Kapitel 2

Arrays

2.1 Konzept

Eine der großen Stärken von MATLAB liegt im einfachen Umgang mit Matrizen bzw. Arrays (Felder), wobei diese beiden Bezeichnungen praktisch gleichbedeutend verwendet werden. In MATLAB werden beinahe alle Größen als Arrays behandelt. An dieser Stelle beschränken wir uns auf numrische Arrays, deren Inhalt Zahlen sind. Später werden auch andere Typen, wie z.B.: Zeichenketten, Zellen, oder Strukturen besprochen werden. Am einfachsten vorstellen kann man sich also ein Array als eine geordnete Anordnung von Zahlen, deren Bedeutung natürlich unterschiedlich sein kann.

So kann man den Inhalt verstehen als,

- Matrix im Sinne der linearen Algebra,
- Tensor oder Vektor im Sinne der Vektor-Tensor-Rechnung,
- Menge von Zahlen im Sinne der Mengenlehre,
- numerisches Ergebnis einer Berechnung, z.B.: der Funktion $f(x, y) = \sin xy$ für verschiedene (geordnete) Werte von x und y,
- Resultat eines Lesevorgangs (Zeilen und Spalten einer Tabelle).

Anders als die meisten anderen Programmiersprachen kann Matlab die meisten Operationen nicht nur auf einzelne Zahlen, sondern auch auf ganze Arrays anwenden. Man kann also beispielsweise Matrizen miteinander multiplizieren, muss sich aber natürlich bewußt sein, dass dies zumindest auf zwei verschiedene Arten geschehen kann:

- Matrizenmultiplikation im Sinne der linearen Algebra.
- Elementweises Multiplizieren für numerische Berechnungen.

Tabelle 2.1: Eigenschaften von Arrays: Dimension, Größe, Länge, Anzahl

Bezeichnung	Elemente	Dimension `ndims`	Größe `size`	Länge `length`	Anzahl `numel`
Leeres Array	0	2	`[0 0]`	0	0
Skalar	1	2	`[1 1]`	1	1
Zeilenvektor	3	2	`[1 3]`	3	3
Spaltenvektor	3	2	`[3 1]`	3	3
2-dim Matrix	3×4	2	`[3 4]`	4	12
3-dim Matrix	$3 \times 4 \times 2$	3	`[3 4 2]`	4	24
⋮					

2.2 Eigenschaften von Arrays

Wichtige Eigenschaften von Arrays sind neben ihrem Inhalt,

- ihre Dimension, und
- ihre Größe, entspricht der Anzahl der Elemente in jeder Dimension, und
- ihre Länge, entspricht der maximalen Ausdehnung in einer beliebigen Dimension.

In Tabelle 2.1 kann man erkennen, dass auch leere Arrays, Skalare und Vektoren die Dimension 2 haben. Daran sieht man, dass in MATLAB jede Zahl als zumindest 2-dim Array aufgefasst wird.

2.3 Hilfe für Arrays

Eine genaue Erklärung der einzelnen Befehle in MATLAB erhält man durch Aufruf des Befehls `help` also z.B.: `help ndims`. Man kann auch den Links in diesem Dokument folgen, bzw. erhält man mit `doc ndims` die Hilfe in MATLAB in HTML Format.

MATLAB HELP: `NDIMS`

```
 Number of dimensions.

N = NDIMS(X) returns the number of dimensions in the array X.
The number of dimensions in an array is always greater than
or equal to 2.  Trailing singleton dimensions are ignored.
Put simply, it is LENGTH(SIZE(X)).
```

In Ergänzung dazu lautet die Hilfe für `SIZE`:

MATLAB HELP: `SIZE`

```
Size of matrix.

D = SIZE(X), for M-by-N matrix X, returns the two-element
row vector D = [M, N] containing the number of rows and columns
in the matrix.  For N-D arrays, SIZE(X) returns a 1-by-N
vector of dimension lengths.

[M,N] = SIZE(X) returns the number of rows and columns in
separate output variables. [M1,M2,M3,...,MN] = SIZE(X)
returns the length of the first N dimensions of X.

M = SIZE(X,DIM) returns the length of the dimension specified
by the scalar DIM.  For example, SIZE(X,1) returns the number
of rows.
```

bzw. für `LENGTH`:

MATLAB HELP: `LENGTH`

```
Length of vector.

LENGTH(X) returns the length of vector X.  It is equivalent
to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.
```

2.4 Erzeugung von Matrizen

Arrays bzw. Matrizen können auf vielfältige Weise erzeugt werden:

- Explizite Eingabe (2.4.1).
- Erzeugung mit Hilfe der Doppelpunkt Notation (2.4.2).
- Erzeugung mit Hilfe eingebauter Funktionen (2.4.3).
- Laden von einem externen File (2.4.4).
- Selbst geschriebene Funktionen (M-files).

2.4.1 Explizite Eingabe

Die explizite Eingabe einer beliebigen Matrix (hier z.B. eines magisches Quadrats),

$$\begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

kann auf folgende Weise durchgeführt werden:

```
A = [16,3,2,13; 5,10,11,8; 9,6,7,12; 4,15,14,1]
```

wobei hier eine Zuweisung der Werte auf eine Variable mit dem Namen `A` erfolgt.
Man muss dabei folgende Regeln beachten:

- Die einzelnen Einträge innerhalb einer Zeile (row) werden durch Leerzeichen (blanks) oder bevorzugt durch Beistriche (commas) getrennt.
- Der Strichpunkt (semicolon) schließt eine Zeile ab.
- Die gesamte Liste der Einträge wird in eckige Klammern `[]` gestellt.

2.4.2 Doppelpunkt Notation

Die `Doppelpunktnotation` ist eine der mächtigsten Bestandteile von MATLAB. Sie kann einerseits zur Konstruktion von Vektoren (Tab. 2.2), aber auch zum Zugriff auf Teile von Matrizen (Index, 2.6) verwendet werden.

Tabelle 2.2: Doppelpunkt Notation zur Erzeugung von Vektoren

Op.	Alt.	Befehl	Resultat	Bedingung
`J:K`	`J:1:K`	`colon(J,K)`	`[J, J+1, ..., K]`	`K>=J`
`J:K`	`J:1:K`	`colon(J,K)`	`[]`	`K<J`
`J:D:K`		`colon(J,D,K)`	`[J, J+D, ..., J+m*D]`	`K>=J & D>0`
`J:D:K`		`colon(J,D,K)`	`[J, J+D, ..., J+m*D]`	`K<=J & D<0`
`J:D:K`		`colon(J,D,K)`	`[]`	`K<J & D>0`
`J:D:K`		`colon(J,D,K)`	`[]`	`K>J & D<0`
`J:D:K`		`colon(J,D,K)`	`[]`	`D=0`

Definition: `m = fix((K-J)/D)`, Umwandlung in ganze Zahlen durch Abschneiden.

Leere Arrays: Symbolisiert durch `[]`.

Logisches UND: Verwendetes Symbol &.

MATLAB Beispiel

Der Befehl `colon` bzw. der Operator `:`

Einige gültige und ungültige Beispiele für die Doppelpunkt Notation.

```
X = 1:5
    1     2     3     4     5

X = 1:2:5
    1     3     5

X = 1:-2:5
    Empty matrix: []

X = 5:-2:1
    5     3     1

X = 5:2:1
    Empty matrix: []
```

2.4.3 Interne Befehle zur Erzeugen von Matrizen

Es gibt eine Reihe von Befehlen zur einfachen Erzeugung von Matrizen.

