MATLAB® – Simulink® – Stateflow®

Grundlagen, Toolboxen, Beispiele

von
Dr.-Ing. Anne Angermann
Dr.-Ing. Michael Beuschel
Dr.-Ing. Martin Rau
Dipl.-Ing. Ulrich Wohlfarth

7., aktualisierte Auflage

Oldenbourg Verlag München

Dr.-Ing. Anne Angermann, TU München
Dr.-Ing. Michael Beuschel, Conti Temic microelectronic GmbH, Ingolstadt
Dr.-Ing. Martin Rau, BMW AG, München
Dipl.-Ing. Ulrich Wohlfarth, Patentanwaltskanzlei Charrier, Rapp & Liebau, Augsburg

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Lektorat: Martin Preuß
Herstellung: Constanze Müller
Einbandgestaltung: hauser lacour
Gesamtherstellung: Grafik + Druck GmbH, München

Dieses Papier ist alterungsbeständig nach DIN/ISO 9706.

ISBN 978-3-486-70585-0

Vorwort zur siebten Auflage

Die 7. Auflage wurde im Hinblick auf das MATLAB Release 2011a vollständig überarbeitet und ergänzt, ohne jedoch den Umfang des Buches wesentlich zu erhöhen. Das Ergebnis ist ein kompaktes Lehrbuch für den Einsteiger und gleichzeitig ein übersichtliches Nachschlagewerk für den fortgeschrittenen MATLAB-Anwender.

Die dem Buch beigelegte CD-ROM ist mit einem benutzerfreundlichen HTML-Navigator versehen. Neben den überarbeiteten Beispielen, Übungsaufgaben und Lösungen enthält die CD-ROM wieder eine Bibliothek der Autoren mit nützlichen Extras für MATLAB und Simulink.

Danken möchten wir zuallererst Herrn Professor i. R. Dierk Schröder für seine umfassende Unterstützung bei der Erstellung dieses Buches. Ausgehend von seiner Idee und seinem unermüdlichen Engagement, eine längst notwendige Vorlesung zum Softwarepaket MATLAB und Simulink für Studenten der Fachrichtungen Energietechnik, Automatisierungstechnik und Mechatronik ins Leben zu rufen, konnten wir in sehr freier und kollegialer Arbeitsweise ein Skriptum zu dieser Vorlesung erstellen.

Nach einem ruhestandsbedingten Wechsel kann dieses Engagement unter Leitung von Herrn Professor Ralph Kennel, Ordinarius des Lehrstuhls für Elektrische Antriebssysteme und Leistungselektronik der Technischen Universität München, in vollem Umfang fortgesetzt werden. Für seine immer wohlwollende Unterstützung sowie für die vertrauensvolle und angenehme Zusammenarbeit danken wir ihm daher sehr herzlich.

Die äußerst positive Resonanz von Studenten unterschiedlichster Fachrichtungen sowie zahlreiche Anfragen aus Forschung und Industrie ermutigten uns, das ursprüngliche Skriptum einem größeren Leserkreis zu erschließen und als Buch zu veröffentlichen. Aufgrund der regen Nachfrage erscheint dieses Buch nun bereits in seiner 7. Auflage. Nicht zuletzt danken wir daher unseren zahlreichen Lesern, allen Dozenten, Studenten und Kollegen, die uns dabei mit ihren Anregungen und ihrer stets wohlwollenden Kritik unterstützten und noch unterstützen werden.

Für Verbesserungvorschläge und Hinweise auf noch vorhandene Fehler sind wir jederzeit dankbar und werden sie auf der Internetseite **www.matlabbuch.de** neben weiteren aktuellen Informationen rund um MATLAB veröffentlichen.

Dem Oldenbourg Verlag danken wir für die Bereitschaft, dieses Buch zu verlegen. Besonderer Dank gilt hierbei Frau Mönch, Frau Dr. Bromm, Herrn Dr. Preuß sowie Herrn Schmid für ihre hilfreiche Unterstützung während der Entstehung und für die Übernahme des Lektorats.

München

Anne Angermann
Michael Beuschel
Martin Rau
Ulrich Wohlfarth

Vorwort zur ersten Auflage

Das vorliegende Buch „Matlab – Simulink – Stateflow“ wendet sich an *Studenten* und *Ingenieure*, die das Simulationswerkzeug MATLAB/Simulink effizient einsetzen wollen.

Zielsetzung dieses Buches ist es, dem Leser einen direkten Zugang zum Anwenden der umfangreichen Möglichkeiten dieses Programmpaketes zu ermöglichen. Es wird prägnant dargestellt, welche wesentlichen Funktionen in MATLAB und Simulink verfügbar sind – beispielsweise Ein- und Ausgabe, grafische Darstellungen oder die verschiedenen Toolboxen und Blocksets für die Behandlung zeitkontinuierlicher und zeitdiskreter linearer und nichtlinearer Systeme sowie ereignisdiskreter Systeme – und wie diese Funktionen zu nutzen sind. Dies wird außerdem an zahlreichen Beispielen erläutert. Um den Ansatz *prägnante Einführung* zu unterstützen, sind zudem zu jedem Abschnitt Übungsaufgaben mit deren Lösungen auf einer CD-ROM beigefügt. Der Leser hat somit die Option, sein Verständnis des betreffenden Kapitels selbständig zu vertiefen und sofort praktisch zu überprüfen.

Um den Umfang dieses Buches zu begrenzen, sind keine theoretischen Abhandlungen z.B. über Integrationsverfahren, die Grundlagen der Regelungstechnik oder der Signalverarbeitung bzw. deren Implementierungen enthalten. Für den interessierten Leser finden sich jedoch in jedem Kapitel Hinweise auf vertiefende Literatur.

Ausgangspunkt dieses Buches war meine Überlegung, Studenten so früh wie möglich mit den Vorteilen des wertvollen Werkzeuges *Simulation* bekannt zu machen. Dies beginnt bei regelungstechnischen Aufgabenstellungen bereits bei der Modellbildung der Komponenten des betrachteten Systems und der Erstellung des Simulationsprogramms. Es setzt sich fort bei der Validierung der Modelle sowie der grafischen Veranschaulichung des Systemverhaltens in den verschiedenen Arbeitspunkten und bei unterschiedlichen Randbedingungen, etwa aufgrund variabler Parameter, der Systemanalyse, der Reglersynthese sowie der Optimierung des Gesamtsystems. Ausgehend von diesen Überlegungen wurde ein Konzept für dieses Buch erarbeitet, damit der Leser die verschiedenen Aufgabenstellungen möglichst anschaulich kennen lernt. Dieses Konzept wurde aufgrund langjähriger Erfahrung bei Vorlesungen, Studien- und Diplomarbeiten, Dissertationen und Industrieprojekten kontinuierlich verbessert.

Meine Mitarbeiter und ich hoffen, allen Interessierten in Studium und Beruf mit diesem Buch einen anschaulichen und effizienten Einstieg in die Simulation mit MATLAB und Simulink geben zu können.

München Dierk Schröder

Inhaltsverzeichnis

1 Einführung

MATLAB ist ein umfangreiches Softwarepaket für numerische Mathematik. Wie der Name MATLAB, abgeleitet von MATrix LABoratory, schon zeigt, liegt seine besondere Stärke in der Vektor- und Matrizenrechnung. Unterteilt in ein Basismodul und zahlreiche Erweiterungspakete, so genannte Toolboxen, stellt MATLAB für unterschiedlichste Anwendungsgebiete ein geeignetes Werkzeug zur simulativen Lösung der auftretenden Problemstellungen dar.

Das **Basismodul** verfügt neben den obligatorischen Ein- und Ausgabefunktionen und Befehlen zur Programmablaufsteuerung über eine Vielzahl mathematischer Funktionen, umfangreiche zwei- und dreidimensionale Visualisierungsmöglichkeiten, objektorientierte Funktionalität für die Programmierung interaktiver Anwenderoberflächen und Schnittstellen zu Programmiersprachen (C, Fortran, Java) und Hardware.

Zusätzlich werden zahlreiche **Toolboxen (TB)** mit erweitertem Funktionsumfang angeboten. Im Rahmen dieses Buches werden die Toolboxen für Regelungstechnik, Control System Toolbox (Kap. 5), Signalverarbeitung, Signal Processing Toolbox (Kap. 6), und Optimierung, Optimization Toolbox (Kap. 7), vorgestellt. Eine Sonderstellung nimmt die Toolbox Simulink ein, die eine grafische Oberfläche zur Modellierung und Simulation physikalischer Systeme mittels Signalflussgraphen zur Verfügung stellt (Kap. 8–11). Hierzu kommt noch *Stateflow* (Kap. 12), eine Ergänzung zu Simulink (Blockset), mit dem endliche Zustandsautomaten modelliert und simuliert werden können.

Das vorliegende Buch soll Einsteigern einen raschen und intuitiven Zugang zu MATLAB vermitteln, sowie erfahrenen Anwendern eine Vertiefung ihres Wissens ermöglichen. Zahlreiche Programmbeispiele aus den Gebieten Elektrotechnik, Mechanik, Regelungstechnik und Physik veranschaulichen das Gelesene und können selbst nachvollzogen werden. Am Ende der einzelnen Abschnitte finden sich die jeweils wichtigsten Befehle in tabellarischen Übersichten zusammengefasst, um allen MATLAB-Anwendern im Sinne eines kompakten Nachschlagewerks zu dienen.

In **Kapitel 2 – MATLAB Grundlagen** wird ein Einstieg in die grundlegende Programmierung mit MATLAB gegeben. Alle wichtigen Typen von Variablen, mathematischen Funktionen und Konstrukte zur Programmablaufsteuerung werden erklärt. Besonders wird die Möglichkeit hervorgehoben, den Funktionsumfang von MATLAB durch selbst geschriebene Funktionen zu erweitern und umfangreiche Berechnungen in MATLAB-Skripts zu automatisieren. In **Kapitel 3 – Eingabe und Ausgabe in MATLAB** liegt der Schwerpunkt auf den Visualisierungsfunktionen: Ein- und Ausgabe von Daten am Command Window, zwei- und dreidimensionale Grafikfunktionen für die Darstellung von Daten und Berechnungsergebnissen und der Import und Export von Daten. Ferner wird in die Programmierung von grafischen Benutzeroberflächen, so genannten Graphical User Interfaces (GUI) zur benutzerfreundlichen Gestaltung von MATLAB-

Programmen eingeführt. Das **Kapitel 4 – Differentialgleichungen in MATLAB** führt in die numerische Lösung von Differentialgleichungen ein, wofür in MATLAB effiziente Algorithmen implementiert sind. Die Lösung von Anfangswertproblemen mit und ohne Unstetigkeiten, von differential-algebraischen Gleichungen (DAE), von Differentialgleichungen mit Totzeiten (DDE), von Randwertproblemen und von einfachen partiellen Differentialgleichungen (PDE) wird behandelt.

Nachdem nun alle wichtigen Grundfunktionen von MATLAB bekannt sind, erfolgt in **Kapitel 5 – Regelungstechnische Funktionen** eine detaillierte Darstellung der Fähigkeiten der Control System Toolbox. Dieses wichtige Thema beinhaltet alle Schritte von der Darstellung linearer zeitinvarianter Systeme als LTI-Objekt über deren Manipulation und Analyse bis zum Reglerentwurf. Hervorgehoben sei hier die sehr gute Verknüpfungsmöglichkeit der Control System Toolbox mit der grafischen Erweiterung Simulink zur Programmierung und Untersuchung dynamischer Systeme.

Die digitale Signalverarbeitung ist für viele Disziplinen der Ingenieurwissenschaften relevant und wird durch die Behandlung der Signal Processing Toolbox in **Kapitel 6 – Signalverarbeitung** berücksichtigt. Es werden numerische Verfahren zur Interpolation abgetasteter Signale, zur Spektralanalyse und Korrelation sowie zur Signalmanipulation durch analoge und digitale Filter behandelt.

Viele Problemstellungen erfordern die Minimierung oder Maximierung mehr oder weniger komplexer Funktionen. Da dies häufig nur numerisch möglich ist, wird in **Kapitel 7 – Optimierung** die Optimization Toolbox mit ihren Algorithmen zur Nullstellensuche und Minimierung bzw. Maximierung behandelt. Insbesondere wird auf die Minimierung unter Nebenbedingungen, die Methode der kleinsten Quadrate, die lineare Programmierung und die Optimierung der Reglerparameter eines Simulink-Modells eingegangen.

Nun erfolgt der Übergang zur grafischen Erweiterung **Simulink**, das zur grafischen Modellierung und Simulation dynamischer Systeme konzipiert ist und nahtlos mit allen Grundfunktionen und Toolboxen von MATLAB zusammenarbeitet. Das **Kapitel 8 – Simulink Grundlagen** stellt einen Einstieg in die blockorientierte grafische Programmierumgebung mit den zugehörigen Funktionsbibliotheken dar. Die Erstellung von Modellen wird von Anfang an erklärt. Ferner werden alle wichtigen Datentypen, Simulationsparameter und Tools für Verwaltung und Hierarchiebildung behandelt. In den **Kapiteln 9 – Lineare und nichtlineare Systeme in Simulink** und **10 – Abtastsysteme in Simulink** wird jeweils auf typisch vorkommende Systemklassen näher eingegangen. In **Kapitel 11 – Regelkreise in Simulink** erfolgt ein kompletter Reglerentwurf mit anschließender Simulation für eine Gleichstrom-Nebenschluss-Maschine, wodurch vor allem das Zusammenspiel zwischen der Control System Toolbox und Simulink aufgezeigt wird.

Das abschließende **Kapitel 12 – Stateflow** behandelt die Modellierung und Simulation ereignisdiskreter endlicher Zustandsautomaten. Diese Klasse von Systemen tritt häufig bei der Steuerung und Überwachung technischer Prozesse auf. Durch die Kombination mit Simulink lassen sich gemischt kontinuierliche und ereignisdiskrete Modelle erstellen.

Für eine übersichtliche Darstellung werden folgende **Schriftkonventionen** verwendet: Variablen werden *kursiv* geschrieben und optionale Parameter in eckigen Klammern [ ] angegeben. Die Kennzeichnung der MATLAB-Befehle, der Ein- und Ausgaben sowie der Dateinamen erfolgt durch `Schreibmaschinenschrift`.

