WiMa-Praktikum 1

Universität Ulm, Sommersemester 2017

Woche 7

Lernziele

In diesem Praktikum sollen Sie üben und lernen:

- Verschiedene Datentypen in MATLAB
- Wiederholung und Vertiefung des Umgangs mit Vektoren und Matrizen
- Wiederholung des bisher Gelernten anhand von Praktikumsaufgaben

Am Anfang geben wir Ihnen einen kurzen Überblick über die benötigten MATLAB-Anweisungen.

Beantworten Sie danach bitte erst einige Fragen und einige off-line-Aufgaben, bevor Sie sich an den Rechner setzen!

Datentypen in Matlab

In MATLAB müssen im Unterschied zu vielen anderen Programmiersprachen Variablen nicht deklariert werden. Deshalb konnten wir bisher Variablen einfach benutzen, ohne uns Gedanken über deren Datentyp machen zu müssen.
MATLAB kennt 16 Grund-Datentypen, die wir teilweise bereits kennen gelernt haben. Einen Überblick bietet die Seite `http://de.mathworks.com/help/matlab/matlab_prog/fundamental-matlab-classes.html`. Im Folgenden werden wir einige dieser Datentypen genauer betrachten.

Numerische Datentypen

Unter den 16 Grund-Datentypen sind allein 10 numerische Datentypen, die sich in Datentypen für Gleitkommazahlen (Floating-Point Numbers) und ganze Zahlen (Integers) aufteilen.

Gleitkommazahlen

Der Datentyp einer Variablen wird durch die erste Zuweisung festgelegt, kann aber durch folgende Zuweisungen geändert werden. Numerische Variablen und Konstanten sind per Voreinstellung vom Typ `double`. So wird durch die Zuweisung

```
≫ x = 7;
```

eine Variable vom Typ `double` angelegt und ihr der Wert `7.0` zugewiesen. Das überprüfen wir mit dem Kommando `whos`:

```
≫ whos
Name      Size            Bytes  Class     Attributes
x         1x1                 8  double
```

Wir sehen, dass die Variable `x`

- die Größe `1x1` hat (Genaueres dazu siehe Abschnitt Vektoren und Matrizen),
- 8 Bytes Speicherplatz belegt und
- vom Typ `double` ist.

Der Datentyp `double` bezeichnet also eine 64-bit-Gleitkommazahl (8 Byte zu je 8 Bit sind 64 Bit), die eine Genauigkeit von rund 16 signifikanten Stellen im Dezimalsystem hat. Die absolute Genauigkeit einer Zahl, das ist der Abstand von der Zahl bis zur nächsthöheren darstellbaren Zahl, hängt von der Größe der Zahl ab und kann mit der Funktion `eps()` ermittelt werden:

```
≫ eps(x)
ans =
   8.8818e-16
≫ eps(7000)
ans =
   9.0949e-13
```

Die betragsmäßig kleinste darstellbare Zahl vom Typ `double` erhält man mit `realmin`, die betragsmäßig größte darstellbare Zahl vom Typ `double` mit `realmax`. Zahlen, die größer als

realmax bzw. kleiner als -realmax sind, wird der Wert plus bzw. minus unendlich zugewiesen. Beachte: Bei Zahlen im Bereich von realmax beträgt der Abstand von einer mit 64 Bit darstellbaren Zahl zur nächsten mit 64 Bit darstellbaren Zahl etwa eps(realmax), also mehr als 10^{292}. Dadurch wird realmax+1 abgerundet zu realmax (1 ist sehr viel kleiner als der halbe Abstand). realmax+1 ergibt also realmax und nicht unendlich.

```
≫ realmin
ans =
   2.2251e-308
≫ realmax
ans =
   1.7977e+308
≫ eps(realmin)
ans =
   4.9407e-324
≫ eps(realmax)
ans =
   1.9958e+292
≫ realmax + eps(realmax)
ans =
   Inf
```

Genügt eine Genauigkeit von weniger als 64 Bit, reicht möglicherweise der Datentyp single aus, der eine 32-bit-Gleitkommazahl bezeichnet. Eine Variable vom Typ single benötigt mit 4 Byte nur halb so viel Speicherplatz wie eine Variable vom Typ double. Diese Speicherplatzersparnis kann beispielsweise bei sehr großen Matrizen von Vorteil sein.
Die betragsmäßig kleinste darstellbare single-Zahl erhält man mit realmin('single'), die betragsmäßig größte darstellbare single-Zahl mit realmax('single'). Zahlen, die größer als realmax('single') bzw. kleiner als -realmax('single') sind, wird der Wert plus bzw. minus unendlich zugewiesen.

```
≫ realmin('single')
ans =
   1.1755e-38
≫ realmax('single')
ans =
   3.4028e+38
```

Konstanten und Variablen können mit der Funktion single() in den Datentyp single umgewandelt werden. Die Funktion double() wandelt in den Datentyp double um.

```
≫ x = pi;
≫ y = single(pi);
≫ z = double(y);
≫ whos
Name      Size            Bytes  Class     Attributes
x         1x1                 8  double
y         1x1                 4  single
z         1x1                 8  double
```

Achtung: Durch die Typumwandlung in `single` geht Genauigkeit verloren, da weniger signifikante Stellen zur Verfügung stehen. Eine anschließende Rückumwandlung in `double` kann die verlorenen Stellen nicht rekonstruieren.

```
≫ format long
≫ x
x =
   3.141592653589793
≫ y
y =
   3.1415927
≫ z
z =
   3.141592741012573
≫ eps(x)
ans =
   4.440892098500626e-16
≫ eps(y)
ans =
   2.3841858e-07
≫ format short
```

Die Variable `x` enthält π mit 64-bit-Genauigkeit. Alle angezeigten Stellen sind korrekt. Die Variablen `y` und `z` enthalten π nur mit einer Genauigkeit von 32 Bit. Die bei `z` zusätzlich angegebenen Stellen resultieren aus der Darstellung im Dezimalsystem und täuschen eine nicht mehr vorhandene Genauigkeit vor.

