MATLAB-Einführung

Dominik Alfke

1 Allgemeines

- MATLAB steht sowohl für die Programmiersprache als auch das Programm mit der graphischen Benutzeroberfläche
- Man kann MATLAB-Code auch ohne die GUI schreiben und ausführen, aber das machen wir hier erstmal nicht
- MATLAB ist eine Script-Sprache, d.h. kein Kompilieren, Befehle können einzeln im Command Window eingegeben werden
- Wichtigste Befehle: `help` und `help Funktionsname`, um Hilfe allgemein bzw. zu einer bestimmten Funktion zu erhalten
- Pfeiltaste nach oben holt zuletzt benutzte Befehle zurück

A1) Öffnen Sie MATLAB, identifizieren Sie das Command Window, und geben Sie den Hilfebefehl ein.

2 Rechnen mit Zahlen

- Intuitive Operationen: `+`, `-`, `*`, `/`, `^`
- Variablen-Zuweisung einfach über `a = 5` und `b = 3/(2-a)^3 + 0.32`
- Variablennamen können aus Buchstaben, Zahlen und `_` bestehen (erstes Zeichen: nur Buchstaben)
- Das Ergebnis jedes Befehls wird im Command Window ausgegeben. Die Ausgabe wird unterdrückt durch `;` am Ende der Zeile (also z.B. `x = 5;`)
- Spezielle Konstanten: `Inf` (unendlich), `NaN` (*not a number*, wird oft bei fehlerhaften Operationen zurückgegeben)
- Wissenschaftliche Notation: `2.3e-4` für $2.3 \cdot 10^{-4}$
- Komplexe Zahlen über `2 + 3i`, imaginäre Einheit am besten über `1i` (nicht nur `i`)
- `pi` und viele Standardfunktionen sind vordefiniert: `sin`, `cos`, `tan`, `sqrt`, `exp`, `log`, `abs`, `arg`, `asin`, ...
- Funktionsauswertung über `sin(x)`
- Im Workspace kann man alle aktuell belegten Variablen sehen
- In der Command History kann man die letzten Befehle ansehen
- Unten links in der Leiste steht *Busy*, wenn gerade ein vorheriger Befehl ausgewertet wird

A2) Berechnen Sie den Betrag der komplexen Zahl mit Realteil 1 und Imaginärteil $\cos(\frac{\pi}{4})$ und speichern Sie das Ergebnis in der Variable `x`, ohne dass es im Command Window angezeigt wird. Geben Sie darauf ausgehend e^{2x-1} aus.

A3) Was gibt MATLAB für $\frac{1}{0}$ und $\sin(\infty)$ zurück?

3 Skripte

- Dateien mit MATLAB-Code immer mit Dateiendung `.m`
- Skripte enthalten einfache Folgen von Befehlen
- Aufruf des Skripts: Zuerst in MATLAB zum übergeordneten Ordner wechseln (über `cd('/my/folder')` oder die Benutzeroberfläche), dann einfach im Command Window den Dateinamen eingeben (ohne `.m`)
- Man kann im Skript alle Variablen des Workspaces aufrufen (und überschreiben!) und alle im Skript benutzten Variablen sind danach vom Command Window aus erreichbar.
- Die MATLAB-GUI enthält einen guten Editor für MATLAB-Code
- Mehrere Befehle in einer Zeile können mit `,` oder besser `;` getrennt werden, das ist jedoch meistens unübersichtlich

A4) Legen Sie ein Skript mit Ihrem Code zu **A2)** an und führen Sie es im Command Window aus.

4 Matrizen

- `A = [1, 2; 3, 4]` gibt die Matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ ein (Kommata können weggelassen werden)
- Kein Unterschied zwischen Skalaren `a = 5` und 1×1-Matrizen `a = [5]`
- Spaltenvektoren `y = [1;2;3]` als $n \times 1$-Matrizen, Zeilenvektoren `z = [1,2,3]` als $1 \times n$-Matrizen
- Blockweiser Aufbau von Matrizen: `A = [B, C; D, E]` bei passenden Teilmatrizen. Analog "Vergrößerung" eines Vektors durch `y = [y; 4]`

- Die Funktion `zeros(m,n)` erstellt eine $m \times n$-Nullmatrix, `zeros(n)` erstellt die quadratische $n \times n$-Nullmatrix
- Analog: `ones` (Matrix voller Einsen), `eye` (Einheitsmatrix), `rand` (zufällige Einträge zwischen 0 und 1)
- `a:b` erstellt den Zeilenvektor $\begin{pmatrix} a & a+1 & a+2 & \cdots & b \end{pmatrix}$ (*colon operator*)
- `a:h:b` erstellt den Zeilenvektor $\begin{pmatrix} a & a+h & a+2h & \cdots & b \end{pmatrix}$ (`h` kann auch negativ sein)
- `linspace(a,b,n)` erstellt den $1 \times n$-Zeilenvektor mit n äquidistanten Punkten zwischen `a` und `b`

A5) Legen Sie die Matrix $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 3 & 4 & 5 & 6 \end{pmatrix}$ blockweise unter Benutzung von `ones`, `zeros` und dem *colon operator* an.

A6) Legen Sie den Zeilenvektor $\begin{pmatrix} 7 & 5 & 3 & 1 \end{pmatrix}$ einmal mit Hilfe der Syntax `a:h:b` und einmal mit `linspace` an.

5 Indexing

- `A(2,3)` gibt den Matrixeintrag in Zeile 2, Spalte 3 zurück
- Die Zählung beginnt bei 1
- Bei Vektoren reicht `y(3)` anstelle von `y(3,1)` bzw. `z(1,3)`
- Der Platzhalter `end` steht für den größtmöglichen Index in die jeweilige Richtung, d.h. `A(2, end-1)` → zweite Zeile, vorletzte Spalte
- Teilmatrizen über Indexvektoren oder den Platzhalter `:` (äquivalent zu `1:end`)
  - `A(2, :)` → gesamte zweite Zeile
  - `A(3:end, 5)` → $\begin{pmatrix} A_{3,5} \\ A_{4,5} \\ \vdots \end{pmatrix}$
  - `A([3,1], 2:4)` → $\begin{pmatrix} A_{3,2} & A_{3,3} & A_{3,4} \\ A_{1,2} & A_{1,3} & A_{1,4} \end{pmatrix}$
  - `A(end:-1:1, :)` gibt die Matrix mit vertauschter Reihenfolge der Zeilen zurück
- Zuweisung auf "zu großen" Index vergrößert die Matrix automatisch (evtl. durch 0 aufgefüllt)

A7) Überschreiben Sie in der Matrix aus **A5)** die Einserblock durch $\begin{pmatrix} 7 & 8 & 9 \end{pmatrix}$. Vertauschen Sie anschließend die zweite und vierte Spalte der resultierenden Matrix.

A8) Legen Sie eine Variable `x=1` an und erweitern Sie diese anschließend um einen Eintrag mit Wert 3 in Zeile 4, Spalte 5. Geben Sie das Resultat für `x` an.

6 Matrixoperationen

- `A+B` und `A-B` sind trivial
- Matrixmultiplikation mit `A*B`, elementweise Multiplikation mit `A.*B`
- `A\B` berechnet $A^{-1}B$ (*backslash operator*), `A/B` entspricht AB^{-1}, `A./B` ist elementweise Division
- Matrixpotenzen `A^k` (nur für quadratisches A und skalares k), elementweises Potenzieren mit `A.^B` (sowohl für `[1,2,3].^2` als auch `2.^[0,1,2]` und Kombinationen)
- Transponieren und evtl. komplex konjugieren mit `A'`, nur transponieren mit `transpose(A)` oder `A.'`

A9) Benutzen Sie `\` und `eye`, um die Inverse von $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ auszugeben.

7 Nützliche Funktionen

- `div(a,b)` und `mod(a,b)` für ganzzahlige Division und deren Rest
- `size(M)` ergibt den Vektor `[m,n]` für eine $m \times n$-Matrix `M`
- `size(M, 1)` und `size(M, 2)` ergeben nur die Anzahl der Zeilen bzw. Spalten
- `sum(v)` und `prod(v)` ergeben Summe bzw. Produkt aller Einträge eines Vektors `v`
- `max(v)` und `min(v)` ergeben das maximale bzw. minimale Element eines Vektors
- `numel(M)` ergibt `prod(size(M))` (*number of elements*)
- `length(M)` ergibt `max(size(M))`, besonders gut für Vektoren geeignet
- `det(M)` ergibt die Determinante der Matrix

- `diag(M)` ergibt den Vektor der Diagonalelemente von `M`
- `diag(v)` ergibt eine Diagonalmatrix mit Einträgen wie in `v`
- `norm(v,p)` ergibt die p-Vektornorm $\|v\|_p$ für $p \in (1, \infty)$ und `norm(M,p)` ergibt die induzierte Matrixnorm für $p \in \{1, 2, \infty\}$.
- Standardfunktionen wie `sin` werden auf Matrizen elementweise ausgewertet
- Vektorfunktionen wie `sum` werden auf Matrizen spaltenweise ausgewertet
- Beispiel: `sum(M)` ergibt den Zeilenvektor mit den Spaltensummen, `sum(sum(M))` gibt die Summe aller Einträge
- Manche Funktionen können mehrere Werte zurückgeben, z.B. ergibt `[x, i] = max(v)` den maximalen Eintrag `x` und dessen Index `i`, und `[n, m] = size(M)` speichert die Größe einer Matrix direkt in zwei Variablen `n` und `m`

A10) Berechnen Sie die Frobenius-Norm $\|A\|_\mathrm{F} = \sqrt{\sum_{i,j} a_{ij}^2}$ einer beliebigen 3×3-Matrix einmal durch `norm(A,'F')` und einmal durch elementweises Quadrieren und `sum`.

8 Zeichen und Textausgabe

- Datentyp `char` steht für einzelne Zeichen, z.B. `'a'` und `'\n'` (Zeilenumbruchszeichen)
- Zeichenketten als Zeilenvektoren mit `char`-Elementen: `'Hello'` ist äquivalent zu `['H', 'e', 'llo']`
- Einfache Ausgabe ins Command Window mit `disp('Text')` oder `disp(pi)`
- Fortgeschrittene Textausgabe mit `fprintf('Format', data1, data2)` – Der `'Format'`-String kann Platzhalter der Form `%g` enthalten, die durch die `data`-Argumente ersetzt werden
- Wichtigste Platzhalter: `%d` für ganze Zahlen, `%f` für Fließkommazahlen, `%e` für Zahlen in wissenschaftlicher Schreibweise, `%g` sucht selbst ein Zahlformat aus (weitere Formatierungseinstellungen unter `doc fprintf`)
- Beispiel: `fprintf('Pi hat den Wert %g \n', pi)` ergibt "Pi hat den Wert 3.14159"
- Es müssen so viele `data`-Argumente übergeben werden, wie es Platzhalter gibt

A11) Geben Sie eine Zeile `sin(0.5) = ?, cos(0.5) = ?` aus, wobei die Fragezeichen durch die tatsächlichen Werte der Ausdrücke ersetzt sind.

9 Plots

- `plot(x, y)` verbindet die Koordinatenpunkte (`x(i)`, `y(i)`) durch eine blaue Linie (Vektoren müssen gleich lang sein)
- Setze z.B. `x = linspace(0, 1, 100)` und `y = x.^2`, um die Funktion $x \mapsto x^2$ auf 100 Gitterpunkten im Bereich $[0, 1]$ zu plotten
- Viele Optionen durch zusätzliche Argumente, z.B. setzt `plot(x,y,'rx')` rote, nicht verbundene Kreuze an jedem Punkt (siehe `help plot`)
- `hold('on')` bewirkt, dass die nächsten `plot`-Befehle zusammen in die gleiche Grafik geplottet werden
- Hinzufügen von Informationen: `title`, `legend`, `xlabel`, `ylabel`, ...
- Dreidimensionale Daten plotten mit `plot3`, `contour`, `contourf`, `surf`, `surfc`

A12) Erstellen Sie eine Grafik, in der die drei Punkte $(1, 1)$, $(2, 5)$ und $(3, 2)$ durch schwarze Linien verbunden sind. Jeder der Punkte soll außerdem durch einen Kreis markiert sein.

A13) Erweitern Sie diese Grafik um einen Plot der Funktion $x \mapsto x \cdot (4 - x)$ auf 100 Gitterpunkten im Intervall $[0, 4]$.

A14) Beschriften Sie die x-Achse als "Zeit (s)" und die y-Achse als "Auslenkung (mm)". Erstellen Sie eine Legende, in der die drei Punkte als "Messpunkte" und die Parabel als "Modellantwort" bezeichnet werden. Speichern Sie die resultierende Grafik als `.PNG`-Datei.

10 Boolsche Operationen

- Relationen zwischen Zahlen: `==`, `~=`, `<`, `>`, `<=`, `>=`
- Logisches "Und": `Ausdruck1 & Ausdruck2`, "Oder": `Ausdruck1 | Ausdruck2`, Negieren: `~Ausdruck`
- Vergleiche mit Matrizen werden elementweise ausgeführt und geben 1-0-Matrizen der entsprechenden Größe zurück
- "Und" / "Oder" mit *lazy evaluation*: `&&` bzw. `||` (nur für skalare Ausdrücke)
- `all(v)` und `any(v)` überprüfen, ob jeder bzw. mindestens ein Vektoreintrag nicht-null ist
- Beispiel: `all((y >= 0)& (y <= 1))` überprüft, ob alle Einträge des Vektors `y` zwischen 0 und 1 liegen

A15) Überprüfen Sie, ob die folgenden Ausdrücke gleich sind: `0` und `-0`, `1/0` und `1/(-0)`, sowie `NaN` und `NaN`.

A16) Benutzen Sie `sum` und `>`, um auszugeben, wie viele Einträge des Vektors `1:100` größer als 32 sind.

11 Kontrollstrukturen

- Verzweigungen mit `if-else-end`:

```
if Bedingung
    Anweisungen
end
```

```
if Bedingung
    Anweisungen
else
    Anweisungen
end
```

- `Bedingung` kann ein beliebiger skalarer Ausdruck sein; er gilt als "wahr", wenn er nicht null ist
- Meistens ist `Bedingung` aber das Ergebnis einer Boolschen Operation, z.B. `if x < 2` oder `if ~all(y > 0)`
- `switch-case`-Blöcke sind möglich, aber selten hilfreich; sh. `help switch`
- Schleifen mit `while` und `for`:

```
while Bedingung
    Anweisungen
end
```

```
for Laufvariable = Laufvektor
    Anweisungen
end
```

- Die `Anweisungen` werden mehrmals durchgeführt; bei `while` wird vor jedem Durchlauf die `Bedingung` überprüft
- Bei `for` wird vor jedem Durchlauf die `Laufvariable` auf das nächste Element des `Laufvektor`s gesetzt
- Der `Laufvektor` kann ein beliebiger Vektor sein, hat meistens jedoch die Form `1:n` (Die Schleife wird `n`-mal ausgeführt, die Laufvariable "zählt" dabei, in welchem Durchlauf man sich gerade befindet)
- `break` bricht die aktuelle Schleife ab und springt sofort zur nächsten Anweisung nach der Schleife
- `continue` bricht den aktuellen Schleifendurchlauf ab und setzt die Schleife mit dem nächsten Laufvariablenwert fort

A17) Legen Sie ein Skript an, das die Ausdrücke e^2 und 2^e vergleicht und in einem Text ausgibt, welcher der beiden Werte größer ist.

A18) Legen Sie ein Skript an, das einen 20-elementigen Vektor anlegt und mit den Fibonacci-Zahlen füllt. Setzen Sie dazu die ersten beiden Elemente manuell und die restlichen Elemente per `for`-Schleife.

A19) Modifizieren Sie ihren Code aus **A18)**, sodass die Schleife abbricht, sobald die erste Zahl über 100 gefunden wurde. Geben Sie diese und ihren Index mit `fprintf` aus.

12 Funktionen

- Eine eigene Funktion `myFunction` kann erstellt werden, indem man eine Datei `myFunction.m` anlegt mit Inhalt

```
function [y1, y2] = myFunction(x1, x2)
    alpha = sqrt(x1^2 + x2^2)
    y1 = 2*x1 + alpha*exp(x2)
    y2 = alpha * sin(x1)
end
```

- Der Funktionsname und Dateiname muss identisch sein
- Die Argumente (hier `x1` und `x2`) und Rückgabewerte (hier `y1` und `y2`) sind innerhalb der Funktion wie normale Variablen benutzbar
- Jede Funktion hat einen eigenen Workspace, d.h. alle Variablen sind nur *lokal* (innerhalb der Funktion) benutzbar und nach dem Funktionsaufruf nicht mehr verfügbar
- `return` beendet den Funktionsaufruf sofort und gibt die aktuellen Werte der Rückgabevariablen zurück
- Auch innerhalb von Funktionen führen Variablenzuweisungen, die nicht mit `;` beendet werden, zu einer Ausgabe im Command Window

A20) Schreiben Sie eine Funktion `frobenius`, die eine Matrix beliebiger Größe als Argument nimmt und ihre Frobenius-Norm zurückgibt. Diese soll im Gegensatz zu **A10)** berechnet werden, indem eine Zählvariable iterativ um das Quadrat jedes Eintrags erhöht wird. Am Ende soll die Rückgabevariable auf die Wurzel der Zählvariable gesetzt werden.

13 Programmierung

- Wichtig: Variablen- und Funktionsnamen sinnvoll wählen und viele Kommentare in den Code schreiben (über `% Kommentar`)
- Skripte können mit "Überschriften" wie `%% Setup` oder `%% Plot` in einzelne Sektionen aufgeteilt werden, die einfach einzeln ausführbar sind
- Wenn ein Befehl zu lang für eine Zeile ist, kann man an einer beliebigen Stelle `...` setzen und in der nächsten Zeile weiterschreiben
- Debugging: Haltepunkte im Code (auch in Funktionen) setzen und Schritt für Schritt durch die Befehle gehen
- Laufzeiten messen mit `tic` und `toc`:

```
tic;
Anweisungen
toc;
```

```
ticVariable = tic;
Anweisungen
runtime = toc(ticVariable);
disp(runtime);
```

A21) Setzen Sie einen Haltepunkt am Anfang ihrer Funktion aus **A20)**. Rufen Sie die Funktion auf und beobachten Sie, wie sich die Variablen im Workspace-Editor verändern, wenn Sie den Code schrittweise durchführen lassen.

A22) Schreiben Sie ein Skript, in dem Sie mit `rand` eine zufällige 10000×10000-Matrix anlegen und die Laufzeiten messen, die die drei Methoden zur Berechnung der Frobenius-Norm aus **A10)** und **A20)** jeweils benötigen.

14 Dünnbesetzte Matrizen

- Für große Matrizen, bei denen "fast alle" Einträge 0 sind, ist es günstiger, nur die Nicht-Null-Einträge zu speichern
- MATLAB stellt so ziemlich alle Matrixoperationen auch für dünnbesetzte Matrizen zur Verfügung
- Konstruktor-Funktion: `sparse` (verschiedene Modi, sh. `help sparse`) und `speye` für die Einheitsmatrix
- `nnz` gibt die Anzahl an Nicht-Null-Einträgen einer Matrix aus (auch bei nicht sparse gespeicherten Matrizen)
- `spy` erstellt einen Plot der Besetztheitsstruktur einer Matrix

A23) Speichern Sie $\begin{pmatrix} 0 & 0 & 1 & 2 \\ 0 & 3 & 0 & 0 \end{pmatrix}$ als dünnbesetzte Matrix und plotten Sie ihre Besetztheitsstruktur.

Fortgeschrittene Themen

15 Function Handles

- Anonyme Funktionen, die als Variable gespeichert werden können
- Syntax: `f = @(x, y) x.^2 .* y` $\rightarrow$ erlaubt Aufruf $z = f(x, y)$
- Eigene und vordefinierte Funktionen können mit `f = @sin` direkt in ein Function Handle umgewandelt werden

A24) Informieren Sie sich über die Funktion `fminunc` und benutzen Sie sie, um vom Ausgangspunkt $x_0 = 1$ aus ein Minimum der Funktion $x \mapsto e^x(x^2 - 1)$ zu finden.

16 Weitere Datentypen

- Jede in Matlab eingegebene Zahl wird erstmal als Datentyp `double` behandelt (64-Bit Fließkommazahl)
- Weitere primitive Datentypen: `single` (halbe Präzision), `logical` (nur 0 oder 1), `char` (Zeichen), verschiedene (nicht benötigte) Ganzzahl-Typen (`int64`, `uint8`, ...)
- Umwandlung zwischen diesen Typen über `Typname(Matrix)`
- Sparse-Matrizen und Function Handles sind eigene Datentypen
- Zusammengesetzte Typen: Structs und Cell Arrays
- `s.feld1 = 5` oder `s = struct('feld1', 5)` erstellt ein `struct` namens `s` mit einem einzelnen Feld namens `feld1` mit Wert `5`
- Weitere Felder können über `s.feld2` zugewiesen und ausgelesen werden; Alternativ `getfield(s, 'feld2')` und `setfield(s, 'feld2')`
- Verschiedene Felder können unterschiedle Datentypen haben, z.B. `s.name = 'John'`
- Cell Arrays: Im Grunde wie Matrizen, nur dass jedes Element ein unabhängiges Objekt ist

- Syntax mit geschweiften Klammern: `c = {3, 'Bla'; [3,4,6], []}` und `c{2,2} = @sin`
- `c = cell(m,n)` erstellt ein "leeres" $m \times n$ Cell Array, bei dem jeder Eintrag die leere Matrix `[]` ist
- Hauptanwendung für Cell Arrays: Listen von unterschiedlich großen Matrizen, Listen von Zeichenketten (z.B. für dynamische `plot`-Legenden)
- Objektorientierte Programmierung mit Klassensystem ist möglich, aber nicht besonders praktikabel
- `class(x)` gibt ein `char`-Array mit dem Datentypnamen der Variable `x` zurück, beispielsw. `class(pi) == 'double'`, `class(@pi) == 'function_handle'`, `class('Hello') == 'char'`
- `isa(x, 'Datentypname')` gibt wahr zurück, falls `x` vom entsprechenden Datentyp ist

17 Weitere praktische Funktionen

- `reshape` formt einen Vektor in eine Matrix um und andersherum
- `sort` sortiert einen Vektor
- `find(v)` gibt einen Indexvektor aus, der alle Indizes enthält, an denen ein Vektor nicht null ist
- `unique` gibt alle Elemente einer Matrix höchstens einmal aus
- `union`, `intersect` und `setdiff` bilden Vereinigung, Schnitt und Differenzmenge von zwei Vektoren
- `ismember` überprüft, ob ein Element in einem Vektor enthalten ist
- `isfield` überprüft, ob ein Struct ein Feld mit einem bestimmten Namen hat
- `meshgrid` erstellt zwei- oder dreidimensionale reguläre Gitter

18 Fortgeschrittenes Indexing

- *Cartesian indexing*: `M(i,j)` wie bekannt
- *Linear indexing*:
  - Eindimensionaler Index `M(i)` auch für Matrizen, Einträge werden spaltenweise von 1 bis `numel(M)` durchnummeriert
  - Spezialfall: `M(:)` wandelt eine Matrix in einen Spaltenvektor um (Spalten werden untereinander geschrieben)
- *Logical indexing*:
  - Wenn `L` eine `logical`-Matrix von der gleichen Größe wie `M` ist, dann gibt `M(L)` einen Vektor zurück, der alle Elemente von `M` enthält, an deren Stelle in `L` eine 1 (wahr) steht
  - Beispiele: `M(M > 0)` gibt einen Vektor aller positiven Einträge von `M` zurück; `v(~isnan(v))` filtert alle `NaN` aus `v` heraus
- Eine ganze Zeile aus einer Matrix löschen: `M(2,:) = []` (Spalten analog)

19 Höherdimensionale Arrays (Tensoren)

- Matrizen können einfach auf höherdimensionale Strukturen verallgemeinert werden
- Die Matrix-Konstruktoren akzeptieren auch mehr als zwei Argumente: `zeros(3,5,4)` erstellt einen $3 \times 5 \times 4$-Nulltensor
- Es gibt keinen Unterschied zwischen einer (2×3)-Matrix und einem $(2 \times 3 \times 1)$-Tensor oder $(2 \times 3 \times 1 \times 1 \times 1)$-Tensor
- Nützliche Funktionen: `permute` (Verallgemeinerung vom Transponieren) und `squeeze`

20 Struct Arrays, Cell Arrays und Comma Separated Lists

- Bekannt: `s.a = 1; s.b = 'Text'` erstellt ein einzelnes *Struct* `s` mit Feldern `a` und `b`
- Dieses Struct kann durch `s(3).a = 4` zu einem 1×3 *Struct Array* erweitert werden
- Alle Elemente eines Struct Arrays sind Structs mit identischen Feldnamen (hier `a` und `b`)
- Nicht initialisierte Felder (wie hier `s(2).a` und `s(3).b` haben zunächst den Wert `[]`
- `s(2).x = eye(2)` fügt ein neues, leeres Feld namens `x` auch zu allen anderen Elementen des Struct Arrays hinzu
- Für das 1×3 Cell Array `c = {1,pi,rand(2)}` erstellt der Struct-Konstruktor `s = struct('a', 1, 'b', c)` ein 1×3 Struct Array mit `s(i).a == 1` und `s(i).b == c{i}`
- `s.a` gibt die Werte von `s(1).a`, `s(2).a`, ... als sogenannte *Comma Separated List* zurück
- Comma Separated Lists sind kein Datentyp und können nicht in Variablen gespeichert werden; `v = s.a` speichert nur den ersten Wert `s(1).a` in `v`

- `w = {s.a}` speichert die einzelnen Einträge in einem einzeiligen Cell Array, äquivalent zu `w = {s(1).a, s(2).a, s(3).a}`
- `y = [s.a]` versucht, die einzelnen Einträge zu einem einzelnen Zeilenvektor oder einer zeilenförmigen Blockmatrix zu verbinden (schlägt fehl bei unpassenden Datentypen oder Matrixgrößen)
- `f(s.a)` ruft die Funktion `f` auf und benutzt die Einträge von `s.a` als einzelne Argumente für `f` (die Funktion muss dementsprechend viele Argumente akzeptieren)
- Andersherum füllt `[s.a] = f(x)` die Felder von `s` mit den Rückgabewerten von `f` (die Funktion muss dementsprechend viele Werte zurückgeben)
- Für mehrelementige Indexvektoren `I` gibt `s(I).a` eine Comma Separated List mit dem entsprechenden Teil der Elemente zurückgabevariablen
- Analog gibt `c{I}` eine Comma Separated List mit den Elementen eines Cell Arrays zurück
- Beispiel: `[c{:}]` versucht, die Einträge von `c` zu einem (Block-)Zeilenvektor zusammenzufügen
- `c(I)` gibt ein Cell Array mit dem entsprechenden Teilen des Original-Inhalts zurück, äquivalent zu `{c{I}}`
- Häufige Fehlerquelle bei Cell Arrays: `c{2}` gibt den Inhalt der zweiten Zelle aus, `c(2)` ergibt ein einelementiges Cell Array mit Inhalt `c{2}`
- Nützliche Funktionen in diesem Zusammenhang: `deal`, `num2cell`, `cell2mat`; nützliche Hilfeseiten: `help lists`, `help paren`, `help cell`, `help struct`

21 Funktionsüberladung

- Wie programmiert man flexible Funktionen wie z.B. `plot` selbst?
- Der Funktionskopf `function [y,z] = myFunction(u,v,x)` legt nur fest, dass die Funktion *höchstens* drei Argumente bekommen darf und *bis zu* zwei Werte zurückgeben kann. Der Aufruf `a = myFunction(2, pi)` führt nur dann zu einem Fehler, wenn das dritte Argument `x` innerhalb von `myFunction` ausgelesen wird. Dass der zweite Rückgabewert nicht verwendet wird, ist egal.
- Wenn man innerhalb von `myFunction` die argumentlose Funktion `nargin` aufruft, erhält man die Anzahl der tatsächlich übergebenen Argumente
- MATLAB unterstützt Default-Werte für optionale Argumente nicht direkt, aber so kann man sie trotzdem implementieren:

```
function y = myFunction(a, b, c)
% b ist optionales Argument mit Defaultwert 4
% c ist optionales Argument mit Defaultwert 5
  if nargin == 1
    b = 4;
  end
  if nargin < 3
    c = 5;
  end
  y = a*b+c;
end
```

- Wenn das letzte Argument den speziellen Namen `varargin` hat, fasst MATLAB die letzten übergebenen Argumente in dieser Variable als Cell Array zusammen, d.h. beim Funktionskopf `function f(x, varargin)` führt `f(1)` zu `varargin == {}` und `f(1,2,3)` zu `varargin == {2,3}`
- Key-Value-Paare, d.h. Funktionsaufrufe der Form `myFunction(x, 'ParameterName', ParameterWert, ...)`, kann man über `varargin` in Structs umwandeln:

```
function y = myFunction(x, varargin)
  % Struct-Konstruktor mit varargin als Comma Separated List aufrufen
  params = struct(varargin{:})
  % Jetzt sind die Key-Value-Paare in den Feldern von params gespeichert
  ...
end
```

- Analog zu `nargin` und `varargin` gibt es für die Rückgabewerte `nargout` und `varargout`
- Falls eine Funktion einen zweiten Rückgabewert besitzt, der eine aufwändige zusätzliche Berechnung erfordert, sollte man vorher durch `nargout > 1` überprüfen, ob der Rückgabewert überhaupt benötigt wird