Auf der beiliegenden **CD-ROM** findet sich der Programm-Code aller im Buch gezeigten Beispiele kapitelweise in Verzeichnissen abgelegt. Weiter enthält jedes Kapitel-Verzeichnis ein Unterverzeichnis `loesungen`, in dem sich die Musterlösungen (MATLAB- und Simulink-Dateien) zu den Übungsaufgaben befinden. Kommentiert werden diese Programme in der Datei `loesung.pdf`, die alle Musterlösungen, evtl. grafische Ausgaben und Kommentare zu den Lösungswegen enthält. Zusätzlich enthält die CD-ROM das Verzeichnis `userlib` mit einer **Extra-Bibliothek**, welche eine Auswahl nützlicher MATLAB-Funktionen und Simulink-Blöcke der Autoren enthält. Zur komfortablen Benutzung der CD-ROM steht ein **HTML-Navigator** zur Verfügung, der durch Aufruf der Datei `start.html` gestartet wird.

Ferner stehen aktualisierte Inhalte der CD-ROM, Errata und weitere aktuelle Informationen zum Buch und rund um MATLAB auf der Internetseite **www.matlabbuch.de** zur Verfügung.

2 MATLAB Grundlagen

2.1 Erste Schritte mit MATLAB

Zum Starten von MATLAB wird die Datei `matlab.exe` oder die entsprechende Verknüpfung im Startmenü oder auf dem Desktop aufgerufen. Beendet wird MATLAB durch Schließen des MATLAB-*Desktops*, durch Eingabe der Befehle `quit` bzw. `exit` oder mit der Tastenkombination *Strg + Q*.

2.1.1 Der MATLAB-Desktop

Nach dem Starten von MATLAB erscheint der so genannte MATLAB-*Desktop*, eine integrierte Entwicklungsumgebung, die folgende Fenster enthalten kann (aber nicht muss – eine mögliche Konfiguration zeigt Abb. 2.1):

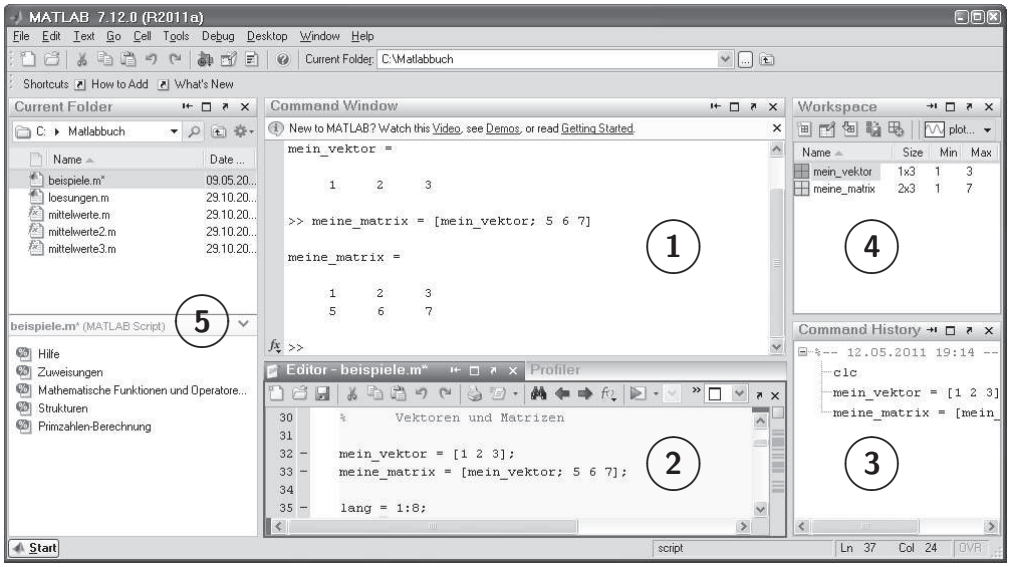

Abb. 2.1: MATLAB-*Desktop (Beispiel-Konfiguration)*

Command Window ①: Dieser Bereich stellt das Kernstück von MATLAB dar. Hier werden alle Eingaben in den so genannten *Workspace* gemacht und die Berechnungen ausgegeben. Der Prompt `>>` signalisiert die Eingabebereitschaft. Jede Eingabe wird mit der Return-Taste abgeschlossen. Die Regel „Punkt vor Strich“ sowie Klammern gelten

wie gewohnt. Zahlen können auch mit Exponent `e` (bzw. `E`) eingegeben werden.

```
>> (40^3 + 3*2e3) / 7
ans =
       10000
```

Tippt man die ersten Buchstaben eines gesuchten Befehls ein, erhält man mit der Tabulator-Taste eine Liste der passenden MATLAB-Befehle.

Editor ②: Mit dem MATLAB-Editor können Skripts und Funktionen erstellt und bearbeitet werden. Er bietet neben den Funktionen zum Editieren von Text die bei Programmier-Umgebungen üblichen Möglichkeiten zum schrittweisen Abarbeiten des Programm-Codes, zur Syntax-Prüfung, zum Debuggen von Fehlern etc.

Command History ③: Hier werden die im Command Window eingegebenen Befehle gespeichert und angezeigt. Durch Doppelklicken können die Befehle wiederholt werden; auch lassen sich einzelne oder mehrere Befehle ausschneiden, kopieren oder löschen. Gespeicherte Befehle können im Command Window ebenso mit den Kursortasten ↑ und ↓ abgerufen werden; tippt man zuvor den Beginn der gesuchten Befehlszeile ein, erscheinen nur die passenden Befehle.

Workspace Browser ④: Hier werden alle im Workspace existierenden Variablen mit Namen und Wert angezeigt. Über das Kontextmenü der Namensleiste lassen sich weitere Eigenschaften wie Größe und Datentyp (`Class`) sowie statistische Größen (`Min`, `Max`, etc.) auswählen. Zusätzlich lassen sich Variablen als Datei abspeichern, aus Dateien einlesen, grafisch ausgeben oder mittels des Variable Editors einfach verändern.

Current Folder Browser ⑤: Mittels des Current Folder Browsers lassen sich das gerade aktuelle Verzeichnis des Workspace einsehen oder ändern, Dateien öffnen, Verzeichnisse erstellen und andere typische Verwaltungs-Aufgaben durchführen.

Profiler (ohne Abbildung, da Registerkarte hinter ②): Der Profiler analysiert die Rechenzeit einzelner Befehle in MATLAB-Skripts und -Funktionen, siehe auch Kap. 2.6.2.

Shortcut-Leiste (unterhalb der Button-Leiste): Shortcuts können mit einem oder mehreren MATLAB-Befehlen belegt werden, die sich dann per Mausklick jederzeit ausführen lassen. Dies eignet sich insbesondere zum Wechseln in häufig genutzte Verzeichnisse sowie zum Vorbelegen oder Löschen von Variablen.

Das Aussehen des Desktops kann auf vielfältige Weise individuell angepasst werden. Die einzelnen Fenster lassen sich durch Ziehen der Titelleisten mit der Maus neu anordnen. Unter dem Menü-Punkt *Desktop* können die einzelnen Fenster zu- und abgeschaltet werden. Insbesondere kann mit *Undock* bzw. dem Button  das aktuelle Fenster vom Desktop separiert und mit *Dock* bzw.  wieder dort integriert werden.

Über das Menü *Desktop/Save Layout* können Desktop-Layouts gespeichert werden. Auf diese Weise gespeicherte sowie verschiedene Standard-Layouts können über das Menü *Desktop/Desktop Layout* jederzeit wiederhergestellt werden. Weitere Vorgaben können im Menü *File/Preferences* eingestellt werden. Dort können unter der Rubrik *Toolbars* auch die Buttonleisten der einzelnen Teilfenster individuell angepasst werden.

2.1.2 Die MATLAB-Hilfe

MATLAB stellt eine umfangreiche Hilfe zur Verfügung. Mit `help` [*befehl*] kann die Hilfe zu einem Befehl direkt im Command Window aufgerufen und dort ausgegeben werden.

```
>> help sqrt
 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also sqrtm, realsqrt, hypot.

    Reference page in Help browser
       doc sqrt
```

Alternativ kann der Hilfe-Browser (siehe Abb. 2.2) über das Menü, den Button im MATLAB-Desktop oder über die Befehle `helpwin` oder `doc` *befehl* aufgerufen werden. Mit dem Befehl `lookfor` *suchstring* kann auch die erste Kommentarzeile aller MATLAB-Dateien im MATLAB-Pfad nach *suchstring* durchsucht werden.

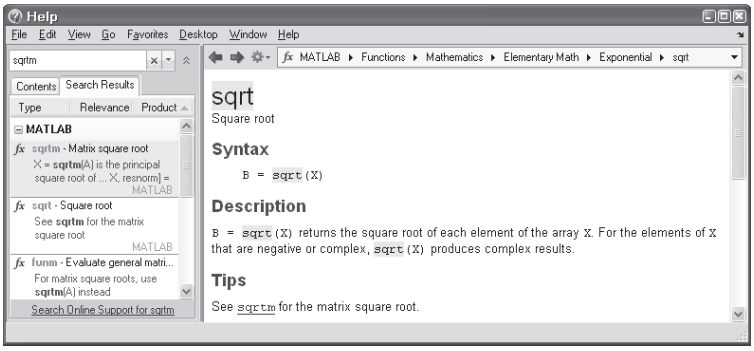

Abb. 2.2: MATLAB-*Hilfe-Browser*

Die Verwendung der MATLAB-Hilfe empfiehlt sich insbesondere bei der Suche nach neuen Befehlen und Funktionen. Oft finden sich am Ende des Hilfetextes Querverweise auf verwandte oder ergänzende Funktionen.

Hilfe

`help` [*befehl*]	MATLAB Online-Hilfe im Command Window
`helpwin` [*befehl*]	MATLAB Online-Hilfe im Hilfe-Browser
`doc` [*befehl*]	MATLAB Online-Hilfe im Hilfe-Browser
`lookfor` *suchstring*	Suche in erster Kommentarzeile aller MATLAB-Dateien

2.1.3 Zuweisungen

Mit = wird in MATLAB eine Variable definiert und ihr ein Wert zugewiesen.[1)] Eine vorherige Deklaration der Variablen ist nicht erforderlich. Die Eingabe des Variablennamens ohne Zuweisung gibt den aktuellen Wert der Variablen aus.

Alle Variablen bleiben im so genannten *Workspace* sichtbar, bis sie gelöscht werden oder MATLAB beendet wird. Variablennamen[2)] können aus bis zu 63 Buchstaben oder (mit Ausnahme des ersten Zeichens) Zahlen sowie Unterstrichen (_) bestehen. Bei Variablen wird Groß- und Kleinschreibung berücksichtigt.

Die Ausgabe eines Ergebnisses lässt sich z.B. bei Aneinanderreihung mehrerer Anweisungen mit Semikolon (;) unterdrücken. Ein Komma (,) zur Trennung von Anweisungen dagegen gibt das Ergebnis aus. Entfällt das Komma am Zeilenende, wird das Ergebnis ebenfalls ausgegeben.

```
>> variable_1 = 25; variable_2 = 10;
>> variable_1
variable_1 =
    25
>> a = variable_1 + variable_2, A = variable_1 / variable_2
a =
    35
A =
    2.5000
```

Bestimmte Variablennamen, wie `pi`, `i`, `j`, `inf` sind reserviert[3)] und sollten daher nicht anders belegt werden; `eps` bezeichnet die relative Fließkomma-Genauigkeit (z.B. `eps = 2.2204e-016`). Bei mathematisch nicht definierten Operationen (z.B. `0/0`) wird `NaN` (*Not a Number*) ausgegeben.

```
>> 1 / 0
ans =
   Inf
```

Zuweisungen

=	Variablenzuweisung	;	Unterdrückung der Ausgabe
		,	Ausgabe (aneinander gereihte Befehle)

Vorbelegte Variablen und Werte

pi	Kreiszahl π	ans	Standard-Ausgabevariable
i, j	Imaginäre Einheit $\sqrt{-1}$	eps	relative Fließkomma-Genauigkeit
inf	Unendlich ∞	NaN	Not a Number (ungültiges Ergebnis)

[1)] Ohne Zuweisung speichert MATLAB das Ergebnis in der Variablen `ans` (siehe Beispiel auf Seite 6).

[2)] Variablen dürfen nicht namensgleich mit MATLAB-Funktionen und selbst geschriebenen Funktionen sein. Im Zweifelsfall wird eine Überschneidung mit `which -all` *name* angezeigt.

[3)] Für die Eulersche Zahl ist keine Variable reserviert; sie kann aber mit `exp(1)` erzeugt werden.

2.1.4 Mathematische Funktionen und Operatoren

In MATLAB steht eine Vielzahl mathematischer Funktionen zur Verfügung; einige sind im Folgenden zusammengestellt. Eine weiterführende Übersicht erhält man durch Eingabe von `help elfun` und `help datafun`. Alle aufgeführten Funktionen können auch auf Vektoren und Matrizen angewandt werden (diese werden in Kap. 2.2.2 behandelt).

Mathematische Funktionen und Operatoren

`+ - * / ^`	Rechenoperatoren	`exp(`x`)`	Exponentialfunktion
`mod(`x,y`)`	x Modulo y	`log(`x`)`	Natürlicher Logarithmus
`rem(`x,y`)`	Rest nach Division x/y	`log10(`x`)`	Zehner-Logarithmus
`sqrt(`x`)`	Quadratwurzel	`erf(`$x/\sqrt{2}$`)`	Normalverteilung $\int_{-x}^{x} \varphi(x)dx$
`abs(`x`)`	Betrag	`real(`x`)`	Realteil
`sign(`x`)`	Signum (Vorzeichen)	`imag(`x`)`	Imaginärteil
`round(`x`)`	Runden	`conj(`x`)`	komplexe Konjugation
`ceil(`x`)`	Runden nach oben	`angle(`x`)`	Phase einer
`floor(`x`)`	Runden nach unten		komplexen Zahl

Trigonometrische Funktionen

`sin(`x`)`	Sinus	`tan(`x`)`	Tangens
`cos(`x`)`	Cosinus	`cot(`x`)`	Cotangens
`sind(`x`)`	Sinus (x in Grad)	`atan(`y/x`)`	Arcus-Tangens $\pm\pi/2$
`cosd(`x`)`	Cosinus (x in Grad)	`atan2(`y,x`)`	Arcus-Tangens $\pm\pi$

2.2 Variablen

Um umfangreiche Daten effizient zu verarbeiten, bietet MATLAB für Variablen unterschiedliche Datentypen an. Aus allen Datentypen können Arrays erzeugt werden. Im Folgenden wird die Verwendung solcher Variablen sowie zugehöriger Funktionen und Operatoren erläutert. Für den schnellen Einstieg kann jedoch Kap. 2.2.1 übersprungen werden, da MATLAB in aller Regel automatisch mit geeigneten Datentypen arbeitet.