Ganze Zahlen

Bei den ganzen Zahlen unterscheidet MATLAB zwischen ganzen Zahlen mit Vorzeichen und ganzen Zahlen ohne Vorzeichen. Zu beiden Gruppen gibt es jeweils Datentypen mit einem, zwei, vier oder acht Byte. Insgesamt hat MATLAB also acht verschiedene Datentypen für ganze Zahlen. Die folgende Tabelle soll einen Überblick geben:

Typ	Vorz.	Größe	Wertebereich	
int8	ja	1 Byte	-2^7 bis 2^7-1	-128 bis 127
int16	ja	2 Byte	-2^{15} bis $2^{15}-1$	-32768 bis 32767
int32	ja	4 Byte	-2^{31} bis $2^{31}-1$	-2147483648 bis 2147483647
int64	ja	8 Byte	-2^{63} bis $2^{63}-1$	-9223372036854775808 bis 9223372036854775807
uint8	nein	1 Byte	0 bis 2^8-1	0 bis 255
uint16	nein	2 Byte	0 bis $2^{16}-1$	0 bis 65535
uint32	nein	4 Byte	0 bis $2^{32}-1$	0 bis 4294967295
uint64	nein	8 Byte	0 bis $2^{64}-1$	0 bis 18446744073709551615

Auch für ganze Zahlen stehen die entsprechenden Funktionen zur Typumwandlung bereit: `int8()`, `int16()`, `int32()`, `int64()`, `uint8()`, `uint16()`, `uint32()` und `uint64()`. Bei der Typumwandlung von Gleitkommazahlen wird entsprechend der Standardrundung gerundet: Abrundung für Werte „$< .5$“ und Aufrundung für Werte „$\geq .5$“. Eine abweichende Rundung kann man

beispielsweise mit Hilfe der Funktionen `fix()`, `floor()` oder `ceil()` und anschließender Typumwandlung erreichen:

```
≫ clear all
≫ x = int8(3.49)
x =
   3
≫ y = int16(-3.5)
y =
   -4
≫ z = int8(ceil(-3.5))
z =
   -3
```

Das Ergebnis arithmetischer Operationen zwischen einem Integer-Datentyp und einer Gleitkommazahl ist wieder vom entsprechenden ganzzahligen Datentyp, natürlich gerundet entsprechend der Standardrundung. Arithmetische Operationen mit verschiedenen Integer-Datentypen sind nicht erlaubt:

```
≫ u = 3.5 * x
u =
   11
≫ v = x * z
v =
   -9
≫ w = x * y
Error using  .*
Integers can only be combined with integers of the same class,
or scalar doubles.
```

Die größte bzw. kleinste darstellbare Zahl erhält man mit den Funktionen `intmax()` bzw. `intmin()`. Anders als bei Gleitkommazahlen haben nicht mehr darstellbare Zahlen nicht den Wert `-Inf` bzw. `Inf`, sondern erhalten jeweils den maximal darstellbaren Wert:

```
≫ clear all
≫ xMax = intmax('int8')
xMax =
   127
≫ yMin = intmin('int16')
yMin =
   -32768
≫ xMax = xMax + 1
xMax =
   127
≫ z = uint8(-100)
z =
   0
```

Komplexe Zahlen

MATLAB unterstützt auch das Rechnen mit komplexen Zahlen. Dazu ist die imaginäre Einheit in den Variablen `i` und `j` vordefiniert. Angezeigt wird die imaginäre Einheit immer durch den Buchstaben `i`. Als Sonderfall braucht bei der Multiplikation mit der imaginären Einheit `i` oder `j` der Multiplikationsoperator `*` nicht angegeben zu werden.

```
≫ clear all
≫ i
ans =
   0.0000 + 1.0000i
≫ j
ans =
   0.0000 + 1.0000i
≫ -1+2*i
ans =
   -1.0000 + 2.0000i
≫ z = (1+i) * (1+2*j)
z =
   -1.0000 + 3.0000i
≫ z = (1+i) * (1+2j)
z =
   -1.0000 + 3.0000i
```

Achtung: Die Variablen `i` und `j` können wie jede andere Variable umdefiniert werden. Dann kann auf die imaginäre Einheit aber noch beispielsweise über den Ausdruck `1i` zugegriffen werden. Also kann man durch Weglassen des Multiplikationsoperators eine häufige Fehlerquelle vermeiden. Mit `clear i` und `clear j` wird der ursprüngliche Zustand wiederhergestellt. Wenn die Variablen `i` und `j` den Wert der imaginären Einheit haben, erscheinen sie nicht im Workspace.

```
≫ clear all
≫ i = 3
i =
   3
≫ z = (1+i) * (1+2j)
z =
   4.0000 + 8.0000i
≫ z = (1+1i) * (1+2j)
z =
   -1.0000 + 3.0000i
≫ clear i
≫ z = (1+i) * (1+2j)
z =
   -1.0000 + 3.0000i
```

