Einführung in Matlab und Python - Einheit 2

Programmieren, Datenstrukturen

Jochen Schulz

Georg-August Universität Göttingen

Aufbau

1. **Programmieren**
   - Schleifen
   - Bedingungen
   - Allgemeines

2. **Datenstrukturen**
   - Zahlen
   - Container
   - Chars und Strings

Aufbau

for - Schleife

```
for <indexvariable> = <Ausdruck>
  <Befehle>
end
```

```
for <indexvariable> in <Liste>:
  <Befehle>
```

```
[<Befehl> for <indexvariable> in <Liste>]
```

Bemerkungen:

- Ausdruck: `start:stepsize:end` oder Vektoren, Matrizen.
- Befehle einrücken (Übersichtlichkeit; in Python Pflicht!)

Schleifen - Beispiele

- Berechne $\sum_{i=1}^{1000} \frac{1}{i}$

```
sum=0; for j=1:1000, sum=sum+1/j; end, sum
```

```
sum([1/j for j in range(1,1000)])
```

```
sum =  7.4855
```

- Berechnen dreier Werte

```
for x=[pi/6 pi/4 pi/3], sin(x), end
```

```
[sin(x) for x in [pi/6,pi/4,pi/3]]
```

```
ans =    0.5000
ans =    0.7071
ans =    0.8660
```

Schleifen - Beispiele II

Matrix als *Ausdruck* bzw. als Schleifeniterator

```
for x=eye(3),  x' ,end
```

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

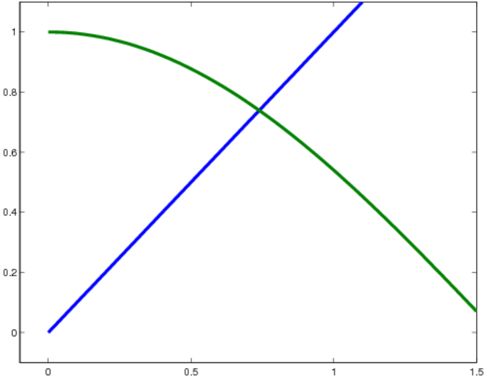
```
for x in eye(3,3): print x;
```

```
[ 1.  0.  0.]
[ 0.  1.  0.]
[ 0.  0.  1.]
```

Fixpunkt

Suche ein $x_f \in \mathbb{R}$ so dass

$$x_f = \cos(x_f)$$



Voraussetzung: Abbildung kontrahierend

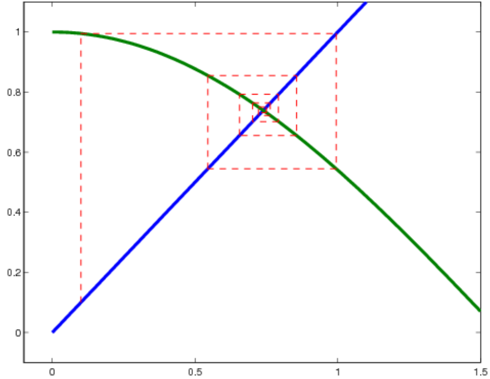
$$|f(x) - f(y)| \leq C\,|x - y|\,,\; C < 1\,\forall x, y \in I$$

Fixpunkt-Iteration

Fixpunkt-Iteration

$$x_{k+1} = cos(x_k)$$

bei geeignetem Startwert x_0.



(Funktioniert wenn die Abbildung kontrahierend ist)

Matlab: Fixpunkt-Iteration

```
% Plot 1
x = linspace(0,1.5,50);
y = cos(x);
plot(x,x,x,y,'LineWidth',3),
axis([-0.1 1.5 -0.1 1.1]);
hold on;
pause; % stoppt bis eine Taste gedrückt wird
z(1) = 0.1; % Anfangswert
it_max = 10; % Iterationsschritte
for i = 1:it_max
    z(i+1) = cos(z(i));
    plot([z(i) z(i)], [z(i) z(i+1)],'r--','LineWidth',1);
    pause;
    plot([z(i) z(i+1)],[z(i+1) z(i+1)],'r--','LineWidth'
       ,1);
    hold on;
    pause; % stoppt bis eine Taste gedrückt wird
end;
```

Python: Fixpunkt-Iteration

```
x = linspace(0,1.5,50)
y = cos(x)
plot(x,x,x,y,linewidth=3)
z = [] # Leere Liste initialisieren
z.append(0.1) # Anfangswert
it_max = 10 # Iterationsschritte
for i in arange(0,it_max):
    z.append( cos(z[i]) )
    plot([z[i], z[i]], [z[i], z[i+1]],'r--',linewidth=1)
    plot([z[i], z[i+1]],[z[i+1], z[i+1]],'r--',linewidth
        =1)
```

Einige Grafikbefehle

- `figure()`
  startet ein Grafik-Fenster.
- `hold on`
  alle Grafiken in einem Fenstier werden übereinander gezeichnet.
  (Python: default!)
- `hold off` (Default)
  bestehende Grafik wird gelöscht und durch die neue Grafik ersetzt.
  (Python: jeweils neue figures erzeugen)

Vandermonde-Matrix I

Berechne zu einem gegebenen Vektor $x = (x_1, \ldots, x_n)$ die Vandermonde-Matrix

$$V := \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix}.$$

Vandermonde-Matrix II

```
function V = vandermonde2(x)
  n = length(x);
  V = zeros(n,n);
  for i = 1:n
    for j = 1:n
      V(i,j) = x(i)^(n-j);
    end
  end
```

```
def vandermonde2(x):
  n = len(x)
  V = zeros((n,n))
  for i in arange(0,n):
    for j in arange(0,n):
      V[i,j] = x[i]**(n-j-1)
  return V
```

Aufbau

Quadratische Gleichung

$$\begin{cases} \text{Suche } x \in \mathbb{R}, \text{ so dass} \\ x^2 + px + q = 0 \end{cases}$$

Fallunterscheidung für $d := \frac{p^2}{4} - q$:

Fall a) : $d > 0$ 2 Lösungen: $x = -\frac{p}{2} \pm \sqrt{d}$

Fall b) : $d = 0$ 1 Lösung: $x = -\frac{p}{2}$

Fall c) : $d < 0$ keine Lösung

Matlab: Implementierung

```
function [anz_loesungen, loesungen]=quad_gl(p,q)
d=p^2/4-q; % Diskriminante
% 2 Loesungen
if d>0
    anz_loesungen=2;
    loesungen=[-p/2-sqrt(d) -p/2+sqrt(d)];
end
% 1 Loesung
if d==0
    anz_loesungen=1;
    loesungen=[-p/2];
end
% 0 Loesungen
if d<0
    anz_loesungen=0;
    loesungen=[];
end
```

Python: Implementierung

```
def quad_gl(p,q):
    d = p**2/4-q # Diskriminante
    # 2 Loesungen
    if d>0:
        anz_loesungen=2
        loesungen=array([-p/2-sqrt(d), -p/2+sqrt(d)])
    # 1 Loesung
    if d==0:
        anz_loesungen=1
        loesungen=array([-p/2])
    # 0 Loesungen
    if d<0:
        anz_loesungen=0
        loesungen=array([])
    return anz_loesungen, loesungen
```

Logische Operationen

- logische Variablen (Datentyp ist logical(Matlab) bool(Python).
- Variablen dieses Typs sind entweder `true` (1) oder `false` (0) (Python: True oder False)
- Matlab: Numerische Werte ungleich 0 werden als `true` gewertet.

```
a = (1<2)
```

```
a = 1
```

```
b = ([ 1 2 3 ] < [ 2 2 2 ])
```

```
b =    1      0      0
```

```
whos
```

```
Name  Size  Bytes  Class
a     1x1   1   logical array
b     1x3   3   logical array
```

Vergleichs-Operatoren

```
a=[1 1 1], b=[0 1 2]
```

Operation	Bedeutung	Ergebnis
`a == b`	gleich	`0 1 0`
`a ~= b`	ungleich	`1 0 1`
`a < b`	kleiner	`0 0 1`
`a > b`	größer	`1 0 0`
`a <= b`	kleiner oder gleich	`0 1 1`
`a >= b`	größer oder gleich	`1 1 0`

Bem: 1 = wahre Aussage, 0 = falsche Aussage

Bem: Komponentenweise Vergleiche sind auch für Matrizen gleicher Größe möglich!

Logische Operatoren

&(Ma) and(Py) |(Ma) or(Py)	logisches und logisches oder	~(Ma) !(Py) xor(Ma)	logisches nicht exklusives oder

Beispiele:

```
x=[-1 1 1]; y=[1 2 -3];
```

```
>> (x>0) & (y>0)
ans =
      0      1      0
```

```
>> ~( (x>0) & (y>0))
ans =
      1      0      1
```

```
>> (x>0) | (y>0)
ans =
      1      1      1
```

```
>> xor(x>0,y>0)
ans =
      1      0      1
```

Bedingung

Matlab

```
if  <Ausdruck>
   <Befehle>
elseif
  <Befehle>
else
   <Befehle>
end
```

Python

```
if <Ausdruck>:
  <Befehle>
elif:
  <Befehle>
else:
  <Befehle>
```

Die Befehle zwischen `if` und `end` (bzw. nach : und bis Ende Einrückung) werden ausgeführt, wenn der *Ausdruck* wahr (True) ist.
Sonst werden die `elseif`-Bedingungen geprüft, und falls keine dieser wahr ist, wird (soweit vorhanden) der `else`-Block ausgeführt.
Hinweis: *Ausdruck* ist wahr, wenn alle Einträge von *Ausdruck* ungleich 0 sind.

While-Schleifen

```
while <Ausdruck>
  <Befehle>
end
```

```
while <Ausdruck>:
  <Befehle>
```

Die Befehle werden wiederholt, so lange die Bedingung *Ausdruck* wahr ist.

Beispiel: Berechne $\sum_{i=1}^{1000} \frac{1}{i}$.

```
n = 1000; sum = 0; i = 1;
while (i <= n)
  sum = sum+(1/i);
  i = i+1;
end
```

```
n = 1000; sum = 0; i = 1;
while (i <= n):
  sum += 1./i
  i += 1
```

Größter gemeins. Teiler (ggT)

Berechnung des ggT von natürlichen Zahlen a und b mit Hilfe des euklidischen Algorithmus

Idee: Es gilt $ggT(a, b) = ggT(a, b - a)$ für $a < b$.

Algorithmus

Wiederhole, bis $a = b$

- Ist $a > b$, so $a = a - b$.
- Ist $a < b$, so $b = b - a$

Implementierung

```
function a = ggt(a,b)
while (a ~= b)
  if (a > b)
    a = a-b;
  else
    b = b-a;
  end
end
```

```
def ggt(a,b):
  while (a != b):
    if (a > b):
      a -= b
    else:
      b -= a
  return a
```

break

- Der Befehl **`break`** verläßt die **`while`** oder **`for`**-Schleife.

```
x=1;
while 1
  xmin=x;
  x=x/2;
  if x==0
    break
  end
end
xmin
```

```
xmin = 4.9407e-324
```

continue

- Durch `continue` springt man sofort in die nächste Iteration der Schleife, ohne die restlichen Befehle zu durchlaufen.

```
for i=1:10
  if i<5
    continue
  end
  x(i)=i;
end
x
```

```
x =   0   0   0   0   5   6   7   8   9   10
```

Aufbau

Operator Rangfolge

Level	Operator
1	Exponent (^, .^), transpose
2	logische Verneinung (~,!)
3	Multiplikation (*,.*), Division (/,./,\, .\)
4	Addition (+), Subtraktion (-)
5	Doppelpunktoperator (:)
6	Vergleichsoperatoren (<,>,<=,>=,==,~=,!=)
7	Logisches und (&, and)
8	Logisches oder (|, or)

Bei gleicher Rangfolge wird von links nach rechts ausgewertet.

Die Rangfolge kann durch Klammersetzung geändert werden.

globale Variablen

Mittels des Befehls `global` können Variablen des globalen Workspace auch für Funktionen manipulierbar gemacht werden.

Funktion

```
function f=myfun(x)
% myfun.m
% f(x)=x^alpha sin(1/x)

global alpha
f=x.^alpha.*sin(1./x);
```

Plotten

```
% plot_myfun
global alpha
alpha_w=[ 0.4 0. 6 1 1.5
    2];
for i = 1:length(alpha_w)
    alpha = alpha_w(i);
    fplot(@myfun,[0.1,1])
    hold on;
end
hold off;
```

Ein Guter Stil I

- Beschreibung: Alle Programme/Funktionen sollten zu Beginn einen Kommentar enthalten, in dem beschrieben wird, was das Programm macht.
  - Programmbeschreibung
  - Eingabevariablen
  - Ausgabevariablen
  - Beispiele
- Vor und nach logischen Operatoren und $=$ sollte ein Leerzeichen gesetzt werden.
- Man sollte pro Zeile nur einen Befehl verwenden.
- Befehle in Strukturen, wie `if`, `for` oder `while`, sollten eingerückt werden (Python erzwingt dies sowieso)

Ein Guter Stil II

- Die Namen der Variablen sollten, soweit möglich, selbsterklärend sein.
- Verfasst man umfangreiche Programme, so sollten Funktionalitäten, die eine logische Einheit bilden in einen separaten Datei, Unterverzeichnis oder Modul gelegt werden.
- Potenzielle Fehler sollten, soweit möglich, aufgefangen werden. Speziell sollten die Eingabeparameter der Funktionen geprüft werden.

Aufbau

Datentypen

- Datentypen werden bestimmt durch ihre Eigenschaften.
- Zuweisung des Datentyps ist *implizit*.
- Operationen können Typen ändern.
- Achtung: Daher ist nicht immer klar, welchen Typ eine Variablen gerade hat.

Aufbau

Gleitkommazahlen / Maschinengenauigkeit

- Standard-Datentyp ist ein Array von Gleitkommazahlen (`double`).
- Abstand von 1 zur nächsten Gleitkommazahl: $\epsilon = 2^{-52} \sim 2.2 \cdot 10^{-16}$ (`eps`(Matlab), `spacing(1)`(Python))
- Sei $x \in \mathbb{R}$ eine reelle Zahl und $\tilde{x}$ die Darstellung im Computer. Dann gilt für den Rundungsfehler
$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{1}{2}\epsilon.$$
- größte bzw. kleinste darstellbare positive Zahl `realmin`(Matlab) bzw. `realmax`(Matlab) und `sys.float_info` (Python)

Ausnahmen

- Ist eine Zahl größer als `realmax`, so meldet MATLAB einen 'Overflow' und gibt als Ergebnis `Inf` zurück.

```
realmax*1.1
```

```
  ans =    Inf
```

- Bei Operationen wie $0/0$ oder ∞/∞, erhält man als Ergebnis `NaN` (*Not a Number*).

```
0/0
```

```
Warning: Divide by zero.
ans =    NaN
```

Umgang mit NaN und Inf

- Mit Hilfe von `isinf` und `isnan` kann auf ∞ bzw. NaN getestet werden.

```
isnan(0/0), isinf(1.2*realmax)
```

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

- Matlab: Test auf NaN durch $==$ ist nicht möglich

```
0/0 == NaN
```

```
ans =      0
```

- Matlab: Bei Inf ist der Test durch $==$ möglich!

```
(1.2*realmax)==Inf
```

```
ans =      1
```

Darstellungsformate am Beispiel $1/7$

`format short`	0.1429
`format short e`	1.4286e-01
`format short g`	0.14286
`format long`	0.142857142857143
`format long g`	0.142857142857143
`format long e`	1.428571428571428e-01

Das Default-Format ist `short`.

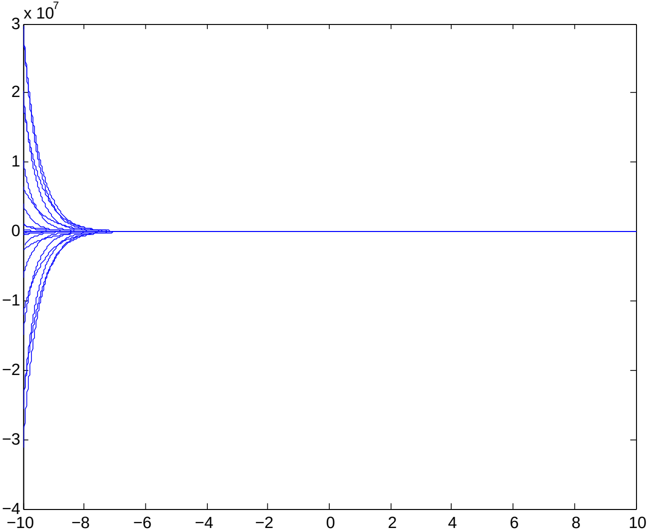
Beispiel - Berechnung von e

Approximation der Exponentialfunktion durch eine Taylor-Reihe

$$P_n(x) = \sum_{j=0}^{n} \frac{x^j}{j!}$$

```
x = -10:0.01:10; % die x-Werte
expx = exp(x); % die wahre Exponentialfunktion
for n=0:1:25
    % so viele Nullen wie x Elemente hat
    sum=zeros(size(x));
    for j=0:n
        % das berechnet die Partialsumme
        sum=sum+x.^j/factorial(j);
    end
    % plottet relativen Fehler
    plot(x,(sum-expx)./expx);
    % wir plotten alles uebereinander
    hold on
end
```

Berechnung von e - Relativer Fehler



Auslöschung

```
% Ausloeschung, mit 6 Dezimalstellen
format long g % sorgt fuer lange Ausgabezahlen
x = 0.344152
xwahr = 0.344152*1.0000001 % Fehler
relfx = abs(xwahr-x)/xwahr
y = 0.344135
z = x-y
zwahr = xwahr-y
relfz = abs(z-zwahr)/abs(zwahr) % relativer Fehler von z
```

```
x = 0.344152
xwahr = 0.3441520344152
relfx = 9.99999900671778e-08
y = 0.344135
z = 1.69999999999892e-05
zwahr = 1.70344152000124e-05
relfz = 0.00202033352005498
```

Komplexe Zahlen

Komplexe Zahlen $z \in \mathbb{C}$ haben die Form

$$z = x + iy, \quad x, y \in \mathbb{R}$$

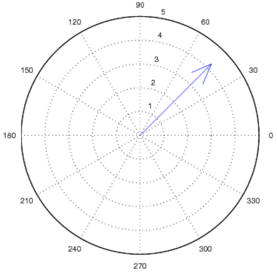
mit $i = \sqrt{-1}$.

- `i,j`(Matlab) `1j`(Python): vordefinierte Variablen für $\sqrt{-1}$.
- `complex(x,y)`: Erzeugung der komplexen Zahl $x + iy$ aus $x, y \in \mathbb{R}$.
- `real(z)`: Realteil für $z = x + iy \in \mathbb{C}$
- `imag(z)`: Imaginärteil für $z = x + iy \in \mathbb{C}$

Polarkoordinaten

$$z \in \mathbb{C}, \quad z = re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$$

- `abs(z)` ergibt den Betrag r von z.
- φ erhält man durch `angle(z)`.
- Matlab: grafische Darst.: `compass(z)` ($z = 3 + 3i$).



Aufbau

Matlab: Structures

(Python: Listen oder Wörterbücher)

Structures:

Strukturen sind eine Möglichkeit verschiedene Objekte in einer Datenstruktur zu bündeln.

Beispiel: komplexe Zahlen

```
komp_Zahl.real=1;
komp_Zahl.imag=1;
komp_Zahl
```

```
komp_Zahl =

    real: 1
    imag: 1
```

Matlab: Structures II

- Alternativ können Strukturen durch

```
struktur = struct('Feld1',<Wert1>,'Feld2',<Wert2>,..)
```

  definiert werden.
- Ein Feld einer Struktur `struktur` kann durch

```
struc2 = rmfield( <struktur> ,'Feld')
```

  gelöscht werden.

Matlab: Cell Arrays

(Python: Listen)

Cell Arrays:

Cell Arrays sind spezielle Matrizen, deren Einträge aus unterschiedlichen Datentypen bestehen können. Erzeugt werden sie durch geschweifte Klammern.

```
C = { 1:10, hilb(4);...
       'Hilbert Matrix', pi}
```

```
C =
        [1x10 double]     [4x4 double]
     'Hilbert Matrix'     [    3.1416]
```

Matlab: Befehle für Cell Arrays

- Zugriff auf Cell-Arrays:

```
C{2,1}
```

```
ans =
Hilbert Matrix
```

```
C{1,2}(2,3)
```

```
ans =
    0.2500
```

- `celldisp(C)`: Der Inhalt von C wird dargestellt.
- `cellplot(C)` stellt C grafisch dar.

Python: Wörterbücher (Dictionaries)

- Index kann Namen enthalten.
- Sind gut geeignet für das Speichern großer Datenmengen, da der indizierte Zugriff sehr schnell ist.
- der Index ist eindeutig
- T = {}: leeres Dictionary
- Wird ein Index nicht gefunden, gibt es eine Fehlermeldung
- T.pop(): Entnehmen (Löschen und Zurückgeben) von Einträgen
- Iterieren:

```python
d = {'a': 1, 'b':1.2, 'c':1j}
for key, val in d.iteritems():
  print key,val
```

```
a 1
c 1j
b 1.2
```

Aufbau

Strings

Characters (char) - Zeichen

- Darstellung durch Integer
- Die Werte zwischen 0 und 128 entsprechen den ASCII Werten (Stichwort Encoding).
- 2 Bytes Speicherbedarf $\Rightarrow$ Zahl zwischen 0 und $2^{16} - 1$

```
s='d'
```

```
s = d
```

Strings - Vektoren von Zeichen:

- Die Zeichen werden wiederum durch die ASCII Werte dargestellt.

```
s='AB6de*'
```

```
s =
AB6de*
```

Befehle für Strings

- Verbinden von Strings: `strcat`(Matlab) $+$(Python)

```
strcat('Hello',' world')
```

```
'Hello'+' world'
```

```
Hello world
```

- `num2str(x,n)`(Matlab) konvertiert x in einen String mit n signifikanten Stellen. (Default: $n = 4$)
- `str(x)`(Python) konvertiert x in einen String.
- `strcmp(s,t)`(Matlab) `s==t`(Python) vergleicht die Strings s und t.

Matlab: Einstieg Standardausgabe

```
fprintf ('Text %<format> und %<format> ... ', x,y,...)
```

Auszug Formatspezifikation:

- `%i` : integer
- `%f` : float
- `%s` : strings

Beispiel:

```
fprintf ('Pi mit %i Nachkomma - Stellen : %f \n',6,pi)
```

```
Pi mit 6 Nachkomma - Stellen : 3.141593
```

Python: Einstieg Standardausgabe

```
"Text {<format>} und {<format>} ... ".format(x,y,...)
```

<format>: kann leer bleiben oder eine Reihenfolge enthalten. Es kann beliebig viele format-Platzhalter geben. (Später Formatspezifikation)

Beispiel:

```
x = 4
y = 6
print ("x ist {0} und y ist {1}".format(x,y))
["x{}".format(k) for k in arange(1,3)]
```

```
x ist 4 und y ist 6
['x1', 'x2']
```