2.2.1 Datentypen in MATLAB

Standardmäßig verwendet MATLAB den **Fließkomma**-Datentyp `double` (64 Bit) und wandelt Daten vor Rechenoperationen falls nötig dahin um. Der Typ `single` (32 Bit) erlaubt die kompakte Speicherung großer Fließkomma-Daten. Eine beliebige Variable wird z.B. mit *klein* = `single(`*gross*`)` in den Typ `single` umgewandelt.

Als **Festkomma**-Datentypen stehen `int8`, `int16`, `int32` und `int64` für vorzeichenbehaftete sowie `uint8`, `uint16`, `uint32` und `uint64` für Zahlen ≥ 0 zur Verfügung. Diese Datentypen dienen der Modellierung von Festkomma-Arithmetik in MATLAB oder der Einsparung von Arbeitsspeicher, bringen jedoch in der Regel keinen Gewinn an Laufzeit. Als weitere Datentypen stehen `logical` für logische Ausdrücke und `char` (16 Bit!) für Strings zur Verfügung. Die erzwungene Typumwandlung erfolgt wie oben gezeigt.

Bei Festkomma-Rechenoperationen müssen alle Operanden denselben (!) Integer-Typ besitzen oder vom Typ `double` sein; andere Datentypen müssen vorher entsprechend umgewandelt werden. Das Ergebnis wird trotzdem immer als Integer ausgegeben!

Achtung: *Das Ergebnis von Integer-Rechenoperationen wird nach unten oder oben gerundet und auf den größten bzw. kleinsten Festkomma-Wert begrenzt! Dies gilt auch für alle Zwischenergebnisse innerhalb eines Rechenausdrucks.*

Im Folgenden werden die automatisch zugewiesenen Standard-Datentypen `double` bzw. `char` verwendet; eine Deklaration oder Umwandlung ist in der Regel **nicht** erforderlich.

Datentypen

`double(`x`)`,	Fließkomma (Standard für Zahlen)
`single(`x`)`	Fließkomma kompakt
`int8(`x`)`, `int16(`x`)`, ... `int64(`x`)`	Festkomma mit Vorzeichen
`uint8(`x`)`, `uint16(`x`)`, ... `uint64(`x`)`	Festkomma ohne Vorzeichen
`char(`x`)`	Zeichen (Standard für Strings)
`logical(`x`)`	Logischer Ausdruck

2.2.2 Vektoren und Matrizen

Die einfachste Art, einen Vektor bzw. eine Matrix zu erzeugen, ist die direkte Eingabe innerhalb eckiger Klammern `[]`. Spalten werden durch Komma (`,`) oder Leerzeichen[4] getrennt, Zeilen durch Semikolon (`;`) oder durch Zeilenumbruch.

```
>> mein_vektor = [1 2 3]
mein_vektor =
     1     2     3
>> meine_matrix = [mein_vektor; 5 6 7]
meine_matrix =
     1     2     3
     5     6     7
```

Vektoren mit fortlaufenden Elementen können besonders einfach mit dem Doppelpunkt-Operator(*start*`:`[*schrittweite*`:`]*ziel*) erzeugt werden. Wird nur *start* und *ziel* angegeben, wird die Schrittweite zu +1 gesetzt. Das Ergebnis ist jeweils ein Zeilenvektor.

```
>> lang = 1:8
lang =
     1     2     3     4     5     6     7     8
>> tief = 10:-2:0
tief =
    10     8     6     4     2     0
```

Des Weiteren stehen die Befehle `linspace(`*start*, *ziel*, *anzahl*`)` und `logspace` für linear bzw. logarithmisch gestufte Vektoren zur Verfügung. Bei `logspace` werden *start* und *ziel* als Zehnerexponent angegeben, d.h. statt 100 ($= 10^2$) wird lediglich 2 eingegeben.

[4] In fast allen anderen Fällen werden Leerzeichen von MATLAB ignoriert.

```
>> noch_laenger = linspace (1, 19, 10)
noch_laenger =
     1     3     5     7     9    11    13    15    17    19
>> hoch_hinaus = logspace (1, 2, 5)
hoch_hinaus =
   10.0000   17.7828   31.6228   56.2341  100.0000
```

Die Funktionen `ones` (*zeilen*, *spalten*) und `zeros` (*zeilen*, *spalten*) erzeugen Matrizen mit den Einträgen 1 bzw. 0. Analog lassen sich Matrizen höherer Dimensionen erstellen. Optional kann ein Datentyp aus Kap. 2.2.1 angegeben werden: `zeros` (*zeilen*, *spalten*, *typ*). Der Befehl `eye` (*zeilen*) erzeugt eine Einheitsmatrix (Spaltenzahl = Zeilenzahl).

```
>> zwo_drei = ones (2, 3)
zwo_drei =
     1     1     1
     1     1     1
```

Der Zugriff auf einzelne Elemente von Vektoren und Matrizen erfolgt durch Angabe der Indizes. Der kleinste Index ist 1 (nicht 0)! Insbesondere zur Ausgabe einzelner Zeilen bzw. Spalten eignet sich der Doppelpunkt-Operator, wobei ein allein stehender Doppelpunkt alle Elemente der zugehörigen Zeile bzw. Spalte adressiert.

```
>> meine_matrix (2, 3)
ans =
     7
>> meine_matrix (2, :)
ans =
     5     6     7
```

Sehr nützlich ist auch der Befehl `end`, der den Index des letzten Eintrags eines Vektors bzw. einer Matrix bezeichnet.

```
>> meine_matrix (end)
ans =
     7
>> meine_matrix (end, :)
ans =
     5     6     7
```

```
>> M = meine_matrix;
>> M (:, end+1) = [10; 11]
M =
     1     2     3    10
     5     6     7    11
```

Ein hilfreicher Befehl zum Erzeugen von Testdaten und Rauschsignalen ist der Befehl `rand` (*zeilen*, *spalten*), der eine Matrix mit gleichverteilten Zufallswerten zwischen 0 und 1 ausgibt. Analog erzeugt `randn` (*zeilen*, *spalten*) normalverteilte Zufallswerte mit dem Mittelwert 0 und der Standardabweichung 1. Weitere Befehle siehe `help elmat`.

```
>> zufall = rand (2, 3)
zufall =
    0.8381    0.6813    0.8318
    0.0196    0.3795    0.5028
```

Vektoren und Matrizen

`[`*x1 x2 ... ; x3 x4 ...*`]`	Eingabe von Vektoren und Matrizen
start`:[`*schrittweite*`:]`*ziel*	Doppelpunkt-Operator (erzeugt Zeilenvektor)
`linspace(`*start*, *ziel*, *anzahl*`)`	Erzeugung linearer Vektoren
`logspace(`*start*, *ziel*, *anzahl*`)`	Erzeugung logarithmischer Vektoren
`eye(`*zeilen*`)`	Einheitsmatrix
`ones(`*zeilen*, *spalten*`[, `*typ*`])`	Matrix mit Einträgen 1
`zeros(`*zeilen*, *spalten*`[, `*typ*`])`	Matrix mit Einträgen 0
`rand(`*zeilen*, *spalten*`)`	Matrix mit Zufallswerten zwischen 0 und 1
`randn(`*zeilen*, *spalten*`)`	Matrix mit normalverteilten Zufallswerten

2.2.3 Mathematische Funktionen und Operatoren für Vektoren und Matrizen

Viele mathematische Operatoren können auch auf Vektoren und Matrizen angewandt werden. Die Multiplikation mit `*` wirkt dann als Vektor- bzw. Matrixprodukt; mit `^` wird eine quadratische Matrix potenziert. Die Linksdivision `A\b` liefert die Lösung $\mathbf{x}$ des linearen Gleichungssystems $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ (ggf. mittels *least squares*-Optimierung).

Die Transponierte eines Vektors bzw. einer Matrix wird durch `transpose` oder `.'` erzeugt. Für die konjugiert-komplexe Transposition steht der Befehl `ctranspose` oder `'` zur Verfügung. Für reellwertige Größen liefern beide Operatoren dasselbe Ergebnis.

```
>> zwo_drei * meine_matrix'
ans =
     6    18
     6    18
```

Soll eine der Operationen `* / ^` elementweise ausgeführt werden, wird dem Operator ein Punkt (`.`) vorangestellt. Operationen mit Skalaren sowie mit `+` und `-` werden immer elementweise ausgeführt.

```
>> zwo_drei ./ meine_matrix
ans =
    1.0000    0.5000    0.3333
    0.2000    0.1667    0.1429
```

Natürlich funktioniert dies auch bei Matrizen mit komplexen Werten:

```
>> komplex = [1+i 1-i; 2 3]
komplex =
   1.0000 + 1.0000i   1.0000 - 1.0000i
   2.0000             3.0000

>> komplex .* komplex
ans =
        0 + 2.0000i        0 - 2.0000i
   4.0000             9.0000
```

Der Befehl `diff` (*vektor* [, *n*]) berechnet den Differenzenvektor (*n*-fache numerische Differentiation). Mit `conv` (*vektor*1, *vektor*2) werden zwei Vektoren gefaltet. Wenn beide Vektoren die Koeffizienten eines Polynoms enthalten, entspricht das Ergebnis den Koeffizienten nach einer Multiplikation beider Polynome.

```
>> diff (mein_vektor)
ans =
     1     1
```

Der Befehl `inv` (*matrix*) invertiert eine quadratische Matrix; die Befehle `det` und `eig` bestimmen deren Determinante und Eigenwerte. Der Rang einer Matrix wird mit `rank` (*matrix*) berechnet.

```
>> quadratische_matrix = [2 1; 4 9]
quadratische_matrix =
     2     1
     4     9
>> det (quadratische_matrix)
ans =
     14
>> rank (quadratische_matrix)
ans =
     2
>> eig (quadratische_matrix)
ans =
    1.4689
    9.5311
```

Werden die nachstehenden Vektorbefehle wie `min`, `sum`, `sort` etc. auf Matrizen angewandt, wirken sie spaltenweise, d.h. für jede Spalte wird eine separate Berechnung durchgeführt und das Ergebnis als Vektor ausgegeben.

Funktionen und Operatoren für Vektoren und Matrizen

`*  ^  \`	Vektor- bzw. Matrixoperatoren, Linksdivision
`.* .^ ./`	Elementweise Operatoren
matrix `.'`, `transpose` (*matrix*)	Transponierte
matrix `'`, `ctranspose` (*matrix*)	Transponierte (konjugiert komplex)
`diff` (*vector* [, *n*])	*n*-facher Differenzenvektor (Standard $n = 1$)
`conv` (*vector*1, *vector*2)	Faltung (Polynom-Multiplikation)
sortiert = `sort` (*vector*)	Sortieren in aufsteigender Reihenfolge

Weitere Funktionen

`min`(*vec*)	kleinstes Vektorelement	`inv` (*m*)	Inverse einer Matrix
`max`(*vec*)	größtes Vektorelement	`det` (*m*)	Determinante
`mean`(*vec*)	Mittelwert	`eig` (*m*)	Eigenwerte
`std`(*vec*)	Standardabweichung	`rank` (*m*)	Rang
`sum`(*vec*)	Summe der Vektorelemente	`cumsum` (*v*)	Kumulierte Summe
`prod`(*vec*)	Produkt der Vektorelemente	`cumprod` (*v*)	Kumuliertes Produkt

2.2.4 Strukturen

Variablen können zu so genannten *Strukturen* zusammengestellt werden, um komplexe Daten übersichtlich zu verwalten. Dabei ist jedem Feld ein Name zugeordnet, der als *String*[5] zwischen einfache Anführungszeichen (`' '`) gesetzt wird. Eine Struktur wird mit dem Befehl `struct('`*name*`1', `*wert*`1, '`*name*`2', `*wert*`2, ...)` oder mit einem Punkt (`.`) als Separator erzeugt.

```
>> meine_struktur = struct ('daten', meine_matrix, 'groesse', [2 3]);
```

Der Zugriff auf die Daten erfolgt ebenfalls mittels eines Punktes als Separator.

```
>> meine_struktur.daten(1,:)
ans =
     1     2     3
```

Geschachtelte Strukturen sind ebenfalls möglich, wie das folgende Beispiel zeigt: Eine Struktur `komponist` wird angelegt und ihrem Feld `name` der String `'Johann Sebastian Bach'` zugewiesen. Eine zweite Struktur namens `datum` mit den drei Feldern `Tag`, `Monat` und `Jahr` enthält die entsprechenden Geburtsdaten. Anschließend wird die Struktur `datum` dem neu erzeugten Feld `geboren` der Struktur `komponist` zugewiesen.

```
>> komponist = struct ('name', 'Johann Sebastian Bach');
>> datum.Tag = 21;
>> datum.Monat = 'März';
>> datum.Jahr = 1685;
>> komponist.geboren = datum;
```

Die Struktur `komponist` soll nun einen zweiten Eintrag mit dem String `'Wolfgang Amadeus Mozart'` als Wert für das Feld `name` erhalten. Die Werte für das Feld `geboren` werden nun direkt eingegeben (dies wäre oben ebenfalls möglich gewesen).

```
>> komponist(2).name = 'Wolfgang Amadeus Mozart';
>> komponist(2).geboren.Tag = 27;
>> komponist(2).geboren.Monat = 'Januar';
>> komponist(2).geboren.Jahr = 1756;
```

Die Struktur `komponist` ist nun eine vektorwertige Struktur, deren einzelne Elemente wie Vektoren behandelt werden können. Indizes, die auf die Strukturelemente verweisen, stehen unmittelbar nach dem Namen der Struktur.

```
>> komponist(2)
ans =
       name: 'Wolfgang Amadeus Mozart'
    geboren: [1x1 struct]
```

```
>> komponist(2).geboren
ans =
      Tag: 27
    Monat: 'Januar'
     Jahr: 1756
```

[5] Siehe auch weitere Ausführungen zum Thema Strings in Kap. 3.2.1.