Tabelle 2.3: MATLAB Befehle zum Erzeugen von Matrizen

`zeros(m)`	Erzeugt eine m×m Nullmatrix
`zeros(m,n)`	Erzeugt eine m×n Nullmatrix
`ones(m)`	Erzeugt eine m×m Matrix mit lauter Einsen
`ones(m,n)`	Erzeugt eine m×n Matrix mit lauter Einsen
`eye(m)`	Erzeugt eine m×m Einheitsmatrix
`eye(m,n)`	Erzeugt eine m×n Einheitsmatrix
`linspace(a,b,n)`	Erzeugt Zeilenvektor mit n äquidistanten Werten von a bis b.
`logspace(a,b,n)`	Erzeugt Zeilenvektor mit n Werten von 10^a bis 10^b mit logarithmisch äquidistantem Abstand.
`rand(m)`	Erzeugt eine m×m Zufallsmatrix (gleichverteilt aus [0,1])
`rand(m,n)`	Erzeugt eine m×n Zufallsmatrix (gleichverteilt aus [0,1])
`randn(m)`	Erzeugt eine m×m Zufallsmatrix (normalverteilt)
`randn(m,n)`	Erzeugt eine m×n Zufallsmatrix (normalverteilt)

Tabelle 2.4: Ergänzende MATLAB Befehle zum Erzeugen von Matrizen

`diag(v,k)`	v ... Vektor, k ... Skalar. Erzeugt eine Matrix mit lauter Nullen, außer auf der k-ten Nebendiagonale, die mit den Werten von v gefüllt wird. k = 0 ist die Hauptdiagonale, k>0 darüber, k<0 darunter. Für k=0 kann man auch diag(v) schreiben.
`diag(m,k)`	m ... Matrix. Extrahiert die k-te Nebendiagonale. (k siehe oben).
`triu(m)`	Extrahiert oberes Dreieck aus der Matrix m.
`triu(m,k)`	Extrahiert Dreieck oberhalb der Nebendiagonale k aus der Matrix m. (k siehe oben).
`tril(m)`	Extrahiert unteres Dreieck aus der Matrix m.
`tril(m,k)`	Extrahiert Dreieck unterhalb der Nebendiagonale k aus der Matrix m. (k siehe oben).
`blkdiag(a,b,...)`	Erzeugt eine blockdiagonale Matrix. a,b, ... sind Matrizen.
`repmat(a,m,n)`	Erzeugt aus einer Matrix a eine neue Matrix durch Replikation in Zeilenrichtung (m-mal) und Spaltenrichtung (n-mal).

Mit Hilfe des Befehls `[z,s]=meshgrid(v1,v2)` ist es sehr leicht zwei gleich große Matrizen zu erzeugen. Sind die beiden Vektoren `v1` und `v2` z.B. die Vektoren `1:n` und `1:m`, dann ergeben sich folgende Matrizen:

$$z = \begin{bmatrix} 1 & 2 & 3 & \dots & n \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \dots & n \end{bmatrix}, \; s = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 2 & 2 & 2 & \dots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m & m & m & \dots & m \end{bmatrix}. \tag{2.1}$$

Die Variablen `m` und `n` müssen dabei vorher definiert werden. Analog kann das natürlich mit allen anderen Vektoren ausgeführt werden. Die so erhaltenen Matrizen eignen sich bestens zum Kombinieren.

Mit dem Befehl `v = z + 100*s` erhält man sofort folgende Matrix:

$$v = \begin{bmatrix} 101 & 102 & 103 & 104 & 105 & \dots \\ 201 & 202 & 203 & 204 & 205 & \dots \\ 301 & 302 & 303 & 304 & 305 & \dots \\ 401 & 402 & 403 & 404 & 405 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{2.2}$$

2.4.4 Lesen und Schreiben von Daten

Neben komplexen Befehlen zum Schreiben und Lesen von Daten und dem Umgang mit externen Datenfiles, gibt es zum Lesen geordneter Strukturen den einfachen Befehl `load`. Er funktioniert nur, wenn die Daten in Tabellenform ohne fehlende Einträge oder Kommentarzeilen gespeichert sind.

