HANSER

Leseprobe

Ulrich Stein

Objektorientierte Programmierung mit MATLAB

Klassen, Vererbung, Polymorphie

ISBN (Buch): 978-3-446-44536-9

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44536-9

sowie im Buchhandel.

Vorwort

Zugegeben, es hat eine Weile gedauert, bis ich die Möglichkeiten der Objektorientierung in MATLAB® schätzen lernte. Im Jahr 2007 schrieb ich ein Lehrbuch, das sich mit dem „Programmieren mit MATLAB“ beschäftigt. Damals standen wir am Fachbereich Maschinenbau und Produktion der HAW Hamburg vor einem Problem: Unsere Studenten waren nicht recht zu motivieren, eine Programmiersprache zu erlernen. Wir „zwangen“ sie, sich längere Zeit mit Compiler, Linker und unverständlichen Fehlermeldungen herumzuquälen, um am Schluss als Ergebnis die Zahl 42 auf dem Bildschirm zu sehen. Ingenieure kommen inzwischen immer weniger mit Programmiersprachen wie C++ in Kontakt, ob in der Industrie oder im weiteren Verlauf des Studiums. Anders liegt die Situation beim Programm MATLAB. Dessen Funktionalität wird bei uns intensiv genutzt, beispielsweise zum Lösen von Differentialgleichungen oder zum Ansteuern von Robotern. Und besonders die grafischen Möglichkeiten von MATLAB fördern das Verständnis für viele technische Anwendungen. Deshalb habe ich damals, zusammen mit Kollegen, unser Bachelor-Modul „Angewandte Informatik“ auf die in MATLAB integrierte Programmiersprache umgestellt. Aus dem Skript zu dieser Vorlesung entstand das oben erwähnte Lehrbuch.

An die objektorientierte Programmierung (OOP) dachte ich dabei nur am Rande. Zwar hatte MATLAB auch damals bereits eingeschränkte OOP-Funktionalität, die ich in meinem Buch beschrieb. Die Vorgehensweise war jedoch noch sehr umständlich. Der große Schritt kam im Jahr 2008 mit der Einführung der *classdef*-Datei, die ähnlich wie in anderen OOP-Sprachen die Struktur einer Klasse festlegt. Durch die Definition von Handle-Klassen erhielt MATLAB außerdem eine Art von Referenzen.

Trotzdem, eine Zeit lang blieb ich noch skeptisch, ob man mit MATLAB wirklich vernünftig objektorientiert programmieren kann. Im Vergleich zu anderen OOP-Sprachen fehlten doch ein paar Dinge, beispielsweise die Möglichkeit, mehrere Methoden mit demselben Namen, aber unterschiedlicher Signatur zu deklarieren. Dies ist in MATLAB nicht vorgesehen. Aber hierfür – und auch für andere fehlende Bereiche – kann man sich recht einfach einen vernünftigen Workaround bauen.

Je länger ich mich mit der OOP-Funktionalität in MATLAB beschäftigte, desto mehr hat sie mich überzeugt. Und ich hoffe, dass ich auch Sie in diesem Buch dafür begeistern kann.

Nun zu Ihnen: Was erwarte ich von meinem Leser?

Idealerweise sollten Sie bereits ein wenig programmiert haben, nicht notwendigerweise in MATLAB. Zwar stelle ich Ihnen in diesem Buch, in den Kapiteln 1 und 2, die Oberfläche und die wichtigsten Sprachelemente von MATLAB vor – soweit es zum Verständnis der folgenden Abschnitte notwendig ist. Das ist jedoch nur eine knappe Zusammenfassung, eher gedacht für Umsteiger von anderen Programmiersprachen wie C++. Einem absoluten

Anfänger würde ich als Einstieg eher mein Buch „Programmieren mit MATLAB“ empfehlen – ein Lehrbuch, in dem Sie auch viele Beispiele und Übungen finden.

Doch zurück zum jetzigen Buch: Kapitel 3 beschreibt den objektorientierten Ansatz von MATLAB. Hier werden die zentralen Begriffe eingeführt, wie Klassen, Eigenschaften, Methoden, Datenkapselung, Vererbung, Polymorphie, Handle-Klassen etc.