2.2.5 Cell Arrays

Noch eine Stufe allgemeiner gehalten sind so genannte *Cell Arrays*. Dies sind multidimensionale Arrays, die in jeder einzelnen Zelle Daten unterschiedlicher Datentypen enthalten können. Erzeugt werden Cell Arrays mit dem Befehl `cell` oder durch Einschließen der Elemente bzw. Indizes in geschweifte Klammern `{ }`.

Die einzelnen Elemente eines Cell Arrays werden ebenso wie normale Vektoren oder Matrizen adressiert, nur werden statt runder Klammern geschweifte Klammern verwendet, z.B. `zelle{1,2}`.[6] Im Folgenden wird ein leeres 2×3-Cell Array namens `zelle` erzeugt:

```
>> zelle = cell (2, 3)
zelle =
     []     []     []
     []     []     []
```

Nun werden den einzelnen Zellen die folgenden Werte zugewiesen:

```
>> zelle {1, 1} = 'Zelle {1, 1} ist ein Text';
>> zelle {1, 2} = 10;
>> zelle {1, 3} = [1 2; 3 4];
>> zelle {2, 1} = komponist (2);
>> zelle {2, 3} = date;
```

Die Eingabe des Namens des Cell Arrays zeigt seine Struktur. Den Inhalt einer oder mehrerer Zellen erhält man durch die Angabe ihrer Indizes (kein Leerzeichen vor der geschweiften Klammer { verwenden!). Verschachtelte Strukturen werden wie oben gezeigt angesprochen.

```
>> zelle
zelle =
    [1x25 char  ]    [10]     [2x2 double]
    [1x1  struct]      []    '25-Jun-2011'

>> zelle{2, 3}                                          % kein Leerzeichen vor { !
ans =
25-Jun-2011

>> zelle{2, 1}.geboren.Monat                            % kein Leerzeichen vor { !
ans =
Januar
```

Cell Arrays eignen sich insbesondere auch zum Speichern unterschiedlich langer Strings. Werden Strings dagegen in einem normalen `char`-Array gespeichert, müssen alle Einträge dieselbe Anzahl an Zeichen aufweisen. Diese Einschränkung lässt sich mit einem Cell Array umgehen.

[6] Die einzelnen Zellen eines (beliebigen) Array-Typs können auch mit nur einem Index adressiert werden. Für ein zweidimensionales `m` $\times$ `n`-Array ist `zelle{k,j}` gleichbedeutend mit `zelle{(j-1)*m+k}` (siehe auch Befehle `ind2sub` und `sub2ind`). Die Reihenfolge der Elemente kann mit `:` ausgegeben werden, z.B. `meine_matrix(:)`.

Strukturen und Cell Arrays

`struct('`$n1$`',` $w1$`, '`$n2$`',` $w2$`, ...)`	Erzeugen einer Struktur
struktur`.`*name*	Zugriff auf Element *name*
zelle `= {`*wert*`}`	Erzeugen eines Cell Arrays
zelle`{`*index*`} =` *wert*	Erzeugen eines Cell Arrays
`cell(`n`)`	Erzeugen eines $n \times n$-Cell Arrays
`cell(`m`,` n`)`	Erzeugen eines $m \times n$-Cell Arrays

2.2.6 Verwalten von Variablen

Im Folgenden werden Befehle vorgestellt, die Informationen über Art, Größe und Speicherbedarf von Variablen liefern sowie das Löschen von Variablen erlauben.

Die Dimension eines Vektors bzw. einer Matrix lässt sich mit `size(`*variable*`)` bestimmen. Für Vektoren eignet sich auch der Befehl `length(`*variable*`)`, der bei Matrizen den Wert der größten Dimension angibt. Eine Variable kann auch die Größe 0 besitzen, wenn sie mit *variable* `= []` erzeugt wurde.

```
>> length (meine_matrix)
ans =
     3
```

```
>> size (meine_matrix)
ans =
     2     3
```

Mit dem Befehl `who` werden alle aktuell im Workspace vorhandenen Variablen aufgelistet. Mit `whos` erhält man zusätzlich deren Dimension (`Size`), Speicherbedarf (`Bytes`) und Datentyp (`Class`). Mit `clear` [*variable*1 *variable*2 ...] können Variablen gezielt gelöscht werden. Der Befehl `clear` alleine löscht alle Variablen im Workspace; `clear all` löscht zudem alle globalen Variablen.[7] Dies eignet sich auch gut als Shortcut auf dem MATLAB-Desktop.

```
>> clear variable_1 variable_2 a A meine_matrix lang tief noch_laenger ...
         hoch_hinaus zwo_drei M zufall quadratische_matrix datum ans

>> whos
  Name                Size             Bytes  Class     Attributes

  komplex             2x2                 64  double    complex
  komponist           1x2               1252  struct
  mein_vektor         1x3                 24  double
  meine_struktur      1x1                312  struct
  zelle               2x3               1110  cell
```

Über das Menü *Desktop/Workspace* oder den Befehl `workspace` lässt sich der *Workspace Browser* öffnen. Dieser erlaubt ebenfalls eine Übersicht der vorhandenen Variablen.

[7] Zu globalen Variablen siehe Kap. 2.5.2

Durch Doppelklicken auf den gewünschten Namen wird der *Variable Editor* aktiviert (siehe Abb. 2.3). Dort können Variablen editiert und gelöscht werden. Ein markierter Datenbereich (im Beispiel die 2. Spalte von `zelle{1,3}`) kann über die Auswahlliste plot() direkt grafisch angezeigt werden. Die Schaltflächen dienen zum Anordnen mehrerer Teilfenster innerhalb des Variable Editors.

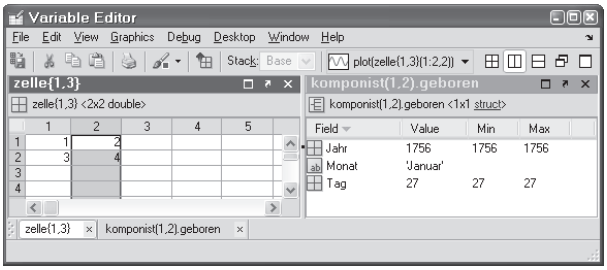

Abb. 2.3: MATLAB *Variable Editor*

Verwalten von Variablen

`size` (*variable*)	Dimensionen einer Variablen
`length` (*variable*)	Länge eines Vektors, größte Dimension einer Matrix
`clear`	Löscht alle Variablen im Workspace
`clear all`	Löscht zusätzlich alle globalen Variablen
`clear` [*v*1 *v*2 ...]	Löscht ausgewählte Variablen
`who`	Liste aller im Workspace existierenden Variablen
`whos`	Ausführliche Liste aller im Workspace existierenden Variablen mit Name, Dimension, Speicherbedarf und Datentyp

2.3 Ablaufsteuerung

Die Ablaufsteuerung umfasst neben den eigentlichen Verzweigungs-, Schleifen- und Abbruchbefehlen die Abfrage von Bedingungen sowie deren logische Verknüpfung.

2.3.1 Vergleichsoperatoren und logische Operatoren

Logische Operatoren können auf alle Zahlen angewandt werden. Werte ungleich 0 sind dabei logisch *wahr* und 0 ist logisch *falsch*. Als Ergebnis erhält man stets 0 bzw. 1.

Neben Vergleichsoperatoren stehen logische Operatoren für UND, ODER, NEGATION und EXKLUSIV ODER zur Verfügung, und zwar meist in der Form eines Zeichens oder als Funktion, z.B. *a* `&` *b* oder `and(`*a*, *b*`)` für *a* UND *b*. Zuerst werden mathematische, dann logische Ausdrücke ausgewertet (für Details siehe `help precedence`). Treten mehrere logische Operatoren auf, empfiehlt sich die Verwendung von Klammern!

```
>> mein_vektor >= 2
ans =
     0     1     1
```

```
>> 1 == 0 | (4 > 5-2 & 4 <= 5)
ans =
     1
```

Hier wird eine Ergebnistabelle für logische Verknüpfungen zweier Binärzahlen erstellt:

```
>> a = [0 0 1 1]';  b = [0 1 0 1]';
>> [ a     b   a&b   a|b xor(a,b) ~a ~(a&b) ~(a|b)]
ans =
     0     0     0     0     0     1     1     1
     0     1     0     1     1     1     1     0
     1     0     0     1     1     0     1     0
     1     1     1     1     0     0     0     0
```

Die *Shortcut-Operatoren* `&&` und `||` brechen im Gegensatz zu `&` und `|` die Auswertung mehrerer logischer Ausdrücke ab, sobald das Ergebnis eindeutig ist: so wird `ausdruck` in `(1 | ausdruck)` ausgewertet, in `(1 || ausdruck)` dagegen nicht. Die Auswertung erfolgt dabei von links nach rechts. Bei vektorwertigen Daten müssen allerdings nach wie vor die Operatoren `&` und `|` bzw. `all` oder `any` verwendet werden (siehe unten).

Die Funktion `any(`*vektor*`)` ist wahr, wenn mindestens ein Element eines Vektors (bestehend aus Zahlen oder logischen Ausdrücken) wahr ist; `all(`*vektor*`)` ist nur wahr, wenn jedes Element wahr ist. Die Funktion `find(`*vektor*`)` liefert die Indizes aller wahren Elemente, `[~,` *index*`]=sort(`*vektor*`)` die Indizes aller Vektorelemente in sortierter Reihenfolge.[8] Eine weiterführende Befehlsübersicht erhält man mit `help ops`.

Der Befehl `exist('`*name*`')` überprüft, ob eine Variable, Funktion oder Datei *name* existiert: Falls nicht, wird 0 zurückgeliefert, ansonsten eine Zahl ungleich 0, z.B. 1 für eine Variable, 2 für eine MATLAB-Datei, 7 für ein Verzeichnis (siehe auch `doc exist`).

Eine spezielle Anwendung logischer Operatoren ist das *logical indexing*. Dabei wird ein Array mit einem Index-Array derselben Größe adressiert, welches nur aus den (logischen) Werten 0 und 1 besteht. Das folgende Beispiel setzt alle negativen Elemente von `a` auf 0. Der Ausdruck `a<0` liefert dabei den Vektor `[1 1 0 0 0 0 0 0 0 0]` zurück; damit werden nur die beiden ersten Elemente von `a` ausgewählt und auf 0 gesetzt.

```
>> a = -2:7
a =
    -2    -1     0     1     2     3     4     5     6     7
>> a (a < 0) = 0
a =
     0     0     0     1     2     3     4     5     6     7
```

In einem weiteren Schritt sollen alle durch 3 teilbaren Elemente von `a` entfernt werden.

```
>> a = a (logical (mod (a, 3)))
a =
     1     2     4     5     7
```

Da der Ausdruck `mod(a,3)` auch andere Werte als 0 und 1 liefert, muss sein Ergebnis mit `logical` explizit in logische Ausdrücke umgewandelt werden.[9]

[8] Das Zeichen `~` ist dabei ein leerer Rückgabeparameter (anstelle des sortierten Vektors).

[9] Alternativ: `a (~mod (a, 3)) = []`. Dabei übernimmt die Negation `~` die Konvertierung in den

Vergleichsoperatoren

`==`	`eq` (*a*, *b*)	gleich
`~=`	`ne` (*a*, *b*)	ungleich
`<`	`lt` (*a*, *b*)	kleiner
`<=`	`le` (*a*, *b*)	kleiner gleich
`>`	`gt` (*a*, *b*)	größer
`>=`	`ge` (*a*, *b*)	größer gleich

Logische Operatoren

`~`	`not` (*a*)	NEGATION
`&`	`and` (*a*, *b*)	UND
`|`	`or` (*a*, *b*)	ODER
	`xor` (*a*, *b*)	EXKLUSIV ODER
`&&`		Shortcut-UND (skalar)
`||`		Shortcut-ODER (skalar)

Weitere Operatoren

	`all` (*vec*)	jedes Element wahr	`exist('x')` Existenz von *x*
	`any` (*vec*)	mind. 1 Element wahr	`find` (*vec*) Indizes wahrer Elemente
[~, *i*] =	`sort` (*vec*)	Indizes *i* sortiert	`logical` (*a*) Typ-Umwandlung

2.3.2 Verzweigungsbefehle `if` und `switch`

Mit Hilfe der oben behandelten Operatoren können Fallunterscheidungen durchgeführt werden. Dafür stellt MATLAB die folgenden Verzweigungsbefehle zur Verfügung:

`if` *ausdruck befehle* [`elseif` *ausdruck befehle* ...] [`else` *befehle*] `end`
`switch` *ausdruck* `case` *ausdruck befehle* [...] [`otherwise` *befehle*] `end`

Bei `case` können mehrere ODER-verknüpfte Möglichkeiten innerhalb geschweifter Klammern `{ }` (d.h. als Cell Array) angegeben werden. Anders als in C wird immer nur eine Verzweigung (`case`) ausgeführt; es wird kein `break` zum Aussprung benötigt.

```
if test <= 2
    a = 2
elseif test <= 5
    a = 5
else
    a = 10
end
```

```
switch test
    case 2
        a = 2
    case {3 4 5}
        a = 5
    otherwise
        a = 10
end
```

Für `test = 5` ergeben diese beiden Beispiele jeweils die Ausgabe

```
a =
     5
```

Eine Verschachtelung mehrerer `if`- und `switch`-Konstrukte ist natürlich möglich.

Datentyp `logical`; die leere Zuweisung mit `[]` löscht die entsprechenden Elemente des Vektors `a`.

2.3.3 Schleifenbefehle `for` und `while`

Mit Schleifen können bestimmte Anweisungen mehrfach durchlaufen werden:

`for` *variable* `=` *vektor befehle* `end`
`while` *ausdruck befehle* `end`

```
for k = 1:0
    k^2
end
```

Die `for`-Schleife im obigen Beispiel wird nicht durchlaufen, da der für `k` angegebene Bereich `1:0` leer ist. Im Unterschied dazu wird im folgenden Beispiel die `while`-Schleife mindestens einmal abgearbeitet, da die Abbruchbedingung (siehe auch Kap. 2.3.4) erst am Ende geprüft wird. Außerdem sind zwei `end` notwendig – für `if` und `while`!

```
n = 1;
while 1
    n = n+1;
    m = n^2
    if m > 5
        break
    end
end
```

2.3.4 Abbruchbefehle `continue`, `break` und `return`

Weitere Befehle zur Ablaufsteuerung sind `continue`, `break` und `return`. Mit `continue` wird innerhalb einer `for`- oder `while`-Schleife sofort zum nächsten Iterationsschritt gesprungen; alle innerhalb der aktuellen Schleife noch folgenden Befehle werden übergangen. Der Befehl `break` dagegen bricht die aktuelle Schleife ganz ab. Der Befehl `return` bricht eine MATLAB-Funktion bzw. ein Skript[10] ab und kehrt zur aufrufenden Ebene zurück (bzw. zum Command Window, falls die Funktion von dort aufgerufen wurde).