Die Form des Aufrufs ist `D = load('d.dat')`, wobei hier `'d.dat'` für eine Zeichenkette mit dem Filenamen steht. Das Gegenstück zum Speichern von lesbaren Daten ist `save`. Dieser Befehl wird in folgender Form verwendet: `save('d.dat','D','-ascii')`

Eine detailierte Beschreibung von Schreibe- und Leseroutinen folgt in einem späteren Kapitel.

2.5 Veränderung und Auswertung von Matrizen

Viele Befehle haben als Inputparameter eine Matrix und liefern eine (im Allgemeinen nicht unbedingt gleich große) Matrix zurück. (Zur Erinnerung: Spalten- bzw. Zeilenvektoren werden ebenfalls als Matrizen angesehen).

Beispiele dafür sind das Bilden von Summen oder Produkten, oder das Transponieren und Konjugieren. Im Folgenden wurden dafür einige einfache Beispiele zusammengestellt.

Der numerische Inhalt von Matrizen muss nicht nur aus reellen Zahlen bestehen, sondern kann auch komplexe Werte enthalten. Dafür ist keine spezielle Deklaration notwendig, MATLAB führt diese automatisch beim ersten Auftreten von komplexen Elementen in einer Matrix durch.

Die Variablen `i` oder auch `j` werden als imaginäre Einheit $i = \sqrt{-1}$ verwendet, und sollen daher sonst nicht verwendet werden. MATLAB hat keinen effektiven Schutz vor dem Überschreiben von wichtigen Variablen. Die beiden Befehle `i=1` und `j=1` legen die Fähigkeit von MATLAB lahm, mit komplexen Zahlen zu rechnen.

MATLAB Beispiel

Einige Befehle stehen in MATLAB zur Verfügung, um Matrizen zu kippen bzw. zu drehen. Außerdem gibt es noch `FLIPDIM(X,DIM)`, für Kippen entlang der Dimension DIM.

`FLIPLR` Flip matrix in left/right direction.
`FLIPLR(X)` returns X with row preserved and columns flipped in the left/right direction.

`FLIPUD` Flip matrix in up/down direction.
`FLIPUD(X)` returns X with columns preserved and rows flipped in the up/down direction.

`ROT90` Rotate matrix 90 degrees.
`ROT90(X)` is the 90 degree counterclockwise rotation of matrix X. `ROT90(X,K)` is the K×90 degree rotation of X, K = ±1, ±2, ...

```
X = [1 2 3; 4 5 6]
     1     2     3
     4     5     6

Y=fliplr(X)
     3     2     1
     6     5     4

Y=flipud(X)
     4     5     6
     1     2     3

Y=rot90(X)
     3     6
     2     5
     1     4
```

MATLAB Beispiel

Drei Befehle stehen in MATLAB zur Verfügung, um transponierte, konjugiert komplex transponierte oder konjugiert komplexe Matrizen zu berechnen.

TRANSPOSE is the non-conjugate transpose.

Operator form: X.' is the transpose of X.

CTRANSPOSE is the complex conjugate transpose.

Operator form: X' is the complex conjugate transpose of X.

CONJ is the complex conjugate of X.
For a complex X,
CONJ(X) = REAL(X) - i*IMAG(X).

```
X = [1+i 2+i 3+i; 4+i 5+i 6+i]
      1 + i   2 + i   3 + i
      4 + i   5 + i   6 + i

Y=transpose(X)
      1 + i   4 + i
      2 + i   5 + i
      3 + i   6 + i

Y=ctranspose(X)
      1 - i   4 - i
      2 - i   5 - i
      3 - i   6 - i

Y=conj(X)
      1 - i   2 - i   3 - i
      4 - i   5 - i   6 - i
```

MATLAB Beispiel

Summation und kummulative Summation in Matrizen.

