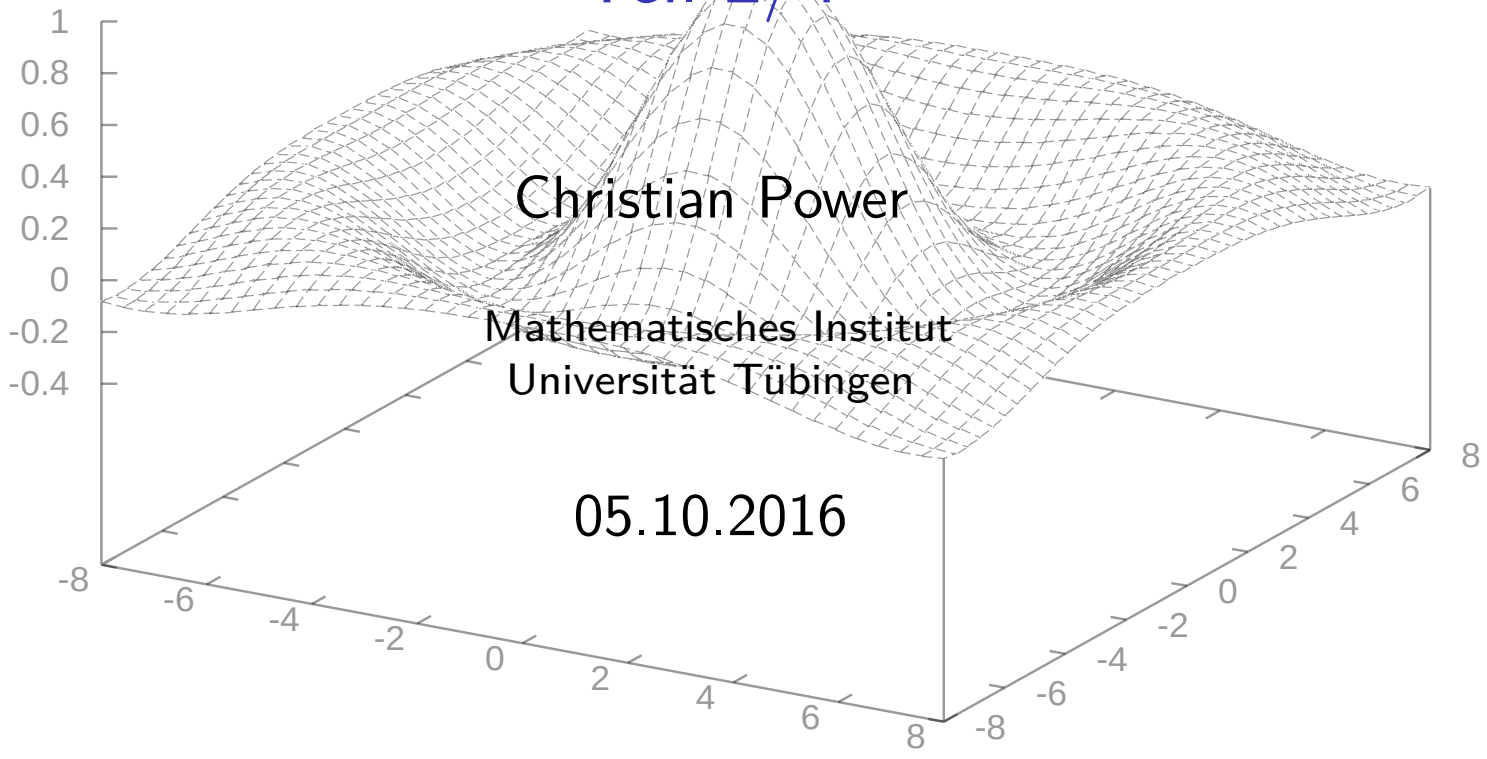
Programmiervorkurs für die Numerik
Teil 2/4



Christian Power

Mathematisches Institut
Universität Tübingen

05.10.2016

Wiederholung

- Allgemeines zu MATLAB
- Anweisungen
  - Operatoren (arithmetische Op. z.B. + −, relationale Op. z.B. < > und logische Op. z.B. &)
  - Kontrollanweisungen (Schleifen, Verzweigungen)
  - Ausgabe von Variablen

Gliederung

Lösungen für die Aufgaben des 1. Übungsblattes

Aufgabe 4: mündlich

Aufgabe 5:

```
1 for i = 1:10
2   println(i)
3 end
```

Aufgabe 6:

```
1 for i = 0:10
2   println(2*i+1)
3 end
```

Aufgabe 7: siehe Beispiel auf Folien des 1. Tages.