In Kapitel 4 finden Sie längere Anwendungen aus der Physik und dem Maschinenbau. Dies soll den Umgang mit dem Erlernten vertiefen und weitere Tipps für ein strukturiertes Vorgehen geben.

Zum Abschluss liefert Kapitel 5 eine Befehlsreferenz und den Vergleich mit anderen OOP-Sprachen.

Dieses Buch ist aber kein Referenz-Handbuch für MATLAB. Die MATLAB-Funktionen werden oft nur so weit vorgestellt, wie es für die aktuelle Aufgabenstellung nötig ist. Für eine vollständige Definition der Funktionen sei auf die MATLAB-Hilfe verwiesen.

Die Idee, ein zweites Buch zu MATLAB zu schreiben, entstand während einer Unterredung mit Frau Franziska Jacob, M. A., meiner Ansprechpartnerin beim Fachbuchverlag Leipzig im Carl Hanser Verlag. Vielen Dank für ihr Engagement, mit dem sie das Projekt im Verlag durchsetzte. Dank auch an Frau Dipl.-Ing. (FH) Franziska Kaufmann, die mir beim Layout zur Seite stand. Dank an alle Kollegen, die mich zu diesem Projekt ermutigten und mir hilfreiche Tipps gaben, speziell Prof. Dr. rer. nat. Ivo Nowak, Prof. Dr. rer. nat. Thorsten Struckmann und Prof. Dr.-Ing. Jürgen Dankert. Und einen besonderen Dank an Elfriede Neubauer, die mir bei der stilistischen Überarbeitung eine große Hilfe war.

Die im Buch beschriebenen und abgebildeten Abläufe beziehen sich auf die Bedienoberfläche der Version MATLAB 2015a. Andere MATLAB-Versionen präsentieren sich dem Anwender zum Teil mit einer leicht abgewandelten Oberfläche. Lassen Sie sich deshalb nicht verwirren. Die vorgestellten Programme wurden mit verschiedenen Versionen getestet. Erweiterungen und die Lösungen der Aufgaben finden Sie auf meiner Homepage

www.Stein-Ulrich.de/Matlab/

Ich wünsche den Lesern, dass Ihnen das Programmieren auch Spaß macht und dass Ihnen möglichst viel vom hier präsentierten Stoff bei Problemlösungen nützt. Und nicht verdrängen oder vergessen: Informatik kann auch Schaden anrichten. Deshalb sollte jeder, der programmiert, sich überlegen, ob er sein Tun verantworten kann und will.

Hamburg, im August 2015

Ulrich Stein

Inhalt

2 Programmieren mit MATLAB

In Kapitel 1 diente uns MATLAB hauptsächlich als Taschenrechner, der einzelne Befehle verarbeitet. Jetzt, in Kapitel 2, wollen wir zeigen, wie man mit MATLAB programmiert – jedoch ohne die erweiterten Möglichkeiten der OOP. Zu Klassen und Objekten kommen wir erst in Kapitel 3.

Denn vorher müssen wir noch ein paar Dinge klären, zum Beispiel:

- Was sind Variablen und Daten?
- Wie schreibt man in MATLAB Funktionen?
- Was sind Kontrollstrukturen?
- Wie kommuniziert ein Programm mit der Außenwelt?
- Welche Grafik-Möglichkeiten bietet MATLAB?
- Was sind Handles?

2.1 Variablen, Daten, Typen

Variablen hatten wir bereits in Kapitel 1 kennengelernt, beispielsweise beim Aufruf:

```
>> x = int32( 5 );
```

Testen wir mit dem Befehl *whos*, welches Objekt dadurch erzeugt wurde:

```
>> whos x
  Name    Size    Bytes    Class
  x       1x1         4    int32 array
```

Es wurde eine Variable mit dem Namen *x* anlegt, vom Datentyp *int32* (ganze Zahl mit 32 Bits = 4 Bytes). Als Überschrift für den Datentyp verwendet MATLAB die Bezeichnung *Class*, was auf die Ähnlichkeit von MATLAB-Typen und benutzerdefinierten Klassen hinweist.

