Skript

Matlab-Tutorium

Institut für Statistik und Operations Research
Karl-Franzens-Universität Graz

6. Auflage

Autor:
Michael Mochart

Letzter Edit:
Nathalie Frieß, Yanick Dickbauer

Aufbauend auf den Vorversionen von:
Veronika Bosankic, Robert Eder

März 2019

Content

1 Einleitung 4

1.1 Allgemeines 4

1.2 Ziel des Tutoriums 4

1.3 Was ist Matlab? 4

1.4 Wieso Matlab? 4

1.5 Matlab starten 5

1.5.1 Von Uni-Rechnern 5

1.5.2 Von beliebigen Rechnern 5

1.6 Benutzeroberfläche 6

1.6.1 Current Folder 6

1.6.2 Editor 6

1.6.3 Workspace 7

1.6.4 Command Window 7

1.6.5 Command History 8

1.7 Wichtige Tipps und Tricks für Einsteiger 9

1.8 Weitere Literatur und Links 10

2 Programmier-Vokabular 11

3 Programmieren mit Matlab 12

3.1 Variablen 12

3.2 Spezielle Zeichen 13

3.2.1 Punkt, Komma, Strichpunkt 13

3.2.2 Klammern 15

3.2.3 Kommentare 15

3.3 Wichtige Matlab-Befehle/Funktionen 16

3.3.1 Zufallszahlen generieren - rand(), randn() 16

3.3.2 Sortieren einer Zahlenfolge - sort() 16

3.3.3 Eingabe von Texten und Zahlen – input() 17

3.3.4 Ausgabe von Texten und Zahlen – disp(), fprintf() 17

3.3.5 Größe einer Matrix ermitteln – size() 18

3.3.6 Suchen von Werten innerhalb einer Matrix – find() 18
3.3.7 Abfrage, ob Liste leer ist – isempty() 19
3.3.8 Ausgabe einer Grafik – plot() 19
3.3.9 Finden eines Maximums/Minimums - max()/min() 20
3.3.10 Berechnen der Summe - sum() 21
3.3.11 Mittelwert und Standardabweichung - mean(), std() 21
3.3.12 Zahl auf gerade/ungerade prüfen - mod() 22
3.3.13 Weitere nützliche Befehle/Funktionen 22
3.4 Vektoren, Matrizen und Operatoren 23
3.4.1 Manuelles Anlegen von Matrizen 24
3.4.2 Automatisches Anlegen von Matrizen mit Befehlen 25
3.4.3 Zugriff auf einzelne Matrixelemente - Indizierung 29
3.4.4 Rechnen mit Matrizen - Verwendung von Operatoren 29
3.5 Skripte und Funktionen 34
3.5.1 Skripte 34
3.5.2 Funktionen 36
3.6 Verzweigungen 37
3.6.1 If-Verzweigung 37
3.7 Schleifen 39
3.7.1 For-Schleife 39
3.7.2 While-Schleife 41
3.8 Weitere Datentypen 42
3.8.1 Datentyp Struktur (struct) 42
3.8.2 Datentyp Block (cell) 43
3.9 Graphen plotten 44
3.10 Debugging (Fehlersuche) 46
4 Übungs-Beispiele 49

1 Einleitung

1.1 Allgemeines

Dieses Skript dient als Unterlage für das Matlab-Tutorium des Instituts für Statistik und Operations Research und des Instituts für Produktion und Logistik der Karl-Franzens-Universität Graz. Es stellt keinen Anspruch auf Vollständigkeit und kann regelmäßig überarbeitet werden. Weiters liegt die inhaltliche Verantwortung beim Autor. Die aktuelle Auflage (6. Auflage) baut an vielen Stellen an den vorhergegangenen Auflagen, welche von Veronika Bosankic, Robert Eder und Yanick Dickbauer verfasst wurden, auf. Es empfiehlt sich für Programmier-Einsteiger dieses Skript ergänzend zum Tutoriums-Besuch zu lesen und Teile daraus auch für die Lehrveranstaltungen am Institut anzuwenden.

1.2 Ziel des Tutoriums

In diesem Tutorium sollen die Teilnehmer grundlegende Kenntnisse im Arbeiten mit Softwarelösungen und deren Anwendung auf praktische Problemstellungen in den Bereichen Operations Research und Produktion/Logistik mit Hilfe der Software-Entwicklungsumgebung Matlab erlernen. Das Skriptum erhebt, wie oben erwähnt, keinen Anspruch auf Vollständigkeit. Es wurde speziell für Programmier-Einsteiger verfasst, die im Masterstudium Betriebswirtschaft die Spezialisierungen „Operations Research“ und/oder „Produktion und Logistik“ gewählt haben, um ihnen einen möglichst einfachen und reibungslosen Einstieg ins Programmieren zu ermöglichen.

1.3 Was ist Matlab?

Matlab ist eine kommerzielle Software-Entwicklungsumgebung, die vor allem zur Lösung von mathematischen Problemstellungen (primär für Matrizenrechnungen) und für grafische Darstellungen verwendet wird. Bei dem Wort Matlab handelt es sich um eine zusammengesetzte Abkürzung der Wörter **MAT**rix **LAB**oratory.

1.4 Wieso Matlab?

Matlab ist in der Industrie sowie an Hochschulen weit verbreitet und dient als häufig verwendetes Tool zur Automatisierung von Datenauswertungen und zur Durchführung von Simulationen. Deshalb gilt Matlab auch bei quantitativorientierten Wirtschaftsstudenten als sinnvolle Zusatzqualifikation, da es in der Wirtschaft, im speziellen in industrienahen Betrieben, oft zum Standard gehört.

Vorteile:

- leichter Einstieg für Programmier-Anfänger
- einfache Bedienung
- detaillierte Fehlermeldungen
- Debugging-Modus (Fehlersuche, Fehlerbehebung)
- einfache Visualisierung (2D, 3D,...)
- umfangreiche Internethilfe

Nachteile:

- hohe Lizenzkosten
- kein Open-Source-Programm
- Kostenlose Alternativen (Python,...)
- Evtl. Schwierigkeiten bei größeren Software-Projekten

Zusammenfassend kann man sagen, dass Matlab dem Anwender bereits viele Schritte der Softwareentwicklung und -anwendung abnimmt und somit einen schnellen Einstieg ins Programmieren ermöglicht. Eine häufig verwendete und kostenlose Alternative zu Matlab stellt die auch sehr einfach bedienbare Entwicklungsumgebung/Programmiersprache *Python* dar. Eine weitere, der Matlab Benutzeroberfläche sehr ähnlichen Umgebung, ist *Scilab*, was ebenfalls kostenlos erhältlich ist.

1.5 Matlab starten

1.5.1 Von Uni-Rechnern

An einigen PCs der Uni Graz ist Matlab bereits vorinstalliert, wie z.B. an den PCs im Max-Jung-Laboratorium, wo das Tutorium stattfindet. Hier können Sie über das Startmenü bzw. über eine Desktop-Verknüpfung das Programm öffnen.

1.5.2 Von beliebigen Rechnern

Befindet sich Matlab nicht vorinstalliert auf dem von Ihnen gewählten Uni-PC oder möchten Sie das Programm von Ihrem eigenen PC von zu Hause aus starten, so können Sie dies per Fernzugriff tun. Für den Aufruf von virtueller Software von Ihrem privaten Arbeitsgerät stellt die uniIT einen Applikationsserver über RDS zur Verfügung. Der Zugriff funktioniert auch außerhalb des Campus und auf Nicht-Windows-Betriebssystemen. Eine ausführlichere Anleitung zum Öffnen eines Programms über den Applikationsserver ist unter folgendem Link abrufbar:

https://static.uni-graz.at/fileadmin/uni-it/docs/RDS_Win10.pdf

1.6 Benutzeroberfläche

Im Folgenden wird die Benutzeroberfläche anhand der Matlab-Version "Matlab2016b" dargestellt, die auf den PCs im Tutoriums-Raum vorinstalliert ist. Sie unterscheidet sich optisch zu der Version "Matlab2017b", auf die man über den Applikationsserver zugreifen kann, ist aber funktional quasi ident.

1.6.1 Current Folder

Dieses Fenster zeigt dem Anwender seinen Arbeitsordner an, in dem er sich gerade befindet. Durch Doppelklick auf eines der Programme öffnet sich dieses im Editor. Es empfiehlt sich, eine übersichtliche Ordnerstruktur der Programme anzulegen, um bei vielen Programmen/Skripten/Funktionen den Überblick zu behalten. Weiters sollte man alle Skripten und Funktionen, die man auf der Uni programmiert, in seinem Z-Laufwerk abspeichern, da man auf die dort gespeicherten Dateien auch von zu Hause aus zugreifen kann. Figure 1.1 zeigt wie der Current Folder in der Matlab-Oberfläche aussieht.

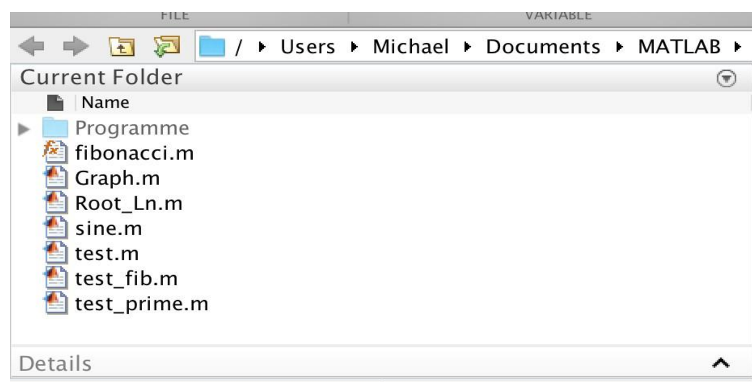

Figure 1.1: Der Current Folder

1.6.2 Editor

Im Editor kann man Programme (Skripten und Funktionen) schreiben. Grundsätzlich sind beim Öffnen des Programms die Skripten und Funktionen als Tabs geöffnet, die auch beim letzten Beenden geöffnet waren. Sollte man Matlab erstmalig öffnen oder man vor dem letztmaligen Beenden von Matlab alle Skripten und Funktionen geschlossen haben, so erscheint kein Editor. Erst wenn man in der linken oberen Ecke auf "New" klickt (Auswahl: Skript, Funktion, ...), öffnet sich der Editor wieder. Figure 1.2 zeigt den Editor in der Matlab-Oberfläche.