Komplexe Zahlen sind kein eigener Datentyp. Wie man an der Ausgabe des Kommandos `whos` sehen kann, sind sie Gleitkommazahlen, bei denen das Attribut `complex` gesetzt ist. Sie benötigen den doppelten Speicherplatz einer nicht-komplexen Gleitkommazahl.

```
≫ clear all
≫ x = 3
x =
   3
≫ x1 = x + 1i
x1 =
   3.0000 + 1.0000i
≫ y = single(3)
y =
   3
≫ y1 = y + 1i
y1 =
   3.0000 + 1.0000i
≫ whos
Name      Size            Bytes  Class     Attributes
x         1x1                 8  double
x1        1x1                16  double    complex
y         1x1                 4  single
y1        1x1                 8  single    complex
```

Der Datentyp `logical`

Der Datentyp `logical` dient der Darstellung logischer Werte. Dieser Datentyp hat nur zwei Werte: `0` mit der Bedeutung `false` und `1` mit der Bedeutung `true`. Variablen vom Typ `logical` benötigen 1 Byte Speicherplatz.

```
≫ clear all
≫ x = 3>4
x =
   0
≫ y = 4>3
y =
   1
≫ whos
Name      Size            Bytes  Class     Attributes
x         1x1                 1  logical
y         1x1                 1  logical
```

Mit der Funktion `logical()` können Werte eines anderen Datentyps in logische Werte umgewandelt werden. Dabei werden Werte gleich 0 in den Wert `false`, Werte ungleich 0 in den Wert `true` konvertiert.
In manchen Situationen, wie beispielsweise bei der Bewertung eines Ausdrucks in einer `If`-Verzweigung oder einer `While`-Schleife, wird der Wert eines Ausdrucks automatisch in einen logischen Wert umgewandelt. Wie das folgende kleine Beispiel zeigt, ist dabei aber besondere Vorsicht nötig. Die Funktion `nprint(n,s)` soll den Text `s` `n`-mal in einer Zeile ausgeben. Der Aufruf `nprint(2, 'bla')` erzeugt auch wie gewünscht den Text `blabla`, der Aufruf `nprint(2.5, 'bla')` allerdings wird zur Endlosschleife.

```
1 function nprint( n, s )
2
3 while(n)
4   fprintf(s)
5   n = n-1;
6 end
7 fprintf('\n');
8
9 end
```

Mit `logical()` können auch Rückgabewerte anderer Funktionen in logische Werte umgewandelt werden. Beispielsweise überprüft `~logical(mod(x,3))`, ob der Wert der Variablen `x` durch 3 teilbar ist.
Achtung: Wie wir im Abschnitt über Vektoren und Matrizen sehen werden, muss in manchen Situationen sorgfältig zwischen den logischen Werten 0 und 1 sowie den numerischen Werten 0 und 1 (bzw. Nicht-Null) unterschieden werden.

Einige nützliche Funktionen

Die folgenden Funktionen können beim Umgang mit verschiedenen Datentypen nützlich sein. Die genaue Erklärung dieser Funktionen würde aber den Rahmen dieses Praktikumsblattes sprengen. Es sei daher auf die MATLAB-Dokumentation verwiesen.

Funktion	Beschreibung
`class(x)`	liefert den Datentyp von `x`
`isa(x, type)`	entscheidet, ob `x` den Datentyp `type` hat
`isnumeric(x)`	entscheidet, ob `x` ein numerischer Wert ist
`isfloat(x)`	entscheidet, ob `x` eine Gleitkommazahl (Floating-Point Number) ist
`isinteger(x)`	entscheidet, ob `x` einen Integer-Datentyp hat
`isreal(x)`	entscheidet, ob `x` eine reelle Zahl ist
`real(x)`	liefert den Realteil von `x`
`imag(x)`	liefert den Imaginärteil von `x`
`complex(x, y)`	konstruiert aus dem Realteil `x` und dem Imaginärteil `y` eine komplexe Zahl
`isfinite(x)`	entscheidet, ob `x` ein endlicher Wert ist
`isinf(x)`	entscheidet, ob `x` ein unendlicher Wert ist
`islogical(x)`	entscheidet, ob `x` ein logischer Wert ist
`char(x)`	wandelt den Wert `x` in einen Character-Wert (bzw. String) um
`ischar(x)`	entscheidet, ob `x` ein Character-Wert bzw. String-Wert ist

Vektoren und Matrizen

In MATLAB sind alle Variablen, bis auf eine Ausnahme, Matrizen. Wir haben in den vorigen Abschnitten beim Aufruf des Kommandos `whos` gesehen, dass die Größe der Variablen mit `1x1`

angegeben wurde. Eine einfache Variable, wie wir sie bisher in diesem Praktikumsblatt betrachtet haben, ist also eigentlich eine 1×1-Matrix. Alles, was wir bisher über Datentypen gesagt haben, gilt somit auch für Matrizen des entsprechenden Datentyps.
Ein Zeilenvektor ist eine Matrix, die nur eine Zeile hat (also eine $1 \times m$-Matrix), ein Spaltenvektor eine Matrix, die nur eine Spalte hat (also eine $n \times 1$-Matrix). Ein String ist ein Zeilenvektor mit Elementen des Datentyps `char`.
Die Ausnahme sind Variablen vom Datentyp `function handle`. `Function handles` sind immer scalar!
Die folgenden Funktionen sind beim Umgang mit Vektoren und Matrizen hilfreich:

Funktion	Beschreibung
`zeros()`	legt eine Matrix der angegebenen Größe vom Typ `double` an, die elementweise auf 0 initialisiert ist
`ones()`	legt eine Matrix der angegebenen Größe vom Typ `double` an, die elementweise auf 1 initialisiert ist
`eye()`	legt eine Einheits-Matrix der angegebenen Größe vom Typ `double` an
`rand()`	legt eine Matrix der angegebenen Größe vom Typ `double` an, deren Elemente Zufallszahlen aus dem Intervall $[0, 1]$ sind
`false()`	legt eine Matrix der angegebenen Größe vom Typ `logical` an, die elementweise auf `logical(0)`, also `false`, initialisiert ist
`true()`	legt eine Matrix der angegebenen Größe vom Typ `logical` an, die elementweise auf `logical(1)`, also `true`, initialisiert ist
`size()`	liefert die Anzahl Zeilen und die Anzahl Spalten einer Matrix
`length()`	liefert das Maximum aus Zeilenanzahl und Spaltenanzahl einer Matrix
`abs()`	liefert eine Matrix mit den Absolutbeträgen der Matrixelemente
`max()`	liefert für einen Vektor das Maximum der einzelnen Einträge des Vektors, liefert für eine Matrix einen Zeilenvektor mit den Maxima der einzelnen Spalten der Matrix
`min()`	analog zu `max()`
`sum()`	liefert für einen Vektor die Summe der einzelnen Einträge des Vektors, liefert für eine Matrix einen Zeilenvektor mit den Summen der einzelnen Spalten der Matrix
`prod()`	liefert für einen Vektor das Produkt der einzelnen Einträge des Vektors, liefert für eine Matrix einen Zeilenvektor mit den Produkten der einzelnen Spalten der Matrix
`sort()`	liefert für einen Vektor den sortierten Vektor, liefert für eine Matrix eine Matrix, deren Spalten sortiert sind
`unique()`	liefert einen Vektor mit den Elementen einer Matrix ohne Wiederholungen

Die mit `zeros()`, `ones()`, `eye()`, `false()` und `true()` erzeugten Matrizen können mithilfe der oben erklärten Funktionen zur Typumwandlung in Matrizen eines anderen Datentyps umgewandelt werden.
Wir wollen uns anhand kleiner Beispiele in Erinnerung rufen, wie wir auf Matrixelemente zugreifen können. Zunächst legen wir uns eine 3×3-Matrix `A` an, deren Elemente spaltenweise durchnummeriert sind, und geben dann mit `A(2,1)` das Element aus Zeile 2 und Spalte 1 im Command Window aus. `A(2,:)` gibt die komplette zweite Zeile aus:

```
≫ clear all
≫ A = [1 4 7; 2 5 8; 3 6 9]
A =
   1     4     7
   2     5     8
   3     6     9
≫ A(2,1)
ans =
   2
≫ A(2,:)
ans =
   2     5     8
```

In MATLAB werden Matrizen immer spaltenweise im Speicher abgelegt. Die einzelnen Spalten einer Matrix liegen also hintereinander im Speicher, und zwar lückenlos. Mit `A(:)` können wir uns die Matrix `A` als Vektor ausgeben lassen. Dabei erhalten wir die Matrixelemente in der Reihenfolge, wie sie im Speicher liegen, also spaltenweise. `A(6)` liefert das sechste Element der als Vektor betrachteten Matrix:

```
≫ A(:)
ans =
   1
   2
   3
   4
   5
   6
   7
   8
   9
≫ A(6)
ans =
   6
```

Wir können auf die Matrixelemente in beliebiger Reihenfolge und auch mehrfach zugreifen. `A(:,[3 2 3 1 3])` gibt beispielsweise die dritte Spalte, die zweite Spalte, wieder die dritte Spalte, dann die erste Spalte und zuletzt nochmals die dritte Spalte der Matrix `A` aus.
`A([8 3 5])` liefert das achte, dritte und fünfte Element der als Vektor betrachteten Matrix:

```
≫ A(:,[3 2 3 1 3])
ans =
   7     4     7     1     7
   8     5     8     2     8
   9     6     9     3     9
≫ A([8 3 5])
ans =
    8     3     5
```

Greifen wir auf ein nicht vorhandenes Matrixelement zu, erhalten wir einen Fehler:

```
≫ A(4,1)
Index exceeds matrix dimensions.
≫ A(10)
Index exceeds matrix dimensions.
```

Wenn wir jedoch andererseits einem nicht vorhandenen Matrixelement einen Wert zuweisen, wird die Matrix entsprechend erweitert. Die neuen Matrixelemente werden, sofern ihnen kein Wert zugewiesen wird, mit `0` initialisiert. Mit `A(:)` überprüfen wir, wie die Matrixelemente danach im Speicher liegen:

```
≫ A(4,1) = 10
A =
     1     4     7
     2     5     8
     3     6     9
    10     0     0
≫ A(:)
ans =
     1
     2
     3
    10
     4
     5
     6
     0
     7
     8
     9
     0
```

