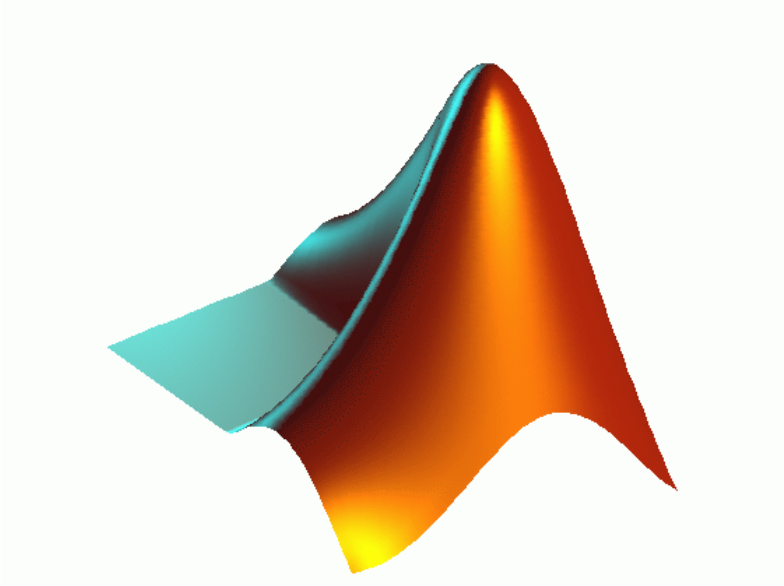
ulm university universität ulm



Wima Praktikum I

Matlab Praktikum - Tag 3

Prof. Dr. Karsten Urban,
Iris Häcker

Institut für Numerische Mathematik

Sommersemester 2013

Funktionen

m-Files

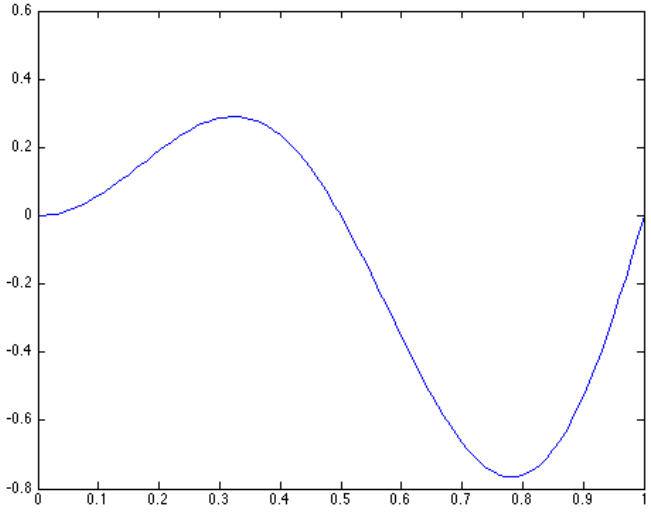
Debuggen

Array-Funktionen

Funktionen in Matlab - Anonyme Funktionen

- Die einfachste Methode in Matlab eine Funktion zu definieren ist mittels anonymer Funktion: `<Funktionsname> = @(<Argumentliste>)Funktionsbeschreibung`.

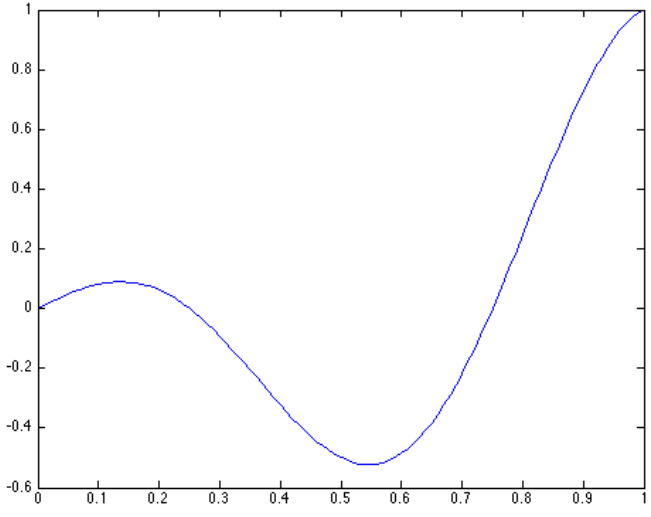
```
>> f=@(x)x.*sin(2*pi*x)
f =
    @(x)x.*sin(2*pi*x)
>> x=0:.01:1;
>> plot(x,f(x));
>> whos
  Name        Size                Bytes  Class                Attributes
  f           1x1                    16  function_handle
  x           1x101                 808  double
```