`SUM` Sum of elements.
For vectors, `SUM(X)` is the sum of the elements of X. For matrices, `SUM(X)` is a row vector with the sum over each column. For N-D arrays, `SUM(X)` operates along the first non-singleton dimension.

`SUM(X,DIM)` sums along the dimension DIM.

`CUMSUM` Cumulative sum of elements. For vectors, `CUMSUM(X)` is a vector containing the cumulative sum of the elements of X. For matrices, `CUMSUM(X)` is a matrix the same size as X containing the cumulative sums over each column. For N-D arrays, `CUMSUM(X)` operates along the first non-singleton dimension.

`CUMSUM(X,DIM)` works along the dimension DIM.

The first non-singleton dimension is the first dimension which size is greater than one.

```
X = [0 1 2; 3 4 5]
     0     1     2
     3     4     5

Y=sum(X)
     3     5     7

Y=sum(X,2)
     3
    12

Y=cumsum(X)
     0     1     2
     3     5     7

Y=cumsum(X,2)
     0     1     3
     3     7    12
```

MATLAB Beispiel

Multiplikation und kummulative Multiplikation in Matrizen.

`PROD` Product of elements.
For vectors, `PROD(X)` is the product of the elements of X. For matrices, `PROD(X)` is a row vector with the product over each column. For N-D arrays, `PROD(X)` operates along the first non-singleton dimension.

`PROD(X,DIM)` works along the dimension DIM.

`CUMPROD` Cumulative product of elements. For vectors, `CUMPROD(X)` is a vector containing the cumulative product of the elements of X. For matrices, `CUMPROD(X)` is a matrix the same size as X containing the cumulative product over each column. For N-D arrays, `CUMPROD(X)` operates along the first non-singleton dimension.
`CUMPROD(X,DIM)` works along the dimension DIM.

```
X = [0 1 2; 3 4 5]
     0     1     2
     3     4     5

Y=prod(X)
     0     4    10

Y=prod(X,2)
     0
    60

Y=cumprod(X)
     0     1     2
     0     4    10

Y=cumprod(X,2)
     0     0     0
     3    12    60
```

Alle Befehle in Matlab, bei denen die Richtung innerhalb der Matrix von Bedeutung ist, wie z.B. der Befehl `sum`, folgen folgenden Regeln:

1. Ist eine Richtung vorgegeben, `sum(X,2)`, erfolgt die Operation in Richtung dieser Dimension.
2. Ist keine Richtung vorgegeben, erfolgt die Summation in Richtung der ersten Dimension, die ungleich eins ist (non-singleton dimension). Das heißt, dass sowohl in einem Spaltenvektor (`size(X)` z.B. `[3 1]`), als auch in einem Zeilenvektor (`size(X)` z.B. `[1 3]`) über alle Elemente summiert wird.

Befehle können in MATLAB beliebig geschachtelt werden, solange die Syntax für jeden einzelnen Befehl korrekt ist. So kann man z.B. die Summe über die Diagonale bzw. die zweite Diagonale (links unten bis rechts oben) einer Matrix mit folgenden Befehlen berechnen:

Summe der Diagonalelemente der Matrix X:

```
S_D = sum(diag(X))
```

Summe der Elemente in der zweiten Diagonale der Matrix X:

```
S_ND = sum(diag(fliplr(X)))
```

Tabelle 2.5: Indizierung von Arrays

Index	Alternative	Zeilen	Spalten	Resultat
INDIZIERUNG MIT ZWEI INDICES				
`X(J,M)`		`J`	`M`	Skalar
`X(J,:)`	`X(J,1:end)`	`J`	`ALLE`	Zeilenvektor
`X(:,M)`	`X(1:end,M)`	`ALLE`	`M`	Spaltenvektor
`X(:,:)`	`X(1:end,1:end)`	`ALLE`	`ALLE`	2-D Array
`X(J:K,M)`		`J:K`	`M`	Spaltenvektor
`X(J:D:K,M)`		`J:D:K`	`M`	Spaltenvektor
`X(J:K,M:N)`		`J:K`	`M:N`	2-D Array
INDIZIERUNG MIT EINEM INDEX (LINEAR)				
`X(:)`		`ALLE`	`ALLE`	Spaltenvektor
`X(I)`		`JI`	`MI`	Skalar
`X(I:H)`		`JI:JH`	`MI:MH`	Zeilenvektor

Die große Vielzahl von verfügbaren Befehlen und die Möglichkeit der Schachtelung führt dazu, dass sehr mächtige Programme in sehr kompakter Form geschrieben werden können.