Wir sehen, dass durch die Erweiterung der Matrix an der vierten, siebten und zehnten Speicherstelle ein Element eingefügt wurde. Die Matrix musste also umkopiert werden. Da das Umkopieren bei großen Matrizen sehr zeitaufwendig ist, sollte es unbedingt vermieden werden. Das bedeutet, dass der Speicherplatz für Matrizen möglichst zu Beginn an einem Stück allokiert werden sollte, beispielsweise durch Aufruf einer der Funktionen `zeros()`, `ones()`, `eye()`, `false()` oder `true()` mit der maximalen Größe der Matrix.
In MATLAB können auf Matrizen auch logische Operatoren oder andere Funktionen angewendet werden. `L5 = A>5` erzeugt eine Matrix `L5` der gleichen Größe wie `A`, aber vom Typ `logical`, die genau an den Stellen den logischen Wert `1` enthält, an denen `A` einen Wert größer als `5` hat:

```
≫ L5 = A>5
L5 =
     0     0     1
     0     0     1
     0     1     1
     1     0     0
```

Mit `L2 = (mod(A,2) == 0)` wird eine logische Matrix erzeugt, die genau an den Stellen den logischen Wert `1` enthält, an denen `A` einen durch `2` teilbaren Wert hat:

```
≫ L2 = (mod(A,2) == 0)
L2 =
     0     1     0
     1     0     1
     0     1     0
     1     1     1
```

Einen Vektor mit den entsprechenden Werten von `A` erhalten wir mit `A(L5)` bzw. `A(L2)`:

```
≫ A(L5)
ans =
    10
     6
     7
     8
     9
≫ A(L2)
ans =
     2
    10
     4
     6
     0
     8
     0
```

Dieser Vektor kann natürlich auch weiter bearbeitet werden. So liefert beispielsweise `max(A(mod(A,2) == 0))` den größten der durch 2 teilbaren Einträge der Matrix `A`:

```
≫ max(A(mod(A,2) == 0))
ans =
    10
```

Achtung: Hier sehen wir den Unterschied zwischen der Indizierung einer Matrix über einen numerischen Vektor und der Indizierung der Matrix über einen Vektor mit logischen Werten. Bei der Indizierung mit logischen Werten werden genau diejenigen Matrixelemente ausgewählt, deren Index den logischen Wert `1` hat.
Dies wollen wir am folgenden Beispiel verdeutlichen. Mit `B = 5*ones(size(A))` erzeugen wir eine Matrix `B` vom Typ `double` der gleichen Größe wie `A`, deren Einträge alle den Wert `5` haben. `A(B)` liefert eine Matrix der gleichen Größe wie `B`, deren Einträge alle den Wert des fünften Eintrags von `A` haben (also nach der Erweiterung der Matrix `A` den Wert `4`). `A(logical(B))` liefert dagegen einen Vektor, der alle Einträge der Matrix `A` enthält:

```
≫ B = 5*ones(size(A))
B =
     5     5     5
     5     5     5
     5     5     5
     5     5     5
```

```
≫ A(B)
ans =
   4     4     4
   4     4     4
   4     4     4
   4     4     4
≫ A(logical(B))
ans =
    1
    2
    3
   10
    4
    5
    6
    0
    7
    8
    9
    0
```

In diesem Zusammenhang wird oft auch die Funktion `find()` benutzt. `find()` lokalisiert alle Nichtnull-Einträge einer Matrix und liefert deren Indizes.
`find(A>5)` liefert also gerade die Indizes der aus der logischen Bedingung `A>5` resultierenden logischen Matrix, an denen diese logische Matrix den Wert `1` hat. Das sind aber gerade auch die Indizes der Matrix `A`, an denen die Matrix `A` einen Wert größer als `5` hat. Mit `A(find(A>5))` erhalten wir gerade diese Werte der Matrix `A`, die größer als `5` sind.

```
≫ find(A>5)
ans =
    4
    7
    9
   10
   11
≫ A(find(A>5))
ans =
   10
    6
    7
    8
    9
```

Strukturen in Matlab

MATLAB unterstützt den Datentyp einer Struktur, wie er aus anderen Programmiersprachen bekannt ist. Eine Struktur besteht aus einem oder mehreren Feldern, die Daten unterschiedlichen Typs speichern können. Der Zugriff auf die Felder erfolgt über den Namen des jeweiligen Feldes. Die Idee der Struktur werden wir an einem einfachen Beispiel verdeutlichen: Um eine Person zu beschreiben, verwenden wir die Attribute `Name`, `Vorname` und `Alter`. Die zugehörigen Feldnamen definieren wir in drei Variablen `field1`, `field2` und `field3`. Mit der Funktion `struct()` legen wir eine Instanz dieser Struktur an. Und zwar, da möglicherweise noch weitere Personen hinzukommen, als erstes Element eines Vektors P:

```
≫ clear all
≫ field1 = 'Name';
≫ field2 = 'Vorname';
≫ field3 = 'Alter';
≫ P(1) = struct(field1, 'Musterfrau', field2, 'Max', field3, 25)
P =
       Name: 'Musterfrau'
    Vorname: 'Max'
      Alter: 25
```

Über die Feldnamen kann auf die Inhalte der Felder zugegriffen werden. So erhalten wir beispielsweise den Namen der ersten Person von `P` mit `P(1).Name`:

```
≫ P(1).Name
ans =
Musterfrau
```

Alternativ kann eine Instanz dieser Struktur durch direkte Zuweisung an die Felder der Struktur angelegt werden:

```
≫ P(2).Name = 'Mustermann';
≫ P(2).Vorname = 'Maxima';
≫ P(2).Alter = 24;
≫ P
P =
1x2 struct array with fields:
    Name
    Vorname
    Alter
≫ P(2)
ans =
       Name: 'Mustermann'
    Vorname: 'Maxima'
      Alter: 24
```

Die Eingabe von `P` liefert die Typ-Informationen über die Variable `P`, mit `P(2)` erhält man die Daten der zweiten Person.

