Kapitel 4

Operatoren

4.1 Arithmetische Operatoren

4.1.1 Arithmetische Operatoren für Skalare

Die vordefinierten Operatoren auf skalaren double-Ausdrücken sind in Tabelle 4.1 zusammengefaßt. Diese Operatoren sind eigentlich Matrixoperatoren, deren genaue Behandlung in Kapitel 6 folgt. Der Grund dafür liegt darin, dass skalare Größen auch als Matrizen mit nur einem Element aufgefasst weren können. Damit bleibt hier die übliche Notation mit `*` und `/` erhalten.

Operatoren haben Prioritäten, die die Abarbeitung bestimmen. Operationen mit höherer Priorität werden zuerst ausgeführt.

Die Reihenfolge der Auswertung eines Ausdrucks kann durch Klammerung beeinflußt werden. In Klammern eingeschlossene (Teil-) Ausdrücke haben die höchste Priorität, d.h., sie werden auf jeden Fall zuerst ausgewertet. Bei verschachtelten Klammern werden die Ausdrücke im jeweils innersten Klammerpaar zuerst berechnet. Zur Klammerung verwendet MATLAB die sogenannten runden Klammern `()`.

Kommen in einem Ausdruck mehrere aufeinanderfolgende Verknüpfungen durch Operatoren mit gleicher Priorität vor, so werden sie von links nach rechts abgearbeitet, sofern nicht Klammern vorhanden sind, die etwas anderes vorschreiben; dies ist vor allem dann zu beachten, wenn nicht-assoziative Operatoren gleicher Priorität hintereinander folgen.

Operatoren können nur auf bereits definierte Variablen angewandt werden. Sie liegen immer in Form von Operatoren (`+`) oder in Form von Befehlen (`plus`) vor.

Tabelle 4.1: Skalare Operationen; `a` und `b` sind skalare Variablen

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	MATH	PRIORITÄT
`^`	`a^b`	`mpower(a,b)`	Exponentiation	a^b	4
`+`	`+a`	`uplus(a)`	Unitäres Plus	$+a$	3
`-`	`-a`	`uminus(a)`	Negation	$-a$	3
`*`	`a*b`	`mtimes(a,b)`	Multiplikation	ab	2
`/`	`a/b`	`mrdivide(a,b)`	Division	a/b	2
`\`	`a\b`	`mldivide(a,b)`	Linksdivision	b/a	2
`+`	`a+b`	`plus(a,b)`	Addition	$a+b$	1
`-`	`a-b`	`minus(a,b)`	Subtraktion	$a-b$	1

4.1.2 Arithmetische Operatoren für Arrays

Eine herausragende Eigenschaft von MATLAB ist die einfache Möglichkeit der Verarbeitung ganzer Felder durch eine einzige Anweisung. Ähnlich wie in modernen Programmiersprachen Operatoren überladen werden können, lassen sich die meisten Operatoren und vordefinierten Funktionen in MATLAB ohne Notationsunterschied auf (ein- oder mehrdimensionale) Felder anwenden. Tabelle 4.2 enthält die vordefinierten Operatoren für Arrays am Beispiel von Zeilenvektoren. Die Anwendung auf mehrdimensionale Felder erfolgt analog.

Die hier vorgestellten Operatoren, die mit einem Punkt beginnen, werden komponentenweise auf Felder übertragen. Andere Operatoren haben unter Umständen bei Feldern eine andere Bedeutung.

Bei Anwendung auf Skalare haben sie natürlich die gleiche Bedeutung wie die Operatoren in 4.1. In diesem Fall ist also das Resultat von z.B. $\star$ und $.\star$ das selbe, da Skalare Matrizen mit einem Element sind. Bei $+$ und $-$ erübrigt sich eine Unterscheidung der Bedeutung überhaupt, was zur Folge hat, dass es keine $.+$ und $.-$ Operatoren gibt.

Durch den Einsatz von Vektoroperatoren kann auf die Verwendung von Schleifen (wie sie etwa in C oder FORTRAN notwendig wären) sehr oft verzichtet werden, was die Lesbarkeit von MATLAB-Programmen fördert.

In MATLAB werden die gleichen Operatoren verwendet, um Vektoren oder allgemein Arrays mit Skalarausdrücken zu verknüpfen. In Tabelle 4.3 findet man die vordefinierten Operatoren zur komponentenweisen Verknüpfung von Feldern und Skalaren.

In 4.3 kommen in einigen wenigen Fällen die Array-Operatoren mit Punkten und die Matrix-Operatoren gleichwertig vor, da sie zum selben Ergebnis führen. Eine genaue Behandlung der Matrix-Operatoren im Sinne der linearen Algebra erfolgt in Kapitel 6 .

