1 Kontrollstrukturen

Verzweigung

Wenn TEST wahr ist, wird der nachfolgende Codeblock (und kein weiterer) ausgeführt. Sowohl `elseif` als auch `else` sind optional. `elseif` kann mehrfach vorkommen. `else` wird ausgeführt, falls keine der vorherigen Bedingungen zutrifft.

```
if (TEST)
  CODE
elseif (TEST)
  CODE
else
  CODE
endif
```

While-Schleife

Wie die Verzweigung, der Codeblock wird aber (als Ganzes!) wiederholt, bis die Abbruchbedingung TEST nicht mehr wahr ist.

```
while (TEST)
  CODE
end
MORE
```

Ausführreihenfolge: TEST ✓ – CODE – TEST ✓ – CODE – ... – TEST ✗ – MORE.

For-Schleife

Die Zählervariable (hier `i`) bekommt bei jedem Durchlauf den nächsten Wert aus dem angegebenen Vektor (hier `1:10`) zugewiesen, bis keine weiteren mehr vorhanden sind.

```
for i=1:10
  CODE
end
MORE
```

Ausführreihenfolge: CODE$|_{i=1}$ – CODE$|_{i=2}$ – CODE$|_{i=3}$ – ... – CODE$|_{i=10}$ – MORE.

Flusskontrolle

Schleifen können frühzeitig verlassen werden.

`continue`

überspringt die verbleibenden Zeilen im Codeblock und macht dann wie gewohnt weiter. In der while-Schleife wird danach wieder die Abbruchbedingung TEST geprüft, in der for-Schleife der nächste Wert des Vektors verwendet.

`break`

bricht die Ausführung der Schleife ganz ab und springt nach das `end` bzw. `endif` zu MORE.

Vergleichende und logische Operatoren

... werden für die TEST-Bedingungen (Logische Ausdrücke) benötigt. Die Rangfolge der Operatoren ist auf der Hilfeseite `doc precedence` angegeben. `E` und `F` sind logische Ausdrücke.

`A == B`	Gleichheit
`A ~= B`	Ungleich
`A < B`	Kleiner als (> für Grösser als)
`A <= B`	Kleiner oder gleich (>=)
`E && F`	Und
`E || F`	Oder
`~ E`	Nicht (Negation)

Als Werte für Wahr und Falsch werden `1` und `0` verwendet (das Resultat des Ausdrucks `4 < 100` ist `1`). Anwendungsbeispiel mit Vergleichsoperator und Zählervariable:

```
count = 0
while (count < 100)
  count = count + 1
  CODE
end
```

Wiederholt CODE 100 Mal.

2 Vektoren und Matrizen

Vektoren

```
r = [1, 2, 3]
r = [1  2  3]
c = [2; 4; 8]
```

Die ersten beiden Befehle sind gleichwertig und erzeugen einen Zeilenvektor, der dritte Befehl erzeugt einen Spaltenvektor. Das Semikolon bedeutet, dass eine neue Zeile beginnt.

$$r = (1\ 2\ 3) \quad c = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Vektoren sind eigentlich $1 \times n$- oder $n \times 1$-Matrizen. Auf das n-te Element wird mit `v(n)` zugegriffen:

`c(3)` → 8

Sequenzen

Sequenzen werden in folgender Form geschrieben:

`low:inc:high`

Damit wird ein Vektor generiert, der bei `low` beginnt und mit `high` endet. Die Schrittweite `inc` ist standardmässig auf 1 gesetzt, wenn sie nicht angegeben wird.

```
    1:3   →   [1  2  3]
  3:-1:1  →   [3  2  1]
0:.4:1.5  →   [0  0.4  0.8  1.2]
```

Im zweiten Fall muss `inc` angegeben werden, `3:1` würde einen leeren Vektor erzeugen. Im letzten Fall wird die 1.6 weggelassen, da sie grösser als `high` ist.