Lösungen für die Aufgaben des 1. Übungsblattes

Aufgabe 8:

```
1  n = 10;
2  S = 0;
3  for i = 0 : n
4     S += .5^i
5  end
6  println(S)
```

Aufgabe 9:

```
1  n = 10;
2  q = .25;
3  S = 0;
4  for i = 0 : n
5     S += q^i
6  end
7  println(S)
```

Lösungen für die Aufgaben des 1. Übungsblattes

Aufgabe 10:

```
limes = 2;
maxit = 30;
tol = .01;
approx = 0; # Partialsumme Σ_{i=0}^n .5^i
for n = 0:maxit  # Keine Endlosschleife
  approx += .5^n
  # Abbr. Bed.
  if abs(limes - approx) < tol
    println(n)
    break
  end
  if n == maxit # letzte Iteration
    println("Zu viele Iterationen")
  end
end
```

Wozu Funktionen?

- Funktionen sind das Hauptwerkzeug um ein Programm zu Strukturieren.
- Gute Namen für Funktionen dokumentieren automatisch deinen Code.
- Die Fehler im Code korrelieren stark mit der Komplexität und der Länge des Codes. Benutze Funktionen, um einen langen Code in kleinere überschaubare Teile zu zerlegen.
- *Faustregel*: Eine Funktion sollte ca. sieben Zeilen lang sein, aber auf keinen Fall länger wie der Bildschirm.

Funktionen und Skripte

MATLAB unterscheidet zwischen Skripten und Funktionen:

- **Skripte** sind Textdateien, die Anweisungen enthalten, die man genauso gut im Command Window einzeln eintippen könnte.
- **Funktionen** sind auch Textdateien mit folgenden unterschieden zu Skriptdateien:
  - Funktionsdataien enthalten spezielle Anweisungen: In der ersten Zeile steht als erstes das Schlüsselwort `function` und weiter hinten in der Zeile der zum Dateinamen gleichlautende Funktionenname.
  - Funktionen können so angelegt werden, dass sie beim Aufruf ein oder mehrere *Funktionsargumente* erwarten z.B. `f(x)`.
  - Funktionen können so angelegt werden, dass sie nach ihrer Beendigung *Rückgabewerte* an den Aufrufer liefern.
  - Es besteht die Möglichkeit, über spezielle Kommentare, eine Hilfe für die Funktion in MATLAB zu importieren.

Beispiel einer Funktion

in MATLAB

```
 1  function [ar, um] = area(a, b)
 2  % Berechne Flaeche und Umfang
 3  %{
 4      Precondition:
 5      - Skalar a,b > 0
 6      Postcondition:
 7      - Skalar ar, um > 0
 8  %}
 9  assert(a>0 & b>0)
10  ar = a * b;
11  assert(ar>0)
12  um = 2 * a + 2 * b;
13  assert(um > 0)
14  end
```

Erläuterung

- Der Dateiname der im obigen Bsp. definierten Funktion namens `area` lautet `area.m`.
- Die Funktion liefert als Rückgabewert in einem Feld zwei Variablen passenden Typs zurück: `ar` und `um`.
- Die Funktion erwartet beim Aufruf zwei Variablen passenden Typs: `a` und `b`
- Bei MATLAB müssen die Rückgabewerte nicht speziell irgendwo gesetzt werden oder speziell zurückgegeben werden.
- Erwartet die Funktion keine Argumente, bleibt die runde Klammer hinter dem Funktionsnamen leer `()`.
- Gibt die Funktion keine Arugment zurück bleibt die eckige Klammer hinter `function` leer `[]`.

Beispiel einer Funktion

in Julia

````
"""
(ar,um) = area(a,b)
Berechne *Flaeche* (ar) und *Umfang* (um).

- **Precondition**: Skalare a,b > 0
- **Postcondition**: Skalare ar, um > 0

# Beispiel
```julia
julia> area(1,1)
(1,4)
```
"""
function area(a, b)
````

Beispiel einer Funktion

in Julia

```
1  function area(a, b)
2      assert(a>0 && b>0)
3      ar = a * b
4      assert(ar>0)
5      um = 2*a + 2*b
6      assert(um>0)
7      return (ar, um)
8  end
```

Erläuterung

- Der Text vor der Funktion wird steht für die Dokumentation zur Verfügung.
- Bei Julia werden am Anfang keine Rückgabewerte angegeben werden. Mit `return` kann man mehrere Rückgabewerte angeben.
- Erwartet die Funktion keine Argumente, bleibt die runde Klammer hinter dem Funktionsnamen leer `()`.