Im folgenden Beispiel wird überprüft, welche ungeraden Zahlen zwischen 13 und 17 Primzahlen sind. Die zu testenden Zahlen `m` werden in der äußeren Schleife, mögliche Teiler `n` in der inneren Schleife hochgezählt.

```
for m = 13:2:17
    for n = 2:m-1
        if mod (m,n) > 0
            continue
        end
        sprintf ('   %2d ist keine Primzahl.\n', m)
        break
    end
    if n == m-1
        sprintf ('!! %2d IST eine  Primzahl!\n', m)
    end
end
```

[10] MATLAB-Skripts und -Funktionen werden in Kap. 2.4 und 2.5 behandelt.

Ist `n` kein Teiler von `m` und damit `mod(m,n)` > 0, wirkt der `continue`-Befehl. Der erste `sprintf`-[11] und der `break`-Befehl werden übersprungen; es wird sofort das nächsthöhere `n` getestet. Ist `n` dagegen Teiler von `m`, wird die innere `for`-Schleife nach der Ausgabe „ist keine Primzahl“ bei `break` verlassen und das nächste `m` getestet.

Ist `m` durch keine Zahl von 2 bis `m-1` teilbar, so ist `m` eine Primzahl. Die Überprüfung mit der zweiten `if`-Abfrage `n == m-1` ist notwendig, da die innere Schleife zwar bei Nicht-Primzahlen vorzeitig abgebrochen wird, aber dennoch alle nachfolgenden Befehle der äußeren Schleife abgearbeitet werden. Die Ausgabe am MATLAB-Workspace ergibt:

```
!! 13 IST eine  Primzahl!
   15 ist keine Primzahl.
!! 17 IST eine  Primzahl!
```

Eine weiterführende Übersicht zur Programm-Ablaufsteuerung bietet `help lang`.

Verzweigungen, Schleifen und Ablaufsteuerung

`if` ... [`elseif` ...] [`else` ...] `end`	If-Verzweigung
`switch` ... `case` ... [`otherwise` ...] `end`	Switch-Verzweigung
`for` *variable* `=` *vektor befehle* `end`	For-Schleife
`while` *ausdruck befehle* `end`	While-Schleife
`break`	Sofortiger Schleifen-Abbruch
`continue`	Sofortiger Sprung zum nächsten Iterationsschritt einer Schleife
`return`	Sofortiger Funktions-Rücksprung

2.4 Der MATLAB-Editor

Neben der direkten Befehlseingabe am MATLAB-Prompt können Befehlsfolgen auch in so genannten MATLAB-*Skripts* (Textdateien mit der Endung *.m*, daher auch *M-Files* genannt) gespeichert werden.

Zur Bearbeitung dieser Dateien steht der MATLAB-Editor zur Verfügung (siehe Abb. 2.4). Dieser kann über den Button auf dem Desktop oder das Menü *File/New/M-File* aufgerufen werden. Ein bestehendes M-File wird über *File/Open*, durch Doppelklicken im Current Folder Browser oder mit dem Befehl `edit` *datei* geöffnet.

Das **Ausführen** eines Skripts geschieht durch Eingabe des Dateinamens (ohne Endung) im MATLAB-Command-Window, in diesem Fall also mit `beispiele`. Einfacher geht dies mit *Run* im Kontextmenü des Current Folder Browsers oder mit dem Editor-Button ; dabei werden Änderungen im Code auch gespeichert.

Zum **Debuggen** eines Skripts (oder einer Funktion) können in der Spalte neben der Zeilennummer Breakpoints per Mausklick gesetzt und gelöscht werden.[12] startet

[11] Die Syntax des Ausgabe-Befehls `sprintf` wird in Kap. 3.3 erklärt.

[12] Über das Kontextmenü kann eine zusätzliche Anhalte-Bedingung angegeben werden.

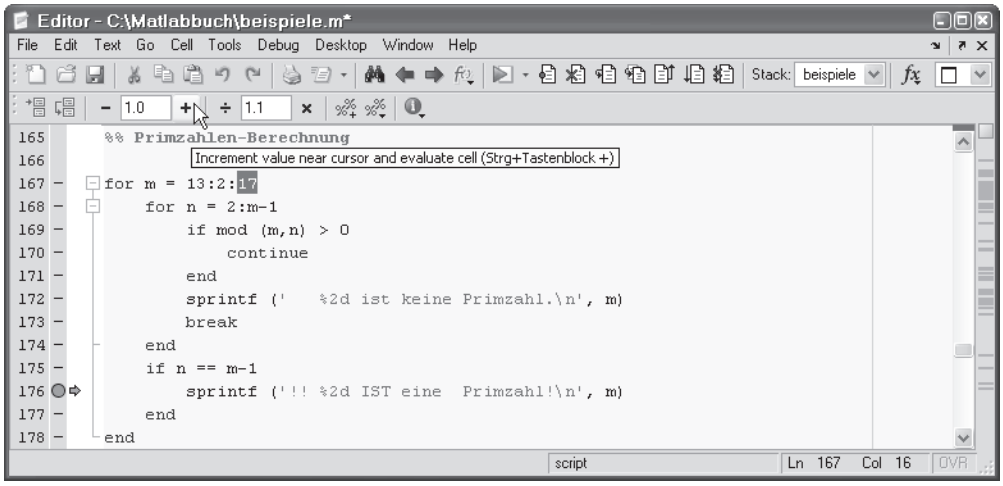

Abb. 2.4: MATLAB*-Editor (mit Beispieldatei zu Kap. 2.3.4)*

den Debugger, springt zum nächsten Breakpoint. Mit können einzelne Zeilen abgearbeitet, aufgerufene Skripts bzw. Funktionen angesprungen und wieder verlassen werden. Die jeweils aktuelle Zeile ist mit einem Pfeil markiert, im Command Window (Prompt `K>>`) und im Workspace Browser sind die dann sichtbaren Variablen verfügbar. Beendet wird der Debugger mit oder automatisch nach der letzten Zeile des M-Files. Der Button löscht alle Breakpoints.

Während der Eingabe wird eine **Syntax-Prüfung** durchgeführt. Auffällige Stellen werden im Code unterringelt und rechts neben dem vertikalen Scrollbalken farbig markiert: Syntaxfehler sind rot, Warnungen orange (also fehleranfällige oder ungünstige Konstrukte). Führt man den Mauszeiger über solch eine Markierung am Rand oder im Code, wird eine Beschreibung oder ein Verbesserungsvorschlag eingeblendet. Rechts oben erscheint als Zusammenfassung eine „Ampel“: Wird keine Stelle beanstandet, ist diese grün. Eine Liste aller Fehlermeldungen und Warnungen des Code-Analyzers (`mlint`) erhält man ebenso im Profiler (siehe Kap. 2.6.1) und über das Menü *Tools/Code Analyzer/Show Code Analyzer Report*.

Kommentare werden durch das Zeichen `%` markiert, d.h. alle Zeichen rechts von `%` bis zum Zeilenende werden von MATLAB ignoriert. Zum Auskommentieren längerer Passagen ist das Menü *Text/Comment* (bzw. *Strg* + *R*) und umgekehrt *Text/Uncomment* (bzw. *Strg* + *T*) hilfreich. Ebenso existiert mit `%{` *kommentar* `%}` eine Maskierung mehrzeiliger Kommentare.

Lange Zeilen können innerhalb eines Befehls mit `...` umgebrochen werden (siehe Beispiel in Kap. 3.2.2). Ebenso verbessert der MATLAB-Editor mit den Symbolen die Übersichtlichkeit durch **Code Folding** für bestimmte Bereiche (z.B. `for`-Schleifen).

Hilfreich für lange Skripts ist auch der so genannte **Cell Mode**. Er wird über das Menü *Cell/Enable Cell Mode* aktiviert und ist an der zusätzlichen Buttonleiste erkennbar (siehe Abb. 2.4). Das Skript wird dabei durch *Cell Divider* `%%` in Abschnitte unterteilt; der Kommentar nach `%%` wird bei der automatischen Dokumentation als

Überschrift verwendet und auch im Vorschaufenster des *Current Folder Browsers* angezeigt, wie in Abb. 2.1. Der Cell Mode bietet unter anderem folgende Funktionen:

- Schnelle Navigation: Mit der Tastenkombination *Strg* + ↓ bzw. *Strg* + ↑ kann man direkt zum nächsten bzw. vorigen Abschnitt springen. Alternativ zeigt der Button eine Auswahlliste aller Abschnitte und eine alphabetische Auswahlliste aller Funktionen in einem M-File.
- Schnelle Evaluierung: Über die Buttons lässt sich ein Skript abschnittsweise ausführen. Mit – 1.0 + ÷ 1.1 × wird eine markierte Zahl entsprechend der eingegebenen Differenz (bzw. Faktor) geändert und anschließend der Abschnitt ausgeführt. Dabei wird das Skript nicht gespeichert!
- Schnelle Dokumentation: Mittels oder über das Menü *File/Publish* wird der Code eines Skripts abschnittsweise samt Überschriften und Ergebnissen (Text und Grafik) als HTML-Dokument ausgegeben. Andere Formate (z.B. PDF) und weitere Einstellungen sind über das Menü *File/Publish Configuration/Edit* wählbar.

Der **Function Browser** bietet über den Button eine hierarchische Online-Hilfe zu allen MATLAB-Funktionen. Mit schließlich können die Stellen der letzten Änderungen (auch über mehrere Dateien) direkt angesprungen werden.

Das **Comparison Tool** eignet sich zum Vergleich von Dateien und ganzen Verzeichnissen. Es wird aus dem Editor-Menü *Tools/Compare Against* aufgerufen oder auch über das Menü *Desktop/Comparison Tool* im Command Window. Es vergleicht zwei Textdateien (M-Files, aber auch andere) zeilenweise und markiert Abweichungen farbig. Ebenso ist ein detaillierter Vergleich der Variablen zweier MAT-Files möglich.[13)]

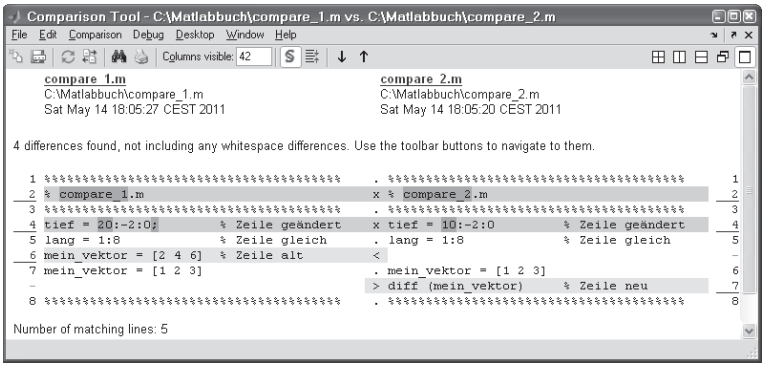

Abb. 2.5: MATLAB *Comparison Tool*

Skripts

...	Fortsetzungszeichen zum Umbruch überlanger Zeilen
%	Beginn einer Kommentar-Zeile
%{ *kommentar* %}	Mehrzeiliger Kommentar
%%	Beginn eines Kommentars als Cell-Divider

13) MAT-Files zum Speichern von MATLAB-Variablen werden in Kap. 3.3 erklärt.

2.5 MATLAB-Funktionen

Eine Sonderform der M-Files stellen so genannte MATLAB-*Funktionen* dar. Dies sind in einer Datei abgespeicherte MATLAB-Skripts, die ihren eigenen, nach außen nicht sichtbaren Workspace (d.h. Variablen-Bereich) haben. Definiert wird eine Funktion durch das Schlüsselwort `function` in der ersten Befehlszeile der Datei:

`function [`*out*1, *out*2, ...`] =` *funktionsname* `(`*in*1, *in*2, ...`)`
befehle

In runden Klammern `()` nach dem Funktionsnamen[14] stehen die Übergabe-Parameter *in*1, *in*2, ..., die beim Aufruf der Funktion übergeben werden können. In eckigen Klammern `[]` stehen die Rückgabewerte *out*1, *out*2, ..., die die Funktion an die aufrufende Funktion bzw. an den Workspace zurückgibt. Zahlreiche MATLAB-Befehle sind als Funktionen programmiert, z.B. `mean(x)` – Anzeigen des Quellcodes mit `edit mean`.

Funktionen können von anderen Funktionen und Skripts aufgerufen werden, Skripts ebenso durch andere Skripts. Lokale Funktionen können in dem M-File der aufrufenden Funktion unten angefügt werden; die lokalen Funktionen sind dann nur für die Funktionen in dieser einen Datei sichtbar. Anders als bei Variablen wird Groß- und Kleinschreibung bei Funktions- und Skriptnamen **nicht** unterschieden!

Als Beispiel sei die Funktion `mittelwerte` in der Datei `mittelwerte.m` gespeichert. Sie berechnet den arithmetischen und geometrischen Mittelwert eines übergebenen Vektors:

```
function [arithm, geom] = mittelwerte (x)  % Datei mittelwerte.m
arithm = mean(x);                          % Arithmetisches Mittel
geom   = prod(x).^(1/length(x));           % Geometrisches Mittel
```

Mit dem Aufruf `mittelwerte(test)` wird nur das erste Ergebnis der Rückgabeliste `[arithm, geom]` zurückgegeben. Werden mehrere Rückgabewerte benötigt, so muss das Ergebnis in ein Array der Form `[A, G]` gespeichert werden:

```
>> test = [2 4 3];
>> [A, G] = mittelwerte (test)
A =
     3
G =
    2.8845
```

Sollen Rückgabewerte übersprungen werden, können diese durch das Zeichen ~ ersetzt werden.[15]

```
>> [~, G] = mittelwerte (test)
G =
    2.8845
```

[14] Der möglichst der Dateiname sein sollte, aber nicht muss!
[15] Siehe auch Beispiel zum Befehl `sort` auf Seite 18.

2.5.1 Funktionen mit variabler Parameterzahl

Eine Stärke von MATLAB-Funktionen ist, mit einer unterschiedlichen Anzahl von Übergabe-Parametern zurechtzukommen. Hierzu existieren innerhalb einer Funktion die Variablen `nargin` und `nargout`, die die Anzahl der übergebenen Parameter bzw. zurückzuschreibenden Werte enthalten. Zusätzlich stehen noch die Befehle `inputname` zum Ermitteln der Variablennamen der Übergabe-Parameter und `nargchk` zum Überprüfen der korrekten Anzahl der übergebenen Parameter zur Verfügung.

Im Folgenden soll die Funktion `mittelwerte2` entweder nur den arithmetischen, nur den geometrischen oder beide Mittelwerte berechnen, wozu ein zweiter Übergabe-Parameter `schalter` definiert wird.

Je nach Anzahl der übergebenen Parameter wird nun unterschiedlich verfahren: Wird nur eine Variable `x` übergeben, so werden beide Mittelwerte zurückgegeben. Wird auch der zweite Parameter `schalter` übergeben, so wird `out1` das geometrische Mittel zugewiesen, wenn `schalter == 'g'` ist, ansonsten das arithmetische Mittel. `out2` wird als leer definiert. Wird eine andere Zahl an Parametern übergeben (Test mit `nargchk`), wird mittels `error` eine Fehlermeldung ausgegeben und die Funktion abgebrochen.

```
function [out1, out2] = mittelwerte2 (x, schalter)      % Datei mittelwerte2.m
error (nargchk (1, 2, nargin))                          % Prüfe Parameterzahl
if nargin == 1
    out1 = mean(x);                                     % Arithmetisches Mittel
    out2 = prod(x).^(1/length(x));                      % Geometrisches Mittel
elseif nargin == 2
    if schalter == 'g', out1 = prod(x).^(1/length(x));  % Geometrisches Mittel
    else                out1 = mean(x);                 % Arithmetisches Mittel
    end
    out2 = [];                                          % Leere Ausgabe für out2
end
```

Damit ergibt sich folgende Ausgabe:

```
>> mittelwerte2 (test, 'g')
ans =
    2.8845
```

Funktionen

`function [`*out*`] = `*name*`(`*in*`)`	MATLAB-Funktion *name* mit Liste der Eingabeparameter *in* und Ausgabewerte *out* definieren
`nargin`, `nargout`	Anzahl der Ein- bzw. Ausgabeparameter
`nargchk(`*min*, *max*, *n*`)`	Anzahl n der Übergabeparameter überprüfen, ob gilt: $min \leq n \leq max$, sonst Fehlertext
`isempty(`*name*`)`	Gültigkeit der Variablen *name* prüfen
`error('`*info*`')`	Abbruch der Funktion und Anzeige von *info*

2.5.2 Lokale, globale und statische Variablen

Die Variablen *innerhalb* jeder Funktion sind *lokal* und werden beim Verlassen der Funktion wieder gelöscht. Soll eine Variable dagegen bis zum nächsten Aufruf derselben Funktion gespeichert bleiben, muss sie zu Beginn der Funktion mit `persistent` *variable ...* als statisch deklariert werden. Statische Variablen werden nur durch `clear functions` oder bei Änderung des M-Files der Funktion gelöscht.

Globale Variablen können mit `global` *VAR*1 ... deklariert werden.[16] Um auf diese Variablen zugreifen zu können, muss die Deklaration zu Beginn jeder betroffenen MATLAB-Funktion sowie bei Bedarf im Workspace des Command Windows erfolgen. Angezeigt werden die globalen Variablen mit `whos global`, gelöscht werden sie mit `clear global`. Globale und statische Variablen zeigt der Editor in hellblauer Schrift an.

So kann in der Funktion `mittelwerte2` die Übergabe des Parameters `schalter` vermieden werden, wenn dieser in der Funktion `mittelwerte3` als global definiert wird:

```
function [out1, out2] = mittelwerte3 (x)  % Datei mittelwerte3.m
global SCHALTER                           % Definiert globale Variable

out1 = mean(x);                           % Arithmetisches Mittel
out2 = prod(x).^(1/length(x));            % Geometrisches Mittel
if ~isempty (SCHALTER)
    if SCHALTER == 'g'
        out1 = out2;                      % Gibt geometrisches Mittel aus
    end
    out2 = [];
end
```

Hierdurch entfällt die Abfrage von `nargin`; sicherheitshalber muss nur z.B. mit `isempty` überprüft werden, ob die globale Variable `SCHALTER` nicht leer ist. Dann wird für `SCHALTER == 'g'` die Rückgabevariable `out1` auf das geometrische Mittel (`out2`) und anschließend `out2` auf `[]` gesetzt.

Ebenso muss im Command Window die Variable `SCHALTER` – vor der ersten Zuweisung! – als `global` deklariert werden. Wird die Variable dann dort gleich `'g'` gesetzt, kann die Funktion `mittelwerte3.m` auf diesen Wert zugreifen.

```
>> global SCHALTER
>> SCHALTER = 'g';
>> test = [2 4 3];

>> [M1, M2] = mittelwerte3 (test)
M1 =
    2.8845
M2 =
     []
```

Die Anzeige aller globalen Variablen ergibt dann:

[16] Zur Unterscheidung sollten für globale Variablen nur GROSSBUCHSTABEN verwendet werden.

```
>> whos global
  Name          Size            Bytes  Class     Attributes

  SCHALTER      1x1                 2  char      global
```

Der Befehl `assignin` erlaubt einer Funktion den Schreibzugriff auf andere (d.h. von ihr nicht sichtbare) Workspaces. Wird die Zeile `assignin ('base', 'name', wert)` innerhalb einer Funktion aufgerufen, weist sie der Variablen *name* im Workspace des Command Windows den Wert *wert* zu. Falls die Variable dort nicht existiert, wird sie dabei erzeugt. Umgekehrt liefert `evalin ('base', 'name')` aus einer Funktion heraus den Wert der Variablen im Workspace des Command Windows zurück. Ein Beispiel zur Verwendung von `assignin` findet sich auf Seite 250.

Achtung: *Globale Variablen und der Befehl* `assignin` *sind innerhalb von Funktionen mit Vorsicht zu genießen, da natürlich auch andere Funktionen, die Zugriff auf die so verwendeten Variablen haben, deren Wert verändern können!*

Globale und Statische Variablen in Funktionen

`persistent` *var*1 ...	Statische (lokale) Variablen deklarieren
`global` *VAR*1 ...	Globale Variablen definieren
`clear global` *VAR*1 ...	Globale Variablen löschen

Zugriff aus Funktion heraus auf Workspace im Command Window

`assignin ('base', 'var', x)`	Zuweisen des Wertes x an die Variable *var*
$x =$ `evalin ('base', 'var')`	Lesen des Wertes der Variable *var*

2.5.3 Hilfetext in Funktionen

Zur Erklärung einer Funktion kann im Kopf des zugehörigen M-Files ein Hilfetext eingefügt werden, der beim Aufruf von `help` *funktion* bzw. mit `helpwin` *funktion* oder `doc` *funktion* ausgegeben wird. Der Text muss mit dem Kommentarzeichen `%` beginnen; die Ausgabe endet mit der ersten Leerzeile. Alle weiteren auskommentierten Zeilen werden unterdrückt. Für die Funktion `mittelwerte` lauten die ersten Zeilen z.B.

```
%MITTELWERTE (X) Berechnungen verschiedener Mittelwerte
%   [A,G] = MITTELWERTE (X) berechnet das arithmetische
%   Mittel A und das geometrische Mittel G des Vektors X.

% Erstellt: 21.01.02
```

Wird nun am MATLAB-Prompt der Befehl `help mittelwerte` eingegeben, so sieht die Ausgabe wie folgt aus:

```
>> help mittelwerte

 MITTELWERTE (X) Berechnungen verschiedener Mittelwerte
    [A,G] = MITTELWERTE (X) berechnet das arithmetische
    Mittel A und das geometrische Mittel G des Vektors X.
```

2.5.4 Function Handles

Üblicherweise wird eine Funktion wie oben gezeigt direkt aufgerufen. Ein zweite, sehr mächtige Möglichkeit ist das indirekte Ansprechen der Funktion über das ihr zugeordnete *Function Handle*.[17] Ein Function Handle stellt den Zeiger auf eine Funktion dar. Dieser dient vor allem der Übergabe einer Funktion als Parameter an eine andere Funktion und wird z.B. beim Aufruf von Gleichungslösern und Optimierungsbefehlen verwendet (siehe Kap. 4 und 7).

Ein Function Handle wird durch den Operator `@` vor dem Funktionsnamen *funktion* erzeugt:

f_handle `=` `@`*funktion*

Die dem Function Handle zugewiesene Funktion kann nun mit dem Befehl `feval` ausgeführt werden. Alternativ kann das Function Handle auch wie der ursprüngliche Funktionsname verwendet werden:

[*out*1, *out*2, ...] `=` `feval` (*f_handle*, *in*1, *in*2, ...)
[*out*1, *out*2, ...] `=` *f_handle* (*in*1, *in*2, ...)

Hierbei sind *in*1, *in*2, ... die Eingabeparameter der Funktion und *out*1, *out*2, ... die Rückgabewerte. Für das obige Beispiel `mittelwerte` wird nun das Function Handle `fh` erzeugt und die Funktion damit aufgerufen:

```
>> fh = @mittelwerte;           % function handle auf mittelwerte erzeugen
>> A = fh (test)                % Ausführen des function_handles fh
A =
     3
```

2.5.5 Funktionen als Inline Object

Eine sehr bequeme Möglichkeit zum Arbeiten mit immer wieder verwendeten kleinen Funktionen sind *inline objects*:[18] Hiermit können im MATLAB-Workspace Funktionen definiert werden, ohne sie in einem M-File abspeichern zu müssen.

Bereitgestellt und ausgeführt wird eine solche Inline-Object-Funktion wie folgt. Sowohl die Befehlszeile (Formel) als auch die Parameter werden als Strings angegeben:

```
>> f1 = inline ('x.^2+x-1', 'x')
f1 =
     Inline function:
     f1(x) = x.^2+x-1
```

```
>> test = [2 4 3];
>> f1 (test)
ans =
     5    19    11
```

[17] Das Function Handle stellt einen eigenen Datentyp dar, der wie eine Struktur aufgebaut ist.
[18] Für eine detaillierte Erklärung zu Inline Objects siehe auch Kap. 7.1.

2.5.6 P-Code und `clear functions`

Wird ein MATLAB-Skript oder eine MATLAB-Funktion zum ersten Mal aufgerufen, erzeugt MATLAB einen **Pseudo-Code**, der dann ausgeführt wird. Bleibt daher die Änderung eines M-Files ohne Wirkung, wurde dieser Übersetzungsvorgang nicht neu gestartet. Dies sollte automatisch geschehen, kann aber auch durch den Befehl `clear functions` erzwungen werden (die M-Files werden dabei natürlich nicht gelöscht!).

Der Pseudo-Code kann auch mit `pcode` *datei* erzeugt und als P-File abgespeichert werden, um z.B. Algorithmen zu verschlüsseln. Existieren sowohl M-File als auch das gleichnamige P-File, führt MATLAB immer das P-File aus. Nach einer Änderung des M-Files muss daher der Befehl `pcode` wiederholt oder das P-File gelöscht werden!

Funktionen (erweitert)

f_handle `= @`*funktion*	Function Handle auf *funktion* erzeugen
`functions (`*f_handle*`)`	Function-Handle-Informationen abrufen
`feval (`*f_handle*`)`	Function Handle (d.h. Funktion) ausführen
f `= inline (`*funktion*`)`	Funktion als Inline Object definieren
`pcode` *datei*	P-Code zu M-File `datei.m` speichern
`clear functions`	Alle P-Codes im Workspace löschen

2.6 Code-Optimierung in MATLAB

Werden in MATLAB umfangreiche Daten bearbeitet, gilt es Hardware-Ressourcen möglichst effizient zu nutzen. Dieser Abschnitt beschreibt zunächst den MATLAB-Profiler als wichtiges Werkzeug für die Suche nach Optimierungspotential. Anschließend werden einige Tipps zur Reduktion der Rechenzeit und des Speicherbedarfs sowie zur Vermeidung typischer Fehler gegeben.

2.6.1 Der MATLAB-Profiler

Der MATLAB-Profiler wird über das Menü *Desktop/Profiler* des MATLAB-Desktops oder den Button aufgerufen. Im Feld *Run this code* wird die zu testende Funktion oder der Name des M-Files (ohne Endung) eingetragen und der Profiler mit dem Button *Start Profiling* gestartet.

Als Ergebnis zeigt der Profiler zunächst eine Übersicht an (*Profile Summary*, siehe Abb. 2.6). Durch Anklicken der gewünschten Funktion oder Datei in der linken Spalte gelangt man zur Detailansicht (siehe Abb. 2.7). Diese zeigt zuerst eine tabellarische Übersicht, die unter anderem folgende Rubriken enthält:

- Laufzeit[19] (`time`)
- Anzahl der Aufrufe (`calls` bzw. `numcalls`)

[19] Alternative Laufzeitmessung mit den Funktionen `tic` und `toc`, siehe Übungsaufgabe in Kap. 2.7.4.

- Nicht erreichter Code (`line` bzw. `coverage` / `noncoverage`)
- Code-Prüfung (`code analyzer`)

Weiter unten folgt schließlich ein Listing mit darüber liegendem Auswahlfeld. Durch Anklicken einer der genannten Rubriken kann dieses Kriterium durch farbige Markierung der relevanten Zeilen hervorgehoben werden.

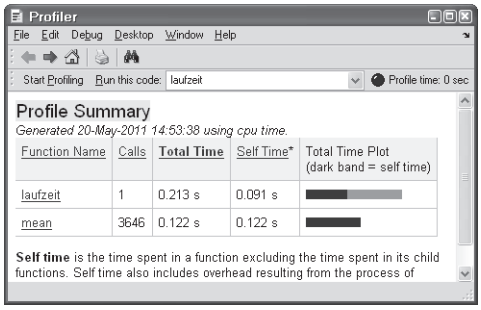

Abb. 2.6: *Profiler mit Summary*

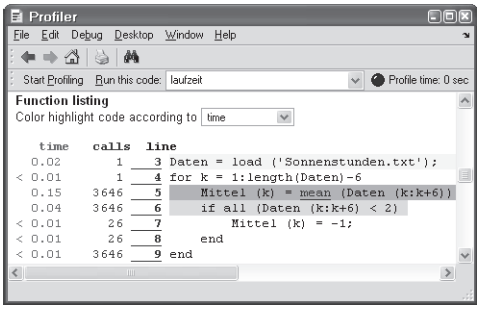

Abb. 2.7: *Listing mit Laufzeit-Markierung*

Wird im genannten Auswahlfeld die Rubrik `code analyzer` gewählt, werden Zeilen mit fehleranfälligen oder zeitintensiven Konstrukten markiert und mit erklärenden Kommentaren versehen.

2.6.2 Optimierung von Rechenzeit und Speicherbedarf

MATLAB kompiliert eingegebene Befehle, Skripts und Funktionen – unbemerkt vom Anwender – bereits vor der Ausführung. Dies geschieht durch den so genannten *JIT-Accelerator*.[20] Dennoch lässt sich die Effizienz von MATLAB-Skripts und -Funktionen durch geeignete Programmierung noch weiter verbessern.

In der Regel wird die meiste Laufzeit an wenigen Stellen innerhalb oft durchlaufener Schleifen des MATLAB-Codes verschenkt; diese findet man am besten mittels des MATLAB-Profilers. Der folgende Abschnitt zeigt, wie solche Stellen optimiert werden können.

- **Große Arrays** sollten vor ihrer Verwendung mit der maximal benötigten Größe **vorbelegt** werden, damit MATLAB den benötigten Speicher alloziert. Dafür eignen sich Befehle wie `zeros(`*zeilen*, *spalten*[, *'typ'*]`)`. Optional kann dabei auch ein **kompakter Datentyp** aus Kap. 2.2.1 angegeben werden.
- **Shortcut-Operatoren** `&&` und `||` (anstelle von `&` und `|`) beschleunigen skalare logische Verknüpfungen z.B. bei `if`- oder `while`-Abfragen.
- Alle **Hardware-Zugriffe** benötigen viel Zeit und sollten daher innerhalb häufig durchlaufener Schleifen unterbleiben. Dies gilt für Ausgaben im Command Window, Lesen und Schreiben von Dateien sowie für grafische Ausgaben jeder Art.

[20] Just In Time Accelerator

- Das **Löschen** nicht mehr benötigter **Variablen** mit `clear` *variable* und das Schließen nicht benötigter **Figures** mit `close` *nummer* gibt den dadurch belegten Speicher (zumindest zum Teil) wieder frei.

Achtung: *Bei der Optimierung sind immer auch die Lesbarkeit und Wartung eines M-Files zu berücksichtigen! Daher gelten die folgenden Anregungen ausschließlich für Stellen mit akuten Laufzeitproblemen:*

- **Strukturen**, **Cell-Arrays** sowie Arrays hoher Dimension erfordern einen großen Verwaltungsaufwand und sollten daher in inneren Schleifen vermieden werden.
- Jeder Aufruf von MATLAB-**Skripts** und -**Funktionen**, die als separates M-File vorliegen, benötigt zusätzlich Zeit. Diese lässt sich verringern, indem der Inhalt des aufgerufenen M-Files (entsprechend angepasst) in die Schleife kopiert wird.
- Viele MATLAB-Funktionen (M-Files) bieten Aufruf-Varianten und detaillierte Fehlerbehandlung; dieser **Overhead** benötigt jedoch Rechenzeit. Bei genau definierten Rahmenbedingungen kann es sich lohnen, stattdessen eine dafür optimierte Funktion zu erstellen und diese in das aufrufende M-File zu integrieren.
- Die Verwendung **globaler Variablen**, um Daten zwischen Funktionen auszutauschen, vermeidet das mehrfache Anlegen derselben Daten im Arbeitsspeicher. Vorsicht: Es besteht die Gefahr des unbeabsichtigten Überschreibens!

Schließlich kann MATLAB mit `matlab.exe -nojvm` auch **ohne** die **Java Virtual Machine** neu gestartet werden. Dies spart deutlich Arbeitsspeicher. Dafür steht allein das *Command Window* ohne den gewohnten Bedienkomfort und (in Zukunft) auch ohne Figures und Grafische Benutzeroberfläche zur Verfügung (siehe Kap. 3).

2.6.3 Tipps zur Fehlersuche

Auch erfahrene MATLAB-Anwender bleiben von Flüchtigkeitsfehlern nicht immer verschont. Manche Fehler sind besonders tückisch, da sie eine MATLAB-Fehlermeldung an ganz anderer Stelle bewirken oder lediglich durch „überraschende“ Ergebnisse auffallen. Einige solche Fehler werden im Folgenden beschrieben und mit Tipps zur Abhilfe ergänzt:

- **Alte Daten** eines vorherigen Projekts „überleben“ im Workspace. Werden gleiche Variablennamen weiter verwendet, sind diese Variablen dann noch falsch belegt. Gleiches gilt, wenn ein M-File nacheinander mit verschiedenen Datensätzen aufgerufen wird. Neben unplausiblen Ergebnissen sind typische Fehlermeldungen *„Index exceeds matrix dimensions“* oder *„Matrix dimensions must agree“*. Abhilfe mit `clear all`.
- Ein mehrdimensionales Array wird mit **zuwenig Indizes** adressiert. MATLAB fasst das Array dann fälschlicherweise als Vektor bzw. Array niedrigerer Dimension auf (siehe auch Fußnote auf Seite 15). Abhilfe durch Debugger und Anzeigen der betreffenden Variable im Variable Editor.

- **Globale** oder Workspace-Variablen werden in einem anderen Skript (oder Funktion) unbeabsichtigt geändert. Abhilfe durch Funktionen mit lokalen Variablen und durch (neue) eindeutige Namen für globale und Workspace-Variablen.

- **Tippfehler bei Variablennamen**, z.B. `varible(k) = 2*variable(k-1)`, bleiben unentdeckt, wenn die Variable bei der Ausführung des fehlerhaften Codes bereits vorhanden ist. In diesem Fall wird eine neue Variable mit falschem Namen `varible` erzeugt; die eigentliche Variable dagegen bleibt unverändert. Abhilfe durch den MATLAB-Profiler: Unter der Rubrik `code analyzer` wird auf nicht verwendete Variablen und damit auf solche Tippfehler hingewiesen.

- **Leerzeichen** vor Klammern bei der Adressierung von Cell-Arrays und beim Zusammensetzen von Strings können zu Fehlermeldungen führen. Fehlerfrei funktioniert z.B. `zelle{2}` statt `zelle {2}` (siehe Beispiel auf Seite 15) sowie `[num2str(2.5), 'm']` statt `[num2str (2.5), 'm']` (siehe Beispiel Kap. 3.2.3).

- Variablen und Funktionen besitzen **identische Namen** oder es existieren mehrere M-Files desselben Namens im MATLAB-Pfad. Die Abfrage `which -all` *name* listet alle Vorkommen eines Namens im Workspace und MATLAB-Pfad auf.

- Insbesondere beim **Debuggen großer Funktionen** oder Skripts ist es hilfreich, die Ausführung genau an der ersten fehlerhaften Stelle im Code anzuhalten. Dazu wird im Menü *Debug/Stop if Errors/Warnings* die Option *Always stop* gewählt. Der Editor öffnet die Funktion dann automatisch an der entsprechenden Stelle. Diese Einstellung wir durch `clear all` sowie beim Löschen aller Breakpoints mit  wieder zurückgesetzt.

2.7 Übungsaufgaben

2.7.1 Rechengenauigkeit

Alle Rechenoperationen in MATLAB werden mit dem Variablentyp `double` (64 Bit) durchgeführt, soweit kein anderer Typ explizit angegeben ist. An einem Beispiel kann die Grenze der Rechengenauigkeit untersucht werden.

Quadrieren Sie die Zahl 1.000 000 000 1 insgesamt 32-mal! Berechnen Sie das Ergebnis zunächst in einer Schleife! Bestimmen Sie zum Vergleich das Ergebnis mit einer einzigen Potenzfunktion! Was fällt Ihnen auf?

Hinweis: Wählen Sie ein geeignetes Format der Zahlendarstellung, z.B. mit dem Befehl `format long g`.

2.7.2 Fibonacci-Folge

Nach dem italienischen Mathematiker *Leonardo Pisano Fibonacci* (ca. 1175 bis 1250 n. Chr.) ist folgende Reihe benannt:

$$n(k) = n(k-1) + n(k-2) \qquad \text{mit} \quad n(1) = n(2) = 1$$

Berechnen Sie die nächsten 10 Elemente mit einer Schleife! Beginnen Sie mit dem Vektor `[1 1]`, an den Sie bei jedem Durchlauf ein Element anfügen! Das wievielte Element überschreitet als Erstes den Wert 10^{20}? Verwenden Sie dazu eine Schleife mit entsprechender Abbruchbedingung!

Die Elemente der Fibonacci-Reihe können nach Gleichung (2.1) auch explizit bestimmt werden:

$$n(k) = \frac{F^k - (1-F)^k}{\sqrt{5}} \qquad \text{mit} \qquad F = \frac{1+\sqrt{5}}{2} \tag{2.1}$$

Berechnen Sie damit **ohne Schleife** das 12. bis 20. Element mit elementweisen Vektoroperationen!

2.7.3 Funktion `gerade`

Die Funktion `gerade` soll aus zwei Punkte-Paaren (x_1, y_1) und (x_2, y_2) in kartesischen Koordinaten die Parameter Steigung m und y_0 (y-Wert für $x = 0$) der Geradengleichung

$$y = m \cdot x + y_0$$

bestimmen. Es müssen also **4 Werte** `x1`, `y1`, `x2`, `y2` an die Funktion übergeben werden, die wiederum 2 Werte `m` und `y0` zurückgibt. Für den Fall einer senkrechten Steigung ($x_1 = x_2$) soll eine Warnung `'Steigung unendlich!'` mittels des Befehls `disp`[21)] ausgegeben werden.

[21)] Zur Verwendung des Befehls `disp` siehe Kapitel 3.2.3.

Werden nur **2 Werte** übergeben, so sollen neben den Parametern `m` und `y0` (in diesem Fall = 0) noch der Abstand `r` des Punktes vom Ursprung (Radius) und der mathematisch positive Winkel `phi` zur x-Achse in Grad zurückgegeben werden.

Wird eine andere Anzahl von Werten übergeben, so soll eine Fehlermeldung mit `'Falsche Anzahl von Parametern!'` und der Hilfetext der Funktion (Aufruf mit `help gerade`) ausgegeben werden.

2.7.4 Berechnungszeiten ermitteln

In MATLAB existieren verschiedene Befehle zur Abfrage der Systemzeit und zum Messen der Berechnungsdauer, darunter `tic` und `toc`: `tic` startet eine Stoppuhr, `toc` hält diese wieder an und gibt die verstrichene Zeit aus. Somit kann die Laufzeit von Programmen ermittelt werden, indem sie innerhalb eines solchen `tic-toc`-Paares aufgerufen werden.

Zum Vergleich der Laufzeit verschiedener Programmstrukturen werden die Funktionen `mittelwerte` aus Kap. 2.5 und `mittelwerte2` aus Kap. 2.5.1 verwendet.

Schreiben Sie ein Skript `mittelwerte_zeit.m`, in dem beide Funktionen jeweils 10 000-mal mit einem Testvektor aufgerufen werden. Stoppen Sie die Zeit für 10 000 Durchläufe für jede der beiden Funktionen. Wiederholen Sie diesen Test 10-mal (mittels einer Schleife) und mitteln Sie anschließend die Zeiten. Welche Funktion ist schneller und warum?

3 Eingabe und Ausgabe in MATLAB

Die in diesem Kapitel vorgestellten Befehle dienen der Steuerung der Bildschirmausgabe, der komfortablen Ein- und Ausgabe von Daten im Dialog mit dem Benutzer sowie dem Import und Export von Dateien. Weitere Schwerpunkte sind die grafische Darstellung von Ergebnissen in MATLAB und der Import und Export von Grafiken.

3.1 Steuerung der Bildschirmausgabe

Die Befehle zur Steuerung der Bildschirmausgabe sind syntaktisch an die entsprechenden UNIX-Befehle angelehnt. Sie sind vor allem zum Debuggen von Programmen oder zum Vorführen bestimmter Zusammenhänge hilfreich.

Soll die **Bildschirmausgabe seitenweise** erfolgen, wird mit `more on` die seitenweise Bildschirmausgabe ein-, mit `more off` ausgeschaltet. `more(`n`)` zeigt jeweils n Zeilen je Seite an. Die Steuerung erfolgt wie in UNIX: Ist die Ausgabe länger als eine Seite, so schaltet die Return-Taste um eine Zeile weiter, die Leertaste zeigt die nächste Seite an und die Taste Q bricht die Ausgabe ab.

Mit dem Befehl `echo on` können die beim Aufrufen eines MATLAB-Skripts oder einer Funktion ausgeführten **Befehle angezeigt** werden; `echo off` schaltet dies wieder aus. Beim Ausführen einer Funktion *funktion* werden die darin aufgerufenen Befehle mit `echo` *funktion* `on` ausgegeben (Ausschalten mit `echo` *funktion* `off`), wie das folgende Beispiel zeigt:

```
>> echo mittelwerte on
>> [A, G] = mittelwerte (1:2:7)

arithm = mean(x);                          % Arithmetisches Mittel
geom   = prod(x).^(1/length(x));           % Geometrisches Mittel
A =
     4
G =
    3.2011
```

Die **Bildschirmausgabe anhalten** kann man mit dem Befehl `pause`, der die Ausgabe beim nächsten Tastendruck fortsetzt, während `pause(`n`)` die Ausgabe für n Sekunden anhält. Mit `pause off` schaltet man alle folgenden `pause`-Befehle aus; es wird also nicht mehr angehalten. Mit `pause on` werden die `pause`-Befehle wieder aktiviert. So gibt die folgende Schleife jeweils den Zähler aus und wartet dann die dem Zähler entsprechende

Zeit in Sekunden.[1] Am Ende muss mit einem Tastendruck quittiert werden.

```
>> for i=1:2:6, disp(i), pause(i), end, disp('Ende'), pause
     1
     3
     5
Ende
>>
```

Auch sehr nützlich ist der Befehl `clc` (*Clear Command Window*), mit dem alle Ein- und Ausgaben am **Command Window gelöscht** werden und der Cursor in die erste Zeile gesetzt wird.

Steuerung der Bildschirmausgabe

`more`	Seitenweise Ausgabe am Bildschirm
`echo`	Zeigt Befehle bei Ausführung von Skripts und Funktionen an
`pause`	Hält die Bildschirmausgabe an (bis Tastendruck)
`pause(`n`)`	Hält die Bildschirmausgabe für n Sekunden an
`clc`	Löscht alle Ein- und Ausgaben im Command Window

3.2 Benutzerdialoge

Benutzerdialoge zur Eingabe und Ausgabe von Text und Daten können mit den im Folgenden beschriebenen Befehlen erstellt werden. Zu diesem Zweck wird zunächst die Behandlung von Text in MATLAB betrachtet.

3.2.1 Text in MATLAB (Strings)

Texte (*Strings*) werden in einfache Anführungszeichen eingeschlossen und können Variablen zugewiesen werden:[2] *string* `= '`*text*`'`. String-Variablen werden als Zeilen-Vektor gespeichert und können mit `['`*text*`1','`*text*`2']` zusammengesetzt werden. Geht ein String in einem Skript über mehrere Zeilen, muss der String am Ende jeder Zeile abgeschlossen werden, da andernfalls der Umbruch mit `...` auch als Teil des Strings interpretiert werden würde.

```
>> string = ['Das ist', ' ', 'ein String!']
string =
Das ist ein String!
```

```
>> whos string
  Name        Size            Bytes  Class    Attributes

  string      1x19               38  char
```

[1] Der Befehl `disp` zur Ausgabe von Zahlen und Strings wird in Kap. 3.2.3 behandelt.
[2] MATLAB wählt automatisch den Datentyp `char`, ohne dass dieser explizit angegeben werden muss.

3.2.2 Eingabedialog

Die Abfrage von Daten erfolgt mit dem Befehl *variable* = `input` (*string*). Der String wird ausgegeben; die Eingabe wird der Variablen zugewiesen. Wird keine Zahl, sondern ein String abgefragt, lautet der Befehl *string* = `input` (*string*, `'s'`). Als Sonderzeichen stehen innerhalb des Strings der Zeilenumbruch `\n`, das einfache Anführungszeichen `''` sowie der Backslash `\\` zur Verfügung. Hier das Beispiel eines Eingabedialogs:

```
preis = input (['Wieviel kostet heuer \n', ...
                'die Wiesn-Maß ?      '])
waehrung = input ('Währung ?             ', 's');
```

Nach Aufruf des Skripts kann man folgenden Dialog führen:

```
Wieviel kostet heuer
die Wiesn-Maß ?      9.20
Währung ?             EUR
```

3.2.3 Formatierte Ausgabe

Für die Ausgabe lassen sich Daten und Strings ebenfalls formatieren. Der Befehl `disp` (*string*) gibt einen String am Bildschirm aus. Interessant wird dieser Befehl, wenn der String zur Laufzeit aus variablen Texten zusammengesetzt wird; Zahlen müssen dann mit `num2str` (*variable* [, *format*]) ebenfalls in einen String umgewandelt werden.

Für vektorielle Daten kann die Ausgabe mit *string* = `sprintf` (*string*, *variable*) formatiert und mit `disp` ausgegeben werden. Die Syntax zur Formatierung entspricht bei `num2str` und `sprintf` im Wesentlichen der der Sprache C (genaue Infos über `doc sprintf`). Alle Variablen müssen in einer einzigen Matrix angegeben werden, wobei jede Spalte einen Datensatz für die Ausgabe darstellt. Fortsetzung des obigen Beispiels:

```
disp (['Aber ', num2str(preis, '%0.2f'), ' ', waehrung, ...
       ' pro Maß wird ein teures Vergnügen!'])
disp (' ')                                                  % Leerzeile ausgeben
ausgabe = sprintf ('Zwei Maß kosten dann %2.2f %s.', ...  % Ausgabe formatieren
                   2*preis, waehrung);
disp (ausgabe)                                              % ausgeben
mehr = sprintf ('%4d Maß kosten dann %2.2f.\n', ...       % Ausgabe formatieren
                [3:5; (3:5)*preis]);                        % Vektoren für Ausgabe
disp (mehr)                                                 % ausgeben
```

Als Ausgabe erhält man:

```
Aber 9.20 EUR pro Maß wird ein teures Vergnügen!

Zwei Maß kosten dann 18.40 EUR.
   3 Maß kosten dann 27.60.
   4 Maß kosten dann 36.80.
   5 Maß kosten dann 46.00.
```

Neben den oben beschriebenen Möglichkeiten erlaubt MATLAB auch die Erstellung grafischer Benutzerschnittstellen, die in Kap. 3.6 behandelt werden.

Benutzerdialoge

variable =	`input (`*string*`)`	Abfrage einer Variablen
string =	`input (`*string*`, 's')`	Abfrage eines Strings
string =	`num2str (`*variable* [, *format*]`)`	Umwandlung Zahl in String
string =	`sprintf (`*string*, *variable*`)`	Formatierten String erzeugen
	`disp (`*string*`)`	Textausgabe auf Bildschirm

Sonderzeichen

`\n`	Zeilenumbruch
`\t`	Tabulator
`\\`	Backslash \
`%%`	Prozent %
`''`	Anführungszeichen '

Formatierung

`%d`	Ganze Zahl (z.B. 321)
`%x`	Ganze Zahl hexadezimal
`%5.2f`	Fließkomma-Zahl (z.B. 54.21)
`%.2e`	Exponentenschreibweise (z.B. 5.42e+001)
`%s`	String

3.3 Import und Export von Daten

3.3.1 Standardformate

Für den Import und Export von Dateien unterstützt MATLAB standardmäßig ASCII-Text sowie ein spezielles MATLAB-Binärformat. Das Laden und Speichern geschieht mit den Befehlen `load` *dateiname* [*variable*1 *variable*2 ...] und `save` *dateiname* [*variable*1 *variable*2 ...]. Die alternativen Klammerversionen `load ('`*dateiname*`'` [`,'`*variable*1`','`*variable*2`',` ...]`)` und `save ('`*dateiname*`'` [`,'`*variable*1`',` `'`*variable*2`',` ...]`)` erlauben, den Dateinamen auch als String-Variable zu übergeben.

Wird nach `load` ein Dateiname **ohne** Endung angegeben, nimmt MATLAB an, dass die Daten im MATLAB-Binärformat (so genannte *MAT-Files*, mit der Endung *MAT*) vorliegen. Bei diesem sind außer den Werten der Variablen auch deren Namen gespeichert und die Daten, falls möglich, immer zusätzlich komprimiert.[3)]

Zum Einlesen von Daten im ASCII-Format wird der Dateiname **mit** Endung angegeben. Die Daten einer Zeile müssen dann durch Leerzeichen oder Tabulatoren getrennt sein (keine Kommata!). Jede Zeile muss gleich viele Elemente besitzen (Kommentarzeilen sind aber erlaubt). Die eingelesenen Werte werden standardmäßig einer Variablen mit dem Namen der Datei zugewiesen. Mit *variable* `= load ('`*dateiname*`')` werden die eingelesenen Werte stattdessen der angegebenen Variablen zugewiesen.

```
>> test_vektor = [0:0.1:10]';                    % Spaltenvektor
>> test_matrix = [test_vektor cos(test_vektor)]; % Matrix
>> save test                                     % Speichern in Datei test.mat
>> clear                                         % Workspace löschen
>> load test                                     % Laden aus Datei test.mat

>> who                                           % Anzeige des Workspace
Your variables are:
test_matrix  test_vektor
```

[3)] Aufrgund der Komprimierung haben auch kompakte Datentypen aus Kap. 2.2.1 hier keinen Vorteil.

Beim Speichern mit `save` erzeugt MATLAB standardmäßig ein MAT-File. Die Option `-append` hängt zu speichernde Variablen an ein bestehendes MAT-File an. Werden beim Laden und Speichern keine Variablen explizit angegeben, lädt MATLAB jeweils alle Variablen des MAT-Files bzw. speichert alle Variablen aus dem Workspace.

Mit der alternativen Option `-ascii` kann ein ASCII-Format ausgegeben werden. Die Namen der Variablen werden dann nicht gespeichert. Speichern im ASCII-Format schreibt alle Variablen untereinander in die Ausgabedatei. Bei unterschiedlicher Spaltenzahl ist ein Einlesen in MATLAB dann nicht mehr ohne weiteres möglich. Selbst bei gleicher Spaltenzahl können die ursprünglichen Variablen aufgrund der nicht gespeicherten Namen nicht mehr separiert werden.

```
>> save test.txt -ascii test_matrix   % Speichern in Datei test.txt
>> clear                              % Workspace löschen
>> load test.txt                      % Laden aus Datei test.txt

>> who                                % Anzeige des Workspace
Your variables are:
test
```

Die Befehle `xlswrite('`*datei*`',`*variable*`)` und `xlsread('`*datei*`')` schließlich schreiben in eine Excel-Datei bzw. lesen daraus. Für Details sei auf die MATLAB-Hilfe verwiesen.

Datenimport und -export Standardformate

	`load` *datei* [*variable* ...]	Laden aus MAT-File
	`save` *datei* [*variable* ...]	Speichern in MAT-File
[*variable* =]	`load` *datei.endung*	Laden aus ASCII-File
	`save` *datei.endung* `-ascii` [*variable* ...]	Speichern in ASCII-File
variable =	`xlsread ('`*datei.xls*`')`	Laden aus Excel-File
	`xlswrite ('`*datei.xls*`',`*variable*`)`	Speichern in Excel-File

3.3.2 Formatierte Textdateien

Ein universeller Befehl zum **Einlesen** beliebig formatierter Textdateien ist *vektor* `= fscanf (`*fid*`,'`*format*`')`, der ähnlich wie in der Sprache C arbeitet. Hauptunterschied ist allerdings die vektorisierte Verarbeitung der Daten, d.h. der Formatstring wird so lange wiederholt auf die Daten angewandt, bis das Dateiende erreicht ist oder keine Übereinstimmung mehr gefunden wird. Dies bringt insbesondere bei großen Textdateien einen deutlichen Zeitgewinn gegenüber zeilenweisem Zugriff.

Die Befehle *string* `= fgetl (`*fid*`)` und *string* `= fgets (`*fid*`,` *anzahl*`)` lesen bei jedem Aufruf jeweils eine ganze Zeile bzw. eine bestimmte (maximale) Anzahl an Zeichen aus.

Zum Öffnen und Schließen der Datei sind dabei jeweils die zusätzlichen Befehle *fid* `= fopen('`*datei.endung*`','`*zugriff*`')` und `fclose (`*fid*`)` notwendig, wobei *fid* das Handle der geöffneten Datei ist. Der Parameter *zugriff* kann aus folgenden Strings bestehen

(siehe auch `doc fopen`): `'w'` und `'a'` für Schreiben bzw. Anfügen (die Datei wird bei Bedarf angelegt) sowie `'r'` für Lesen.

Für die formatierte **Ausgabe** in eine Textdatei kann der Befehl `fprintf` verwendet werden. Die Syntax entspricht der von `sprintf`; es können jeweils nur reelle Zahlen verarbeitet werden. Die Bierpreise von Kap. 3.2.3 werden mit den nachstehenden Befehlen in eine Datei `bier.txt` geschrieben:[4)]

```
>> bier_id = fopen ('bier.txt', 'w');
>> fprintf (bier_id, '%s\r\n', 'Bierpreis-', 'Hochrechnung');   % Kopfzeilen
>> fprintf (bier_id, '%4d Maß kosten dann %2.2f.\r\n', ...
                    [3:5; (3:5)*preis]);
>> fclose (bier_id);
```

Die Ausgabe der unterschiedlich langen Kopfzeilen kann auch elegant mit einem Cell Array erfolgen, indem zunächst jede Zeile k in eine Zelle `kopfzeilen`$\{k\}$ gespeichert und dann mit `fprintf(bier_id,'%s\r\n',kopfzeilen{:})` ausgegeben wird.

Für das **Einlesen beliebiger Textdaten** stellt MATLAB die sehr flexible Funktion *cellarray* = `textscan(`*fid*, *'format'*`)` zur Verfügung. Optional können weitere Parameter und zugehörige Werte angegeben werden (siehe `doc textscan`). Das folgende Beispiel liest die Datei `chaos.txt` zeilenweise in ein Cell Array aus Strings ein:

```
>> fid = fopen ('chaos.txt', 'r');
>> zeilen = textscan (fid, '%s', ...       % Datei-Handle und Formatstring
                      'delimiter', '\n');  % nur Zeilenumbruch als Trennzeichen
>> fclose (fid);
>> zeilen = zeilen{:}                      % Umwandeln in einfaches Cell-Array
zeilen =
    'Kopfzeile'
    '4,5,6,'
    '1, 2,,'
    '3,*,0,'
    'Fusszeile'
```

Die so eingelesenen Zeilen können dann z.B. analog zu `fscanf` mit dem Befehl *vektor* = `sscanf(`*string*, *'format'*`)` weiterverarbeitet werden.

Ein weiteres Beispiel liest dieselbe Datei als numerische Daten ein; dabei werden leere bzw. ungültige Werte (hier `*`) durch `NaN` (*Not a Number*) ersetzt.

```
>> fid = fopen ('chaos.txt', 'r');
>> matrix = textscan (fid, '%f %f %f',     ...  % Datei-Handle und Formatstring
                      'delimiter',   ',', ...  % Komma als Trennzeichen
                      'whitespace', '* ', ...  % überspringt * und Leerzeichen
                      'emptyvalue',  NaN, ...  % ersetzt leere Felder durch NaN
                      'headerlines',   1, ...  % überspringt die erste Zeile
                      'collectoutput', 1);     % fasst Daten zu Matrix zusammen
>> fclose (fid);
```

Für jeden einzelnen Formatstring gibt der Befehl ein Cell-Array zurück. Mit der Option

[4)] Das Beispiel erzeugt einen Text im DOS-Format: Der Format-String `\r\n` gibt die beiden Zeichen *carriage return* (CR) und *linefeed* (LF) für den Zeilenumbruch aus. Unter UNIX genügt `\n`.