Die führende `0` vor dem Komma darf bei Gleitkommazahlen weggelassen werden: Abgekürzt `0.1 == .1`

Matrizen

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

Erzeugt eine Matrix mit zwei Zeilen und drei Spalten.

$$A = \begin{matrix} \\ m{=}1 \\ m{=}2 \end{matrix} \begin{matrix} \begin{matrix} n{=}1 & n{=}2 & n{=}3 \end{matrix} \\ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \end{matrix}$$

Auf einzelne Elemente wird mit `A(m,n)` zugegriffen.

`A(2,3)` → 6

Auf Zeilen oder Spalten mit Vektoren als Parameter:

```
A(1,[1 3])   →   [1  3]
A(1,2:end)   →   [2  3]
A(1, 1:3 )   →   [1  2  3]
A(1,  :  )   →   [1  2  3]
```

wobei `end` für das letzte Element steht und `:` alleine die ganze Zeile oder Spalte auswählt.

3 Allgemeines

Datentypen

```
x = 1.25    s = 'abc'    A = [1 2; 3 4]    c = 1+2i
```

Von links nach rechts:

- Zahlen (Skalare). Dazu gehören auch `Inf` (Unendlich; `1/0`) und `NaN` (Not-a-Number; `0/0`).
- Text (Strings). Um ein ' zu schreiben, muss es verdoppelt werden: `'ist''s'` ergibt `ist's`.
- Vektoren und Matrizen
- Komplexe Zahlen; $i = \sqrt{-1}$. Sie können auch mit `complex([1,2])` erzeugt werden. `real(c)` gibt den Realteil und `imag(c)` den Imaginärteil zurück.

Zahlen können auch in wissenschaftlicher Notation (`2.87e3` für 2870) oder hexadezimal mit dem `0x`-Präfix (`0x2a`) eingegeben werden, Binärzahlen mit `bin2dec('101010')`.

Textausgabe mit `fprintf`

```
fprintf(FORMAT, var1, var2, ...)
```

Gibt den String `FORMAT` aus. `%f`, `%s` etc. dienen als Platzhalter, die der Reihe nach mit den folgenden Argumenten `var1` etc. ersetzt werden. Die häufigsten Platzhalter:

`%f`	Floating-Point-Zahlen, einfache Darstellung
`%e`	Wissenschaftliche Darstellung (E-Notation)
`%s`	Strings. `%f` gibt Buchstaben als Zahlen aus!
`%.4f`	Floating-Point mit vier Nachkommastellen

Weiter wird `\n` durch einen Zeilenumbruch, `\\` durch einen Backslash und `%%` durch ein Prozentzeichen ersetzt. Genauere Informationen sind wie gewohnt unter `doc fprintf` (Formatting Strings) zu finden. Beispiel mit Ausgabe:

```
fprintf('%d Uhr: Albedo = %.3f \n', zeit, albedo)
  3 Uhr: Albedo = 0.289
```

Praktische Funktionen

```
abs(X)
```

Berechnet elementweise den Absolutwert $|x_{ij}|$.

```
max(X)  min(X)  abs(X)  sum(X)  mean(X)
```

Geben den Maximal-/Minimalwert, die Summe und das arithmetische Mittel ($\frac{1}{n}\sum x_i$) des übergebenen Vektors zurück. In Matrizen wird das Resultat spaltenweise berechnet, das Resultat über eine ganze Matrix erhält man durch zweifache Anwendung. Beispiel Maximum: `max(max(A))`

```
ceil(X)  floor(X)  round(X)
```

`ceil` rundet `X` auf die nächste Ganzzahl (in Richtung $+\infty$) auf, `floor` rundet nach $-\infty$ ab und `round` rundet zur nächsten Ganzzahl. `0.1*round(10*x)` rundet auf eine Nachkommastelle.

```
mod(A, M)  rem(A, M)
```

Rest bei Division `A/M`. Das Resultat hat bei `rem` das selbe Vorzeichen wie `A`, bei `mod` das selbe wie `M`. `mod(M,+3)` ist immer positiv, bei negativem Rest wird 3 addiert: $-5 = (-2)\cdot 3 + 1$

4 Visualisierung: 2-dimensionale Plots

Plotten

```
figure(ID)
```

Öffnet ein neues Plot-Fenster. Mit der optionalen `ID` wird dieses Fenster später mit dem selben Befehl wieder aktiviert.

```
plot(X, Y, FORMAT)
```

Plottet die Werte aus dem Vektor `Y` zu den entsprechenden x-Werten, also $(X_1|Y_1)$, $(X_2|Y_2)$, ... etc. `X` ist optional, ohne den Parameter werden die x-Werte `1,2,3,...` verwendet.

