UNIVERSITÄT
DES
SAARLANDES

MATLAB
für Naturwissenschaftler

5. Grundlegende Sprachkonzepte

Till Biskup

Lehrstuhl für Physikalische Chemie und Didaktik
Universität des Saarlandes

Übersicht

Motivation

Warum sich mit Konzepten einer Programmiersprache befassen?

Ausgangslage

- Wir wollen, dass uns der Computer Arbeit abnimmt.

Vorgehen

- verbale Formulierung der Problemstellung
  - Nur wenn wir konkret wissen, was wir wollen, können wir das auch einem Computer beibringen.
- Aufteilung in kleine Blöcke
  - Selten sehen wir sofort die Lösung für das ganze Problem.
  - Kleine Teilprobleme sind dagegen oft einfach lösbar.
- Umsetzung dieser Blöcke in Programme
  - Programme = Sprache, die der Computer versteht

Motivation

Warum sich mit Konzepten einer Programmiersprache befassen?

Die „schlechte“ Nachricht

- Programmieren lernen ist wie eine Sprache lernen.
- Grundlegende Sprachkonzepte müssen bekannt sein.

Die „gute“ Nachricht

- Es gibt nur wenige fundamentale Programmierparadigmen.
- Es gibt nur wenige Grundkonzepte.

- ☛ Ähnlich wie beim Erlernen einer menschlichen Sprache: Die grundlegende Grammatik ist (oft) ähnlich.
- ☛ Die hier vorgestellten Konzepte sind recht universell...

Grammatik

- Operatoren
- Datentypen
- Entscheidungsstrukturen
- Schleifen

Syntax

- Kommentare
- Zeilenenden
- Groß- und Kleinschreibung
- Zeilenumbrüche und Leerzeilen
- Einrückungen

Syntax: MATLAB-Spezifika

- Kommentare
  - Werden durch Prozentzeichen (%) eingeleitet
  - Alles nach dem „%“ in einer Zeile wird ignoriert.
- Zeilenenden
  - Befehle normalerweise mit Semikolon (;) beenden
  - Ansonsten wird der Variableninhalt ausgegeben
- Groß- und Kleinschreibung
  - MATLAB unterscheidet zwischen Groß- und Kleinschreibung
- Zeilenumbrüche und Leerzeilen
  - Zeilenumbrüche in einem Befehl mit „. . . “
  - Leerzeilen ansonsten beliebig

Syntax: MATLAB-Beispiele

Listing 1: Beispiele für die grundlegende Syntax in MATLAB

```
1  % Das ist ein Kommentar
2
3  sin(2*pi) % Beispiel fuer einen Kommentar nach einem Befehl
4
5  % Hier wird das Ergebnis ausgegeben
6  sin(2*pi)
7
8  % Hier wird nichts ausgegeben
9  sin(2*pi);
10
11 % Diese beiden Befehle sind _nicht_ identisch
12 sin(2*pi);
13 Sin(2*pi);
14
15 % Zeilenumbruch innerhalb eines Befehls
16 x = [ ...
17       1 2 3 ; ...
18       2 3 4 ; ...
19       3 4 5 ...
20     ];
```

```
#!/usr/bin/perl -w                                      # camel code
use strict;

                                           $_='ev
                                       al("seek\040D
           ATA,0,                  0;");foreach(1..3)
       {<DATA>;}my               @camel1hump;my$camel;
  my$Camel  ;while(             <DATA>){$_=sprintf("%-6
9s",$_);my@dromedary           1=split(//);if(defined($
_=<DATA>)){@camel1hum        p=split(//);}while(@dromeda
 ry1){my$camel1hump=0      ;my$CAMEL=3;if(defined($_=shif
        t(@dromedary1    ))&&/\S/){$camel1hump+=1<<$CAMEL;}
       $CAMEL--;if(d   efined($_=shift(@dromedary1))&&/\S/){
      $camel1hump+=1  <<$CAMEL;}$CAMEL--;if(defined($_=shift(
     @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL--;if(
     defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
     L;;}$camel.=(split(//,"\040..m`{/J\047\134}L^7FX"))[$camel1h
      ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
      camel1hump){chomp;$Camel=$_;y/LJF7\173\175`\047/\061\062\063\
      064\065\066\067\070/;y/12345678/JL7F\175\173\047`/;$_=reverse;
       print"$_\040$Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y
        /LJF7\173\175`\047/12345678/;y/12345678/JL7F\175\173\0 47`/;
         $_=reverse;print"\040$_$Camel\n";}';;s/\s*//g;;eval;   eval
           ("seek\040DATA,0,0;");undef$/;$_=<DATA>;s/\s*//g;(   );;s
             ;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__   \124
               \1   50\145\040\165\163\145\040\157\1 46\040\1  41\0
                    40\143\141  \155\145\1 54\040\1   51\155\  141
                    \147\145\0  40\151\156 \040\141    \163\16 3\
                     157\143\   151\141\16  4\151\1     57\156
                     \040\167  \151\164\1   50\040\      120\1
                     45\162\   154\040\15    1\163\      040\14
                     1\040\1   64\162\1      41\144       \145\
                     155\14    1\162\       153\04        0\157
                      \146\     040\11     7\047\         122\1
                      45\15      1\154\1  54\171          \040
                      \046\         012\101\16            3\16
                      3\15           7\143\15             1\14
                      1\16            4\145\163           \054
                     \040            \111\156\14         3\056
                    \040\         125\163\145\14         4\040\
                    167\1        51\164\1  50\0         40\160\
                  145\162                              \155\151
                \163\163                                \151\1
              57\156\056
```