Funktionen in Matlab - Inline Funktionen

- Eine weitere Möglichkeit ist mit `inline` eine Inline Funktion zu definieren. Hierbei wird die Funktion durch einen String definiert.

```
>> g=inline('cos(2*pi*x).*x')
g =
     Inline function:
     g(x) = cos(2*pi*x).*x
>> plot(x,g(x));
>> whos
  Name        Size                Bytes  Class     Attributes
  g           1x1                   844  inline
  x           1x101                 808  double
```



Funktionen

m-Files

Debuggen

Array-Funktionen

m-Files Teil 1: Skripte (Wiederholung)

- Anweisungen können in Textdateien mit Endung `.m` geschrieben werden.
- Die Anweisungen werden im aktuellen Workspace ausgeführt und können auf die im Workspace definierten Variablen zugreifen.
- Skripten können keine Argumente übergeben werden.
- Variablen, die im Skript angelegt oder gelöscht werden, bleiben im Workspace bzw. werden aus dem Workspace gelöscht.

shift.m

```
n=1;
clear y;
y=x+n;
clear x;
```

```
>> x=1:5
x =
     1     2     3     4     5
>> who
Your variables are:
x
>> shift
>> who
Your variables are:
n  y
>> y
y =
     2     3     4     5     6
```

m-Files Teil 2: Funktionen

- Für komplexere Funktionen können Funktions-m-Files verwendet werden. Hier werden, wie in Skripten, Anweisungen in einer Textdatei gespeichert, die der Reihe nach abgearbeitet werden.
- Um die Datei als Funktion zu kennzeichnen, wird die Funktion durch

  `function <Ausgabeargumente> = <Funktionsname>(<Eingabeargumente>)`

  eingeleitet.
  Hierbei sollte `<Funktionsname>` mit dem Dateinamen übereinstimmen. Falls der Funktionsname und der Dateiname *nicht* übereinstimmen, wird der Dateiname zum Funktionsaufruf verwendet.

gerade.m

```
function [a, b] = gerade(x1, x2, y1, y2)
%GERADE Berechnet die Koeffizienten
% der linearen Funktion ax+b durch die
% Punkte (x1, y1) und (x2, y2)

a=(y2-y1)./(x2-x1);
b=y1-a.*x1;
```

```
>> [c1,c2]=gerade(0, 3, 1, 2)
c1 =
    0.3333
c2 =
     1
```

Funktions-Workspace

- Im Gegensatz zu Skripten wird eine Funktion in einem eigenen Workspace ausgeführt.
- In diesem Workspace werden Variablen unabhängig vom Matlab-Workspace angelegt und gelöscht. Nach dem Beenden der Funktion werden die Variablen des Funktionsworkspace gelöscht.
- Soll auf eine Variable des Matlab-Workspace zugegriffen werden, die nicht als Argument übergeben wird, so muss diese mit `global <Variablenname>` im Matlab-Workspace und im Funktions-Workspace sichtbar gemacht werden.
- Variablen, die nach beenden der Funktion ihren Wert behalten sollen, müssen mit `persistent <Variablenname>` deklariert werden.

modulfunc.m

```
function y=modulfunc(x)
modul=7;
global letzterAufruf;
persistent funcnt;
if isempty(funcnt)
    funcnt=1;
else
    funcnt=funcnt+1;
end
disp('Funktionsaufrufe:')
disp(funcnt);
letzterAufruf=datestr(now,13);
y=x.^2+3.*x+2;
y=mod(y,modul);
```

```
>> global letzterAufruf
>> letzterAufruf=0
letzterAufruf =
     0
>> modulfunc(3)
Funktionsaufrufe:
     4
ans =
     6
>> disp(letzterAufruf)
12:57:09
>> who
Your variables are:
ans             letzterAufruf
```

m-File Funktionen: Kommentare

- Mit % kann ein einzeiliger Kommentar und mit %{ }% ein Kommentarblock definiert werden. Kommentare mit % beginnen beim Kommentarzeichen und gehen bis zum Ende der Zeile. Bei der Definition eines Kommentarblocks mössen die %{ %} Zeichen jeweils in einer eigenen Zeile stehen.
- Die ersten zusammenhängenden Kommentarzeilen vor der ersten Anweisung werden beim Aufruf von `help <Funktionsname>` angezeigt.
- Die erste Kommentarzeile dieses Blocks wird beim Aufruf von `lookfor` durchsucht.