Mit `FORMAT` kann der Zeichenstil geändert werden: `'-ok'` zeichnet schwarze (`k`) Linien (`-`) zwischen den Datenpunkten, die als Kreise (`o`) erscheinen. `'.xg'` zeichnet nur Punkte (`.`), die als grüne (`g`) Kreuze (`x`) erscheinen. Weitere Optionen sind unter `doc plot` erklärt.

```
hold on
```

Erlaubt das Zeichnen mehrerer Kurven im selben Plot: Der nächste Plot-Befehl zeichnet, ohne bereits existierende Kurven zu löschen. Standardeinstellung: `hold off`

```
clf(ID)
```

Leert das aktive Plotfenster. Wenn die `ID` vom `figure`-Befehl angegeben wird (optional), wird dieses Fenster geleert. Einstellungen (zum Beispiel von `subplot`) werden zurückgesetzt.

Formatierung und Beschriftung

```
axis ARG
```

Ändert die Darstellungsoptionen der Achsen. Beispiele für `ARG` sind `equal` für gleichen Massstab beider Achsen, `off` für unsichtbare Achsen oder `tight` für optimale Platzausnutzung.

```
xlabel(TEXT)  ylabel(..)  title(..)
```

Beschriftet x- und y-Achse und den Plot als Ganzes.

```
legend(str1, str2, ...)
```

Setzt für die geplotteten Kurven der Reihe nach eine Legende. Das folgende Beispiel plottet eine Sinus- und eine x^2-Kurve und beschriftet sie anschliessend:

```
hold on
x = linspace(0,pi,50)
plot(x, sin(x), 'r');  plot(x, x.^2, 'g');
legend('sin x', 'x^2')
```

Dateiausgabe

```
print(FILENAME)
```

Speichert den Plot im aktiven Fenster als Grafik in der Datei `FILENAME` (zum Beispiel `'population.png'` oder `'population.svg'`).

Mehrere Plots pro Fenster

```
subplot(M, N, i)
```

Mit `subplot` kann das Plot-Fenster in $m \times n$ Felder aufgeteilt werden, so dass verschiedene Plots im selben Fenster gezeichnet werden. Das aktive Feld wird mit dem Index `i` (zeilenweise durchgezählt) ausgewählt. Das folgende Beispiel zeichnet links $\sqrt{x}$ und rechts x^2, wobei die y-Achsen unterschiedliche Skalen aufweisen.

```
subplot(1,2, 1); plot([1:100].^.5);
subplot(1,2, 2); plot([1:100].^2 );
```

5 Rechnen mit Matrizen

Indizierung

Die Indizierung von Matrizen ist 1-basiert, das erste Element besitzt den Index 1. Zugriffe ausserhalb des gültigen Bereichs ergeben einen Indexfehler. Beispiel für `v = [4 6]`:

`v(0)` oder `v(3)` → *Fehler*
`v(1)` → 4

Zahlen mit gleichmässigem Abstand

`linspace(min, max, N)`

generiert `N` Zahlen zwischen `min` und `max` mit gleichmässigem Abstand. Gegenüber der Sequenzangabe mit `low:inc:high` hat dieser Befehl den Vorteil, dass die Sequenz genau mit `max` endet. Beispiel, das mit `linspace` einfacher ist:

```
linspace(0, pi, 10)
  [0  0.628  1.256  1.885  2.513  3.1415]
```