Cell Arrays

Ein weiterer Datentyp in MATLAB sind Cell Arrays. Cell Arrays können wie Strukturen Daten unterschiedlichen Typs speichern. Im Unterschied zu Strukturen werden die einzelnen Komponenten aber nicht über Feldnamen identifiziert, sondern über Indizes wie bei Vektoren oder bei Matrizen. Um auf einzelne Zellen zuzugreifen, werden die Indizes in geschweiften Klammern angegeben. Indizes in runden Klammern bezeichnen Zellbereiche.
Die Daten aus dem Beispiel zu den Strukturen können wir auch in einem Cell Array speichern. Dazu legen wir mit `C = cell(1,3);` einen leeren Cell Array an, der aus einer Zeile mit drei Spalten besteht. Den drei Zellen weisen wir die Werte der Person zu. Auf das Alter können wir mit `C{3}` zugreifen. Den Namen und den Vornamen liefert `C(1:2)`, in der Reihenfolge Vorame gefolgt von Name erhalten wir die Werte mit `C(2:-1:1)`.

```
≫ C = cell(1,3);
≫ C{1} = 'Musterfrau';
≫ C{2} = 'Max';
≫ C{3} = 25;
≫ C{3}
ans =
    25
≫ C(1:2)
ans =
    'Musterfrau'    'Max'
≫ C(2:-1:1)
ans =
    'Max'    'Musterfrau'
```

Cell Arrays können mit der Funktion `celldisp()` komplett ausgegeben werden.
Eine Anwendung von Cell Arrays in MATLAB wäre bei der Parameterübergabe an Funktionen, die eine unterschiedliche Anzahl Parameter erhalten können. Der Funktion im folgenden Beispiel müssen mindestens zwei Parameter und können beliebig viele weitere Parameter übergeben werden. Diese werden mit `celldisp` ausgegeben.

```
1  function varInputArgs( p1, p2, varargin )
2
3    fprintf('Total number of inputs = %d\n',nargin);
4    disp(p1)
5    disp(p2)
6
7    nVarargs = length(varargin);
8    fprintf('Number of Inputs in varargin = %d:\n',nVarargs)
9    celldisp(varargin)
10
11 end
```

Offline Aktivitäten

Übereinstimmen

Schreiben Sie vor jeden Begriff auf der linken Seite den passenden Buchstaben der Beschreibung, die am besten mit der aus der rechten Spalte übereinstimmt.

_____	1.	`A>5`	a.	plottet Oberflächen
_____	2.	`double`	b.	Typumwandlung
_____	3.	`int32`	c.	logischer Wert
_____	4.	`uint8`	d.	64 Bit
_____	5.	`real(x)`	e.	4 Byte
_____	6.	`double(x)`	f.	Programmiersprachen wie C
_____	7.	Struktur	g.	immer größer oder gleich 0
_____	8.	`A(5,:)`	h.	Zeile
_____	9.	`surf`	i.	Realteil
_____	10.	`mex`	j.	Zugriff über Feldnamen

Ihre Antwort:

Füllen Sie die Lücken aus

Ergänzen Sie die folgenden Sätze.

11. Die Variable `i` kann mit _____ oder _____ auf den Wert der imaginären Einheit zurückgesetzt werden.
12. `realmin` liefert die _____ _____ darstellbare Zahl vom Typ `double`.
13. Der Speicherplatz für eine Matrix kann z. B. mit _____ allokiert werden.
14. Variablen vom Typ _____ sind in MATLAB nie eine Matrix, sondern immer scalar.
15. Mit _____ und _____ können Werte unterschiedlichen Typs in einer Variablen zusammengefasst werden.
16. Bei der Indizierung einer Matrix über einen Vektor mit _____ Werten muss der Vektor genau gleich viele Elemente haben wie die Matrix.
17. Die Funktion `find()` lokalisiert alle _____-Einträge einer Matrix und liefert deren Indizes.
18. Matrizen werden in Matlab immer _____ im Speicher abgelegt.
19. Der Datentyp _____ hat nur zwei Werte.

Ihre Antwort:

Kurz und knapp

Geben Sie bitte eine kurze Antwort zu jeder der folgenden Fragen. Ihre Antwort sollte so kurz und präzise wie möglich sein; versuchen Sie es mit zwei bis drei Sätzen.

20. Warum sollte der Speicherplatz für Matrizen in Matlab immer vor der ersten Zuweisung an die Matrix in der endgültigen Größe allokiert werden?

    Ihre Antwort:

21. Warum liefert `x = realmax('single'); x = x + 1` nicht den Wert `Inf`?

    Ihre Antwort:

Programmausgaben

Für jedes der folgenden Programmsegmente lesen Sie zuerst die Zeilen und schreiben Sie die Ausgabe an die dafür vorgesehene Stelle.