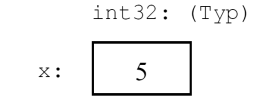

Bild 2.1 Variable *x*

Variablen sind Objekte, denen man einen Namen gegeben hat. Der Name einer Variablen beginnt in MATLAB immer mit einem Buchstaben. Dann können Ziffern und das „_“-Zeichen folgen. Zwischen Groß- und Kleinschreibung wird unterschieden. Bis zur Länge von 31 Zeichen sind die Namen eindeutig. Die Namen dürfen aber auch länger sein.

Dieser Name erlaubt den Zugriff auf die Objekte, um ihren Wert zu ändern oder um den Wert auszulesen – beispielsweise um ihn mit der Funktion *disp* auf dem Bildschirm auszugeben:

```
>> disp( x );
     5
```

Durch den Aufruf „`x = int32( 5 );`“ wurde die Variable *x* auch bereits mit Daten belegt. Als **Initialisierung** bekam *x* gleich zu Beginn den Wert 5 zugewiesen. Dafür benötigen die Objekte im Rechner einen Speicherplatz, je nach Datentyp unterschiedlich groß. Bei *int32* sind dies 4 Bytes = 32 Bits. 1 Byte ist eine Gruppierung von 8 Bits. Wie die Bits, also die abgelegten Daten im Speicher, vom Programm interpretiert werden, hängt vom Typ des Objekts ab.

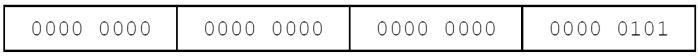

Bild 2.2 Bit-Folge (0 oder 1), 4 Bytes = 32 Bits

Die Zuordnung eines Namens und eines Datentyps zu einer Variablen bezeichnet man als **Deklaration** – als eine Vereinbarung, wie man den Inhalt der Variablen zu interpretieren hat. Wird bei einer Anweisung zusätzlich noch Speicherplatz belegt, nennt man dies **Definition**. Unsere Variable *x* wurde als Objekt vom Typ *int32* deklariert und durch die Zuweisung der Daten, der Zahl 5, auch gleichzeitig definiert.

In MATLAB verwendet man eine reine Deklaration für Variablen eher selten. Deklarationen sind normalerweise mit einer Definition verbunden. Eine Ausnahme bildet die *global*-Spezifikation, die darauf hinweist, dass diese Variable nicht eine lokale Variable der Funktion ist, sondern außerhalb definiert wurde und von dort beeinflusst werden kann. In der OOP wird die reine Deklaration bei den abstrakten Methoden eine wichtige Rolle spielen.

Objekte bezeichnet man auch als Exemplare oder **Instanzen** eines Typs (wobei der deutsche Begriff Instanz im täglichen Leben etwas anderes meint als das englische Wort *instance*).

In Abschnitt 1.2 hatten wir ein Objekt vom Typ *Auto* angelegt:

```
>> HH_BT_21 = Auto( 'VW_Golf', 2010 );
>> whos HH_BT_21
  Name      Size     Bytes  Class
  HH_BT_21   1x1       102  Auto
```

Hierbei bekam die Variable den Namen *HH_BT_21*. Bei der Initialisierung wurden dem Auto-Objekt nicht nur ein, sondern sogar zwei Werte zugewiesen. Während ein Ganzzahl-

Objekt vom Typ *int32* nur für eine einzige Zahl gedacht ist, kann ein beliebiges Objekt, je nach seinem internen Aufbau, auch eine größere Zahl von Daten enthalten. Diese Daten werden als seine Eigenschaften (*properties*) in den Objektvariablen abgespeichert.

Doch zu selbst definierten Klassen und deren Objekten kommen wir erst in Kapitel 3. In diesem Kapitel beschränken wir uns auf die Typen, die standardmäßig in MATLAB eingebaut sind, wie *double*, *int32*, *char* etc.