eratosthenes.m

```
function x = eratosthenes(n)

%ERATOSTHENES Berechnet alle Primzahlen bis zu einer oberen Schranke
%
% INPUT : n  Obere Schranke der Primzahlen
% OUTPUT: x  Vektor der Primzahlen der groesse kleiner oder gleich n
%
% Die Funktion berechnet Primzahlen mit dem Sieb des Eratosthenes

...
```

```
>> lookfor eratosthenes
ERATOSTHENES Berechnet alle Primzahlen bis zu einer oberen Schranke
```

m-File Funktionen: Ein- und Ausgabeargumente

- Eingabeparameter werden via call-by-value übergeben, d.h. Änderungen der Eingabevariablen in der Funktion ändert die beim Aufruf benutzte Variable nicht.
- Zur Verwendung einer Ein- oder Ausgabevariablenliste mit variabler Länge können in der Funktionsdefinition die Argumente `varargin` und `varargout` verwendet werden.
- Zur Abfrage der Anzahl von Ein- und Ausgabeargumenten können die Funktionen `nargin` und `nargout` verwendet werden.
- Mit `exist` kann geprüft werden, ob eine Variable definiert ist.
- Die Funktionen `nargchk` und `nargoutchk` überprüfen die Anzahl der Eingabe- bzw. Ausgabeparameter. Beispiel aus der Matlab-Hilfe:

mysize.m

```
function [s, varargout] = mysize(x)
msg = nargoutchk(1, 3, nargout);
if isempty(msg)
    nout = max(nargout, 1) - 1;
    s = size(x);
    for k = 1:nout, varargout(k) = {s(k)}; end
else
    disp(msg)
end
```

Argumentprüfung

Datentypen von Variablen und Funktionsargumenten können mit den `is*` Funktionen überprüft werden. Die wichtigsten Funktionen dazu sind

- `isempty`: überprüft, ob das Argument eine leere Matrix ist.
- `isnumeric`: überprüft, ob das Argument numerisch ist (Skalar, Vektor, Matrix).
- `isstr`: überprüft, ob das Argument ein String ist.
- `issparse`: überprüft, ob das Argument eine sparse Matrix ist.
- `isscalar`, `isvector`: überprüft, ob das Argument ein Skalar bzw. Vektor ist.
- `isreal`: überprüft, ob das Argument reell ist.
- `iscell`, `isstruct`: überprüft, ob das Argument ein Cell-Array bzw. Struct ist.

Weitere Funktionen können in der Hilfe mit `doc is` nachgelesen werden.

Funktion vs. Kommando

In Matlab wird zwischen Funktionsaufruf und Kommandoaufruf unterschieden.

- Bei Funktionsaufrufen werden die Argumente in runden Klammer aufgelistet. Dabei werden Variablen interpretiert und ihr Wert wird an die Funktion übergeben.

```
>> x=1:3
x =
     1     2     3
>> disp(x)
     1     2     3
```

- Bei Kommandoaufrufen werden die Argumente ohne Klammerung aufgelistet. Dabei werden die Argumente als Strings interpretiert.

```
>> disp x
x
```

- Viele Funktionen sind auch als Kommando aufrufbar und liefern dann mit dem falschen Aufruf unerwartete Ergebnisse.

Funktionen zur Datei- und Verzeichnisverwaltung

Zur Verwaltung von Verzeichnissen und Dateien, insbesondere m-Files, stehen in Matlab einige Funktionen zur Verfügung:

- `cd`: Wechselt das Arbeitsverzeichnis.
- `mkdir`: Legt ein neues Verzeichnis an.
- `rmdir`: Löscht ein Verzeichnis.
- `ls, dir`: Zeigt den Inhalt eines Verzeichnisses.
- `delete`: Löscht eine Datei.
- `open`: Öffnet eine Datei im Matlab-Editor.
- `which`: Zeigt an, welches m-File beim Aufruf einer Funktion ausgeführt wird.
- `type`: Zeigt den Inhalt eines m-Files an.