Matrizen generieren

`zeros(m,n)` `ones(..)` `eye(..)` `rand(..)`

erstellt eine Matrix der Grösse $m \times n$ (`m` Zeilen und `n` Spalten). Sie wird folgendermassen initialisiert:

- mit Nullen bei `zeros`
- mit Einsen bei `ones`
- als Identitätsmatrix ($A_{ij} = 0, A_{ii} = 1$) bei `eye`
- zufällig mit Werten $\in [0,1]$ bei `rand`

`diag(X, d)`

erzeugt eine Diagonalmatrix mit dem Vektor `X` auf der Diagonalen und Nullen sonst. Mit dem optionalen `d` kann die Diagonale nach oben oder (negativer Index) unten verschoben werden. Beispiel: `D = diag(rand(3,1),1)` erzeugt eine 4×4-Matrix mit einer um 1 Element nach oben verschobenen Diagonalen.

$$D = \begin{bmatrix} 0 & .5 & 0 & 0 \\ 0 & 0 & .9 & 0 \\ 0 & 0 & 0 & .2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrix- und Vektormultiplikationen

Alle Arten von Multiplikationen werden mit dem $*$-Operator berechnet. (Matrizen sind unten wie üblich in Grossbuchstaben geschrieben, Vektoren sind hier Spaltenvektoren.)

- `s = u' * v`
  Skalarprodukt $\vec{u} \cdot \vec{v} = |\vec{u}|\,|\vec{v}| \cos\varphi$
- `A = u * v'`
  Kreuzprodukt $\vec{u} \times \vec{v}$
- `b = A * x`
  Matrix-Vektor-Multiplikation mit `A`: $m \times n$, `x`: $n \times 1$, `b`: $m \times 1$
- `C = A * B`
  Matrix-Matrix-Multiplikation mit `A`: $m \times k$, `B`: $k \times n$, `C`: $m \times n$

Transponieren

Der Postfix-Operator ′ (einfaches Anführungszeichen) spiegelt Matrizen an der Diagonale. Das heisst, A^T wird als `A'` geschrieben. Damit werden auch Zeilen- in Spaltenvektoren (und umgekehrt) umgewandelt.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Elementweise Operatoren

Mit einem Punkt vor den Operatoren `* / ^` wird die Operation für jedes Element einzeln durchgeführt. `A^2` ist gleichbedeutend mit `A*A` (Matrixmultiplikation), aber `A.^2` quadriert jedes Element der Matrix separat.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad B = \begin{bmatrix} s & t \\ u & v \end{bmatrix}$$

$$\texttt{A * B} = \begin{bmatrix} as+bu & at+bv \\ cs+du & ct+dv \end{bmatrix} \qquad \texttt{A .* B} = \begin{bmatrix} as & bt \\ cu & dv \end{bmatrix}$$

$$\texttt{A .\^{} B} = \begin{bmatrix} a^s & b^t \\ c^u & d^v \end{bmatrix} \qquad \texttt{A ./ B} = \begin{bmatrix} \frac{a}{s} & \frac{b}{t} \\ \frac{c}{u} & \frac{d}{v} \end{bmatrix}$$

Für die restlichen Operatoren (`+ - ==` etc.) und bei Multiplikation/Division mit Skalaren (`2*A`, `A/4`) ist der Punkt nicht notwendig.

Auswahl von Submatrizen

`A(x, y)`

So wie einzelne Elemente oder Zeilen/Spalten einer Matrix ausgewählt werden können, funktioniert das auch für Submatrizen. Dazu wird einfach für die Zeilen- *und* die Spaltenauswahl ein Vektor angegeben.

$$\text{A,B,C,D,E} = \begin{array}{c} \\ m{=}1 \\ m{=}2 \\ m{=}3 \end{array} \begin{array}{c} \begin{array}{cccc} n{=}1 & n{=}2 & n{=}3 & n{=}4 \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{array}$$

Die Elemente können so mittels Zuweisung auch direkt in der Matrix geändert werden (`B` und `C`).

```
A = A([1,2],[2,4])
B([1,2],[2,4]) = 0
C( 1:3 , 1:3 ) = [20 22 24; 26 28 30; 32 34 36]
```