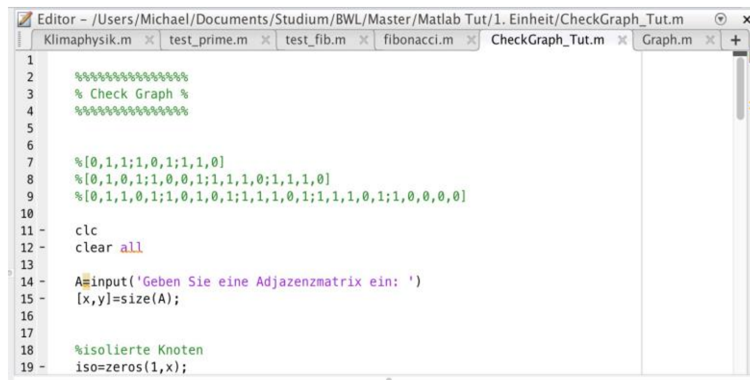

Figure 1.2: Der Editor

1.6.3 Workspace

Im Workspace werden alle Variablen (Zahlen, Vektoren, Matrizen) und deren aktueller Wert(e) aufgelistet, die zuvor vom Benutzer angelegt wurden. Durch Doppelklick auf die Variable öffnet sich der Variablen-Editor, in dem der aktuelle Variablenwert manuell angepasst werden kann. Hier kann man auch einzelne Variablen löschen. Wenn man alle Variablen löschen möchte, kann man dies durch den Befehl "clear all" tun. Figure 1.3 zeigt, wie der Workspace in der Matlab-Oberfläche aussieht.

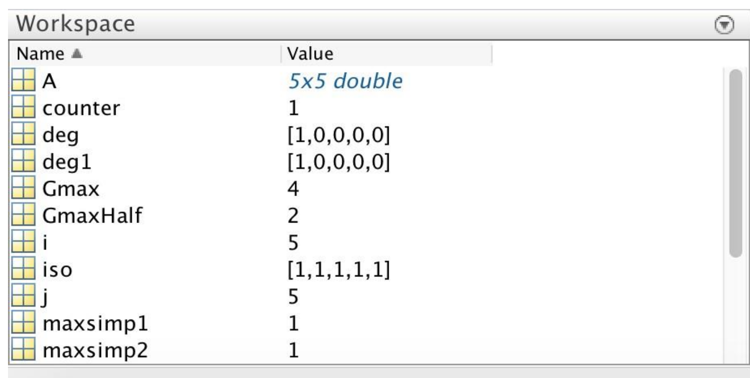

Figure 1.3: Der Workspace

1.6.4 Command Window

Das Command Window kann bereits zu Programmierzwecken verwendet werden, indem man Rechenoperationen oder Funktionsaufrufe eingibt und die Eingabe mit [ENTER] abschließt. Über die Pfeile [up] und [down] der Tastatur können, ähnlich wie bei einem Taschenrechner, zuvor eingegebene Befehle wieder aufgerufen werden. Weiters sind hier alle Ausgaben eines Skriptes oder einer Funktion zu sehen. Im Folgenden nun einige wichtige Befehle, die im Command Window benutzt werden können:

Befehl	Funktion
help	Aufruf der Hilfe
help *Befehl*	Aufruf der Hilfe zu einem bestimmten Befehl
doc *Befehl*	Matlab öffnet die Hilfe zum Befehl in einem eigenen Fenster
lookfor xy	Matlab sucht im Hilfesystem nach dem Schlüsselwort xy. Dieser Befehl ist vor allem dann hilfreich, wenn der gesuchte Befehlsnamen nicht bekannt ist.
clc	Löschen des Command Window Inhalts. Variablen im Workspace bleiben allerdings erhalten. Dieser Befehl sollte immer angewendet werden, bevor man eine neue Aufgabe beginnen möchte.
clear all	Löschen aller angelegten Variablen. Die erste Zeile eines jeden Skripts sollte diesen Befehl beinhalten, um sicherzustellen, dass zuvor angelegte Variablen nicht den Ablauf des Programms beeinflussen.
who	Auflisten aller verwendeten Variablen
whos	Auflisten aller verwendeten Variablen inkl. Details
exist Variable	Ermittelt ob eine bestimmte Variable noch existiert (1=Ja, 0=Nein)

Table 1.1: Nützliche Befehle in Matlab

Figure 1.4: Das Command Window

1.6.5 Command History

In der Command History werden alle bisherigen Eingaben in das Command Window als Liste dargestellt. Um Zeit zu sparen, kann mittels Drag & Drop ein Befehl aus der History direkt in das Command Window übertragen werden, um ihn erneut auszuführen.

1.7 Wichtige Tipps und Tricks für Einsteiger

Regeln für "sauberes" Programmieren gibt es viele. Im Folgenden sind einige derer, die vor allem für Einsteiger hilfreich sein können, aufgelistet. Es wird sehr empfohlen, sich diese Tipps und Tricks anzueignen und beim Programmieren von Anfang an zu verwenden, um spätere Probleme mit "unsauberen" Codes zu vermeiden.

- **Vor dem Start ein Konzept überlegen**

  Eine Möglichkeit ist es, dass Sie sich mit Hilfe von Kommentaren eine Überschriften-Struktur in Ihrem noch leeren Skript anlegen und anschließend die Lücken dazwischen mit Code füllen. Im besten Fall machen Sie sich jedoch im Vorhinein (altmodisch) Notizen mit Stift und Papier und erstellen daraus ein sogenanntes Flussdiagramm, in dem Ihr Lösungsansatz bereits abgebildet ist. An das halten Sie sich während der gesamten Arbeit, um nie die Übersicht zu verlieren und strukturiert vorzugehen.

- **KISS – Keep It Short and Simple**

  Eine kurze Lösung hat oft Vorteile gegenüber einer langen oder komplexen. Oft ist es sinnvoller den direkten Weg zu nehmen als ein komplexes Programm zu entwickeln, das mehr kann als es muss. Nicht zuletzt, weil Sie das Programm vielleicht Wochen später noch einmal benötigen und es Ihnen dann leichter fallen wird, nachvollziehen zu können, was Ihr Code im Detail macht, wenn Sie es kurz und einfach gehalten haben.

- **Verständlich programmieren**

  Grundsätzlich sollte Ihr Programm für jeden einfach lesbar sein. Versuchen Sie es (auch visuell) in seine Unterabschnitte zu gliedern und ihm eine Struktur zu geben. Rücken Sie, wenn Sie Schleifen oder Verzweigungen benutzen, immer ein, um deutlich zu machen, in welcher Zeile man sich in welchen Schleifen/Verzweigungen befindet. Auch ein außenstehender Programmierer sollte Ihren Code möglichst schnell verstehen können.

- **Kommentieren!**

  Wenn Struktur alleine nicht ausreicht, verwenden Sie Kommentare um zu erklären, was Ihr Code an welcher Stelle gerade macht. Die eigenen logisch erscheinenden Gedankengänge sind für Außenstehende und auch oft für Sie selbst einige Tage/Wochen später nicht immer offensichtlich und nachvollziehbar. Geben Sie Ihrem Programm einen Titel, versehen Sie die Unterabschnitte mit Überschriften und erklären Sie möglichst viele Stellen des Programms, alles mittels der Kommentar-Funktion, sodass auch Sie selbst einige Zeit nach Erstellen des Codes diesen rasch nachvollziehen können.

- **Sprechende Variablennamen**

  Man sollte bereits durch den Variablennamen erkennen, welche Daten sich dahinter verstecken. Dadurch können Sie die Lesbarkeit Ihres Programmes deutlich verbessern. Beispiel: Summe der Produktkosten = sumProdCosts *Achtung:* Verwenden Sie nie Funktionsnamen als Variablennamen! (z.B.„sum“). Welche Funktionen es im Detail gibt finden Sie in Kapitel 3.3

- **DRY – Don‘t Repeat Yourself**

  Mehrmals aufgerufene, identische Programmteile sollten in Funktionen ausgelagert werden. Dadurch wird Ihr Hauptprogramm übersichtlicher und Änderungen sind mit weniger Aufwand verbunden. Wie man Funktionen schreibt und sie in ein Programm/Skript einbindet, finden Sie in Kapitel 3.5

- **Üben, Üben, Üben!**

  Man benutzt in Matlab eine eigene Programmiersprache. Und wie es der Name schon sagt, muss man diese neue "Sprache" lernen wie eine Art Fremdsprache. Und das geht nur, wenn man sie auch anwendet. Ohne regelmäßiges Üben wird man schnell die gelernten Skills verlieren. Wenn man das Gelernte hingegen immer wieder anwendet, wird es sich bald so anfühlen, als "spräche" man mit mit dem Computer. Wie bei fast allem im Leben gilt: Übung macht den Meister.

1.8 Weitere Literatur und Links

Dieses Skript bietet einen Einstieg für Programmier-Anfänger in die Software Matlab. Wer sich über diese Basics hinaus weiter mit der Software auseinandersetzen möchte, dem seien hier noch diverse weiterführende Literatur und Links empfohlen. Vor allem die Online-Hilfe (Documentation) ist an dieser Stelle für schnelle Hilfestellung sehr zu empfehlen.