Tabelle 4.2: Array-Array Operationen; `a` und `b` sind Felder der gleichen Größe, in diesem Beispiel Zeilenvektoren der Länge `n`.

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	PRIO. PRIO.
`.^`	`a.^b`	`power(a,b)`	$[a_1^{b_1}\ a_2^{b_2}\ \ldots\ a_n^{b_n}]$	4
`.*`	`a.*b`	`times(a,b)`	$[a_1b_1\ a_2b_2\ \ldots\ a_nb_n]$	2
`./`	`a./b`	`rdivide(a,b)`	$[a_1/b_1\ a_2/b_2\ \ldots\ a_n/b_n]$	2
`.\`	`a.\b`	`ldivide(a,b)`	$[b_1/a_1\ b_2/a_2\ \ldots\ b_n/a_n]$	2
`+`	`a+b`	`plus(a,b)`	$[a_1+b_1\ a_2+b_2\ \ldots\ a_n+b_n]$	1
`-`	`a-b`	`minus(a,b)`	$[a_1-b_1\ a_2-b_2\ \ldots\ a_n-b_n]$	1

Tabelle 4.3: Skalar-Array Operationen; `a` ist in diesem Beispiel ein Zeilenvektor der Länge `n` und `c` ist ein Skalar.

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	PRIO. PRIO.
`.^`	`a.^c`	`power(a,c)`	$[a_1^c\ a_2^c\ \ldots\ a_n^c]$	4
`.^`	`c.^a`	`power(c,a)`	$[c^{a_1}\ c^{a_2}\ \ldots\ c^{a_n}]$	4
`.*`	`a.*c`	`times(a,c)`	$[a_1c\ a_2c\ \ldots\ a_nc]$	2
`./`	`a./c`	`rdivide(a,c)`	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
`./`	`c./a`	`rdivide(c,a)`	$[c/a_1\ c/a_2\ \ldots\ c/a_n]$	2
`.\`	`a.\c`	`ldivide(a,c)`	$[c/a_1\ c/a_2\ \ldots\ c/a_n]$	2
`.\`	`c.\a`	`ldivide(c,a)`	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
`+`	`a+c`	`plus(a,c)`	$[a_1+c\ a_2+c\ \ldots\ a_n+c]$	1
`-`	`a-c`	`minus(a,c)`	$[a_1-c\ a_2-c\ \ldots\ a_n-c]$	1
`*`	`a*c`	`mtimes(a,c)`	$[a_1c\ a_2c\ \ldots\ a_nc]$	2
`/`	`a/c`	`mrdivide(a,c)`	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
`\`	`c\a`	`mldivide(c,a)`	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2

Tabelle 4.4: Vergleichsoperatoren

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	MATH
<	a<b	lt(a,b)	kleiner als	$a < b$
<=	a<=b	le(a,b)	kleiner oder gleich	$a \leq b$
>	a>b	gt(a,b)	größer als	$a > b$
>=	a>=b	ge(a,b)	größer oder gleich	$a \geq b$
==	a==b	eq(a,b)	gleich	$a = b$
~=	a~=b	ne(a,b)	ungleich	$a \neq b$

4.2 Vergleichsoperatoren

Vergleichsoperatoren sind <, <=, >, >=, ==, and ~=. Mit ihnen wird ein Element-für-Element Vergleich zwischen zwei Feldern durchgeführt. Beide Felder müssen gleich groß sein. Als Antwort erhält man ein Feld gleicher Größe, mit dem jeweiligen Element auf logisch TRUE (1) gesetzt, wenn der Vergleich richtig ist, oder auf logisch FALSE (0) gesetzt wenn der Vergleich falsch ist.

Die Operatoren <, <=, > und >= verwenden nur den Realteil ihrer Operanden, wohingegen die Operatoren == und ~= den Real- und den Imaginärteil verwenden.

Wenn einer der Operanden ein Skalar ist und der andere eine Matrix, dann wird der Skalar auf die Größe der Matrix expandiert. Die beiden folgenden Beispiele geben daher das gleiche Resultat.

```
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]

ans =
     1     1     1
     1     1     0
     0     0     0
```

4.3 Logische Operatoren

Die Symbole &, |, and ~ stehen für die logischen Operatoren and, or, and not. Sind die Operanden Felder, wirken alle Befehle elementweise. Der Wert 0 representiert das logische FALSE (F), und alles was nicht Null ist, representiert das logische TRUE (T). Die Funktion xor(A,B) implementiert das "exklusive oder". Die Wahrheitstabellen für diese Funktionen sind in 4.5 zusammengestellt.

Tabelle 4.5: Logische Operatoren

INPUT		and	or	xor	not
A	B	A&B	A|B	xor(A,B)	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Wenn einer der Operanden ein Skalar ist und der andere eine Matrix, dann wird der Skalar auf die Größe des Feldes expandiert. Die logischen Operatoren verhalten sich dabei gleich wie die Vergleichsoperatoren. Das Ergebnis der Operation ist wieder ein Feld der gleichen Größe.