22. Wie lautet die Ausgabe des folgenden Skripts?

```
1   x = -386.123;
2   i1 = uint8(x)
3   i2 = int8(x)
4   i3 = int16(x)
5   i4 = floor(int16(x))
6   i5 = int16(floor(x))
7   i6 = i1 / i5
```

Ihre Antwort:

23. Wie lautet die Ausgabe des folgenden Skripts?

```
1   clear all;
2   x = 1 + i;
3   i = 3;
4   y = 1 - 1i;
5   x * y
```

Ihre Antwort:

24. Welche Werte haben x und y nach Ausführen des folgenden Skripts? Welchen Datentyp haben x und y?

```
1   A = [5 -3; 2 -4; 3 -1]
2   x = abs(A) > 2
3   y = A(x)
```

Ihre Antwort:

25. Wie lautet die Ausgabe des folgenden Skripts? Warum?

```
1   A = [1  2 -3; -4  5  6;  7 -8  9];
2   max(abs(A(A<2)))
```

Ihre Antwort:

Korrigieren Sie den Code

Für jedes der folgenden Codesegmente sollen Sie feststellen, ob ein Fehler enthalten ist. Falls ein Fehler vorliegt, markieren Sie diesen und spezifizieren Sie, ob es sich dabei um einen Semantik- oder Syntaxfehler handelt. Schreiben Sie die korrigierten Anweisungen jeweils in jeden dafür vorgesehenen Bereich unter der Problemstellung. Bemerkung: Es kann sein, dass ein Programm mehrere oder keine Fehler enthält.

26. Die folgende Funktion `nprint` haben Sie bereits im Abschnitt über den Datentyp `logical` kennen gelernt. Dort wurde darauf hingewiesen, dass diese Funktion bei bestimmten Parametern zur Endlos-Schleife wird. Wie können Sie das verhindern?

```
1   function nprint( n, s )
2
3   while(n)
4       fprintf(s)
5       n = n-1;
6   end
7   fprintf('\n');
8
9   end
```

Ihre Antwort:

27. Das Kommando `whos('X')` liefert die Attribute der Variablen `X`. Diese können als Struktur in einer anderen Variablen, z. B. `xAttr`, gespeichert werden.

```
≫ X = zeros(1000,1);
≫ xAttr = whos('X')
xAttr =
          name: 'X'
          size: [1000 1]
         bytes: 8000
         class: 'double'
        global: 0
        sparse: 0
       complex: 0
       nesting: [1x1 struct]
    persistent: 0
```

Das folgende Codesegment soll der Variablen `memSize` die Größe des von der Variablen `X` benötigten Speichers zuweisen.

```
1   X = zeros(1000,1);
2   xAttr = whos('X');
3   memSize = xAttr{bytes}
```

Ihre Antwort:

Praktikumsaufgabe

Lesen Sie die Aufgabenstellung, studieren Sie dann die vorgegebenen Programmzeilen. Ersetzen Sie dann die %% Kommentare im vorgegebenen Code durch Matlab-Anweisungen und führen Sie das Programm aus.

---

28. In dieser Praktikumsaufgabe sollen Sie vier MATLAB-Funktionen zur Berechnung von $n!$ implementieren. Diese sollen $n!$ jeweils auf eine andere Weise berechnen.

    (a) Schreiben Sie eine MATLAB-Funktion, die $n!$ mit Hilfe einer For-Schleife berechnet.

    (b) Schreiben Sie eine MATLAB-Funktion, die $n!$ mit Hilfe einer While-Schleife berechnet.

    (c) Schreiben Sie eine MATLAB-Funktion, die $n!$ rekursiv berechnet. Ihre Funktion soll, in Abhängigkeit von n, sich also selbst aufrufen.

    (d) Schreiben Sie eine MATLAB-Funktion, die $n!$ mit Hilfe einer der im Abschnitt über Vektoren und Matrizen vorgestellten Funktionen berechnet. Wenden Sie diese Funktion auf einen geeigneten Vektor an.

29. Wir möchten das MATLAB-Praktikum mit einer größeren Praktikumsaufgabe abschließen. Diese könnte Ihnen so oder so ähnlich auch in einer Numerik-Vorlesung begegnen.

    Sie dient zur Vorbereitung der Abschluss-Projekte und kann in dieser und der kommenden Woche bearbeitet werden.

    Stellen Sie sich vor, Sie haben für eine Funktion f die Funktionswerte $f_i = f(s_i)$ an mehreren Stellen s_i, $i = 0, \ldots, n$. Die Stellen s_i bezeichnen wir als Stützstellen. Die Funktion f selbst kennen Sie nicht.

    Nun möchten Sie ein Polynom p vom Grade kleiner oder gleich n finden, das die Funktion f an diesen Stützstellen s_i interpoliert. Es soll also gelten: $p(s_i) = f_i = f(s_i)$ für alle $i = 0, \ldots, n$.

    In der Numerik-Vorlesung werden Sie sehen, dass dieses Interpolationspolynom gegeben ist durch:

    $$p(x) = \sum_{i=0}^{n} f_i L_i^{(n)}(x).$$

    Dabei bezeichnet $L_i^{(n)}(x)$ das i-te Lagrange-Basis-Polynom zu den Stützstellen $s_0, \ldots, s_n$. Dieses ist gegeben durch

    $$L_i^{(n)}(x) = \prod_{j=0, j\neq i}^{n} \frac{x - s_j}{s_i - s_j}, \quad i = 0, \ldots, n.$$

    Ziel dieser Praktikumsaufgabe soll sein, dieses Interpolationsverfahren zu testen. Dazu zeichnen Sie eine gegebene Funktion f, die Sie in diesem Testbeispiel kennen, und das

zugehörige Interpolationspolynom p in ein Schaubild ein. Die Abweichung des Interpolationspolynoms p von der Funktion f ist ein Maß für die Güte der Interpolation. Sie werden sehen, dass die Wahl der Stützstellen erheblichen Einfluss auf die Güte der Interpolation hat. Die theoretischen Hintergründe werden Sie in der Vorlesung „Numerische Analysis“ kennen lernen.