2.6 Zugriff auf Teile von Matrizen, Indizierung

Sehr häufig ist es wichtig, auf bestimmte Teile einer Matrix in Abhängigkeit von ihrer Position in der Matrix zuzugreifen. Dazu braucht man die sogenannte Indizierung, die hier am Beispiel einer 2-dim Matrix erläutert werden soll. Bei höher dimensionalen Matrizen ist das Konzept analog anzuwenden.

In MATLAB bezieht sich der Befehl `A(i,j)` auf das Element a_{ij} der Matrix A. Diese Bezeichnung ist praktisch in allen Programmiersprachen üblich. MATLAB bietet jedoch einen viel weitergehenden Aspekt der Indizierung, der es auf einfache Weise erlaubt auf bestimmte Regionen innerhalb einer Matrix zuzugreifen. Diese Eigenschaft macht die Matrix Manipulation einfacher als in vielen anderen Programmiersprachen. Außerdem bietet es eine einfache Möglichkeit die "vektorisierte" Natur von Berechnungen in MATLAB zu benutzen.

Die meisten Programme werden dadurch viel lesbarer und übersichtlicher, da man sich eine große Anzahl von Schleifen (und damit auch eine große Anzahl von Fehlerquellen) sparen kann.

In der Folge wird nun auf die verschiedenen Möglichkeiten der Indizierung eingegangen. In Tabelle 2.5 werden die einzelnen Regeln erläutert, und in 2.6 die Zuweisung von Werten gezeigt, und in 2.7 der Zugriff auf bestimmte Regionen gezeigt.

Tabelle 2.6: Zuweisung von Werten an bestimmten Positionen eines Arrays

`X`	`X(3,2)=1`	`X(:,2)=1`
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 1 0 0	0 1 0 0 0 1 0 0 0 1 0 0
`X(2,:)=1`	`X(:,:)=1`	`X(:)=1`
0 0 0 0 1 1 1 1 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1
`X(:,1:2:4)=1`	`X(1:2:3,:)=1`	`X(1:2:3,1:2:4)=1`
1 0 1 0 1 0 1 0 1 0 1 0	1 1 1 1 0 0 0 0 1 1 1 1	1 0 1 0 0 0 0 0 1 0 1 0
`X(7:10)=1`	`X(1:2,3)=1`	`X(2,1:3)=1`
0 0 1 1 0 0 1 0 0 0 1 0	0 0 1 0 0 0 1 0 0 0 0 0	0 0 0 0 1 1 1 0 0 0 0 0

Die Umrechnung zwischen dem linearen Index und mehrfachen Indices erfolgt mit den Befehlen `ind2sub` und `sub2ind`:

Mehrfacher Index von linearem Index: `[JI,MI] = ind2sub(size(X),I)`

Linearer Index von mehrfachem Index: `[I] = sub2ind(size(X),JI,MI)`

In beiden Befehlen muss natürlich die Größe, `size(X)`, angegeben werden, da nur mit diesem Wissen der Zusammenhang zwischen den Indices eineindeutig ist. Wie bei dem Befehl `sum` folgt der lineare Index zuerst der ersten, dann der zweiten, dann der nächsten Dimension. Der Zusammenhang sollte aus folgender Darstellung klar werden,

$$\begin{bmatrix} (1,1) & (1,2) & (1,3) & (1,4) \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & (3,2) & (3,3) & (3,4) \end{bmatrix} \equiv \begin{bmatrix} (1) & (4) & (7) & (10) \\ (2) & (5) & (8) & (11) \\ (3) & (6) & (9) & (12) \end{bmatrix} . \tag{2.3}$$