- Folien zum Tutorium: https://static.uni-graz.at/fileadmin/sowi-institute/sor/Dokumente/Downloads/MATLAB_Tutorium.pdf
- Jim Sizemore, *Matlab für Dummies*, John Wiley & Sons, 1. Auflage, 2016
- Stefan Wicki, *Die nicht zu kurze Kurzeinführung in MATLAB: Erste Schritte in MATLAB*, Books on Demand, 2. Auflage, 2015
- Ulrich Stein, *Programmieren mit MATLAB: Programmiersprache, Grafische Benutzeroberflächen, Anwendungen*, Hanser Verlag, 5. Auflage, 2015
- Wolfgang Schweizer, *MATLAB kompakt*, Oldenburg Verlag, 5. Auflage, 2013
- Matlab Online Documentation: *http://de.mathworks.com/help/matlab/?refresh=true*

2 Programmier-Vokabular

In diesem Kapitel sollen kurz einige Standard Begriffe, die in der SoftwareWelt immer wieder vorkommen, vorgestellt und erklärt werden. Vor allem bei Online-Recherchen oder in Gesprächen mit Software-Entwicklern ist es daher sehr nützlich, sich einige Begrifflichkeiten zu merken. Dadurch findet man schneller die Lösung, nach der man sucht.

- **Debugging**

  Bezeichnet die Fehlersuche und anschließende Fehlerbehebung in einem Softwareprogramm. Viele Entwicklungsumgebungen, darunter auch Matlab, haben einen eigenen Debugg-Modus, der die Fehlersuche erleichtert. Wie dieser im Detail bei Matlab funktioniert, finden Sie in Kapitel 3.10.

- **Programmiersprache**

  Jede Programmierumgebung "versteht" eine eigene Programmiersprache. So verwendet auch Matlab seine eigene Programmiersprache. Daneben gibt es viele andere, wie z.B.: C, C++, Python, Java, PHP. Die unterschiedliche Sprachen sind sich oft ähnlich, weshalb man sich beim Erlernen einer weiteren Programmiersprache oft leichter tut, wenn man eine einmal gut verstanden hat.

- **Pseudo-Code**

  Ein Pseudo-Code ist ein informeller Programmcode, der maschinell nicht verarbeitet werden kann und nur der Veranschaulichung der Programmstruktur dient. Es empfiehlt sich, bevor man den eigentlichen Code, den Source-Code, schreibt, einen Pseudo-Code zu erstellen, um beim späteren Programmieren Anhaltspunkte zu haben und nicht den Überblick zu verlieren.

- **Source-Code**

  Der Source-Code, zu Deutsch „Quelltext“, bezeichnet den in einer Programmiersprache geschriebenen Text eines Softwareprogrammes.

- **Syntax**

  Bedeutet soviel wie „Reihenfolge“ oder „Ordnung“. Damit Matlab Ihr Programm (Source-Code) versteht, müssen Sie seine Sprache verwenden. Das bedeutet zum Beispiel, dass nur standardisierte Befehle verwendet werden können, oder, dass Komma, Semikolon und Klammern an bestimmten Positionen zu setzen sind. Wird eine Regel verletzt, kann Matlab nicht verstehen, was Sie ihm sagen wollen und wird Ihnen eine Fehlermeldung ausgeben. An dieser Stelle sei ein weiteres Mal darauf hingewiesen, dass andere Programmiersprachen zwar eine andere Syntax verwenden, die Grundlagen der Softwareentwicklung jedoch gelten weiterhin!

3 Programmieren mit Matlab

In diesem Kapitel wird dargestellt, wie man mit Matlab programmiert. Sollten Sie dieses Kapitel verstanden haben, beherrschen Sie die Grundwerkzeuge, mit denen Sie wohl fast alle Ihnen jemals begegnenden Programmieraufgaben lösen können.

3.1 Variablen

Man muss Matlab nicht zwingend zum Programmieren verwenden. Matlab kann auch als „großer Taschenrechner“ benutzt werden. Dazu muss man lediglich Rechnungen (unter Berücksichtigung allgemeiner Rechenregeln wie „Punkt vor Strich“) in das Command Window (1.4) eingeben.

Figure 3.1: Rechnen im Command Window

Diese Rechenergebnisse können auch Variablen zugewiesen werden. Dazu verbindet man im Command Window die gewünschte Variable (z.B. „result“) und die Rechnung mit einem „=“. Nachdem die Eingabe mit [ENTER] bestätigt wurde, wird die Variable und ihr Inhalt (Ergebnis der Rechnung) im Command Window ausgegeben. Weiter wurde die Variable im Workspace angelegt, wo auch der Wert der Variable überprüft werden kann.

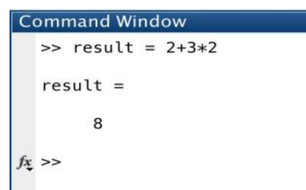

Figure 3.2: Variable anlegen

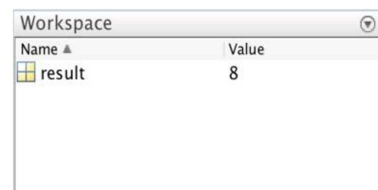

Figure 3.3: Variable in Workspace

Im Gegensatz zu anderen Programmiersprachen kann man in Matlab, ohne den Datentyp zu beachten, bestehende Variablen mit neuen Werten oder Texten überschreiben. Verwendet man also beispielsweise eine Variable zwei Mal hintereinander, so wird lediglich der letzte festgelegte Wert der Variable gespeichert und der alte überschrieben. Das mehrmalige Beschreiben der gleichen Variable innerhalb eines Programmdurchlaufs sollte jedoch möglichst vermieden werden, da dies dazu führen kann, dass man nicht mehr weiß, an welcher Stelle im Programm die Variable auf welchen Wert gesetzt wurde.

Hier empfiehlt es sich dann evtl. mit mehreren Variablen zu arbeiten. Variablen, die bereits im Workspace gespeichert wurden, können durch Eingabe des Variablennamens im Command Window und anschließender Bestätigung mittels [ENTER] wieder aufgerufen werden.

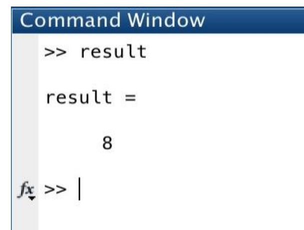

Figure 3.4: Aufrufen einer Variable im Command Window

Hinweis: Oft möchte man Variablen inkrementieren oder erweitern. Dies geschieht, indem man die Variable mit sich selbst beschreibt. Anbei zwei Beispiele. Figure 3.5 zeigt wie man beispielsweise einen Zähler hinauf zählen lassen kann. Figure 3.6 zeigt, wie man eine Liste (Vektor) um ein Element erweitern kann. Wie dies im Detail funktioniert, lernen wir in den Kapiteln 3.2 und 3.4.

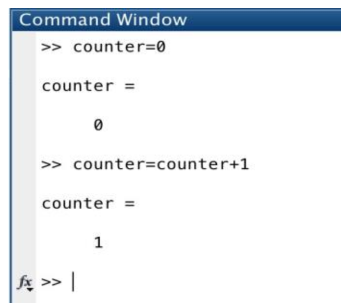

Figure 3.5: Variable mit sich selbst addieren

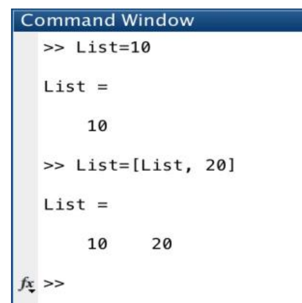

Figure 3.6: Erweitern von Listen

3.2 Spezielle Zeichen

Dieses Unterkapitel handelt von der korrekten Verwendung diverser Zeichen wie Punkt ("."), Komma (","), Strichpunkt (";") (Semikolon) und diversen Arten von Klammern. Weiters wird erstmals gezeigt, wie in einem Programm kommentiert wird.

3.2.1 Punkt, Komma, Strichpunkt

Trennzeichen bei Dezimalzahlen: Punkt

Komma und Strichpunkt (Semikolon) haben je nach Anwendung unterschiedliche Bedeutungen, im Allgemeinen dienen sie aber als Trennzeichen. Wichtig ist, dass man bei Dezimalzahlen, die im deutschen Sprachraum mit einem Komma getrennt werden (z.B. 3,14), den Punkt verwendet (z.B. 3.14).

Matlab kennt bei Dezimalzahlen (rationale, irrationale Zahlen) nur den Punkt als Trennzeichen und wird bei Verwendung des Kommas in diesem Zusammenhang eine Fehlermeldung ausgeben oder etwas vom Programmierer nicht Erwünschtes tun.

Trennzeichen zwischen Spalten: Komma

Komma und Strichpunkt benötigt man bei der Erstellung von Matrizen und Vektoren. Um bei der Definition einer Matrix die Elemente verschiedener Spalten zu trennen, verwendet man das Komma, vgl. Figure 3.7. Das Resultat, wenn man bei der Definition eines Vektors nur Kommas verwendet, ist ein *Zeilenvektor*.

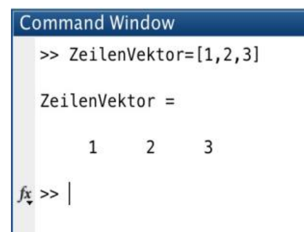

Figure 3.7: Trennung zwischen den Spalten einer Matrix mittels Komma

Trennzeichen zwischen Zeilen: Strichpunkt

Die Trennung von Elementen einer Matrix in unterschiedliche Zeilen erfolgt mittels Strichpunkt, vgl. Figure 3.8. Das Resultat, wenn man bei der Definition eines Vektors nur Strichpunkte verwendet, ist ein *Spaltenvektor*.

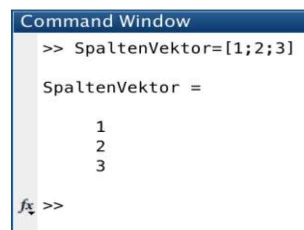

Figure 3.8: Trennung zwischen den Zeilen einer Matrix mittels Strichpunkt (Semikolon)

Hinweis: Wenn man eine Matrix manuell befüllt, ist zu beachten, dass jede Zeile gleich viele Einträge hat, da ansonsten die Dimensionen nicht übereinstimmen und Matlab eine Fehlermeldung ausgibt.