Funktionen zur File I/O

Es gibt verschiedene Möglichkeiten Daten in Dateien zu schreiben oder aus Dateien zu lesen:

- `save`, `load` zum Laden bzw. speichern von Variablen aus einem Matlab-Workspace.

```
>> A=rand(2);
>> save 'ws.mat'  A
>> clear
>> load ws.mat
>> who
Your variables are:
A
```

- `csvwrite`, `csvread` zum schreiben bzw. lesen von Matrizen in bzw. aus einer Textdatei, in der die Elemente durch Komma getrennt sind.

```
>> csvwrite('A.txt', A)
>> B=csvread('A.txt')
B =
    0.3171    0.0344
    0.9502    0.4387
```

- `dlmwrite`, `dlmread` zum schreiben bzw. lesen von Matrizen in bzw. aus einer Textdatei, in der die Elemente durch ein beliebiges Zeichen getrennt sind.
- `textread` zum formatierten Einlesen von Daten aus einer Datei.

Funktionen zur File I/O (cont'd)

Analog zu den Programmiersprachen C, C++, Java gibt es in Matlab Funktionen zur I/O mit File-Handels.

- `fopen, fclose` zum Öffnen und Schließen eines Dateihandles.
- `fprintf, fwrite` zum Schreiben von Daten in eine Datei über ein Handle.
- `fscanf, fgets, fread, textscan` zum Einlesen von Daten über ein Handle.

```
>> fid=fopen('A.txt', 'r');
>> a=fscanf(fid, '%g,%g',[2 inf])
a =
    0.3171    0.9502
    0.0344    0.4387
>> fclose(fid);
```

Unterfunktionen

Beim Aufruf einer Funktion in einem m-File wird nur die erste Funktion der Datei ausgeführt. Zur Strukturierung können Unterfunktionen in einem m-File definiert werden. Diese können *nur* aus den Funktionen des m-Files aufgerufen werden.

funktion.m

```
function y = funktion(calls)
y=0;
for i=1:calls
    unterfunktion(i)
    y=y+i;
end

function []=unterfunktion(n)
fprintf(1, '%d-ter␣Aufruf\n', n);
```

```
>> funktion(2)
1-ter Aufruf
2-ter Aufruf
ans =
     3
```

Dabei hat jede Unterfunktion einen eigenen Workspace. Variablen, die in mehreren Funktionen verwendet werden sollen bzw. ihren Wert behalten sollen, müssen wieder als `global` oder `persistent` deklariert werden.

Verschachtelte Funktionen

In Matlab ist es auch erlaubt Funktionen in Funktionen zu definieren. In diesem Fall müssen die Funktionen mit einem `end` abgeschlossen werden.

funktion.m

```
function y = funktion(calls)
y=0;
for i=1:calls
    unterfunktion(i)
    y=y+i;
end

function []=unterfunktion(n)
    fprintf(1, '%d-ter␣Aufruf\n', n);
end

end
```

```
>> funktion(2)
1-ter Aufruf
2-ter Aufruf
ans =
     3
```

Die Funktionen haben dabei wieder einen eigenen Workspace, können aber auf den Workspace der umschliessenden Funktion zugreifen.

Funktionen in Strukturen und Cell-Arrays

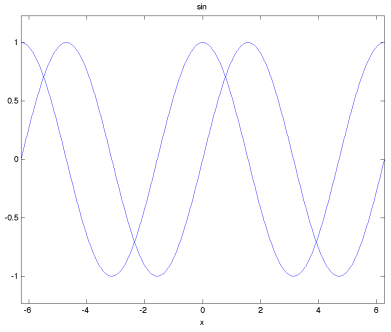
- Für Funktionen in m-Files können, wie für die built-in Funktionen von Matlab, mit `@<Funktion>` Funktionen Handles definiert werden.

```
>> myfun=@sin
myfun =
    @sin
>> myfun(pi/4)
ans =
    0.7071
```

- Somit können Funktionen als Handle auch Elemente von Cell-Arrays und Strukturen zugewiesen werden:

```
>> C={@sin, @cos}
C =
    @sin    @cos
>> C{:}
ans =
    @sin
ans =
    @cos
>> C{1}(pi/4)
ans =
    0.7071
```

```
>> hold on
>> cellfun(@ezplot, C)
```



Funktionen

m-Files

Debuggen

Array-Funktionen

Debuggen von Skripten und Funktionen mit Standardfunktionen

Wie in anderen Programmiersprachen auch gibt es mehrere Methoden m-File-Funktionen und Skripte zu debuggen. Eine Möglichkeit ist das Einfögen von Ausgaben und Anweisungen zur Überprüfung:

- `echo`, `disp`, `fprintf` zur Ausgabe von Werten und zur Ablaufkontrolle;
- `size`, `who`, `whos` zur Überprüfung von Variablen im Workspace und deren Grööe;
- `lasterr`, `lasterror`, `lastwarn` zur Ausgabe der letzten Fehlermeldung oder Warnung;
- `keyboard`, `input`, `return` zur Unterbrechung des Programmflusses durch Benutzereingaben.
- Mit `ginput` kann bei graphischen Benuzteroberlächen auf eine Mauseingabe gewartet werden. Die Funktion liefert dann die Koordinaten des Mauszeiger zurück.

Matlab Debugger

Darüber hinaus bietet Matlab einen Debugger, mit dem die Programmausführung an beliebiger Stelle unterbrochen, Kommandos einzeln ausgeführt und Werte von Variablen überprüft werden können.

- Mit `dbstop` und `dbclear` können Haltepunkte gesetzt und wieder entfernt werden, `dbstatus` zeigt die gesetzten Haltepunkte an.

```
>> dbstop in modulfunc at 11
>> dbstop in modulfunc at 3
>> dbstatus
Breakpoints for modulfunc are on lines 3, 11.
>> dbclear in modulfunc at 3
```

- Mit `dbtype` können m-Files mit Zeilennummern angezeigt werden.

```
>> dbtype modulfunc.m
1       function y=modulfunc(x)
2
3       modul=7;
4       global letzterAufruf
5       persistent funcnt
6       if isempty(funcnt)
7           funcnt=1;
8       else
        ...
```

Matlab Debugger (cont'd)

- Beim Aufruf der Funktion wird der Programmlauf an den Haltepunkten unterbrochen. Wird die Abarbeitung einer Funktion an einem Haltepunkt unterbrochen, so können an dem Punkt beliebige Funktionsaufrufe durchgeführt werden.
- Mit `dbstep` und `dbcont` kann die Funktion in Einzelschritten oder bis zum nächsten Haltepunkt fortgesetzt werden.

```
>> modulfunc(3)
11  disp('Funktionsaufrufe:')
K>> funcnt
funcnt =
     1
K>> dbstep
Funktionsaufrufe:
12  disp(funcnt);
K>> dbcont
     3
ans =
     6
```

- `dbstack` zeigt den Namen der aktuell ausgeführten Funktion an

```
K>> dbstack
> In modulfunc at 11
```

Matlab Debugger (cont'd)

- Mit den Funktionen `dbup` und `dbdown` kann zwischen dem aufrufendem und dem Funktions-Workspace gewechselt werden.