Da mit Hilfe der Doppelpunkt Notation ja eigentlich Vektoren als Indices erzeugt werden (2.4.2), ist natürlich auch folgende Schreibweise erlaubt:

Tabelle 2.7: Zugriff auf bestimmte Positionen eines Arrays

`X`	`X(3,2)`	`X(:,2)`
$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$	10	$\begin{matrix} 2 \\ 6 \\ 10 \end{matrix}$
`X(2,:)`	`X(:,:)`	`X(:)`
$\begin{matrix} 5 & 6 & 7 & 8 \end{matrix}$	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$	$\begin{matrix} 1 \\ 5 \\ \vdots \\ 8 \\ 12 \end{matrix}$
`X(:,1:2:4)`	`X(1:2:3,:)`	`X(1:2:3,1:2:4)`
$\begin{matrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{matrix}$	$\begin{matrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{matrix}$	$\begin{matrix} 1 & 3 \\ 9 & 11 \end{matrix}$
`X(7:10)`	`X(1:2,3)`	`X(2,1:3)`
$\begin{matrix} 3 & 7 & 11 & 4 \end{matrix}$	$\begin{matrix} 3 \\ 7 \end{matrix}$	$\begin{matrix} 5 & 6 & 7 \end{matrix}$

- `X([1 2],[2 3])` äquivalent zu `X(1:2,2:3)`
- `X([1 3],[2 4])` äquivalent zu `X(1:2:3,2:2:4)`

Eine wichtige Rolle spielt auch das Keyword `end`, das im richtigen Kontext die entsprechende Größe angibt. Damit ist es nicht notwendig bei der Indizierung die Größe der Arrays zu kennen:

- `X(1:2:end,3)` für die dritte Spalte jeder 2.ten Zeile.
- `X(2:end-1,2:end-1)` für die 2.te bis vorletzte Zeile bzw. Spalte.

2.6.1 Logische Indizierung

In Ergänzung zur normalen Indizierung erlaubt MATLAB auch die sogenannte logische Indizierung mit Arrays die nur die Werte 1 (entspricht TRUE) bzw. 0 (entspricht FALSE) enthalten. Dadurch ist auch der Zugriff auf völlig ungeordnete Bereiche möglich (Tab. 2.8).

Wichtig dabei ist Folgendes:

- Das Array `L` muss die gleiche Größe wie das Array `X` haben.
- Das Array `L` muss ein logisches Array sein, das entstanden ist durch
  - logische Operationen (`and`, `or`, `xor`, `not`),
  - Vergleichsoperationen (z.B.: `<`),
  - durch Verwendung des Befehls `logical(Y)`, wodurch ein numerisches Array in ein logisches umgewandelt wird.
- Ein logisches Array darf nicht nur die Werte 0 und 1 beinhalten, MATLAB folgt der Konvention, dass alle Zahlen die ungleich 0 sind als TRUE gelten.
- Wegen der möglicherweise ungeordneten Anordung der Zielelemente in der Matrix, geht die Form verloren. Das Ergebnis liegt immer in Form eines Spaltenvektors vor, außer beide Matrizen sind ein Zeilenvektor, dann bleibt ein Zeilenvektor erhalten.
- Der Verlust der Form spielt natürlich bei einer Zuweisung von Werten auf diese Positionen keine Rolle, die Form der Matrix bleibt dabei erhalten.
- Bei jeder Zuweisung muss entweder die Anzahl der Werte gleich sein wie die Anzahl der ausgewählten Positionen, oder ein Skalar wird auf eine beliebige Anzahl von Positionen zugewiesen.