Die Priorität der logischen Operatoren ist folgendermaßen geregelt:

- **not** hat die höchste Priorität.
- **and** und **or** haben die gleiche Priorität und werden von links nach rechts abgearbeitet.

Die "Links vor Rechts" Ausführungspriorität in MATLAB macht `a|b&c` zum Gleichen wie `(a|b)&c`. In den meisten Programmiersprachen ist `a|b&c` jedoch das Gleiche wie `a|(b&c)`. Dort hat `&` eine höhere Priorität als `|`. Es ist daher in jedem Fall gut, mit Klammern die notwendige Abfolge zu regeln.

Eine Besonderheit stellen die beiden logischen Operatoren `&&` und `||`, die man als logische Operatoren mit short circuit bezeichnet. Dabei wird z.B. beim Befehl

```
x = (b ~= 0) && (a/b > 18.5)
```

der zweite Teil mit der Divison nicht mehr ausgeführt, wenn `b` gleich `0` ist. Dies ist nämlich nicht mehr notwendig, da das Ergebnis unabhängig vom zweiten Teil das Resultat `FALSE` liefern muss. Man spart sich damit in diesem Fall die Warnung, dass durch Null dividiert wird.

Besonders praktisch ist dieses Feature, wenn eine Variable zum Zeitpunkt der Berechnung nicht existiert. Wenn also `a` nicht als Variable existiert (exist), liefert der Befehl

```
~exist('a','var') | isempty(a)
```

die Fehlermitteilung `Undefined function or variable 'a'`, da der zweite Befehl (isempty) nicht ausgeführt werden kann. Verwendet man hingegen

```
~exist('a','var') || isempty(a)
```

kann die Zeile auch in diesem Fall ausgewertet werden und liefert den Wert `1` (`TRUE`), wenn `a` nicht existiert oder ein leeres Feld ist.

Die Verwendung von `&&` und `||` stellt also sicher, dass jene Teile des logischen Konstrukts nicht mehr ausgeführt werden, wenn sie das Ergebnis nicht mehr beeinflussen können.

Bei der Verwendung von Vergleichsoperatoren und logischen Operatoren in Steuerkonstrukten, wie z.B. `if-Strukturen`, ist zur Entscheidung natürlich nur ein skalarer logischer Wert möglich. Einen solchen kann man aus logischen Arrays durch die Befehle:

`any(M)` **oder** `any(M,DIM)`**:** Ist `TRUE`, wenn ein Element ungleich Null ist.

`all(M)` **oder** `all(M,DIM)`**:** Ist `TRUE`, wenn alle Elemente ungleich Null sind.

> Wenn die Befehle `any(M)` und `all(M)` auf Felder angewandt werden, verhalten sie sich analog zu anderen Befehlen (wie z.B. `sum(M)`) und führen die Operation entlang der ersten von Eins verschiedenen Dimension aus. Das Ergebnis ist dann in der Regel kein Skalar.

Die Ergebnisse von Vergleichsoperationen und logischen Operarationen können für die logische Indizierung, Kapitel 3 , verwendet werden.

Ist man nur an den Positionen interessiert, kann man mit `I = find(L)` die linearen Indices, bzw. mit `[m,n] = find(L)` die 2-dim Indices erhalten, für die die Bedingung in `L` erfüllt ist.

Beispiel mit `find`, `ind2sub` und `sub2ind`:

```
m = reshape([1:12],3,4);        m = [ 1     4     7    10
                                      2     5     8    11
                                      3     6     9    12 ]

l = m>3 & m<8;                  l = [ 0     1     1     0
                                      0     1     0     0
                                      0     1     0     0 ]

i = find(l);                     i = [ 4;    5;    6;    7 ]

[si,sj] = find(l);              si = [ 1;    2;    3;    1 ]
                                sj = [ 2;    2;    2;    3 ]

Umrechnung: [si,sj] = ind2sub(size(m),i);
                  i = sub2ind(size(m),si,sj);
```

Beispiel mit `any` und `all`:

```
m = reshape([1:12],3,4);
```

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

```
l = m>=2 & m<=11;
```

```
l = [ 0     1     1     1
      1     1     1     1
      1     1     1     0 ]
```

```
an1 = any(l)
al1 = all(l);
```

```
an1 = [ 1     1     1     1 ]
al1 = [ 0     1     1     0 ]
```

```
an2 = any(l,2); al2 = all(l,2);
```

```
an2 = [ 1          al2 = [ 0
        1                  1
        1 ]                0 ]
```

```
an = any(l(:));
al = all(l(:));
```

```
an = 1
al = 0
```