Sie müssen in MATLAB den Datentyp bei einer Variablendefinition nicht explizit angeben. Wenn Sie den Typ nicht spezifizieren, werden Zahlen automatisch als *double* behandelt:

```
>> y = 5;
>> whos y
  Name      Size     Bytes  Class
  y         1x1          8  double array
```

Texte erscheinen automatisch als *char*, genauer: als ein Feld von Zeichen, als ein Charakter-Array mit einer Zeile und mehreren Spalten, hier 1x5:

```
>> t = 'Willy';
>> whos t
  Name      Size     Bytes  Class
  t          1x5        10  char array
```

Dieser Typ-Automatismus beschränkt sich in MATLAB nicht nur auf die Definition von Variablen. Sie bekommen auch keinerlei Warnung, wenn Sie beispielsweise Text einer Variablen zuweisen, die eigentlich für Zahlen gedacht ist. MATLAB ist nicht **typsicher**. Typverletzungen werden nicht registriert, obwohl explizite Datentypen und deren Überprüfung die Programme sicherer machen. Ein Teil der Fehler würde dadurch automatisch erkannt werden.

Manchmal ist es notwendig, dass man Datenwerte in ein anderes Format umwandelt. Beispielsweise kann man zu einem Zeichen (vom Typ *char*) die ganze Zahl (vom Typ *int32*) bestimmen, mit der das Zeichen intern dargestellt wird, seinen sogenannten ASCII-Wert:

```
>> z = 'a';
>> whos z
  Name      Size     Bytes  Class
  z         1x1          2  char array
>> n = int32( z )
  n = 97
>> whos n
  Name      Size     Bytes  Class
  n         1x1          4  int32 array
```

Die int32-Anweisung erzeugt aus *z* die neue, ganzzahlige Variable *n*. Das Zeichen 'a', das in der Variablen *z* gespeichert ist, hat demnach den ASCII-Wert 97.

Typ-Umwandlungen bezeichnet man als Cast. MATLAB erzeugt einen Cast, wenn man den neuen Typ vor die Variable schreibt, beispielsweise „`int32(z)`“, analog der Definition von Variablen.

Wie bereits im ersten Kapitel erwähnt, sind die Standardtypen von MATLAB nicht einzelne Zahlen, sondern Matrizen, also zweidimensionale **Felder** (Arrays) mit einer gewissen Anzahl von Zeilen und Spalten. Eine Zelle des Arrays ist eindeutig durch die Angabe des Tupels „(Zeile,Spalte)“ definiert, im Beispiel die `Zelle(2,3)` in der 2. Zeile und 3. Spalte:

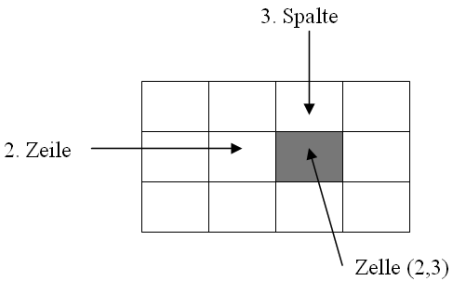

Bild 2.3 Zelle (2,3) einer 3x4-Matrix

Arrays kann man entweder durch eine Wertliste in eckigen Klammern erzeugen – beispielsweise legt die folgende Anweisung das Feld *c* an, mit zwei Zeilen und zwei Spalten:

```
>> c = [ 1 2; 3 4 ]
c =  1     2
     3     4
```

Oder man definiert explizit die Zellen des Arrays durch ihren Inhalt, beispielsweise:

```
>> a(1,1) = 0;
>> a(1,2) = 3;
>> a(1,3) = -1;
>> a
   a = 0  3  -1
```

Auf dieselbe Art kann man auf die Elemente eines Arrays zugreifen:

```
>> b = a(1,2)
b = 3
```

Um regelmäßig aufgebaute Arrays zu erzeugen, bietet MATLAB verschiedene Funktionen, etwa den *„:“-Operator*, mit dem man eindimensionale Felder definieren kann, beginnend mit einem *Startwert* im ersten Element und dann mit festem Abstand *Inkrement* bis zu einer Zahl, die kleiner oder gleich dem *Endwert* ist:

```
x = Startwert : Inkrement : Endwert;
```

Zum Beispiel für Zahlen zwischen 10 und 30 im Abstand von 5:

```
>> x = 10 : 5 : 30
x = 10  15  20  25  30
```