Tabelle 2.8: Zugriff mit Hilfe logischer Indizierung

`X`	`L`	`X(L)`
1 2 3 4 5 6 7 8 9 10 11 12	0 0 1 0 1 0 0 0 0 0 0 1	5 3 12
`X`	`L`	`X(L)=0`
1 2 3 4 5 6 7 8 9 10 11 12	0 0 1 0 1 0 0 0 0 0 0 1	1 2 0 4 0 6 7 8 9 10 11 0
`X`	`L`	`X(L)`
1 2 3 4 5 6 7 8 9 10 11 12	1 0 0 0 0 1 0 0 0 0 1 0	1 6 11
`X`	`L`	`X(L)=0`
1 2 3 4 5 6 7 8 9 10 11 12	1 0 0 0 0 1 0 0 0 0 1 0	0 2 3 4 5 0 7 8 9 10 0 12

- Ist man nur an den Positionionen interessiert, kann man mit `I = find(L)` die linearen Indices, bzw. mit `[m,n] = find(L)` die 2-dim Indices erhalten.
- Details über Vergleichsoperatoren und logische Operatoren finden sich in den Abschnitten 3.3 und 3.2.

2.7 Zusammenfügen von Matrizen

Für das Zusammenfügen von Matrizen zu einer Einheit stehen die Befehle `cat`, `vertcat` (untereinander) und `horzcat` (nebeneinander) zur Verfügung. Der Befehl `cat(DIM,A,B)` fügt die beiden Matrizen entlang der Dimenssion `DIM` zusammen. Alle anderen Dimension müssen natürlich übereinstimmen.

BEFEHL	ALTERNATIVE	KURZFORM

`cat(1,A,B)`
`cat(2,A,B)`
`cat(3,A,B)`
	`vertcat(A,B)`
	`horzcat(A,B)`
		`[A;B]`
		`[A,B]`

`cat(1,A,B,C,...)` `cat(2,A,B,C,...)`	`vertcat(A,B,C,...)` `horzcat(A,B,C,...)`	`[A;B;C;...]` `[A,B,C,...]`

2.8 Initialisieren, Löschen und Erweitern

Eine Initialisierung bzw. Deklaration von Matrizen in MATLAB ist nicht unbedingt notwendig. Bei Matrizen kann jederzeit ihr Inhalt, ihre Größe oder ihr Typ verändert werden. Trotzdem ist es meist sinnvoll, Matrizen mit Typ und Größe zu initialisieren, wie sie später benötigt werden.

Vor allem bei großen Matrizen und bei sogenannten dynamischen Matrizen, dass sind solche, deren Inhalt sich in Schleifen dauert ändert, ist dies ein wichtiger Schritt. Beim Initialisieren wird ein kontinuierlicher Bereich im Computerspeicher angelegt (alloziert), auf den rasch zugegriffen werden kann. Ändert sich der Typ oder die Größe muss neu alloziert werden, was jedesmal Zeit kostet.

Zum Initialisieren bietet sich der Befehl `zeros(m,n)` an. Benötigt man eine Matrix, die gleich groß wie eine bestehende Matrix `X` sein soll, kann man den Befehl auch so `zeros(size(X))` schreiben.

2.9 Umformen von Matrizen

Zum Umformen von Matrizen steht im Wesentlichen der Befehl `reshape` zur Verfügung.

Der Befehl `Y=reshape(X,SIZ)` liefert ein Array mit den gleichen Werten aber der Größe `SIZ`. Natürlich muss `prod(SIZ)` mit `prod(SIZE(X))` übereinstimmen (gleiche Anzahl von Elementen), sonst meldet MATLAB einen Fehler.

Der Befehl `reshape` kann auf zwei verschiedene Weisen geschrieben werden:

- `reshape(X, M, N, P, ...)`
- `reshape(X, [M N P ...])`

Die zweite Form eignet sich bestens um einen Vektor einzusetzen, der automatisch z.B. mit `size` erhalten wurde.