Trennzeichen nach Befehlen mit/ohne Ausgabe: Strichpunkt

Es gibt die Möglichkeit, verschiedene Befehle in einer Zeile einzugeben und sie gleichzeitig an Matlab zu übergeben. Will man, dass Matlab diese Befehle im Command Window ausgibt, so trennt man die Befehle mit einem Komma. Möchte man jedoch, dass Matlab die Befehle zwar abarbeitet, aber nicht ausgibt, trennt man diese mit einem Strichpunkt und setzt auch am Ende einen Strichpunkt. Dies verhindert die anschließende Darstellung des Ergebnisses, was vor allem bei größeren Programmen sinnvoll ist, um die Übersicht im Command Window nicht zu verlieren.

Die Befehle in Figure 3.9 ohne Strichpunkt werden ausgegeben, die Befehle in Figure 3.10 mit Strichpunkt (Semikolon) hingegen nicht.

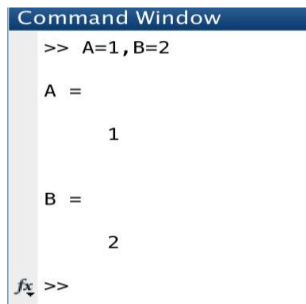

Figure 3.9: Ausgabe bei Komma

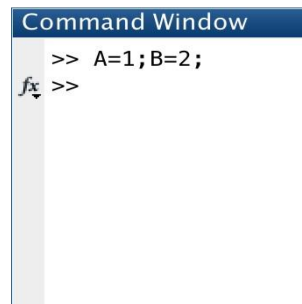

Figure 3.10: Keine Ausgabe bei Strichpunkt

3.2.2 Klammern

Runde Klammern

Runde Klammern werden als „Parenthesis“ bezeichnet. Sie werden bei Rechenoperationen, für den Zugriff auf Matrizen und für Funktionsaufrufe verwendet, vgl. Figure 3.11.

Eckige Klammern

Eckige Klammern werden in Matlab als „Brackets“ bezeichnet und dienen vor allem zur Definition von Vektoren und Matrizen, vgl Figure 3.12.

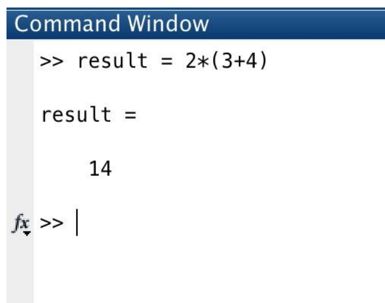

Figure 3.11: Benutzung runder Klammern

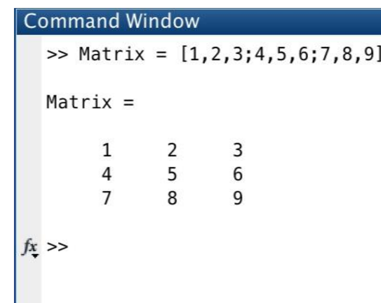

Figure 3.12: Benutzung eckiger Klammern

3.2.3 Kommentare

Wie bereits im Kapitel 1.7 erwähnt, sollten Kommentare zur besseren Lesbarkeit verwendet werden. Dies kann durch das Hinzufügen eines Prozent-Zeichens „%“ getan werden. Die folgenden Zeichen werden danach in dieser Zeile grün dargestellt und dadurch vom Programm nicht mehr berücksichtigt, vgl. Figure 3.13. Es empfiehlt es sich, gewisse Anmerkungen an jenen Stellen im Programm zu machen, an denen nicht offensichtlich ist, was getan wird.

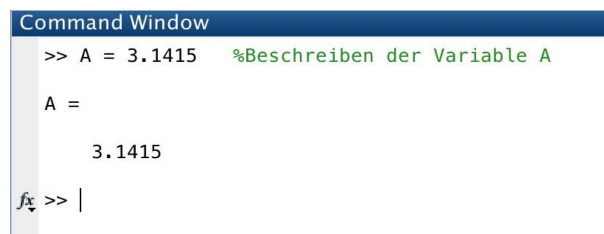

Figure 3.13: Kommentare werden von Matlab nicht berücksichtigt

3.3 Wichtige Matlab-Befehle/Funktionen

Matlab stellt dem Benutzer bereits eine Vielzahl an Funktionen bereit, die einem das Programmieren entscheidend erleichtern können. Im Folgenden werden einige – für die SBWL wichtige – Funktionen vorgestellt.

3.3.1 Zufallszahlen generieren - rand(), randn()

Die Funktion rand() gibt einem in Matlab eine stetig gleichverteilte Zufallszahl zwischen 0 und 1. Darüber hinaus können Sie auch eine normalverteilte Zufallszahl mit der Funktion randn() generieren. Optional kann auch eine n x m Matrix aus Zufallszahlen ausgegeben werden - rand(n,m) bzw. randn(n,m), vgl Figures 3.14 und 3.15.

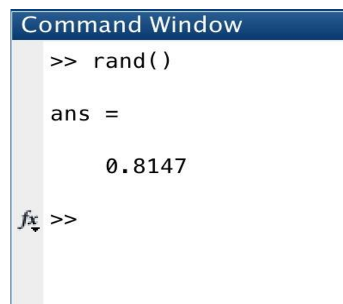

Figure 3.14: Erstellen einer Zufallszahl

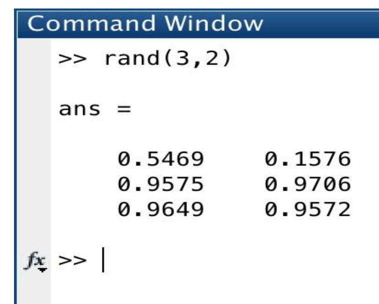

Figure 3.15: Erstellen einer Zufallsmatrix

3.3.2 Sortieren einer Zahlenfolge - sort()

Mit der Funktion sort() können Zahlenfolgen in auf- oder absteigender ('ascend', 'descend') Reihung sortiert werden, wie Figures 3.16 und 3.17 zeigen. Als Argument der Funktion muss der Vektor angegeben werden, der zu sortieren ist. Wird eine Matrix als Argument angegeben, so wird jede Spalte der Matrix sortiert.

```
Command Window
>> Vector

Vector =

     4     2     1     3     5

>> sort(Vector, 'ascend')

ans =

     1     2     3     4     5

fx >>
```

Figure 3.16: Aufsteigendes Sortieren einer Zahlenfolge

```
Command Window
>> Vector

Vector =

     4     2     1     3     5

>> sort(Vector, 'descend')

ans =

     5     4     3     2     1

fx >> |
```

Figure 3.17: Absteigendes Sortieren einer Zahlenfolge

3.3.3 Eingabe von Texten und Zahlen – input()

Mit Hilfe des Befehls input() kann eine Eingabe vom Anwender des Programmes eingefordert und einer Variable zugeordnet werden, vgl. Figure 3.18. Im Argument der Funktion schreibt man unter Anführungszeichen den Text, der bei der Aufforderung zur Eingabe ausgegeben werden soll.

```
Command Window
>> InputNumber=input('Bitte geben Sie eine Zahl eine: ')
Bitte geben Sie eine Zahl eine: 3.14

InputNumber =

    3.1400

fx >>
```

Figure 3.18: Abfrage von Texten und Zahlen

3.3.4 Ausgabe von Texten und Zahlen – disp(), fprintf()

Mit dem Befehl disp() kann auf einfache Weise ein Text oder eine Zahl ausgegeben werden. Wenn man einen Variable ausgeben lassen möchte, einfach die Variable ins Argument der Funktion schreiben. Wenn man einen Text ausgeben möchte, dann diesen unter Anführungszeichen ins Argument der Funktion schreiben. Soll ein Text, der eine Zahlen-Variable beinhaltet, ausgegeben werden, so kann man sich der Funktion fprintf() bedienen. Der Text wird unter Anführungszeichen eingegeben. Als Platzhalter im Text für die auszugebende Zahl innerhalb des Ausgabetextes kann für ganze Zahlen „%d“, für Dezimalzahlen (irrationale, rationale Zahlen) „%f“ verwendet werden. Die Zeichen „\n“ führen einen Zeilenumbruch aus und setzen den Cursor in die nächste Zeile. Nach Beendigung des Textes wird, mit einem Komma getrennt, die Variable angeführt, die an Stelle des Platzhalters eingefügt werden soll. Sollen mehrere Variablen eingefügt werden, werden diese der Reihe nach den Platzhaltern zugewiesen. Figure 3.19 zeigt die zwei Funktionen anhand eines Beispiels.

Hinweis: Für weitere Details zur Handhabung dieser Funktion wird an dieser Stelle jedoch auf die Matlab-Hilfe verwiesen.

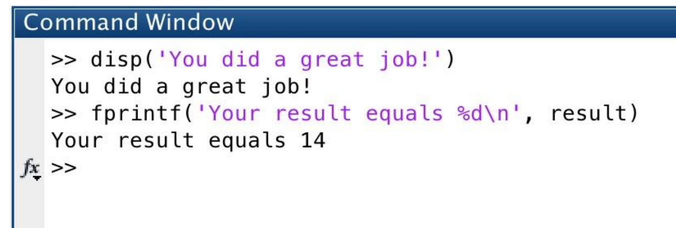

Figure 3.19: Die disp- und die fprintf-Funktion

3.3.5 Größe einer Matrix ermitteln – size()

Diese Funktion ermöglicht es, die Größe bzw. die Dimensionen einer Matrix zu bestimmen. Dabei können bereits beim Aufruf die beiden Outputs (die zwei Dimensionswerte einer *n*×*m*-Matrix) auf beliebige Variablen gespeichert werden. Dafür einfach die zwei Variablen in eckige Klammern stellen und gleich der sizeFunktion mit der Matrix als Argument setzen. Figure 3.20 zeigt dies anhand einer 2×3-Matrix.

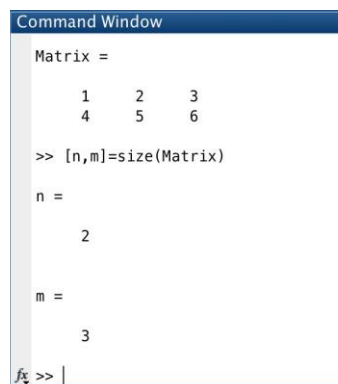

Figure 3.20: Bestimmung der Dimensionen einer Matrix

3.3.6 Suchen von Werten innerhalb einer Matrix – find()

Die Funktion find() ermöglicht es dem Anwender beispielsweise in Matrizen nach bestimmten Werten oder Wertbereichen zu suchen und diese im Format [Zeile, Spalte] auszugeben (ähnlich der size-Funktion, vgl. Kapitel 3.3.5). Die auf die neuen Variablen geschriebenen Werte entsprechen den Indizes der Position des gesuchten Objekts in der Matrix. Wichtig ist hier in der Syntax nicht auf das doppelte == zu vergessen, siehe Figure 3.21.

```
Command Window
Matrix =

     1     2     3
     4     5     6

>> [n,m]=find(Matrix==5)

n =

     2


m =

     2

>>
```

Figure 3.21: Auffinden der Position eines bestimmten Wertes in einer Matrix

3.3.7 Abfrage, ob Liste leer ist – isempty()

Mit der isempty-Funktion kann man auf einfachem Wege ermitteln, ob eine Variable noch Elemente beinhaltet oder ob sie leer ist. Wenn eine Liste/Matrix leer ist, gibt diese Funktion den Wert 1 zurück. Befinden sich noch Elemente in der Liste/Matrix, wird der Wert 0 ausgegeben, vgl. Figure 3.22 und Figure 3.23.

```
Command Window
>> List=[]

List =

     []

>> isempty(List)

ans =

     1

>>
```

Figure 3.22: isempty-Funktion = 1

```
Command Window
>> List=[1,2,3]

List =

     1     2     3

>> isempty(List)

ans =

     0

>>
```

Figure 3.23: isempty-Funktion = 0

Hinweis: Diese Funktion kann auch als if-Abfrage verwendet werden. Die if-Abfrage wird in Kapitel 3.6 genauer beschrieben, daher gehen wir an dieser Stelle nicht genauer darauf ein.

3.3.8 Ausgabe einer Grafik – plot()

Eine große Stärke von Matlab ist die sehr intuitive Möglichkeit zu plotten, was so viel bedeutet wie eine Funktion darzustellen bzw. ein Diagramm zu erstellen. Figures 3.24 und 3.25 zeigen die sehr einfache Vorgehensweise an einem Beispiel.

```
Command Window
X =

     1
     4
     9
    16
    25
    36
    49
    64
    81
   100

>> plot(X)
fx >>
```

Figure 3.24: Zu plottender Vektor

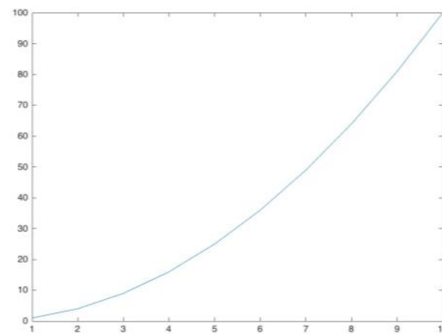

Figure 3.25: Dazugehöriger Plot

Hinweis: Da das Plotten eine der größten Stärken von Matlab ist, gibt es hier unzählige Möglichkeiten Funktionen, Kurven, Punkt, Diagramme, Graphen, etc. zu plotten. An dieser Stelle sei auf die Matlab-Hilfe verwiesen, in der man Informationen zu jeder Art von Plots erhält, z.B: Änderung der Linienfarbe, Beschriftung der Achsen, Diagrammbeschriftung, ...

3.3.9 Finden eines Maximums/Minimums - max()/min()

Mit der max-Funktion findet man sehr einfach das Maximum eines Vektors oder einer Liste an Zahlen. Die min-Funktion findet das jeweilige Minimum. Einfach den Vektor/die Liste als Argument der Funktion schreiben, siehe Figure 3.26. Schreibt man in das Argument der max- oder min-Funktion eine Matrix, so gibt Matlab einen Vektor mit den Maxima/Minima der jeweiligen Spalte aus.

```
Command Window
Vector =

     3     1     2     5     4

>> max(Vector)

ans =

     5

>> min(Vector)

ans =

     1

fx >>
```

Figure 3.26: Bestimmung des Maximums/Minimums eines Vektors

3.3.10 Berechnen der Summe - sum()

Die sum-Funktion berechnet die Summe der Elemente eines Vektors/einer Liste. Sie funktioniert ganz gleich wie die eben vorgestellten max- und minFunktionen. Ein Beispiel findet sich in Figure 3.27. Auch bei der sum-Funktion wird bei Anwendung der Funktion auf eine Matrix ein Vektor mit der Summe der Elemente jeder Spalte ausgegeben.

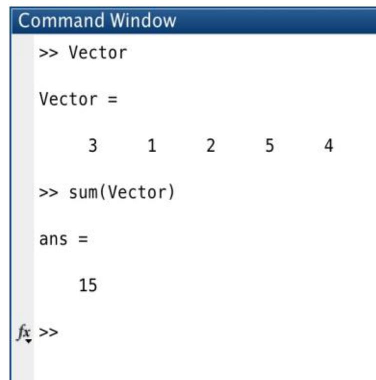

Figure 3.27: Berechnung der Summe der Elemente eines Vektors

3.3.11 Mittelwert und Standardabweichung - mean(), std()

Für simple statistische Auswertungen benötigt man häufig den Mittelwert und die Standardabweichung. Auch hierfür stellt Matlab vorgefertigte Funktionen bereit, die in ihrer Anwendung ganz gleich wie die gerade eben vorgestellten Funktionen arbeiten. Die mean-Funktion berechnet den Mittelwert der Elemente eines Vektors. Die std-Funktion ermittelt die Standardabweichung der Werte eines Vektors. Ein illustrierendes Beispiel ist in Figure 3.28 dargestellt. Wie auch bei den letzten erwähnten Funktionen gibt Matlab bei Anwendung einer der Funktionen auf eine Matrix einen Vektor mit den jeweiligen Funktionswerten für jede Spalte der Matrix aus.

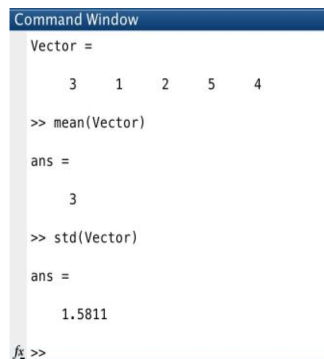

Figure 3.28: Berechnung des Mittelwertes und der Standardabweichung der Elemente eines Vektors

Hinweis: Bei der Standardabweichung handelt es sich um die Standardabweichung für eine Stichprobe. Demnach geht im Nenner der Formel der Faktor $N-1$ ein. Die exakte Formel lautet:

$$S = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}|x_i - \bar{x}|^2}$$

3.3.12 Zahl auf gerade/ungerade prüfen - mod()

Oft will man von einer Zahl wissen, ob sie gerade oder ungerade ist. Dafür gibt es keine eigene Funktion, man kann es aber sehr leicht mit der Funktion mod() abfragen. Die mod-Funktion berechnet den Modulo, also den (ganzzahligen) Rest bei einer Division durch eine Zahl. Bei der Division durch 2 hat man immer entweder Rest 0, wenn die Zahl, die dividiert wurde, gerade war, oder 1, wenn diese Zahl ungerade war. Demnach weiß man, das Ergebnis 0 = gerade, das Ergebnis 1 = ungerade. Bei dieser Funktion benötigt man zwei Argumente. Die Zahl, die man dividieren will, und die Zahl, durch die man dividieren will. Man gibt in dieser Reihenfolge die Zahlen/Variablen im Argument der Funktion durch ein Komma getrennt ein, wie es in Figure 3.29 durch ein Beispiel gezeigt wird. Zuerst wird die Zahl 4 durch 2 dividiert, was natürlich 0 Rest ergibt, danach 5 durch 2, was einen Rest von 1 ergibt.

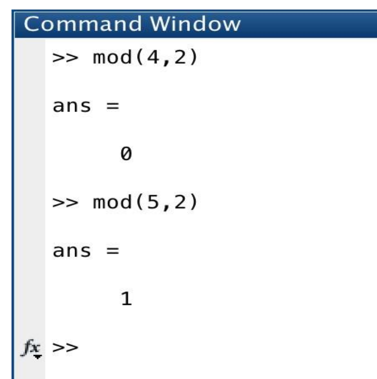

Figure 3.29: Prüfen ob eine Zahl gerade oder ungerade ist

3.3.13 Weitere nützliche Befehle/Funktionen

Man könnte die Liste an nützlichen Funktionen und Befehlen noch beliebig verlängern. Hier sollen exemplarisch noch ein paar interessante Befehle vorgestellt werden.

- **Runden**
Matlab kennt verschieden Formen des Rundens. Die Funktion round() rundet das eingegebene Element zum nächstgelegenen Integer-Wert, also zur nächsten ganzen Zahl. Die Funktion ceil() rundet den eingegeben Wert zum nächst größeren ganzzahligen Wert auf, die Funktion floor() rundet den Wert zum nächst kleineren ganzzahligen Wert ab. Ihre Syntax ist ganz gleich wie bei den bisher vorgestellten Funktionen.

- **Unendlich / ∞**
Manchmal benötigt man beim Programmieren große Zahlen, die sicher größer sind als alle anderen Zahlen, die man in seinem Programm verwendet. In Matlab kann man dafür den Wert "inf" angeben. Dieser ist sicherlich immer größer als jede Zahl, die noch als solche vom Computer dargestellt werden kann.

- **Not a Number**
Hin und wieder ergibt sich auch das Problem, dass man in seinem Programm Rechenoperationen einbaut, die mathematisch nicht zulässig sind oder per Definition kein Ergebnis besitzen (unbestimmt sind). Matlab kann auch, wenn solche Operationen durchgeführt werden sollen, das Programm weiter laufen lassen und gibt den aus diesen Operationen entstandenen Werten die Bezeichnung "NaN", welche für "Not a Number" steht (z.B.: $0/0$).

- **Wurzel ziehen**
In Matlab gibt es auch eine Funktion, die einem schnell die Quadratwurzel aus einer Zahl ziehen kann. Die Funktion sqrt() zieht die Quadratwurzel aus jeder Zahl. Auch komplexe Ergebnisse sind dabei für Matlab kein Problem $\left(\sqrt{-1} = i\right)$.

- **Winkelfunktionen**
Matlab kennt alle Winkelfunktionen wie Sinus, Cosinus und Tangens. Die dazugehörigen Funktionen lauten in selber Reihenfolge sin(), cos() und tan(). Auch die in der BWL seltener gebräuchlichen inversen Winkelfunktionen sind mit asin(), acos() und atan() verfügbar.

- **Pi / π**
Wenn man schon bei Winkeln ist, ist die in diesem Zusammenhang wohl wichtigste Zahl nicht zu übersehen. Die Zahl π kennt Matlab natürlich ebenfalls. Mit der Eingabe von "pi" weiß Matlab, dass Sie π (3.14159...) von Ihrem Programm benötigen.

3.4 Vektoren, Matrizen und Operatoren

In diesem Abschnitt werden wir lernen, wie man Vektoren und Matrizen manuell anlegt, was in vorherigen Kapiteln bereits teilweise behandelt wurde, wie man mittels spezieller Befehle Matrizen automatisch erstellen lassen kann, wie man auf einzelne Matrixelemente zugreift und wie man schließlich mit Matrizen rechnet.

3.4.1 Manuelles Anlegen von Matrizen

Für das manuelle Anlagen gibt es verschiedene Möglichkeiten, die im Folgenden vorgestellt werden.

Explizite Eingabe
Möchte man eine Matrix explizit bestimmen, sprich jedes Element direkt an die Matrix übergeben, so zeigt Figure 3.30 an einem Beispiel, wie dies getan werden kann. Wem das jetzt bekannt vorkommt, der irrt nicht, denn wir haben diese Art Matrizen oder Vektoren (sind im allgemeinen auch Matrizen, bei denen eine der zwei Matrix-Dimensionen gleich 1 ist, je nachdem, ob es sich um einen Zeilen- oder einen Spaltenvektor handelt) zu definieren bereits in Kapitel 3.1 kurz angeschnitten.

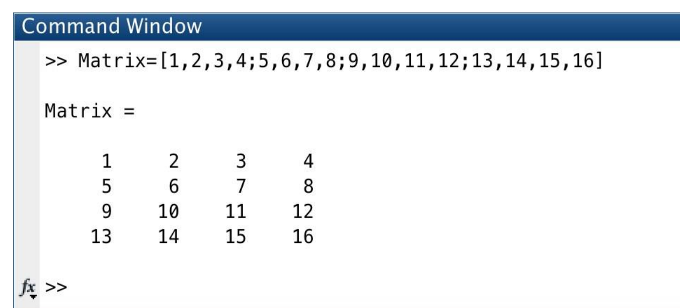

Figure 3.30: Explizite Definition einer Matrix

Die Elemente innerhalb einer Zeile können entweder durch Leerzeichen oder durch Kommas (wie in Figure 3.30 dargestellt) getrennt werden. Durch den Strichpunkt wird eine Zeile abgeschlossen und die darauf folgenden Elemente werden in die nächste Zeile eingetragen. Die gesamte explizite Eingabe muss durch eckige Klammern begrenzt werden.

Hinweis: An dieser Stelle sei ein weiteres Mal darauf hingewiesen, dass die Übersichtlichkeit bei dieser Art der Eingabe sehr schnell verloren gehen kann. Wichtig ist, dass die Abschnitte, die man durch Strichpunkte trennt, immer die gleiche Anzahl an Elementen besitzen, da ansonsten die Dimensionen der Matrix nicht korrekt ermittelt werden können und Matlab eine Fehlermeldung ausgibt. Um die Übersichtlichkeit zu gewährleisten könnte man z.B.: später im Programm bei der Matrix-Definition jede Zeile der Matrix in eine neue Zeile im Programm schreiben. Matlab versteht diese Notation.

Doppelpunkt-Notation

Die Doppelpunkt-Notation ist eine sehr gängige und praktische Anwendung in Matlab, die uns noch an verschiedenen Stellen begegnen wird. Sie kann auch zur Definition von Matrizen und Vektoren verwendet werden. Bei der Erstellung eines Vektors bedeutet der Doppelpunkt in diesem Zusammenhang frei übersetzt "bis". So kann ein Vektor mit Hilfe der Doppelpunkt-Notation, wie in Figure 3.31 dargestellt, erzeugt werden.

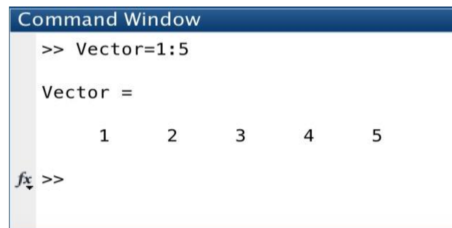

Figure 3.31: Definition eines Vetors mittels Doppelpunkt-Notation

Bei diesem Vorgehen bildet Matlab stets, wie man aus der Graphik gut sehen kann, einen Vektor mit Elementen zwischen den beiden übergebenen Zahlen (hier 1 und 5), wobei der Abstand zwischen zwei Elementen des Vektors immer „1“ beträgt. Möchte man den Abstand zwischen den Elementen des Vektors ändern, kann man dies über einen zusätzlichen Parameter vorgeben. Dieser wird zwischen den "Anfangs-" und "Endwert" des Vektors geschrieben und von beiden wiederum durch einen Doppelpunkt getrennt. Figure 3.32 zeigt dies anhand eines Beispiels. Hier wird ein Vektor mit Elementen von 0 beginnend in 5er Schritten bis 50 erstellt.

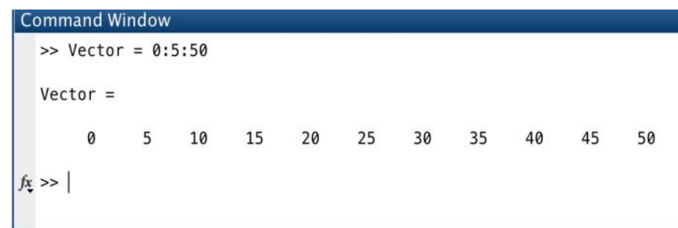

Figure 3.32: Definition eines Vetors mittels Doppelpunkt-Notation

3.4.2 Automatisches Anlegen von Matrizen mit Befehlen

Neben den bisher genannten Möglichkeiten bietet Matlab außerdem bereits vorgefertigte Befehle zur automatisierten Erstellung von Matrizen. Im Folgenden werden ein paar der wichtigsten und für die SBWL relevantesten dieser Befehle vorgestellt.

Einsmatrix - ones()
Eine Einsmatrix (nicht zu verwechseln mit der Ein*hei*tsmatrix) ist eine *n×m* Matrix mit beliebiger Dimension *n,m,* deren Elemente alle gleich 1 sind. Die Funktion ones() erstellt einem eine solche Matrix. Im Argument der Funktion sind die Dimensionen der Matrix zu übergeben und mit einem Komma zu trennen, vgl. Figure 3.33.

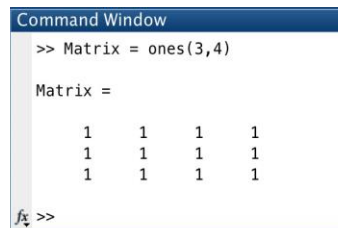

Figure 3.33: Erstellen einer 3×4-Einsmatrix

Nullmatrix - zeros()
Eine Nullmatrix ist eine Matrix, deren Elemente alle gleich 0 sind. Der Befehl zeros() erzeugt einem eine solche Matrix. Wiederum sind die gewünschten Dimensionen der Matrix als Argument der Funktion zu übergeben, vgl. Einsmatrix und Figure 3.34.

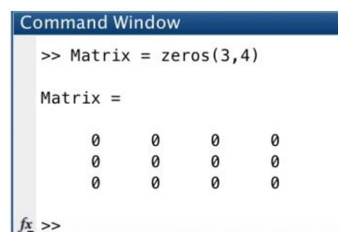

Figure 3.34: Erstellen einer 3 × 4-Nullmatrix

Einheitsmatrix - eye()
Die Einheitsmatrix ist eine Matrix, deren Hauptdiagonalelement alle gleich 1 und alle übrigen Elemente gleich 0 sind. Sie ist per Definition quadratisch. Darum reicht bei der Eingabe der Dimensionen meist eine Zahl, wie Figure 3.35 zeigt. Sollte man doch die Dimensionen der Matrix mit *n = 3, m = 4* wählen, so interpretiert das Matlab wie in Figure 3.35 dargestellt.

```
Command Window
>> Matrix=eye(4)

Matrix =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> Matrix=eye(3,4)

Matrix =

     1     0     0     0
     0     1     0     0
     0     0     1     0

fx >>
```

Figure 3.35: Erstellen einer Einheitsmatrix

Not a Number - nan()

Dieser Befehl erzeugt eine Matrix der Größe $n \times m$ mit NaN (= Not a Number). Wir haben diese Art von Werten bereits etwas früher in diesem Skript kennengelernt. NaN ist besonders dann hilfreich, wenn man sich noch nicht sicher ist, welche Elemente die Matrix beinhalten soll, diese aber dennoch initialisieren möchte (z.B. vor einer Schleife). Man muss allerdings beachten, dass NaN durch Rechenoperationen kombiniert mit Zahlen wieder NaN ergibt, also:

$NaN + 2 = NaN$
$NaN * 3 = NaN$
usw.

Man kann Elemente jedoch explizit mit Zugriff über deren Indizes auf Zahlenwerte setzen, mit denen Matlab in weiterer Folge wieder rechnen kann, vgl. Figure 3.36.

```
Command Window
>> Matrix=nan(3,4)

Matrix =

   NaN   NaN   NaN   NaN
   NaN   NaN   NaN   NaN
   NaN   NaN   NaN   NaN

>> Matrix(1,1)=2

Matrix =

     2   NaN   NaN   NaN
   NaN   NaN   NaN   NaN
   NaN   NaN   NaN   NaN

fx >>
```

Figure 3.36: Erstellen einer NaN-Matrix

Infinity-Matrix - inf()

Den Wert "inf" haben wir, wie auch den Wert "NaN" bereits etwas weiter oben im Skript kennen gelernt, und wie auch zu letzterem gibt es zu "inf" einen Matrixgenerierungs-Befehl. Die Funktion lautet inf(). Dieser Befehl erzeugt eine Matrix mit den Werten "inf" (unendlich, ∞) der Größe *n×m*. Es gelten für "inf" die selben Rechenregeln wie für "NaN"-Werte.

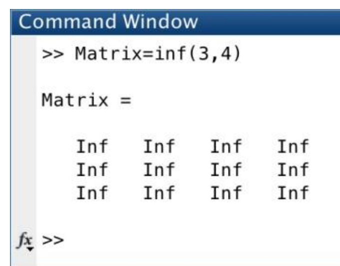

Figure 3.37: Erstellen einer Inf-Matrix

Zeilenvektor mit Abstand Eins - colon(a,b)

Mit dieser Funktion kann die Doppelpunkt-Notation umgangen werden. Die Funktion colon(a,b) erstellt einen Zeilenvektor mit Elementen von a beginnend bis b mit Abstand 1 zwischen allen Elementen, vgl. Figure 3.38.

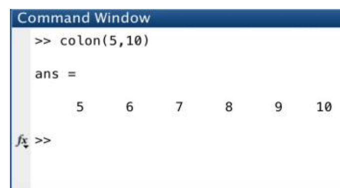

Figure 3.38: Erstellen eines Zeilenvektors mit der colon-Funktion

Zeilenvektor mit äquidistanten Werten erstellen – linspace(a,b,n)

Es kann vorkommen, dass man eventuell einen Vektor mit n Elementen von a bis b mit a ≤ b, die alle den selben Abstand voneinander haben sollen, benötigt. Dann ist der linspace-Befehl der richtige. Dieser Befehl erzeugt einen Zeilenvektor mit n äquidistanten Werten von a bis b, vgl. Figure 3.39.

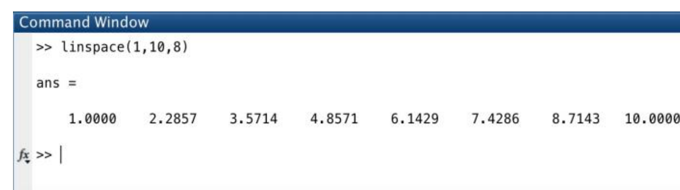

Figure 3.39: Erstellen eines Zeilenvektors mit der linspace-Funktion

Erstellung von Zufallszahlen-Matrizen - rand(), randn()
Es sei an dieser Stelle der Vollständigkeit wegen noch einmal erwähnt, dass auch Matrizen, deren Elemente Zufallszahlen sind, problemlos erstellt werden können. Dabei sei auf Figure 3.15 verwiesen, wo die Anwendung der randFunktion für Matrizen dargestellt ist.

3.4.3 Zugriff auf einzelne Matrixelemente - Indizierung

Manchmal will man auf gewisse Teile der Matrix (z.B. eine gewisse Zeile oder Spalte), oder sogar auf ein einzelnes Element der Matrix zugreifen. Auf einzelne Elemente einer Matrix kann man einfach zugreifen, indem man die jeweiligen Indizes des gesuchten Elementes in runden Klammern hinter der Matrixvariable angibt. Bei der Indizierung mit mehrfachen Indizes übergibt man Matlab zwei Indizes, die die Zeile und Spalte der Matrix identifizieren, in der sich das Element befindet, vgl. Figure 3.40. Der erste Index beschreibt immer die Zeile, der zweite Index die Spalte.

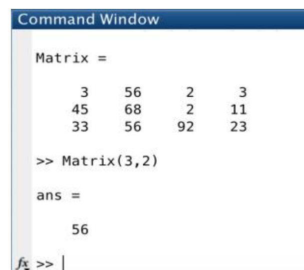

Figure 3.40: Zugriff auf ein Matrixelement mittels mehrfachen Indizes

3.4.4 Rechnen mit Matrizen - Verwendung von Operatoren

Matrizen nur zu erzeugen wird früher oder später langweilig. Deswegen wollen wir nun lernen, wie man mit ihnen rechnet. Im Folgenden werden die wichtigsten Rechenoperationen mit den dazugehörigen Operatoren vorgestellt.

Addition und Subtraktion
Für das Rechnen mit Matrizen gelten auch bei Matlab die allgemeinen Matrizen-Rechenregeln. So müssen z.B.: bei der gewöhnlichen Matrixaddition die Dimensionen der Matrizen übereinstimmen, ansonsten ist diese nicht durchführbar. Das bedeutet im mathematischen Sinne bei einer Addition der Matrizen A mit den Dimensionen $n_a \times m_a$ und B mit den Dimensionen $n_b \times m_b$ muss gelten: $n_a = n_b$ und $m_a = m_b$. Anderenfalls wird Matlab eine Fehlermeldung auswerfen, die wie folgt aussieht:

```
Command Window

A =

     1     2     3
     4     5     6
     7     8     9

>> C

C =

     1     2

>> A+C
Error using  +
Matrix dimensions must agree.

fx >>
```

Figure 3.42: Fehlermeldung nach Addition von Matrizen mit nicht übereinstimmenden Dimensionen

Sollte man jedoch Matrizen mit übereinstimmenden Dimensionen addieren, so geschieht dies elementweise, was so viel bedeutet, dass, wenn man mit den Matrizen *A* und *B* die Rechnung $A+B = C$ durchführen möchte, $a_{i,j} + b_{i,j} = c_{i,j}$ gilt. Figure 3.43 zeigt ein praktisches Beispiel einer Matrixaddition, in dem die Matrizen *A* und *B* zur Matrix *C* addiert werden.

```
Command Window
A =

     1     2     3
     4     5     6
     7     8     9

>> B

B =

     2     4     6
     8    10    12
    14    16    18

>> C=A+B

C =

     3     6     9
    12    15    18
    21    24    27

fx >>
```

Figure 3.43: Addition von zwei Matrizen

Die Subtraktion von zwei Matrizen folgt den gleichen Regeln wie die Addition, nur das hier elementweise subtrahiert und nicht addiert wird, also $a_{i,j} - b_{i,j} = c_{i,j}$. Auch die Übereinstimmung der Dimensionen muss in selber Form wie bei der Addition gegeben sein.

Multiplikation

Die gewöhnliche Matrixmultiplikation ist, im Gegensatz zur Addition und Subtraktion, nicht elementweise definiert. Möchte man Matrizen miteinander multiplizieren, so muss darauf geachtet werden, dass die inneren Dimensionen der beteiligten Matrizen übereinstimmen. Mathematisch ist sie wie folgt definiert:

$$A_{l \times m} \times B_{m \times n} = C_{l \times n}$$

Oder simpel gesprochen muss die erste Matrix demnach gleich viele Spalten haben, wie die zweite Matrix Zeilen hat. Die Dimensionen der daraus resultierenden Matrix entsprechen den beiden äußeren Dimensionen der multiplizierten Matrizen. Für alle Interessierten, die gewöhnliche Matrixmultiplikation ist für die einzelnen Elemente der resultierenden Matrix definiert mit:

$$c_{i,k} = \sum_{j=1}^{m} a_{i,j} * b_{j,k}$$

Sollten die inneren Dimensionen nicht übereinstimmen, so gibt Matlab eine Fehlermeldung ähnlich der in Figure 3.42. Ein praktisches Beispiel wie eine erfolgreiche Matrixmultiplikation in Matlab durchgeführt werden kann zeigt Figure 3.44.

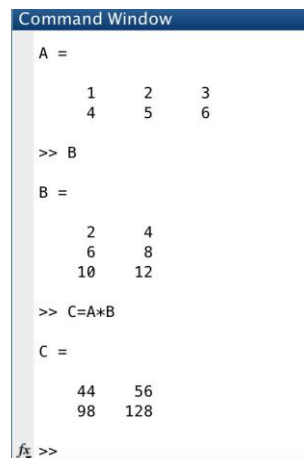

Figure 3.44: Gewöhnliche Multiplikation von zwei Matrizen

Matlab kennt jedoch neben der gewöhnlichen auch die elementweise Matrixmultiplikation. Dies bedeutet das einfach gilt:

$$c_{i,j} = a_{i,j} * b_{i,j}$$

Hierfür muss man einfach anstatt des bisher verwendeten Operators "*" nun den Operator ".*" setzen, in Matlab also einfach C = A.*B eingeben. Hier ist nun wieder darauf zu achten, dass die Dimensionen der beiden Matrizen komplett übereinstimmen (vgl. Addition und Subtraktion). Figure 3.45 zeigt ein praktisches Beispiel in Matlab. Man sieht hier sofort, dass immer Elemente mit jeweils gleichen Indizes miteinander multipliziert werden.

```
Command Window
A =

     1     2
     3     4

>> B

B =

     2     4
     6     8

>> C=A.*B

C =

     2     8
    18    32

fx >>
```

Figure 3.45: Elementweise Multiplikation von zwei Matrizen

Nachdem diese Operation elementweise geschieht, kann hier nun auch eine Division definiert werden, die wie folgt aussieht:

$$c_{i,j} = \frac{a_{i,j}}{b_{i,j}}$$

In Matlab verwendet man dafür einfach den Operator "./". Dabei handelt es sich jedoch nicht um eine Matrixdivision im eigentlichen Sinne. Figure 3.46 zeigt ein Beispiel. Man sieht schnell, dass jedes Element der Matrix *A* durch das Element der Matrix B mit selben Indizes dividiert wird.

```
Command Window
A =

     1     2
     3     4

>> B

B =

     2     4
     6     8

>> C=A./B

C =

    0.5000    0.5000
    0.5000    0.5000

fx >>
```

Figure 3.46: Elementweise Division von zwei Matrizen

Hinweis: Manche werden sich an dieser Stelle vielleicht fragen, wie die gewöhnliche Division von Matrizen in Matlab funktioniert. Die Antwort lautet: gar nicht. Die Division von Matrizen ist mathematisch nicht definiert. Matrizen zu dividieren ist also, unabhängig von Matlab, nicht möglich. Umgehen kann man das Problem mit der Umkehrung der Matrixmultiplikation, indem man die Gleichung mit dem Inversen der linken Matrix auf der linken Seite der Gleichung multipliziert, was dann in einer Gleichung mit der Form $B_{mxn} = A_{lxm}^{-1}\ x\ C_{lxn}$ resultieren würde. Dafür gibt es in Matlab auch Befehle. *A/B* steht in Matlab für die Operation $A \times B^{-1}$. Da man hier jedoch Acht auf die Invertierbarkeit von Matrizen geben muss und dies kein Mathematik-Skript ist, wird dieses Thema nicht weiter vertieft.