Eine ähnliche Funktionalität bietet die Funktion *linspace*, die ein Intervall in *n* Teile aufteilt, hier das Intervall von 5 bis 10 in *n* = 6 Teile:

```
>> y = linspace( 5, 10, 6 )
y =  5  6  7  8  9  10
```

Zur Analyse der Eigenschaften von Feldern stellt MATLAB eine Reihe von Funktionen zur Verfügung. Die Funktion *size* informiert über die Zahl der Zeilen und Spalten eines Arrays. Der Rückgabewert von *size* ist ein Feld mit der Zahl der Zeilen im ersten Element und der Zahl der Spalten im zweiten:

```
>> a = [1 0; 2 1; 3 4];
>> sz = size( a )
sz =  3   2
>> zeilen = sz(1)
zeilen = 3
>> spalten = sz(2)
spalten = 2
```

Bei eindimensionalen Feldern, also Zeilen- oder Spaltenvektoren, kann man die Länge auch über die Funktion *length* abfragen:

```
>> b = [1 2 3];
>> len = length( b )
len = 3
>> sz = size( b )
sz =  1   3
```

Es gibt noch weitere Funktionen zur Analyse von Feldern, zum Beispiel die Funktion *numel*, die auch für mehrdimensionale Felder die Gesamtzahl der Elemente liefert, oder die Funktion *nnz*, die für ein Feld die Zahl der Elemente bestimmt, die ungleich null sind.

Texte (Strings) werden von MATLAB als Character-Arrays mit einer Zeile und *n* Spalten behandelt. Für Strings gibt es spezielle Funktionen, wie *strcmp* zum Vergleich zweier Texte oder die Funktion *upper* zum Wandeln von Klein- in Großbuchstaben.

Sie können auch Arrays von Strings anlegen. Nur müssen Sie hierbei darauf achten, dass alle Texte in so einem Array exakt dieselbe Länge haben, was für die Praxis wenig brauchbar ist.

Um Felder zu erzeugen, die Elemente mit unterschiedlichen Typen aufnehmen können, zum Beispiel unterschiedlich lange Texte, gibt es den Datentyp **Cell-Array**. Cell-Arrays können Sie wie normale Arrays mittels einer Wertliste erzeugen. Nur müssen Sie dazu geschweifte Klammern verwenden, beispielweise `c = { 1, 'Willy', 2 }`. Auch zur Abfrage des Inhaltes einer Zelle brauchen Sie die geschweiften Klammern, zum Beispiel für das dritte Element `c{3}`. Bei der Verwendung von runden Klammern, wie `c(3)`, erhalten Sie den Wert als Inhalt einer Zelle:

```
>> c = {1, 'Willy', 2 }
c = [1]    'Willy'    [2]
>> c3 = c(3)
```

```
c3 = [2]
>> whos c3
  Name   Size  Bytes  Class   Attributes
  c3     1x1      68  cell
>> v3 = c{3}
v3 = 2
>> whos v3
  Name   Size  Bytes  Class   Attributes
  v3     1x1       8  double
```

Des Weiteren bietet MATLAB den Datentyp *struct*, der einzelne Daten zu einer Struktur zusammenfasst, beispielsweise für ein Adressenverzeichnis mit den Daten der einzelnen Personen:

```
>> p = struct( 'Name', 'Willy', 'Adresse', 'Berliner Tor 21' )
p =    Name: 'Willy'
    Adresse: 'Berliner Tor 21'
```

Bei der Definition eines Struct gibt man paarweise die Komponenten des Struct und die zugehörenden Werte an. In unserem Beispiel besitzt der Struct *p* die beiden Komponenten *Name* und *Adresse*, die mit den Daten 'Willy' bzw. 'Berliner Tor 21' belegt sind. Auf den Inhalt der Komponenten können Sie mit Hilfe des Punktoperators zugreifen, analog dem Lesen der Eigenschaften eines Objekts, wie in Abschnitt 1.2 gezeigt:

```
>> Adr = p.Adresse
Adr = Berliner Tor 21
```

Welche Operationen mit den Daten möglich sind, hängt vom Datentyp ab. Zahlen können Sie zum Beispiel addieren, Texte ausdrucken. Es macht aber wenig Sinn, zwei Textzeichen zu addieren. Was ergäbe wohl die Addition 'a' + 'b'?

Als **Datenverarbeitung** bezeichnet man jede Art von Operation mit Daten, also wie oben angeführt die Addition von Zahlen oder das Ausdrucken von Text. Datenverarbeitung erfolgt typischerweise in drei Schritten:

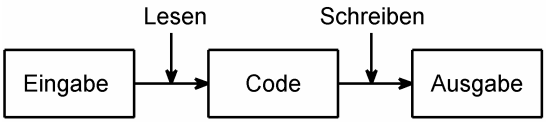

Bild 2.4 Datenfluss

- Eingabe der Daten, zum Beispiel über die Tastatur,
- Verarbeitung der Daten, durch den Programm-Code von Funktionen,
- Ausgabe des Ergebnisses, zum Beispiel auf dem Bildschirm.

Für größere Aufgaben zerlegt man ein Programm in Teilprobleme, das heißt in einzelne Funktionen mit meist nicht mehr als einer Seite Programmcode. Größere Funktionen sind erfahrungsgemäß nicht zu überblicken und führen schnell zu Fehlern.

In den Funktionen werden die einzelnen Berechnungen durchgeführt. Hierzu dienen Ausdrücke und Zuweisungen, wie die Berechnung des Umfangs aus dem Radius durch die Multiplikation mit 2 π:

```
Umfang = 2 * pi * Radius;
```

Im Programm werden diese Anweisungen **sequentiell** ausgeführt, also zeilenweise nacheinander abgearbeitet. Innerhalb einer Zeile erfolgt die Auswertung zuerst von innen nach außen, wie Sie es bei Klammern in der Mathematik gewohnt sind, und danach in den Ausdrücken, wie beim Lesen, von links nach rechts. **Wertzuweisungen** erfolgen an Variablen, die links vor einem Gleichheitszeichen stehen.

Zum Beispiel im folgenden Aufruf die Addition des Wertes von `r = 1.0` mit der Zahl 3, dem Ziehen der Wurzel aus 4 (= 3 + 1) mit der Funktion *sqrt* und der anschießenden Multiplikation mit 0.5, was für *u* letztendlich den Wert 1.0 ergibt:

```
r = 1.0;
u = 0.5 * sqrt( 3 + r );
```

Mit den Kontrollstrukturen, den Auswahlanweisungen und Schleifen, können Sie innerhalb einer Funktion den sequentiellen Ablauf verändern. Dies ist aber erst Thema in einem der späteren Abschnitte. Als Nächstes wollen wir uns anschauen, wie man in MATLAB Funktionen schreibt.

2.2 Funktionen

Funktionen dienen dazu, Teilprobleme zu lösen. Dazu übergibt man ihnen eine Anzahl von Daten, aus denen die Funktion die nötigen Ergebnisse berechnet und diese dem Aufrufer zurückgibt.

Vom Blickpunkt des Aufrufers aus sind Funktionen eine Art Black Box, mit Eingangs- und Rückgabewerten – beispielsweise kann man die Zahl 9 als Eingabe an die Wurzel-Funktion *sqrt* geben, die dann als Rückgabe hoffentlich die Zahl 3 liefert.

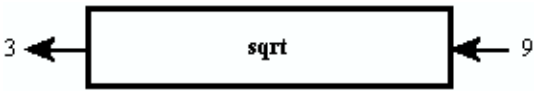

Bild 2.5 Black Box *sqrt*-Funktion

In MATLAB lautet dieser Aufruf folgendermaßen:

```
>> y = sqrt( 9 )
y = 3
```

Wie MATLAB oder eine andere Programmiersprache die Wurzel von 9 berechnet, das wird dem Anwender im Allgemeinen nicht mitgeteilt. Dieser verlässt sich darauf, dass das Ergebnis richtig ist. Es kann (bei etwas komplizierteren Funktionen) auch vorkommen, dass bei einem neuen Release von MATLAB der Berechnungsalgorithmus verändert wird, weil man zum Beispiel ein Verfahren entdeckt hat, das schneller ist.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z