

# Typische Stolperfallen beim Programmieren in MATLAB und wie man sie vermeidet

Von Loren Shure

Genau wie in natürlichen Sprachen gibt es auch in Programmiersprachen Idiome. Das Beherrschen dieser syntaktischen oder inhaltlichen Nuancen ist wesentlich, um anvisierte Programmierziele zu erreichen – ganz gleich, ob ein Programm robust sein oder einfach die richtige Antwort geben soll. Hier sind einige Tipps und Best Practices zu sprachlichen Feinheiten in MATLAB®.

## DIE ERWARTETE ANTWORT ERHALTEN

Computerprogramme machen genau das, was ihr Programmcode ihnen vorschreibt. Eine exakte Notation ist darum entscheidend. Hier einige Beispiele dafür.

### Matrizen- und elementweise Multiplikation

Angenommen, Sie wollen die korrespondierenden Elemente zweier großer, quadratischer Matrizen **A** und **B** miteinander multiplizieren. Mit **A\*B** erhalten Sie zwar ein Ergebnis der erwarteten Dimensionen (denen von **A**), aber die falschen Zahlen. Denn statt die korrespondierenden Elemente in **A** und **B** zu multiplizieren, haben Sie gerade die Matrizen multipliziert. Um nur die korrespondierenden Elemente zu multiplizieren müssen Sie **A.\*B** eingeben:

```
>> A = [1 2; 3 4]; B = [1 0; -1 1];
>> C = A.*B
C =
     1     0
    -3     4
>> D = A*B
D =
    -1     2
    -1     4
```

### Transponierte und konjugiert Transponierte

In MATLAB kann man mit reell- und komplexwertigen Arrays mit doppelter Genauigkeit arbeiten, ohne diese gesondert behandeln zu müssen. Trotzdem muss der Code dafür richtig formuliert sein.

Für reelle Matrizen ist das Ergebnis von **A'** und **A. '** identisch. Für Matrizenoperationen an komplexwertigen Arrays ist der Transponierungs-Operator (**'**) oft der richtige. Aus komplexen

Matrizen dagegen erzeugt er die konjugiert Transponierte. Um diese Konjugation zu vermeiden, verwendet man den nicht-konjugierten Transponierungs-Operator (**. '**):

```
>> C= A+i*B
C =
     1.0000 + 1.0000i     2.0000
     3.0000 - 1.0000i     4.0000 + 1.0000i
>> E = C'
E =
     1.0000 - 1.0000i     3.0000 + 1.0000i
     2.0000                4.0000 - 1.0000i
>> F = C.'
F =
     1.0000 + 1.0000i     3.0000 - 1.0000i
     2.0000                4.0000 + 1.0000i
```

### Imaginäre Einheit **i** oder **j**

MATLAB definiert die Konstanten **i** und **j** beide als `sqrt(-1)`, die Imaginäre Einheit. Wenn Sie aber **i** oder **j** als Variablen in Ihrem Code verwenden und der nachfolgende Code ebenfalls eine dieser beiden Konstanten nutzt, kann dies zu unerwarteten Fehlern führen.

Um dieses Problem zu vermeiden, verwenden Sie andere Variablenamen oder löschen Sie die Variablen nach Gebrauch:

```
>> for i=1:3
    M{i} = magic(i+2)
end
M =
    [3x3 double]
M =
    [3x3 double]    [4x4 double]
M =
    [3x3 double]    [4x4 double]    [5x5 double]
>> clear i
```

### Befehls / Funktions-Dualität

Die korrekte Zeichensetzung kann in MATLAB wichtig sein, besonders in Klammersausdrücken. Die folgenden MATLAB-Anweisungen sind äquivalent:

```
foo baz 5
foo('baz', '5')
```

Durch Weglassen der Klammern beim Aufruf einer Funktion ruft man die Funktion als Befehl auf. MATLAB behandelt dann alle Argumente nach dem Funktionsnamen als Strings. Um das Ergebnis eines Funktionsaufrufs einer Variablen zuzuweisen, verwendet man daher die Funktionsform, nicht die Befehlsform.

### Vergleich von Fließkommazahlen

Der Standarddatentyp in MATLAB sind Fließkomma-Doubles. Da der Double-Datentyp Zahlen mit einer endlichen Zahl von Bits (64) darstellt (davon 53 Bits für die Mantisse, also etwa 16 Dezimalstellen), lassen sich viele Zahlen wie etwa  $1/7$  nicht exakt darstellen. Beim Vergleich von Rechenergebnissen empfiehlt es sich daher, nicht auf exakte Gleichheit von Fließkommazahlen zu prüfen. Mit `eps(target)` erhält man dagegen ein Ergebnis, das dem Ziel mit angemessener Toleranz entspricht.

Im folgenden Beispiel wurde willkürlich das Zehnfache der kleinstmöglichen Toleranz gewählt:

```
>> A = 1/7
A =
    0.142857142857143
>> B = 0.142857142857143
B =
    0.142857142857143
>> dAB = A-B
dAB =
    -1.387778780781446e-016
>> ApproxB = abs(A-B) < (10 * eps(B))
ApproxB =
    1
```

### ERSTELLUNG ROBUSTER PROGRAMME

Man sollte sich frühzeitig Gedanken machen, welche Probleme bei der Nutzung eines Programms durch den Endanwender auftreten könnten und so Fehler oder falsche Berechnungen vermeiden. Hier darum einige Tools zur Erzeugung robuster Programme.

### Verwendung von `MException` statt `lasterror`

Um sicherzustellen, dass ein Programm robust ist, muss man potenzielle Fehler frühzeitig abfangen und beheben. Nutzt man hierfür die ältere `lasterror`-Methode, riskiert man allerdings, dass noch nicht abgearbeitete Fehler überschrieben werden. Mit dem in R2007b eingeführten `MException`-Objekt vermeidet man dies:

#### Altes Codeschema

```
try
    doSomething;
catch
    rethrow(lasterror)
end
```

#### Neues Codeschema

```
try
    doSomething;
catch myException
    rethrow(myException) % or throw or throwAsCaller
end
```

## Verwendung von try/catch Exception statt der Zwei-Argumenten-Form von eval

Mit der Funktion `eval` lässt sich praktisch jede gültige MATLAB-Anweisung ausführen. Da aber das Argument für `eval` ein String ist, kann MATLAB nicht vorhersehen, was hier eigentlich ausgeführt wird und so eventuell bestimmte Optimierungen nicht nutzen. Das gleiche gilt für die Ausführung von `eval` mit zwei Argumenten, bei der die zweite Eingabe nur ausgewertet wird, wenn bei der Ausführung der ersten ein Fehler auftritt. Mit `try/catch` wird der Code dagegen für MATLAB transparent. Das folgende Skript fügt einen Wert `value` in die Struktur `S` ein, sofern `value` existiert; andernfalls bleibt das Strukturfeld leer:

### Altes Codeschema

```
eval(['S.field = value'],'S.field = []')
```

### Neues Codeschema

```
try
    S.field = value;
catch myException
    S.field = [];
end
```

## Verwendung dynamischer Feldnamen statt eval

Um ein beliebiges Feld eines `struct`-Arrays zu erzeugen, verwendet man statt `eval` dynamische Feldnamen. Mit dynamischen Feldnamen lassen sich Felder eines `struct`-Arrays ansteuern, ohne dass deren Name beim Schreiben des Programms explizit bekannt sein muss. Dynamische Feldnamen machen außerdem MATLAB-Code effizienter und sie erleichtern MATLAB die Programmanalyse. Die folgenden zwei Programme erzeugen eine Struktur `S` mit den Feldnamen `magic1`, ..., `magic5`:

### Altes Codeschema

```
n = 5;
for ind = 1:n
    eval(['S.magic' int2str(ind) '= magic(ind+2)']);
end
```

### Neues Codeschema

```
n = 5;
for ind = 1:n
    S.(['magic' int2str(ind)]) = magic(ind+2);
end
```

## Definition von Funktionen durch Function Handles statt Strings

Bei einigen MATLAB-Funktionen muss eine andere Funktion als Eingabeargument definiert werden. Die Funktion `fzero` dient beispielsweise zur Auffindung von Nullstellen anderer Funktionen. Mit den folgenden Codebeispielen findet man den Wert in der Nähe von Punkten `pts`, an denen die Funktion `sin` gleich 0 ist.

Definiert man eine auszuwertende Funktion als String, dann muss MATLAB den Pfad nach ihr durchsuchen. Geschieht diese Berechnung innerhalb einer Schleife, dann sucht MATLAB die Zielfunktion bei jedem Durchlauf erneut, denn der Pfad könnte sich geändert haben und sich nun auf eine andere Zielfunktion beziehen. Ein Function Handle legt dagegen die Zielversion eindeutig fest und MATLAB kann die Zielfunktion aufrufen, ohne sie in jedem Durchlauf wieder suchen zu müssen.

### Altes Codeschema

```
pts = 0:1000;
x = zeros(size(pts));
fun = 'sin';
for npts = 1:length(pts)
    x(npts) = fzero(fun,pts(npts));
end
```

### Neues Codeschema

```
pts = 0:1000;
x = zeros(size(pts));
fun = @sin;
for npts = 1:length(pts)
    x(npts) = fzero(fun,pts(npts));
end
```

## Datentypen von Variablen in Programmen durchgängig beibehalten

MATLAB versucht durch Optimierung von Berechnungen Speicher zu sparen. Führt man ein M-File aus, dann untersucht MATLAB zunächst das File und versucht es dann in optimaler Weise auszuführen. Wenn diese Analyse feststellt, dass die Variable `D` einen bestimmten Datentyp hat, etwa `double`, und dass dieser durchgehend unverändert bleibt, kann MATLAB seine Optimierungen anbringen.

### Altes Codeschema

```
D = 17.34;
doSomething;
D = single(D);
```

### Neues Codeschema

```
D = 17.34;
doSomething;
S = single(D);
```

## Ausgabeargumente bei `load` verwenden

MATLAB-Funktionen können sich auf unerwartete Weise verhalten, wenn MATLAB nicht "sehen" kann, dass einige der im Programm verwendeten Namen Variablen sind. Um dies zu vermeiden, verwendet man eine explizite Ausgabevariable mit `load`. MATLAB kann das Programm dann besser analysieren und erkennt zuverlässiger, welche Codeelemente Variablen sind. Eine Variable kann aber auch unvermittelt erzeugt werden, wenn man mit `load` einen Datensatz lädt, ohne dessen Variablen zu definieren. Angenommen, der Datensatz `mydat.mat` enthält die Variable `var` und ein Programm soll diese anzeigen.

```
function mydisp
load mydat
disp(var)
```

Wenn man `mydisp` ausführt, erhält man diesen Fehler:

```
>> mydisp
??? Input argument "x" is undefined.

Error in ==> var at 55
if isinteger(x)

Error in ==> mydisp at 3
disp(var)
```

Er tritt auf, weil `var` auch der Name einer Funktion im MATLAB-Pfad ist und MATLAB dies bei seiner Analyse festgestellt hat, bevor es das Programm ausführt.

Zur Vermeidung dieses Fehlers gibt es zwei Möglichkeiten:

```
function mydisp1
load mydat3 var
disp(var)
```

```
function mydisp2
S = load('mydat.mat');
disp(S.var)
```

In beiden Fällen versteht MATLAB, welche Elemente im Programm Variablen sind (`var` und `S`) und vermeidet so den Konflikt mit der im MATLAB-Pfad gefundenen Funktion. ■

## Einfache Tipps

- Löschen Sie beim Debuggen mit `clear variables` (ab R2008a mit `clearvars`) die Variablen aus dem Workspace. `clear` allein löscht auch alle Breakpoints.
- Verwenden Sie `{}` zur Extraktion von Zelleninhalten aus Cell-Arrays. Soll ein Cell-Array wie jedes andere Array behandelt werden, etwa für Umformungen, verwendet man dagegen `()`.
- Benutzen Sie Handles zu Grafikobjekten, die später aktualisiert werden sollen, anstatt sich vom jeweiligen Status abhängig zu machen. Erzeugen Sie beispielsweise eine Achse einmal mit `hAx = axes;` und verwenden Sie danach für diese Achse immer `hAx` statt `gca`. Die Verwendung dieser direkten Referenz ist robuster in Fällen, in denen der Fokus in MATLAB gerade auf etwas anderem liegt als auf diesem Grafikenfenster oder dieser Achse.

## Quellen

**BLOG: Loren on the Art of MATLAB**  
[www.mathworks.de/nn8/loren](http://www.mathworks.de/nn8/loren)

**MATLAB-HILFE**  
[www.mathworks.de/nn8/matlabdoc](http://www.mathworks.de/nn8/matlabdoc)