https://www.perlmonks.org/?node_id=45213 (abgerufen am 17.02.2018)

Syntax: Einrückung

- Nicht jeder Programmiersprache ist Einrückung egal
  - Python verzichtet auf Klammern, aber erzwingt Einrückung
- Code wird viel öfter gelesen als geschrieben
  - Verständlichkeit und Lesbarkeit sind Trumpf
  - Konsistente Einrückungen sind einfach, aber sehr wirksam
- (Korrekte) Einrückung erhöht die Lesbarkeit
  - Zusammenhänge sofort erkennbar
  - Blöcke in Schleifen und Bedingungen offensichtlich
  - Inkorrekte Einrückungen sind mühsam

☛ Viele Editoren beherrschen automatische Einrückung (auch der MATLAB-Editor)

Operator (Mathematik)

mathematische Vorschrift, durch die man aus mathematischen Objekten neue Objekte bilden kann

Operator (Informatik)

Konstrukt, das sich allgemein wie eine Funktion verhält,
sich aber syntaktisch/semantisch von Funktionen unterscheidet

Operatoren – Unterscheidung nach Typen

- arithmetisch
  - $+, -, *, /, \hat{}$
- relational
  - $<, >, <=, >=$
  - $==, \sim=$
- logisch
  - $\&, |, \&\&, ||$
- Zuweisung
  - $=$
- weitere
  - bitweise Operatoren

Operatoren – Unterscheidung nach Stelligkeit

- unär (einstellig, monadisch)
  - nur ein Operand
  - Beispiel: `-1`
  - Beispiele in C: `++i`, `k--`
- binär (zweistellig, dyadisch)
  - zwei Operanden
  - Beispiele: `a+b`, `c = d`
- ternär (dreistellig, tryadisch)
  - drei Operanden
  - nicht in MATLAB
  - Beispiel in C: `a ? b : c`
    (Bedingungsoperator: „Wenn a dann b sonst c“)

Operatorrangfolge

- Immer eine Frage der Definition
  - Einfaches Beispiel: „Punkt vor Strich“
- Für jede Programmiersprache anders
  - Einzige Chance: in der Dokumentation nachschauen
  - MATLAB lehnt sich stark an die Regeln der Mathematik an.
- Nicht immer eindeutig
  - Was immer hilft: Klammern setzen
  - Beispiel aus MATLAB: `-1^2` $\neq$ `(-1)^2`

Typisierung

Zuweisung eines Objekts einer Programmiersprache (zum Beispiel einer Variable) zu einem Datentyp

- Datentypen
  - Numerisch
  - Zeichen und Zeichenketten (*strings*)
  - Boolesche Ausdrücke
  - Zeiger (Referenzen)
  - Komplexe Datentypen
- Bedeutung unterschiedlicher Datentypen
  - Mit Zeichenketten kann man (meist) nicht rechnen.
  - Befehle erwarten meist bestimmte Datentypen.

Numerische Datentypen

- Zwei Unterscheidungsmöglichkeiten
  1. Dimension
  2. Präzision/Wertebereich

Dimension

- Skalar (1×1)
- Vektor ($1 \times n$, $n \times 1$)
- Matrix ($n \times m$)

Präzision/Wertebereich

- (signed) integer
- real/float/double

☛ MATLAB rechnet normal mit Gleitkommazahlen (double).
- Standard: ANSI/IEEE 754-1985

Numerische Datentypen

Listing 2: Numerische Datentypen unterschiedlicher Dimension in MATLAB

```
1  % Skalar
2  number = 1;
3  emptyScalar = [];
4
5  % Vektoren
6  rowVector    = [1 2 3 4 5];
7  rowVector    = [1, 2, 3, 4, 5];
8  rowVector    = 1:5;
9
10 columnVector = [1; 2; 3; 4; 5];
11 columnVector = rowVector';
12
13 % Matrix
14 matrix = [ 1 2 3 ; 2 3 4 ; 3 4 5 ; 4 5 6 ];
```

- Zeilenvektor: eine Zeile/Reihe ($1 \times n$)
- Spaltenvektor: eine Spalte ($n \times 1$)

Indizierung numerischer Datentypen

- Gilt strenggenommen für alle geordneten Listen

  Geordnete Liste Struktur, deren Felder über einen ganzzahligen numerischen Index adressiert werden.

- MATLAB-Spezifika
  - Indices in runden Klammern, *immer* positiv, beginnen mit 1
  - Reihenfolge bei zweidimensionalen Matrizen: Reihe, Spalte
- Spezielle Indices
  - `end` – das letzte Element einer geordneten Liste
  - `:` – alle Elemente einer Dimension
- Zugriff auf Bereiche einer Dimension
  - Über eine Liste oder einen Bereich von Indices

Indizierung numerischer Datentypen

Listing 3: Indizierung geordneter Listen in MATLAB

```
1  % Matrix
2  matrix = [ 1 2 3 ; 2 3 4 ; 3 4 5 ; 4 5 6 ];
3
4  % Erste Reihe der Matrix
5  firstRow = matrix(1,:);
6
7  % Zweite Spalte der Matrix
8  firstRow = matrix(:,2);
9
10 % Zweite und dritte Reihe der dritten Spalte
11 selection = matrix(2:3,3);
12 selection = matrix([2,3],3);
13
14 % Letzte Reihe der Matrix
15 lastRow = matrix(end,:);
16
17 % Erste bis vorletzte Spalte der Matrix
18 selection = matrix(:,1:end-1);
```

Genauigkeit von Gleitkommazahlen

- Grundsätzliches Problem
  - Es gibt unendlich viele rationale (und reelle) Zahlen.
  - Unendliche Genauigkeit erforderte unendlich viel Platz.
  - Begrenzter Speicher erlaubt nur endliche Genauigkeit.
- Die Situation vor 1985
  - Jede Architektur hatte ihre eigene Lösung.
  - Die Genauigkeit war rechnerabhängig.
  - Ergebnisse numerischer Rechnungen waren inkompatibel.
- Der Standard ANSI/IEEE 754-1985
  - Standard für die Repräsentation von Gleitkommazahlen
  - Zumindest für 64-bit-Repräsentation eindeutig

Zeichen und Zeichenketten

- `character`, `char`
  - Einzelnes Zeichen
- `string`
  - Zeichenkette aus einzelnen Zeichen (`char`)
- Zeichenketten in MATLAB
  - Zeichenketten immer in Hochkommata ('...') eingeschlossen
  - Mehrdimensionale Strings (Array von Zeichenketten): Reihen müssen gleiche Spaltenzahl haben

☛ Mehrzeilige Texte in `cell arrays` ablegen

Boolesche Ausdrücke

- Zwei Werte
  - wahr (*true*), unwahr (*false*)
- Rückgabewerte relationaler Operatoren
- MATLAB
  - `true`, `false`
  - `0` gilt als `false`
  - $\neq$`0` gilt als `true`

- Entscheidungsstrukturen
- logische Indizierung

George Boole
(1815–1864)

Mathematical Association of America

Komplexe Datentypen

- `cell array` (Liste)
  - Daten unterschiedlicher Typen und Größen
  - In den Feldern eines Datenfeldes (*array*) gespeichert
  - „Generalisiertes“ Datenfeld (*array*)
  - Felder numerisch (mit ganzen Zahlen) indiziert
- `structure` (Wörterbuch)
  - Daten unterschiedlicher Typen und Größen
  - In den Feldern einer Struktur gespeichert
  - Assoziatives Datenfeld
  - Felder mit Namen (*strings*) indiziert

☛ Beide sind hierarchisch verschachtelbar.

Komplexe Datentypen

Geordnete Listen

#	Wert
1	0.0000
2	0.0025
3	0.0050
⋮	
n-1	0.2475
n	0.2500

#	Wert
1	'Im'
2	'Anfang'
3	'war'
⋮	
n-1	'die'
n	'Tat'

Assoziative Datenfelder

Schlüssel	Wert
Name	'K. Racht'
Alter	42
Adresse	
Hobbies	{'...','...'}

Schlüssel	Wert
Straße	'Talstraße'
Nummer	21

cell arrays

Listing 4: cell arrays in MATLAB

```
% Empty cell array
C = cell(0);

% Cell array of strings
C = {'Im','Anfang','war'};

% Cell array with different types
C = {'String',[1 2 3],'String',[1 2 3; 2 3 4; 3 4 5]};

% Same cell array as above
C{1} = 'String';
C{2} = [1 2 3];
C{3} = 'String';
C{4} = [1 2 3; 2 3 4; 3 4 5];

% Accessing a field
foo = C{1}; % Returns a string
foo = C(1); % Returns a 1x1 cell
```

structures

Listing 5: structures in MATLAB

```
% Empty structure
S = struct();

% Structure with some field and value
S.field = 'value'

% Structure with fields of different types
S.field1 = 'value';
S.field2 = pi;
S.field3 = [1 2 3];

% Same structure as above
S = struct(...
    'field1','value',...
    'field2',pi,...
    'field3',[1 2 3] ...
    );

% Accessing a field
foo = S.field3;
```

Konditionale Strukturen

- `if...else`
  - Testet auf bestimmte Bedingung
  - Mehrere Bedingungen über logische Operatoren verknüpft

Logische Operatoren

- Arten logischer Operatoren
  - AND (`&`), OR (`|`), EQUAL (`eq()`, `==`), NOT (`not()`,`~`)
  - Klammern zur Gruppierung logischer Ausdrücke
- „Kurzschluss-Operatoren“
  - `&&`, `||`
  - Überprüfung bricht ab, sobald die Bedingung erfüllt ist
  - Beispiel: `A && B && C` bricht nach `A` ab, wenn `A` unwahr

Listing 6: Einfachste Form einer if-Struktur in MATLAB

```
1 if <condition>
2     % do something
3 end
```

- Abbruch der weiteren Abarbeitung und Rückkehr zum Aufrufer über `return`

Listing 7: if-Struktur mit Alternativzweig

```
1 if <condition>
2     % do something
3 else
4     % do something else
5 end
```

☛ Tipp: Invertierte Logik spart häufig den `else`-Zweig

Entscheidungsstrukturen – Zwei praktische Beispiele

Listing 8: Reales Beispiel einer if-Struktur in MATLAB

```
% Compare current year
if str2double(datestr(now,'yyyy')) < 2016
    disp('You''re outdated.');
else
    disp('You''re in time.');
end

% "now"        - returns current date and time
% "datestring" - formats date - here, "yyyy" means four-digit year only
% "str2double" - converts string into number for comparison
```

Listing 9: Überprüfung der Zahl der Eingabeparameter

```
% nargin returns the number of input arguments of a function
if nargin ~= 2
    return;
end
```

Listing 10: if-Struktur mit mehreren Alternativbedingungen

```
1 if <condition1>
2     % do something
3 elseif <condition2>
4     % do something else
5 else
6     % do something else
7 end
```

Listing 11: if-Strukturen lassen sich verschachteln

```
1  if <condition1>
2      if <additionalCondition>
3          % do something
4      else
5          % do something else
6      end
7  elseif <condition2>
8      % do whatever
9  else
10     % give up
11 end
```

Fallunterscheidungen

- `switch...case`
- Vorteile gegenüber `if...elseif...else`
  - Oft übersichtlicher
  - Gut für Unterscheidung mehrerer Fälle (>2) geeignet
- Beschränkungen von MATLAB
  - Nur Skalare oder Zeichenketten (*Strings*) als Schalter
  - Keine Bedingungen in den Fällen

☛ Sollten immer einen `otherwise`-Zweig haben, um definierte Bedingungen zu schaffen.

Listing 12: Einfachste Form einer switch-case-Struktur in MATLAB

```
1 switch switch_expression
2     case case_expression
3         statements
4     case case_expression
5         statements
6     % ...
7 end
```

Listing 13: switch-case-Struktur mit otherwise-Zweig

```
1 switch switch_expression
2     case case_expression
3         statements
4     case case_expression
5         statements
6     % ...
7     otherwise
8        statements
9 end
```

Schleifen

- `for`-Schleifen
  - Definierte Anzahl an Schleifendurchläufen
  - Typisches Einsatzgebiet:
    Iterieren über die Elemente eines Vektors
- `while`-Schleifen
  - Schleifendurchlauf, solange eine Bedingung wahr ist
  - Typisches Einsatzgebiet:
    Zeilenweises Einlesen einer Datei

☛ Abbruch einer Schleife über `break`

`for`-Schleifen

Listing 14: Einfachste Form einer for-Schleife in MATLAB

```
1 for loopIndex = start : stop
2     % Do something
3 end
```

- `loopIndex` wird in jedem Durchlauf um 1 erhöht

Listing 15: for-Schleife mit angegebenem Inkrement

```
1 for loopIndex = start : increment : stop
2     % Do something
3 end
```

- `increment` kann negativ und nicht-ganzzahlig sein

`for`-Schleifen: Ein praktisches Beispiel

Listing 16: Iterieren über alle Elemente eines Vektors

```
1 % Define vector
2 x = 1:0.1:2*pi;
3
4 % Loop over each element of vector x
5 for k = 1 : length(x)
6     y = sin(x(k));
7 end
```

- Anmerkungen
  - Als Laufvariable *nie* `i` oder `j` verwenden (komplexe Zahl)
  - Als Laufvariable *nie* `l` verwenden (`1` oder `l`?)
- Eigenheiten von MATLAB
  - `for`-Schleifen sind langsam, lassen sich oft vermeiden

`while`-Schleifen

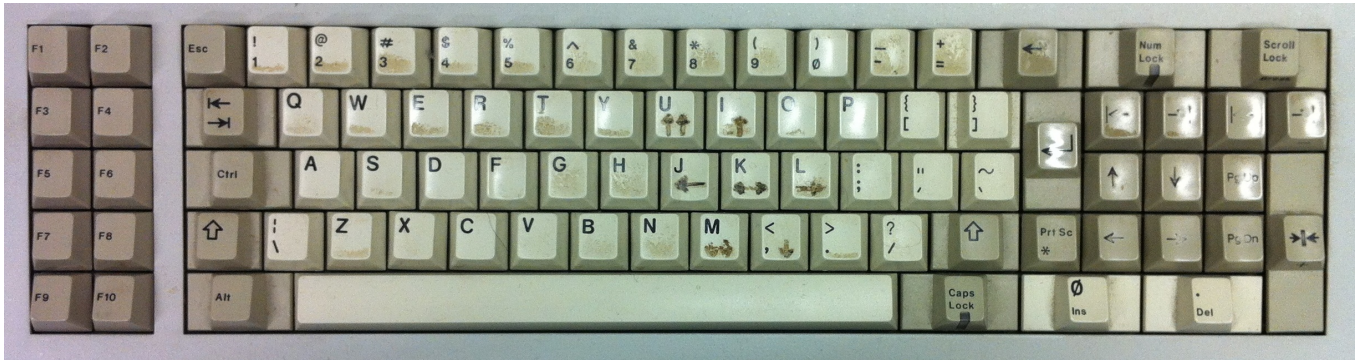
Listing 17: Einfachste Form einer while-Schleife in MATLAB

```
1 while condition
2     % Do something
3 end
```

- Bedingung wird am Anfang jedes Durchlaufs überprüft.
  - MATLAB kennt (anders als andere Sprachen) keine Schleifen, die die Bedingung erst am Ende überprüfen.
- Bedingung muss sich innerhalb der Schleife ändern.
  - Wird sonst zur „Endlosschleife“

Ausblick

Wie geht's weiter?



...Zeit für eigene praktische Arbeit...

Vorschau: Dokumentation

- Dokumentation
- Dokumentation im Code
- Probleme von Dokumentation