Resultate dieser Zuweisungen:

$$A = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 5 & 0 & 7 & 0 \\ 9 & 10 & 11 & 12 \end{bmatrix} \quad C = \begin{bmatrix} 20 & 22 & 24 & 4 \\ 26 & 28 & 30 & 8 \\ 32 & 34 & 36 & 12 \end{bmatrix}$$

Lineare Indizierung

`A(n)`

Intern werden Matrizen als Liste abgespeichert, wobei die Elemente spaltenweise durchnummeriert werden. Mit der Schreibweise `A(n)` wird in der Liste das `n`-te Element angesprochen.

```
D(      [1 5 9]      ) = 21
E(1:(size(E,1)+1):end) = [70 73 77]
```

$$D = \begin{bmatrix} 21 & 2 & 3 & 4 \\ 5 & 21 & 7 & 8 \\ 9 & 10 & 21 & 12 \end{bmatrix} \quad E = \begin{bmatrix} 70 & 2 & 3 & 4 \\ 5 & 73 & 7 & 8 \\ 9 & 10 & 77 & 12 \end{bmatrix}$$

Die Funktion `find` zum Beispiel gibt lineare Indizes von Nicht-Null-Elementen zurück (vgl. `E > 12` und `doc find`):

`find(E > 12)` → `[1 5 9]`

6 Funktionen

Funktions- und Variablennamen

müssen mit einem Buchstaben a-zA-Z beginnen, die folgenden Zeichen dürfen auch Zahlen 0-9 oder Unterstriche _ sein. Gross- und Kleinschreibung wird unterschieden.

Funktionen definieren

Beim Funktionsaufruf wird CODE mit den angegebenen Parametern ausgeführt. Pro Funktion wird eine Datei `funcName.m` benötigt. Sowohl die durch Kommas getrennten Parameter arg_1,... als auch Rückgabewert ret und Beschreibung sind optional.

```
function ret = funcName(arg1, arg2, ...)
% Description for «help funcName»
  CODE
end
```

Mehrere Rückgabewerte

Für mehrere Rückgabewerte wird die Zeilenvektorschreibweise verwendet: [ret_1, ret_2] statt ret. Für die linke Seite der Zuweisung wird ebenfalls ein Zeilenvektor mit Variablen angegeben.

```
function [q,k] = quadKub(x)
  q = x^2;
  k = x^3;
end

[a,b] = quadKub(2)
  a = 4
  b = 8
```

Für unbenötigte Rückgabewerte wird statt eines Variablennamens eine Tilde angegeben.

```
[~, b] = quadKub(3)
  b = 27
```

Rückgabewerte werden der Reihe nach zugewiesen, fehlende Output-Variablen auf der linken Seite der Zuweisung werden ignoriert. Im folgenden Beispiel wird nur der erste Rückgabewert verwendet:

```
a = quadKub(3)
  a = 9
```

Anonyme Funktionen

`funcName = @(ARGS) EXPR`

Anonyme Funktionen lassen sich auch innerhalb einer normalen Funktion und im Befehlsfenster definieren. Sie haben genau einen Rückgabewert, der mit dem Ausdruck EXPR berechnet wird, Kontrollstrukturen können nicht verwendet werden. Das folgende Beispiel verwendet elementweise Operatoren für den Fall, dass x und y Vektoren sind.

```
f = @(x,y) x.^2 + 2*x.*y + y.^2;

f(2,3)
  25
```

7 Workspace und Arbeitsverzeichnis

Sichtbarkeitsbereich (Scope)

Funktionen besitzen einen eigenen Scope (in Matlab *Workspace* genannt). Variablen ausserhalb dieses Scopes können weder verwendet noch verändert werden.

```
function assignA()
  a = 33; fprintf('a in assignA: %d\n', a);
end
```

Beim Aufruf dieser Beispielfunktion im Befehlsfenster wird die Variable a in diesem Scope nicht überschrieben.

```
a = 42;
assignA();
  a in assignA: 33
a
  a = 42
```

Der Befehl whos zeigt alle im aktuellen Scope definierten Variablen an.

Shadowing

Beim Aufruf von Funktionen sucht Matlab zuerst im Workspace, dann im aktuellen Ordner und schliesslich in der internen Bibliothek. Wird zum Beispiel eine Funktion/Variable sum definiert, verdeckt diese die interne sum-Funktion.

`path`

Listet auf, welche Verzeichnisse nach dem Workspace der Reihe nach durchsucht werden. Zusätzliche Pfade werden mit `addpath` hinzugefügt. Beispiel, wenn benötigte Skripte im Unterverzeichnis `rungeKutta` liegen:

```
addpath('./rungeKutta');
```

Workspace leeren

`clear ARG`

löscht die Variable ARG. Ohne den optionalen Variablennamen werden alle Variablen im Workspace gelöscht. Beispiel, wenn die interne Funktion sum verdeckt wurde (Shadowing) und wieder benötigt wird:

```
clear sum
```

Arbeitsverzeichnis

`pwd`

Gibt das aktuelle Arbeitsverzeichnis aus. Hier wird als Erstes nach .m-Dateien für Skripte und Funktionen gesucht. Beim Abspeichern (zum Beispiel mit `print`) werden Dateien ohne Pfadangabe hier gespeichert.

`ls`

Zeigt die Dateien im aktuellen Verzeichnis an.

`cd PATH`

Wechselt das Arbeitsverzeichnis. Zwei Punkte .. stehen fürs übergeordnete Verzeichnis, ~ fürs Home-Verzeichnis. Das zweite Beispiel wechselt unter Windows aufs Laufwerk F (zum Beispiel den USB-Stick):

```
cd ../t5
cd f:/matlab/code
cd ~
```

8 Rekursion

Idee

Ein rekursives Programm ruft sich zur Berechnung des Resultates mit einer «kleineren» Probleminstanz selbst auf. Das heisst, zur Lösung eines Problems wird die Lösung eines kleineren – weniger schwierigen – Problems verwendet.

$$f(x) = \begin{cases} 1, & x <= 1 \\ x + f(x-1), & x > 1 \end{cases}$$

Diese Funktion verwendet zur Berechnung der Summe $\sum x$ keine Schleife, sondern addiert nur zwei Werte: $\sum x = x + \sum(x-1)$. Der Basisfall $x = 1$ ist notwendig, damit die Funktion terminiert und nicht unendlich weiterläuft.

Beispielausführung für die Summe $\sum_{i=1}^{4} i$:

$$\begin{aligned} f(4) &= 4 + \overbrace{f(3)} \\ &= 4 + 3 + \overbrace{f(2)} \\ &= 4+3 + \overbrace{2+f(1)} \\ &= 4+3+2+ \overbrace{1} \end{aligned}$$

Call Stack

Durch Rekursion lassen sich gewisse Funktionen wie zum Beispiel die Fibonacci-Folge elegant definieren. Nachteil bei prozeduralen Programmiersprachen ist der hohe Speicherverbrauch; bei Matlab/Octave wird für jeden Funktionsaufruf der aktuelle Workspace zwischengespeichert, bis der Aufruf beendet ist.

```
function y = fib(x)
if x <= 2
  y = 1; % Basisfall
else
  y = fib(x-1) + fib(x-2);
endif
end
```

Da in rekursiven Funktionen wieder eine Funktion aufgerufen wird, muss immer wieder ein weiterer Worspace zwischengespeichert werden, bis der Basisfall erreicht ist.

