Programm sich selbst aufrufen kann. Das bedürft eine gewisse Vorsicht, denn es könnte zu unendlichen Schleifen führen. Andererseits ist die sorgfältige rekursive Programmierung äusserst schlagkräftig, denn sie ist die algorithmische Version der vollständigen Induktion aus der Mathematik.

Wir erläutern die rekursive Programmierung durch ein Beispiel: wiederum die Fakultät. Die kann man nämlich auch rekursiv definieren:

$$n! = n\,(n-1)!.$$

Dank dieser Eigenschaft, können wir ein neues Programm für MATLAB schreiben, welches die Fakultät berechnet. Wir nennen es diesmal `rfak`. Wir beginnen mit

```
>> edit rfak.m
```

Die einfachste Version des Programms sieht dann wie folgt aus:

```
function r = rfak(n)
% Fakultaet: rekursive Definition
if n == 0
    r = 1;
else
    r = n * rfak(n-1);
end
```

Das ergibt in Beispielen:

```
>> rfak(5)


ans =

   120


>> rfak(0)


ans =

     1


>> rfak(5.3)

??? Maximum recursion limit of 100 reached. Use set(0,'RecursionLimit',N)
to change the limit.  Be aware that exceeding your available stack space can
crash MATLAB and/or your computer.
```

```
Error in ==> /home/asc/rfak.m
On line 6  ==>     r = n * rfak(n-1);


>> rfak(-7)

??? Maximum recursion limit of 100 reached. Use set(0,'RecursionLimit',N)
to change the limit.  Be aware that exceeding your available stack space can
crash MATLAB and/or your computer.

Error in ==> /home/asc/rfak.m
On line 6  ==>     r = n * rfak(n-1);
```

Wie zu erwarten ist, führt diese rekursive Definition zu grossen Problemen, wenn man sie ausserhalb ihres Definitionsbereichs (in diesem Fall $\mathbb{N}$) verwendet. Schreibt man das Programm für sich selbst und man ist sicher, dass man es immer nur unter den richtigen Bedingungen verwenden wird, so braucht man sich keine grosse Sorge zu machen. Andernfalls ist es empfehlenswert, `if`-Zweigungen einzuführen, um Probleme zu vermeiden. Das sei dem Leser überlassen.

3.6. **Beispiel: Überführung in Zeilenstufenform.** Als weiteres Beispiel diskutieren wir die Implementierung des Eliminationsverfahrens von Gauss zur Überführung einer Matrix in Zeilenstufenform. Wir verfolgen [F, 0.4.7]. Wir schreiben das Programm `zsf.m` wie folgt.

```
function B = zsf(A)
% Die Matrix A wird durch elementare Zeilenumformungen
% in die Matrix B in Zeilenstufenform ueberfuehrt
if A == 0
    B = A;
else
    [m n] = size(A);
    if m<=0 | n<=0
        B = A;
    else
        j1 = 1;
        while A(:,j1) == 0
            j1 = j1+1;
        end
        [a,i1] = max(abs(A(:,j1)));
        a = A(i1,j1);
        B = A([i1:m 1:(i1-1)],:);
        v = B(1,:)/a;
        for i = 2:m
            B(i,:) = B(i,:) - B(i,j1)*v;
        end
        B(2:m,(j1+1):n)=zsf(B(2:m,(j1+1):n));
    end
end
```

Zur Erklärung des Programms:

- In der 4. Zeile verifizieren wir, ob die Matrix `A` gleich Null ist, in welchem Fall sie schon in Zeilenstufenform ist, und wir sind fertig.
- In der 7. Zeilen berechnen wir die Dimension der Matrix `A`, was wir später brauchen werden.
- In der 8. Zeilen verifizieren wir, ob die Matrix eine nicht-positive Anzahl Zeilen oder Spalten hat (wie bereits gesehen ist das in MATLAB möglich). In diesem Fall ist die Matrix schon definitionsgemäss in Zeilenstufenform.
- In der 12., 13. und 14. Zeile haben wir eine `while`-Schleife zur Bestimmung der ersten von Null verschiedenen Spalte, die wegen der obigen Überprüfungen existiert. Eine `while`-Schleife ist oft besser als eine `if`-Schleife. Sie hat die folgende Struktur

  `while` Bedingung; Anweisungen; `end`

  und die folgende Wirkung: solange (=*while* auf Englisch) die Bedingung erfüllt ist, wird die Anweisung ausgeführt.
- In der 15. Zeile bestimmen wir die erste Stelle (`i1`) in der `j1`-ten Spalte, wo sich der Eintrag mit grösstem Betrag befindet, und in der 16. Zeile nennen wir diesen von Null verschiedenen Eintrag `a`.
- In der 17. Zeile führen wir folgende Zeilenpermutation

$$\begin{pmatrix} i_1 & i_1+1 & \dots & m & 1 & \dots & i_1-1 \\ 1 & 2 & \dots & m-i_1+1 & m-i_2+2 & \dots & m \end{pmatrix},$$

  durch, nach welcher sich der Eintrag `a` in der ersten Zeile befindet und zum ersten Pivot wird. Wir nennen `B` die so erhaltene Matrix.
- In der 18. Zeilen berechnen wir einen Vektor `v` als Vielfaches der ersten Zeile, aber mit 1 als j_1-Komponente.
- In der 19., 20. und 21. Zeile haben wir eine `for`-Schleife, um alle Einträge zu löschen, die unterhalb des Pivots sind.
- In der 22. Zeile überführen wir durch rekursive Programmierung das Kästchen $\mathtt{B}_{ij}$, $i > 1$ und $j > j_1$, in Zeilenstufenform.

Um das Verfahren zu sehen, können wir die Strichpunkte in der 16. und 19. Zeilen weglassen. Dann bekommt man z.B.:

```
>> A=[1 2 3 4;5 6 7 8; 9 10 11 12]


A =

     1     2     3     4
     5     6     7     8
     9    10    11    12


>> zsf(A)


B =
```

```
     9    10    11    12
     1     2     3     4
     5     6     7     8


B =

    9.0000   10.0000   11.0000   12.0000
         0    0.8889    1.7778    2.6667
    5.0000    6.0000    7.0000    8.0000


B =

    9.0000   10.0000   11.0000   12.0000
         0    0.8889    1.7778    2.6667
         0    0.4444    0.8889    1.3333


B =

    0.8889    1.7778    2.6667
    0.4444    0.8889    1.3333


B =

    0.8889    1.7778    2.6667
         0   -0.0000   -0.0000


B =

   1.0e-15 *

   -0.8882   -0.2220


ans =

    9.0000   10.0000   11.0000   12.0000
         0    0.8889    1.7778    2.6667
         0         0   -0.0000   -0.0000
```

3.7. **Beispiel: Überführung in Zeilenstufenform über $\mathbb{Z}/p\mathbb{Z}$.** In diesem Abschnitt bezeichnet p eine Primzahl. Wir wollen jetzt das Programm `zsf` modifizieren, so dass wir auch Matrizen mit Einträgen aus dem Körper $\mathbb{Z}/p\mathbb{Z}$ in Zeilenstufenform überführen können. Problematisch ist nur die 17. Zeile, wobei wir eine

Division, jetzt modulo p, durchführen müssen. Dazu benötigen wir einen kleinen theoretischen Exkurs.

Es sei zuerst gemerkt, dass die Binomialkoeffizienten

$$\binom{p}{k} := \frac{p!}{k!\,(p-k)!} = \frac{p(p-1)\cdots(p-k+2)(p-k+1)}{k!}$$

durch p teilbar sind, falls $0 < k < p$. (Das könnte nicht wahr sein, wenn p keine Primzahl wäre, denn einer der Faktoren von $k!$ könnte den Faktor p im Zähler teilen.) Also zeigt die Binomialformel

$$(a+1)^p = \sum_{k=0}^{p} \binom{p}{k} a^k,$$

dass

$$(a+1)^p \equiv a^p + 1 \mod p \qquad \forall a \in \mathbb{Z}. \tag{3.1}$$

Eine Folgerung davon ist der

Lemma 1.

$$a^p \equiv a \qquad \forall a \in \mathbb{Z}.$$

Beweis. Durch Induktion über a. Für $a \equiv 0$ gilt die Formel trivialerweise. Aus (3.1) und aus der Induktionsannahme $a^p \equiv a$ folgt, dass $(a+1)^p \equiv a+1$. □

Da $\mathbb{Z}/p\mathbb{Z}$ ein Körper ist, folgt der

Satz 5 (Fermat)**.**

$$a^{p-1} \equiv 1 \qquad \forall a \not\equiv 0.$$

Dividieren wir nochmal durch a, so kriegen wir die erwünschte Inversionsformel:

$$a^{p-2} \equiv a^{-1} \qquad \forall a \not\equiv 0. \tag{3.2}$$

Wir können die obigen Resultate in Beispielen mit MATLAB verifizieren, indem wir den Befehl `mod(a,p)` verwenden, der die Restklasse von `a` modulo `p` berechnet. Z.B.:

```
>> mod(125,11)


ans =

     4
```

Der Satz von Fermat verifizieren wir anhand des folgenden Beispiels:

```
>> mod(3^6,7)


ans =

     1
```

und das Inverse von 3 modulo 13 berechnen wir wie folgt:

```
>> mod(3^11,13)


ans =

     9


>> mod(3*9,13)


ans =

     1
```

Es wäre dann verlockend, einen Befehl `rinv` (= Inversion von Restklassen) folgendermassen zu definieren:

```
function b=rinv(a,p)
% berechnet das Inverse von a in Z/pZ.
%
% Komische Resultate falls a nicht ganz ist
% oder falls p keine Primzahl ist.
b=mod(a^(p-2),p);
```

Das funktioniert reibungslos für kleine Zahlen, z.B.,

```
>> rinv(3,13)


ans =

     9


>> rinv(6,11)


ans =

     2
```

ist aber sonst problematisch :

```
>> rinv(15,17)
```

```
ans =

     0
```

Grund dafür ist, dass 15^{15} eine riesige Zahl ist,

```
>> 15^15


ans =

   4.3789e+17
```

und MATLAB ist wegen des Rundungsfehlers nicht imstande, die Restklasse richtig zu berechnen:

```
>> mod(ans,17)


ans =

     0
```

Ein möglicher Ausweg besteht darin, dass man immer nur kleine Zahlen verwendet. Das ist immer möglich, denn die Restklasse von a^k ist das Produkt der Restklassen von a und a^{k-1}. So brauchen wir immer nur Zahlen zu multiplizieren, die kleiner sind als p. Wir verbessern also das Programm `rinv.m` wie folgt:

```
function b=rinv(a,p)
% berechnet das Inverse von a in Z/pZ.
%
% Komische Resultate falls a nicht ganz ist
% oder falls p keine Primzahl ist.
a1=mod(a,p);
b=a1;
for i=1:p-3
    b = mod(a1*b,p);
end
```

Und wir kriegen z.B:

```
>> rinv(15,17)


ans =

     8
```

```
>> mod(15*8,17)


ans =

     1
```

Jetzt können wir `zsf.m` modifizieren. Wir nennen das neue Programm `rzsf.m` (= Zeilenstufenform mit Restklassen). Es ist eine gute Idee nicht nur die 18. Zeile von `zsf.m` zu verändern, in welcher die Division durch `a` mit `rinv` durchgeführt werden kann, sondern auch modulo `p` zu reduzieren, dort wo es möglich ist. Aus dieser Art werden alle Berechnungen vereinfacht. Es ist immerhin noch zu erwähnen, dass der Befehl `mod(a,p)` funktioniert, auch wenn `a` eine Matrix ist: in diesem Fall wird jeder Eintrag modulo `p` reduziert. Schliesslich brauchen wir nicht mehr Beträge von Einträgen zu betrachten, wie in der 15. Zeile von `zsf.m` der Fall war. Diese Überlegungen führen dann zum folgenden Programm:

```
function B = rzsf(A,p)
% Die Matrix A mit Koeffizienten in Z/pZ
% wird durch elementare Zeilenumformungen
% in die Matrix B in Zeilenstufenform ueberfuehrt
%
% Komische Resultate falls die Eintraege von A nicht ganz sind
% oder falls p keine Primzahl ist.
A1=mod(A,p);
if A1 == 0
    B = A1;
else
    [m n] = size(A);
    if m<=0 | n<=0
        B = A1;
    else
        j1 = 1;
        while A1(:,j1) == 0
            j1 = j1+1;
        end
        [a,i1]=max(A1(:,j1));
        B = A1([i1:m 1:(i1-1)],:);
        v = mod(B(1,:)*rinv(a,p),p);
        for i = 2:m
            B(i,:) = mod(B(i,:) - B(i,j1)*v,p);
        end
        B(2:m,(j1+1):n)=rzsf(B(2:m,(j1+1):n),p);
    end
end
```

Es sei dem Leser überlassen, die Bedeutung der einzelnen Zeilen zu verdeutlichen.

Danksagung. Nicola Kistler hat mich in der Bearbeitung der deutschen Version unterstützt; ich danke ihm.

Literatur

[F] G. Fischer, *Lineare Algebra*, 13. Auflage, Vieweg Studium, 2002.
[M] D. R. Hill, *Experiments in Computational Matrix Algebra*, Random House, 1988.
[B] H. Benker, *Mathematik mit MATLAB*, Springer, 2000.

Index