Beispiel

```
g1 = 3;
[a,b] = area(g1, 2)

include("area.jl")  # Lade Defi.
g1 = 3
(a, b) = area(g1, 2)
```

Variablen I

MATLAB und Julia ist ein auf Matrizen basierendes Werkzeug. Alle Daten die in MATLAB eingegeben werden, werden als Matrix oder mehrdimensionales Array abgespeichert.

- In MATLAB sind Skalare und Vektoren nur der Spezialfall von Matrizen.
- Skalare sind (1×1)-Matrizen.
- Vektoren der Dimension n sind $(1 \times n)$ oder $n \times 1$-Matrizen.
- MATLAB und Julia nummerieren die Elemente beginnend von 1 nicht mit 0!
- Mit dem Befehl `whos` bzw. `whos()` kann man rausfinden, welche Variablennamen schon vergeben wurden.

Initialisierung

von Vektoren

```
a = [1, 2, 3, 4] % Zeilenvektor der Dim. 4
b = [1 2 3 4]    % a == b
c = [1; 2]       % Spaltenvektor der Dim. 2
d = 1:.1:2       % ZEILENvektor der Dim. 11
a = []           % leerer Vektor

a = [1, 2, 3, 4]     # Spaltenvek. Dim. 4
b = [1 2 3 4]        # Zeilenvek. Dim. 4
c = [1; 2]           # Spaltenvek. Dim. 2
d = collect(1:.1:2) # Spaltenvek. Dim. 11
a = []               # Leerer Vektor
```

Der letzte Ausdruck kann zum Löschen einer Variable (hier Vektor a) benutzt werden.

Zugriffsarten

auf Vektoren

- als Ganzes:

```
g = a;          % g == a

g = copy(a);    # g == a
```

- Elementweise bei MATLAB

```
g = a(1);     % g ist ein Skalar
i = 2;
g = a(i);     % g enthaelt i-tes Element
g = a(end); % g enthaelt letztes Element
```

Zugriffsarten

auf Vektoren

- Elementweise bei Julia

```
g = a[1];    # g ist ein Skalar
i = 2;
g = a[i];    # g enthaelt i-tes Element
g = a[end]; # g enthaelt letztes Element
```

- Bereichswahl:

```
g = a(1:3); % length(g) == 3

g = a[1:3]; # Spaltenvektor, length(g)==3
```

Wichtige Methoden

für Vektoren

Wichtig: Indizes können nur *natürliche Zahlen* ohne Null sein!

- Transponieren (’):

```
x = [1 2 3]; % Zeilenvektor
x = x';      % Spaltenvektor
```

- Länge eines Vektors ermitteln z.B. für Schleifen

```
a = [3 7 3];  # 3 Elemente
for i = 1:length(a)
    println(a[i])
end
```

Beispiel

Mittelwertberechnung

Berechnung des Mittelwertes $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$ für $n = 4$ und die Werte $x = (x_1, \ldots, x_4) = (3, 7, 5, 1)$.

```
1  input = [3 7 5 1];
2  size = length(input);
3  summe = 0;
4  # Berechne summe
5  for i = 1 : size
6      summe += input[i]
7  end
8  mittelwert = summe / size;
9  println(mittelwert)
```

Operatoren

auf Vektoren

Operatoren wirken auf Vektoren anders als auf Skalare. Unter anderem existieren folgende Operatoren auf Vektoren, wobei die Vektoren dafür dieselbe Dimension besitzen müssen:

- Zuweisungsoperator: `=`
  `a = b` $\Longrightarrow$ `a(i)=b(i)` für alle i.
- Arithmetische Operatoren: `+ -`
  `a + b` $\Longrightarrow$ `a(i)+b(i)` für alle i.
- Arithmetische Operatoren: `* a * b` hat zwei Bedeutungen
  1. Falls `a`, `b` Vektoren sind siehe Übungen.
  2. Falls `a` ein Skalar und `b` ein Vektor ist, dann ist gilt `a * b(i)` für alle i.
- relationale Operatoren: $==$ $\sim=$ bzw. bei Julia `== !=`
  `a == b` $\Longrightarrow$ `a(i) == b(i)` für alle i.

Weiter Operatoren sind definiert, werden aber nicht besprochen.