(a) Schreiben Sie in einem ersten Schritt eine MATLAB-Funktion `lagrPol`, die das i-te Lagrange-Basis-Polynom an einem Vektor von x-Werten auswertet. Eingabe-Parameter für diese Funktion sollen sein:

- `i`: Bezeichnet das i-te Lagrange-Basis-Polynom,
- `s`: Ein Spaltenvektor $s = (s_0, \ldots, s_n)^T$ mit den $n+1$ Stützstellen und
- `x`: Ein Spaltenvektor $x = (x_0, \ldots, x_m)^T$ mit den Stellen, an denen das i-te Lagrange–Basis-Polynom ausgewertet werden soll.

Die Funktion soll einen Spaltenvektor $y = (y_0, \ldots, y_m)^T$ mit den Werten des Lagrange-Basis-Polynoms an den Stellen $x = (x_0, \ldots, x_m)^T$ zurück liefern.

(b) Im zweiten Schritt wollen wir Ihre MATLAB-Funktion `lagrPol` testen, und zwar mit dem sogenannten Gegenbeispiel von Runge. Schreiben Sie hierzu ein MATLAB-Skript und verwenden Sie zunächst die folgenden Daten:

- Das Gegenbeispiel von Runge verwendet die Funktion $f(x) = \frac{1}{1+x^2}$.
- Diese Funktion soll auf dem Intervall $[-5, 5]$ durch ein Polynom p interpoliert werden.
- Wählen Sie $n = 10$, das heißt, Sie haben insgesamt 11 Stützstellen $s_0, \ldots, s_{10}$.
- Wählen Sie diese Stützstellen zunächst äquidistant. Die erste Stützstelle s_0 soll die linke Intervallgrenze sein, also $s_0 = -5$, die letzte Stützstelle s_{10} soll die rechte Intervallgrenze sein, also $s_{10} = +5$. Der Abstand von einer Stützstelle zur nächsten soll für alle Stützstellen gleich sein.
- Wählen Sie eine Diskretisierung des Intervalls $[-5, 5]$ von $m = 1000$ äquidistanten Punkten.
- Berechnen Sie mit Ihrer Funktion `lagrPol` die Funktionswerte der $n + 1 = 11$ Lagrange-Basis-Polynome $L_i^{(n)}(x),\ (i = 0, \ldots, 10)$ an diesen Punkten. Es ist sinnvoll, diese Werte in einer $m \times (n+1) = 1000 \times 11$-Matrix zu speichern: Jede Zeile entspricht einem x-Wert aus dem Intervall $[-5, 5]$, jede Spalte einem der 11 Lagrange-Basis-Polynome.
- Plotten Sie die 11 Lagrange-Basis-Polynome in einem Schaubild.

(c) Im dritten Schritt werden wir die Funktion f des Gegenbeispiels von Runge und das berechnete Interpolationspolynom p in ein weiteres Schaubild einzeichnen.

- Berechnen Sie die Funktionswerte $f_i = f(s_i)$ an den Stützstellen und speichern Sie diese in einem Spaltenvektor $fs = (f_0, \ldots, f_n)^T = (f(s_0), \ldots, f(s_n))^T$.
- Machen Sie sich klar, dass Sie für einen festen, konkreten x-Wert aus Ihrer Diskretisierung des Intervalls $[-5, 5]$ die Summe $p(x) = \sum_{i=0}^{n} f_i L_i^{(n)}(x)$ einfach durch Multiplikation einer Zeile Ihrer Ergebnismatrix aus dem zweiten Schritt mit dem Spaltenvektor fs berechnen können.

- Berechnen Sie die Werte Ihres Interpolationspolynoms p an allen $m = 1000$ Stellen aus dem Intervall $[-5, 5]$.
- Plotten Sie die Funktion f und das Interpolationspolynom p in ein Schaubild. Beschriften Sie dieses Schaubild und geben Sie eine Legende an.

(d) Im vierten Schritt wollen wir ein weiteres Interpolationspolynom berechnen, das wir durch eine andere Wahl der Stützstellen erhalten.

- Wählen Sie die Stützstellen nach der Formel $s_i = 5\cos(\frac{2i+1}{2(n+1)}\pi), \quad i = 0, \ldots, n$ (wieder für n $= 10$).
- Berechnen Sie analog zum zweiten Schritt mit Ihrer Funktion `lagrPol` die Funktionswerte der $n + 1 = 11$ neuen Lagrange-Basis-Polynome und plotten Sie diese in ein drittes Schaubild.
- Berechnen Sie analog zum dritten Schritt die Werte des neuen Interpolationspolynoms und zeichnen Sie dieses in das Schaubild aus dem dritten Schritt ein.

(e) Welches Interpolationspolynom interpoliert die gegebene Funktion f besser? Entscheiden Sie diese Frage anhand der maximalen Abweichung von Interpolationspolynom p und Funktion f.

Ihre Antwort: