M2 Matlab Programme

Auswahlanweisungen (if-then-else),
Schleifen (for und while),
Skripte und Funktionen (m-Files),
anonyme Funktionen, Funktions-Handles

12.11.2014

Skripte (m-Files)

Ein Skript ...

- ... ist eine Sequenz von Befehlen
- ... hat den gleichen Effekt wie bei Eingabe in die Konsole

Variablen in Skripten

- gleiche Variablen wie in der Konsole (gleicher Workspace)

Spezialzeichen

- Zeilenende: Befehlstrenner (ausser nach `...`)
- `,` (Komma): Befehlstrenner
- `;` (Strichpunkt): Befehlstrenner, sowie Ausgabe unterdrücken
- `%` (Prozent): Kommentar (bis Zeilenende)
- `...` (3 Punkte): Fortsetzungszeile, Kommentar bis Zeilenende

Beispiele

ScriptA.m

```
I = [1:5].' * ones(1,5);
J = ones(5,1) * [1:5];
M = I + J -1
```

```
>> ScriptA
M =
    1   2   3   4   5
    2   3   4   5   6
    3   4   5   6   7
    4   5   6   7   8
    5   6   7   8   9
```

ScriptB.m

```
B = ones(size(A));
C = diag(A);
```

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


If-Then-Else

Syntax

```
if ⟨Bedingung⟩
   ⟨Anweisungen⟩
elseif ⟨Bedingung⟩
   ⟨Anweisungen⟩
...
else
   ⟨Anweisungen⟩
end
```

BspIf.m

```
a = 5;
b = 7;
if a < b
   str = 'kleiner';
elseif a == b
   str = 'gleich';
else
   str = 'groesser';
end
```

Bemerkungen

- Mehrere Anweisungen erlaubt, keine Blöcke notwendig
- == ist Vergleich, = ist Zuweisung

Bedingungen

Eine Bedingung ist ein Ausdruck vom Typ 1 × 1 logical

Matlab ist flexibel

Operatoren

- Vergleiche: $==, \sim=, <, >, <=, >=$ (auf `double`, `single`, `intXX`)
- Verknüpfungen: && (and), || (or), $\sim$ (not)
- Bei `if`: immer short-circuit Evaluation (&, | nicht verwenden!)

Short-Circuit in Matlab

Gut.m

```
if numel(A) >= 3 && A(3) < 2
   disp(1)
end
  ⇒ 1
```

keine Klammern nötig

Böse.m

```
if true || '#$%@'
   disp(1)
end
  ⇒ 1
```

```
if false || '#$%@'
   disp(1)
end
  ⇒ Fehlermeldung
```


While-Schleifen

Syntax

```
while ⟨Bedingung⟩
   ⟨Anweisungen⟩
end
```

BspWhile.m

```
a = 91;
b = 143;
% a <- ggT(a,b)
while a ~= b
   if a > b
      a = a - b;
   else
      b = b - a;
   end
end
```

Vorzeitiger Schleifenabbruch

- `break` – Abbruch der Schleife
- `continue` – Abbruch der Iteration

For-Schleifen

Syntax

```
for ⟨Var⟩ = ⟨Start⟩:⟨End⟩
  ⟨Anweisungen⟩
end
```

```
for ⟨Var⟩ = ⟨Start⟩:⟨Schritt⟩:⟨End⟩
  ⟨Anweisungen⟩
end
```

```
for ⟨Var⟩ = ⟨Matrix⟩
  ⟨Anweisungen⟩
end
```

$1 \times n$ Matrix
⇒ iteriert über Elemente

Beispiele

```
% Summiert Elemente in A
s = 0;
for i = 1:numel(A)
  s = s + A(i);
end
```

```
% Ungerade Indizes
s = 0;
for i = 1:2:numel(A)
  s = s + A(i);
end
```

```
% Summiert Spaltenvektoren
s = 0;
for v = A
  s = s + v;
end
```


Funktionen

Syntax

```
function ⟨Res⟩ = ⟨Name⟩ (⟨Arg1⟩,⟨Arg2⟩,...)
% ⟨Name⟩ Kurze Hilfe
% Lange Hilfe

  ⟨Anweisungen⟩
  ...
end
```

Matlab ist flexibel

Matlab ist flexibel

Variablen

- Alle Variablen sind lokal
- Ausnahme: `global(⟨VarName⟩)`

Aufruf

```
>> t = ggT(91,143)
t = 13
```

ggT.m

```
function y = ggT(a,b)
% ggT - berechnet ggT(a,b)
%   Das ggT wird berechnet,
%   indem immer wieder ...

  while a ~= b
    if a > b
      a = a - b;
    else
      b = b - a;
    end
  end
  y = a;

end
```


Funktionen mit mehreren Rückgabewerten

Syntax

```
function [⟨Res1⟩,⟨Res2⟩,...] = ⟨Name⟩ (⟨Arg1⟩,⟨Arg2⟩,...)
% ...
```

Rucksack.m

```
function [bestVal,bestMatrix] = Rucksack(Gewichte,Werte,Max)
% Berechnet den besten Rucksack-Wert
...
```

Nur Tabelle:
[∼, tabelle] = Rucksack(G,W,10)

```
>> w = Rucksack(G,W,10)
w =  1519
>> [w,tabelle] = Rucksack(G,W,10)
w =  1519
tabelle =
    0     0     0     0     0     0     0     0     0     0
  269   269   269   269   269   269   269   269   269   269
  ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
  269   270   749  1018  1019  1178  1179  1179  1250  1519
```


Funktionen mit optionalen Argumenten 1/2

Matlab: Alle Argumente sind optional!

- Aufruf mit *zu vielen* Argumenten: Fehlermeldung
- Aufruf mit *zu wenigen* Argumenten: ok
- *Zugriff* auf nicht übergebene Argumente: Fehlermeldung

OptArgs.m

```
function y = OptArgs(a,b,c)
  if a==1
    y = 'one';
  elseif b==1
    y = 'two';
  elseif c==1
    y = 'three';
  else
    y = 'none';
  end
end
```

```
>> OptArgs(1,1,1,1)
 ⇒ Fehler (zuviele Argumente)
>> OptArgs(1)
     one
>> OptArgs(2,1)
     two
>> OptArgs(2,3)
 ⇒ Fehler ('c' nicht definiert)
```


Funktionen mit optionalen Argumenten 2/2

Anzahl Argumente

- Eine Funktion kann die Anzahl übergebene Argumente feststellen
- `nargin` → Anzahl Argumente

twos.m

```
function M = twos(m,n)
% twos(m,n)  m x n Matrix mit 2
  if nargin == 0
    M = 2;
  elseif nargin == 1
    M = repmat(2,m,m);
  else
    M = repmat(2,m,n);
  end
end
```

```
>> twos
ans =  2
>> twos(2)
ans =
     2     2
     2     2
>> twos(2,5)
ans =
     2     2     2     2     2
     2     2     2     2     2
```


Funktionen mit beliebig vielen Argumenten

Syntax

```
function ⟨Res⟩ = ⟨Name⟩ (⟨Arg1⟩,...,varargin)
% ...
```

Zugriff auf die Argumente

- `nargin` → Gesamtanzahl Argumente
- `length(varargin)` – Anzahl zusätzliche Argumente
- `varargin{i}` – Zugriff auf i-tes Zusatzargument

AddAll.m

```
function s = AddAll(varargin)
  s = 0;
  for i = 1:length(varargin)
    s = s + varargin{i};
  end
end
```

```
>> AddAll()
ans =   0
>> AddAll(1,2,3,4,5)
ans =   15
```

Funktionen mit optionalen Rückgabewerten

Anzahl Rückgabewerte

- Eine Funktion kann die Anzahl geforderter Rückgabewerte feststellen
- `nargout` = Anzahl geforderter Werte, `varargout{i}` = i-ter Wert

randi2.m

```
function [varargout] = randi2(varargin)
% Berechnet beliebig viele Zufallsmatrizen
  for i = 1:nargout
    varargout{i} = randi(varargin{:});
  end
end
```

```
>> [A,B] = randi2(10,2,5)
A =   6     2     7     6     1
      6     9     4     5     3
B =   2     3     1    10     5
      2     5    10     5     4
```


Lokale Funktionen

Im gleichen m-File:

QuickSort.m

```
function B = QuickSort(A)
  ...
end

function [L,R] = split(M,pivot)
  ...
end
```

Eigenschaften

- Split ist „von aussen“ nicht sichtbar
- ansonsten ist Split eine normale Funktion

Verschachtelte Funktionen

Im gleichen m-File:

QuickSort.m

```
function B = QuickSort(A)
  ...
  function [L,R] = split(M,pivot)
    ...
  end
end
```

Eigenschaften

- Split ist „von aussen“ nicht sichtbar
- Split kann auf die lokalen Variablen (A und B) von QuickSort zugreifen

Anonyme Funktionen

Syntax

⟨***Name***⟩ `= @(`⟨*Arg1*⟩`,...)` ⟨*Ausdruck*⟩

```
>> rand10 = @(m,n) randi(10,m,n);
>> rand10(3,5)
ans =
     3     2     3     9     6
     6     4     3     1     1
     6     7     9    10     1
```

Eleganter:

```
rand10 = @(varargin) ...
       randi(10,varargin{:})
```

Eigenschaften

- können einer Variablen zugewiesen werden (kein m-File nötig)
- nur für *sehr kurze* Funktionen geeignet
- z.B. beim Plotten: `fplot(@(x) cos(x)*x, [0,2*pi])`

Funktions-Handles

Zuweisung

- `f = @(x) x.^3 - 5`
- `g = @sin`

Verwendung

- Aufruf: `y = f(7)`
- Als Parameter: `fplot(g,[0,5])`

MkMatrix.m

```
function M = MkMatrix(m,n,func)
% Create matrix from m, n, and func(I,J) -> elements
  I = [1:m].' * ones(1,n);
  J = ones(m,1) * [1:n];
  M = func(I,J);
end
```

```
>> MkMatrix(5,5, @(I,J) I+J-1)
ans =
     1     2     3     4     5
     2     3     4     5     6
     ...
     5     6     7     8     9
```


Aufruf von Funktionen

Aufruf

```
>> t = ggT(91,143)
```

Suche nach 'ggT' (unvollst.)

1. Variable
2. Verschachtelte Funktion
3. Lokale Funktion
4. m-File im aktuellen Verzeichnis
5. m-File im Suchpfad

Was wird gefunden?

```
>> which ggT
/home/hirt/.../VL/ggT.m
>> which sin
built-in (/usr/ethz/.../elfun/@double/sin)
>> ggT = 5;
>> which ggT
ggT is a variable.
```