Initialisierung

von Matrizen 1/2

```
A = [1 2 3 4; 5 6 7 8]    % 2 x 4 - Matrix
B = [1 5; 2 6; 3 7; 4 8] % A == B'
```

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

```
A[2,:] = [9 10 11 12]      # Neue 2. Zeile
```

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
A = []                     # Leere Matrix
```

Dieser Ausdruck kann zum Löschen der Matrix benutzt werden.

Initialisierung

von Matrizen 2/2

```
I = eye(3)       # 3 x 3 Einheitsmatrix
A = zeros(3,4)   # 3 x 4 Nullmatrix
B = ones(4,2)    # 4 x 2 Einsmatrix
v = [1;2;3;4]    # Zeilenvektor
D = diagm(v)     # Diagonalmatrix
# Im Zweifelsfall diagm(v[:])

v = [1 2 3 4]    % Ein Vektor
D = diag(v)      % Diagonalmatrix
```

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

Zugriffsarten

auf Matrizen in MATLAB

- als Ganzes:

```
B = A % Kopiere A
```

- Elementweise:

```
g = A(2,1)   % g ist (2,1)-Element
i = 1;
g = A(i,end) % g ist oben rechts
g = A(:,1)   % g ist erste Spalte
```

- Bereichswahl:

```
g = A(:,1)   % g ist erste Spalte
g = A(1,2:3) % length(g) == 2
```

Zugriffsarten

auf Matrizen in Julia

- als Ganzes:

```
B = copy(A) # entspricht Matlab B = A
```

- Elementweise:

```
g = A[2,1]   # g ist (2,1)-Element
i = 1;
g = A[i,end] # g ist oben rechts
g = A[:,1]   # g ist erste Spalte
```

- Bereichswahl:

```
g = A[:,1]   # g ist erste Spalte
g = A[1,2:3] # length(g) == 2
```

Wichtige Methoden

für Matrizen

- Transponieren (’):

```
A = [1 2; 3 4; 5 6]    # 3 x 2 - Matrix
B = copy(A) # entspricht Matlab B = A
```

- Länge eines Vektors ermitteln z.B. für Schleifen

```
A = A’                 # Transponieren

A = A’                 % Transponieren
```

Beispiel

Matrizenmultiplikation

Für eine gegeben $(m \times n)$-Matrix A und eine $(n \times r)$-Matrix B ist die Matrix-Multiplikation folgendermaßen definiert:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj},$$

für $i = 1, \ldots, m$ und $j = 1, \ldots, r$.

Beispiel

Matrizenmultiplikation

```
 1  A = [1 2 3 4; 5 6 7 8]
 2  B = [9 10; 11 12; 13 14; 15 16]
 3  assert(size(A,2)==size(B,1)) # Kompatibel
 4  C = zeros(size(A,1),size(B,2))
 5  for i = 1 : size(C,1)  # Spalte
 6      for j = 1 : size(C,2) # Zeile
 7          # Innere Summe
 8          C[i,j] = 0
 9          for(k = 1 : size(A,2))
10              C[i,j] += A[i,k] * B[k,j]
11          end
12      end
13  end
```

Operatoren

auf Matrizen 1/2

Es gelten die üblichen Rechenregeln für Matrix-Matrix-, Matrix-Vektor- und Skalar-Matrix-Operationen:

- Zuweisungsoperator: `=`
  `A = B` $\Longrightarrow$ `A(i,j) = B(i,j)` für alle i, j.
- Arithmetische Operatoren: `+ -`
  `A + B` $\Longrightarrow$ `A(i,j) + B(i,j)` für alle i, j.

Operatoren

auf Matrizen 2/2

- Arithmetischer Operator: `*`
  `A * B` hat drei Bedeutungen
  1. Falls `A` eine Matrix und `B` eine Matrix ist, dann haben wir eine Matrixmultiplikation. Siehe Beispiel.
  2. Falls `A` ein Skalar und `B` eine Matrix ist, dann gilt `A * B(i)` für alle i.
  3. Falls `A` eine $(m \times n)$-Matrix ist und `B` eine $(n \times 1)$-Vektor ist, dann haben wir eine Matrix-Vektor-Multiplikation.
- Relationale Operatoren: $==$ $\sim=$ bzw. bei Julia `==` `!=`
  `A == B` $\Longrightarrow$ `A(i,j) == B(i,j)` für alle i, j.
- Oben gibt es einen kleinen Unterschied. Bei MATLAB erzeugt der == Operator eine Matrix mit den Einträgen `0` oder `1`. Bei Julia wird einfach der Wert `true` oder `false` zurück gegeben. Möchte man den von MATLAB Operator, dann muss man `.==` benutzen. Analog für die anderen Operatoren.

Variable II

- Löschen von Variablen bei MATLAB (sinnvoll wenn Variable andere Dimension erhalten soll)

```
A = 5
clear A    % Vernichte A
A = 7
A = []     % Pseudo vernichte A
clear all % Vernichte alles
```

- Julia besitzt kein `clear`, da es über ein *Garbage collector* verfügt. `workspace()` entspricht aber `clear all`.
- Die Variable `ans` nimmt Ergebnis einer Berechnung auf, wenn keine Ziel-Variable angegeben wurde.