Rekursionstiefe

Die folgende Funktion verdeutlicht bei der Ausführung, wie sie sich selbst rekursiv aufruft. Während `f(1)` ausgeführt wird, sind alle Workspaces bis `f(n)` auf dem Stack im Hauptspeicher zwischengespeichert. Dies kann viel Speicher belegen, weshalb die Rekursionstiefe beschränkt ist (in Matlab auf 500). Wird sie überschritten, wird das Programm abgebrochen.

```
function y = f(x)
fprintf('f(%d) aufgerufen; ', x);
if x <= 1
  y = 1;
else
  y = x + f(x-1);
end
fprintf('f(%d) beendet. ', x);
end
```

9 Komplexität

Landau- oder O-Notation

Zu Algorithmen wird normalerweise die *Komplexität* angegeben. Sie kann zum Beispiel linear (geschrieben als $\mathcal{O}(n)$) oder quadratisch, $\mathcal{O}(n^2)$, sein. Damit wird der Rechnenaufwand – zum Beispiel die Anzahl Multiplikationen – in Abhängigkeit von der Grösse n des Inputs angegeben.

- Um die Summe von n Zahlen zu bilden, werden $n-1$ Additionen benötigt: $\mathcal{O}(n)$
- Matrix-Vektor-Multiplikation (Seitenlänge n) benötigt insgesamt $2n^2$ Operationen: $\mathcal{O}(n^2)$
- Einfache Matrix-Matrix-Multiplikation benötigt insgesamt $2n^3$ Operationen: $\mathcal{O}(n^3)$
- Sortieralgorithmen für Datensätze mit n Einträgen besitzen eine Komplexität von $\mathcal{O}(n \log_2 n)$

Beispiel Matrixmultiplikation

Pro Element im y-Vektor sind n Multiplikationen notwendig, und y hat n Elemente.

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ A_{21} & A_{22} & A_{23} & A_{24} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \cdot \\ y_2 \\ \cdot \\ \cdot \end{bmatrix}$$

Im folgenden Code wird die innere Schleife wird pro Durchlauf der äusseren Schleife n Mal ausgeführt, die äussere insgesamt auch n Mal. Das führt zu je n^2 Additionen und Multiplikationen.

```
for r=1:n
  for c=1:n
    y(r) = y(r) + A(r,c) * x(c)
  end
end
```

Optimierung

Falls die Matrix eine Bandmatrix mit zum Beispiel konstant drei Einträgen pro Zeile ist, kann die innere Schleife so umgeschrieben werden, dass dort nur noch je drei statt n Additionen/Multiplikationen berechnet werden, denn die Position der Nicht-Null-Einträge ist bekannt.

Damit kann die Matrix-Vektor-Multiplikation bereits in $n \cdot 3$ statt in $n \cdot n$ Schritten berechnet werden, die Laufzeit sinkt von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$.

Für grosse Matrizen mit zum Beispiel $n = 10000$ wird dieser Unterschied bemerkbar: Mit 10^9 Operationen pro Sekunde (1 GHz) liegt die Multiplikation mit quadratischer Laufzeit bereits im Sekundenbereich ($n^2 = 10^8$ Operationen), für Bandmatrizen selber Grösse aufgrund der linearen Laufzeit noch im Millisekundenbereich.

10 Handles

Allgemein

Ein Handle ist eine Art «Zeiger» auf ein anderes Objekt, zum Beispiel eine Funktion oder eine geplottete Kurve. Handles können Funktionen übergeben werden, welche dann auf das Objekt zugreifen können.

Grafik-Handles

Funktionen wie `plot`, `line` und `text` geben ein Handle zum erstellten Grafikobjekt zurück.

```
H = plot([1:10].^2);
```

Damit können im Nachhinein Eigenschaften des Grafikobjektes verändert oder Plot-Kurven selektiv beschriftet werden:

```
set(H, 'linewidth', 2, 'color', [1 .8 .2]);
legend(H, 'Power function');
```

`gcf` `gca`

geben ein Grafik-Handle zur aktuellen Figur/Achse zurück.

`get(H)` `set(H, name, value)`

Ruft Eigenschaften des Grafikhandles ab respektive setzt sie, zum Beispiel `'Color'` oder `'LineWidth'`.

Funktionshandle

```
H = @funcName
```

Mit dem `@` vor dem Funktionsnamen wird ein Funktionshandle erstellt. Diese werden etwa für `fplot` oder Runge-Kutta-Methoden `ode23`, `ode45` usw. benötigt. `fplot` erwartet ein Funktionshandle mit einem Parameter und plottet diese Funktion im angegebenen Bereich:

```
fplot(@sin, [0,2*pi]);
```

Standardwerte für Funktionsparameter

Funktionen, die als Parameter ein Funktionshandle benötigen, müssen Annahmen über dessen Parameter treffen. `ode23` etwa erwartet ein Handle zu einer Funktion mit einem oder zwei Parametern, `f(t)` oder `f(t,y)`.

Die folgende Beispielfunktion soll nur über `t` integriert werden. Mit Hilfe einer anonymen Funktion werden die Parameterwerte `S` und `albedo` festgelegt und eine Funktion `f(t)` generiert. Die Werte sind notwendig, dürfen aber nicht integriert werden.

```
S = 1305; albedo = .306;
f = @(t) temperature(t, S, albedo);
```

Die Zuweisung kann man sich so vorstellen, dass der Programmcode der Funktion `temperature` in `f` gespeichert wird und dabei `S` und `albedo` durch die aktuellen Werte ersetzt werden. Nachträgliche Änderungen an den Variablen ändern an der Funktion nichts.

11 Structs

Allgemein

Strukturen sind Objekte, die mehrere Variablen enthalten können. Variablen können damit logisch gruppiert und auch als einziges Argument einer Funktion übergeben werden.

Die im Struct enthaltenen Variablen *(Fields)* können von beliebigem Datentyp sein, einschliesslich Funktionshandles und wiederum Structs.

Strukturen werden in der Dokumentation detailliert erklärt:
`doc structures`

Structs erstellen

`struct(name1, value1, name2, value2, ...)`

Die `struct`-Funktion erstellt aus den übergebenen Name/Wert-Paaren eine Struktur. Beispiel für Structs mit zwei Feldern:

```
erde = struct('albedo', 0.306, 'S', 1367);
mars = struct('albedo', 0.25,  'S',  589);
```

Mit der impliziten Schreibweise die Werte direkt zugewiesen:

```
neptun.albedo = .29; neptun.S = 1.5;
```

Zugriff auf Felder

```
struct.field
struct.('field')
```

Beide Schreibweisen haben semantisch die selbe Bedeutung und ermöglichen den Zugriff aufs Feld `field` in der Struktur `struct`. Mit der zweiten Schreibweise kann der als String angegebene Feldname auch in einer Variablen stehen.

```
name = 'albedo';
erde.(name) → 1367
erde.albedo → 1367
```

12 Verschiedenes

Textausgabe in Variablen

Anders als die Funktion `fprintf`, welche den Text im Command Window ausgibt, gibt ihn `sprintf` als String zurück. Damit kann er in eine Variable gespeichert oder für Funktionsaufrufe – wie im folgenden Beispiel – verwendet werden.

```
a = albedo(t, S);
legend( sprintf('Albedo: %f', a) );
```

Oberflächenplots (3D)

`surf(X, Y, Z)` `repmat(A, m, n)`

Zeichnet einen 3D-Plot mit den `Z`-Werten an den jeweiligen `X`- und `Y`-Koordinaten. Wie beim `plot`-Befehl werden ohne `X` und `Y` Standardkoordinaten verwendet. Koordinatengitter können mit `repmat` generiert werden, der Befehl setzt die Matrix `A` $m \times n$ Mal zusammen. Beispiel: `X = repmat(1:10, 10, 1)`

Farben

`[r g b]`

definiert eine Farbe mit angegebenen `R`ot-, `G`rün- und `B`lauwerten $\in [0 \ldots 1]$ und erweitert damit die vordefinierten Farben `rgbcmyk`. Beispiel für RGB-Farbwerte $\in \{0, 255\}$ aus einem Grafikprogramm wie Inkscape oder GIMP, mit Normierung:

```
plot(sin(1:.1:10).^2, 'color', [132 186 241]/255)
```