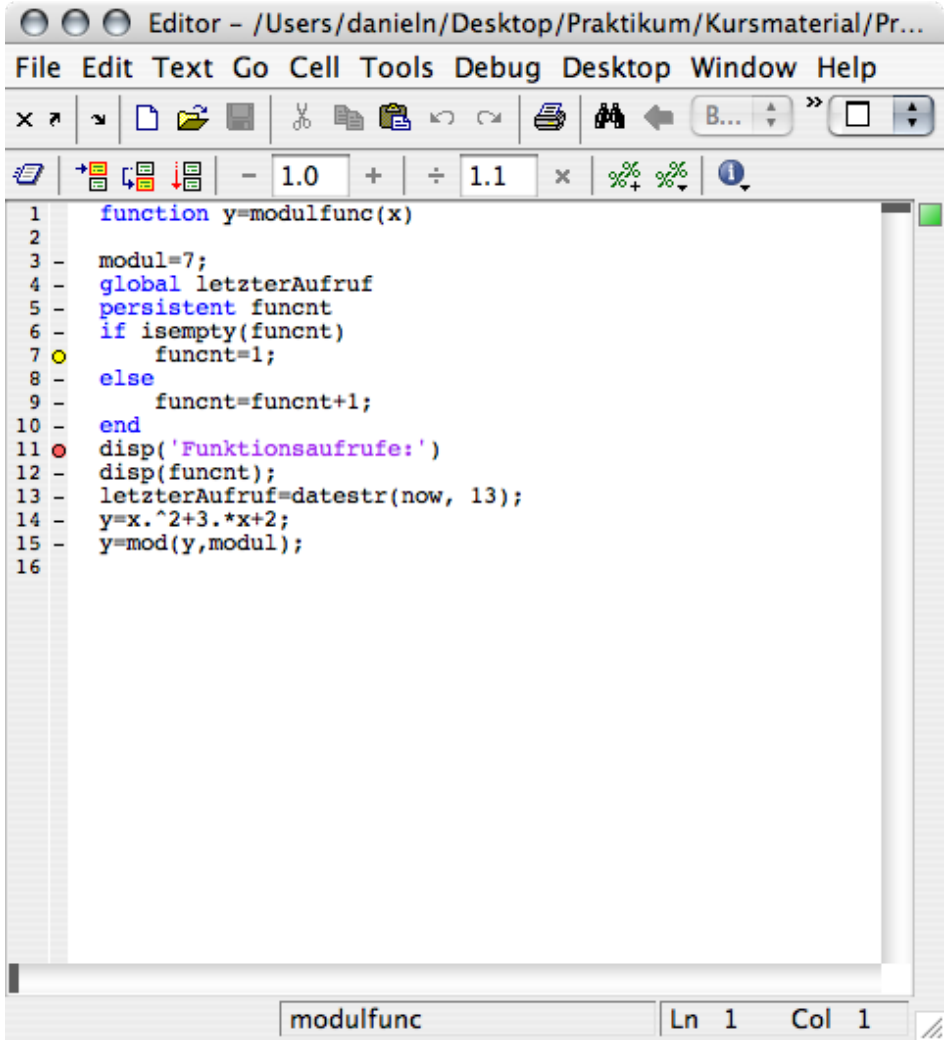
```
K>> dbup
In base workspace.
K>> who
Your variables are:
ans
K>> dbdown
In workspace belonging to modulfunc at 11
K>> who
Your variables are:
funcnt          modul
letzterAufruf   x
```

- Mit dbquit kann die Ausführung abgebrochen werden.

```
K>> dbquit
>>
```

Graphischer Debugger

Der Debugger kann auch über den Matlab-Texteditor bedient werden.



Funktionen

m-Files

Debuggen

Array-Funktionen

Array-Funktionen

Um eine Funktion auf jedes Element eines normalen Arrays oder Cell-Arrays anzuwenden gibt es die Array-Funktionen:

- `O = arrayfun(fun,I)` und
- `O = cellfun(fun,I)`.

Dabei ist

- `fun` eine Funktion mit *einem*, *skalarem* Ausgabeargument und mit *einem* Eingabeargument,
- `O` ein normales Array der gleichen größe wie `I`
- und für jeden Eintrag von `O` gilt:
  `O(n) = fun(I(n))`, bzw. `O(n) = fun(I{n})`.

Array-Funktionen

Beides kann auch verallgemeinert werden zu:

- `[O1,...,OM] = arrayfun(fun,I1,...,IN)` und
- `[O1,...,OM] = cellfun(fun,I1,...,IN)`,

wobei `I1,...,IN` alle die gleiche Größe haben müssen.

Dabei ist

- `fun` eine Funktion mit `M` *skalaren* Ausgabeargumenten und mit `N` Eingabeargumenten,
- jedes Ausgabeargument `O1,...,OM` ein normales Array der gleichen größe wie `I1`
- und für jeden Eintrag gilt:
  `[O1(n),...,OM(n)] = fun(I1(n),...,IN(n))`, bzw.
  `[O1(n),...,OM(n)] = fun(I1{n},...,IN{n})`.

Array-Funktionen

Sind die Ausgabeargumente der Funktion `fun` *nicht* skalar, so ergibt:

- `[O1,...,OM] = arrayfun(fun,I1,...,IN),'UniformOutput',false)` und
- `[O1,...,OM] = cellfun(fun,I1,...,IN),'UniformOutput',false)`,

das gleiche Ergebnis wie vorher, mit dem Unterschied, dass die Ausgabeargumente `O1,...,OM` nun Cell-Arrays sind und es gilt:

- `[O1{n},...,OM{n}] = fun(I1(n),...,IN(n))`, bzw.
- `[O1{n},...,OM{n}] = fun(I1{n},...,IN{n})`.

Bei der Funktion `cellfun` kann `fun` auch einer der folgenden Strings sein (vgl. Hilfe):

- 'isreal', 'isempty', 'islogical',
- 'length', 'ndims', 'prodofsize'.