Determinante - det()
Die Determinante einer Matrix lässt sich sehr leicht über den Befehl det() bestimmen. Einfach die Matrix in das Argument der Funktion schreiben und die Ausgabe ist die Determinante dieser Matrix, vgl. Figure 3.47.

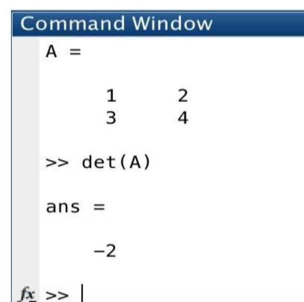

Figure 3.47: Berechnung der Determinanten einer Matrix

Inverse - inv()
Wie die Determinante lässt sich auch die Inverse einer Matrix sehr schnell und mit der Funktion inv() auf gleiche Weise ermitteln, vgl. Figure 3.48.

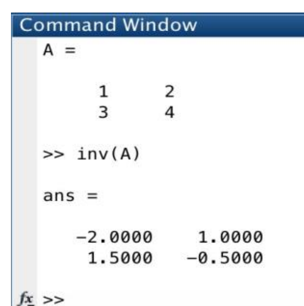

Figure 3.48: Berechnung der Inversen einer Matrix

Auswählen von Zeilen oder Spalten einer Matrix
Manchmal benötigt man genau eine Zeile oder eine Spalte einer Matrix. Hierfür gibt es in Matlab einen sehr einfachen Befehl, wie man genau eine Zeile oder eine Spalte aus einer Matrix auslesen kann. Dafür wird in gewissem Sinne wieder der weiter oben erwähnte Doppelpunkt-Operator verwendet. Wenn man zum Beispiel die erste Zeile einer Matrix *M* bekommen möchte, schreibt man in Matlab: *M*(1,:). Wir sagen Matlab damit, dass wir die Werte der *ersten* Zeile von *allen* Spalten haben wollen. Gewissermaßen steht der Doppelpunkt hier also für "alle". Demnach ist es auch weiter intuitiv, dass man für die Ausgabe der ersten Spalte einer Matrix *M* den Befehl *M*(:,1) eingibt. Wiederum sagt man, man möchte die Werte von *allen* Zeilen der *ersten* Spalte. Figure 3.49 zeigt an einem Beispiel wie man eine gewisse Zeile einer Matrix in Matlab auslesen lassen kann.

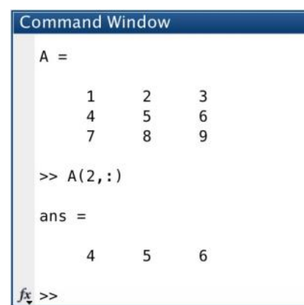

Figure 3.49: Ausgabe einer bestimmten Zeile einer Matrix

3.5 Skripte und Funktionen

Bis jetzt haben wir einige Befehle gelernt, diese jedoch stets nur im Command Window ausgeführt. Wir wollen nun erstmalig damit beginnen "richtige" Programme zu schreiben, sprich Matlab dazu zu bringen, beim Starten eines von uns geschriebenen Programms gewisse Befehle und Operationen hintereinander auszuführen. Dafür gibt es in Matlab die sogenannten Skripte.

3.5.1 Skripte

Wie gerade erwähnt gibt es neben der Möglichkeit Befehle im Command Window einzugeben, noch sogenannte Matlab-Skripte. Schreibt man ein Programm im Command Window, so ist dieses nach Schließen von Matlab nur mehr über die Command History erreichbar. Mit Hilfe eines Matlab-Skripts kann man das Programm jedoch als File speichern und nach Belieben wieder aufrufen und ausführen oder als Datei an andere versenden. Durch klicken auf "New" ganz links in der Befehlsleiste im Bereich "File" kann man ein solches neues Skript öffnen und später auch speichern, vgl. Figures 3.50 und 3.51.

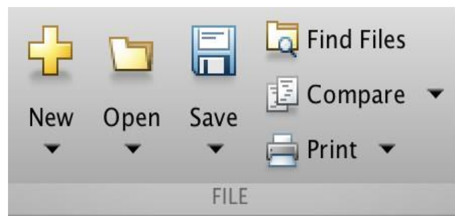

Figure 3.50: File-Command Leiste

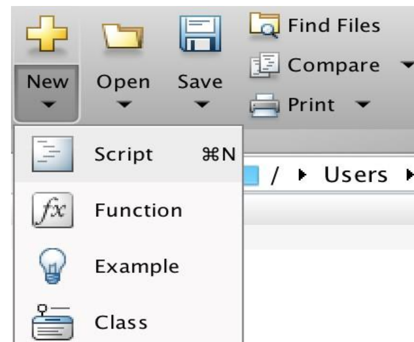

Figure 3.51: Aufrufen eines neuen Skripts

Innerhalb eines Skripts können Sie nun eine Abfolge von Matlab-Befehlen anlegen, wie Sie zuvor auch im Command Window ausgeführt wurden. Die Befehle werden nun erst nach klicken des Buttons „Run“, der in Figure 3.52 oben mittig in der Command-Leiste in Bereich "Run" zu sehen ist und durch ein grünes Dreieck gekennzeichnet ist, in der Regel hintereinander abgearbeitet, also zuerst Zeile 1, dann Zeile 2, etc. Ausnahmen zu dieser Regel sind u.a. Schleifen, die in einem späteren Kapitel näher behandelt werden. Ein Skript beginnt stets mit einem Header, in dem Titel, Autor, Datum und eine Erklärung des Programmes zu finden sein sollten. Darauf folgt das vorsorgliche Löschen aller Variablen, die sich noch im Speicher befinden und die Initialisierung (Definition) aller benötigten Variablen. Wird im Skript eine Variable verwendet, die noch nicht definiert wurde, gibt Matlab einen Fehler aus. Figure 3.52 zeigt wie der Anfang eines Skripts beispielsweise aussehen könnte.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %
% Beispiel 30                                                              %
% Expresskasse                                                             %
%                                                                          %
% Mochart Michael  | 1011466                                               %
%                                                                          %
% WS 2015                                                                  %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Clear all
clc

n=input('Wie viele Stundn sollen simuliert werden? ')*60;
Waitlist1X=[];
Waitlist2X=[];
WaitlistExp=[];
Waitlist1=[];
Waitlist2=[];
Customers=zeros(2,1);
arrive_n=3;
```

Figure 3.52: Beispielhafter Anfang eines Skriptes

Führt man ein Skript aus, in dem Variablen definiert wurden, so können diese Variablen im Anschluss auch im Command Window verwendet werden. Dies ist vor allem dann nützlich, wenn man mit den Ergebnissen des Skripts im Anschluss weiter arbeiten möchte.

3.5.2 Funktionen

Wenn man in seinem Programm bzw. seinem Skript einen gewissen Befehl oder eine gewisse Abfolge an Befehlen immer wieder benötigt, eignen sich Funktionen sehr gut dafür, diese Befehlsfolge auszulagern, sich somit viel Programmierarbeit zu ersparen und gleichzeitig für mehr Übersichtlichkeit im Programm zu sorgen.
Im Unterschied zu Skripten enthalten Funktionen eine sogenannte Deklarationszeile. Ihre Deklaration enthält alle Output-Variablen (= Variablen, die die Funktion ausgeben soll) und Input-Variablen (= Variablen, die der Funktion von außen übergeben werden müssen, damit die Output-Variablen berechnet werden können). Ein einfaches Beispiel, wie ein Skript und eine Funktion zusammenarbeiten wird in den Figures 3.53 und 3.54 dargestellt.

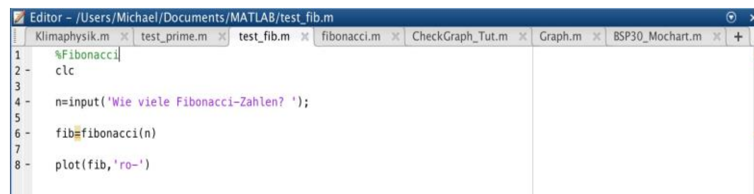

Figure 3.53: Aufrufen einer Funktion in einem Skript

Dieses sehr simple Programm berechnet einem n Fibonacci-Zahlen, je nachdem, wie viele man haben möchte. Der Eingabe-Parameter n wird danach an die Funktion fibonacci übergeben, siehe Zeile 6 im Code in Figure 3.53. Mit diesem Parameter rechnet die Funktion in Figure 3.54 die ersten n Fibonacci-Zahlen aus und gibt sie zurück.

Figure 3.54: Eine von einem Skript aufgerufene Funktion

Die Funktion hat als Output-Variable den Wert f (siehe erste Zeile) und als Input-Variable n. Zu Beginn wird für f ein Vektor mit n Stellen angelegt, der nach und nach mit den Fibonacci-Zahlen beschrieben wird. Wir sehen hier ab Zeile 5 eine weitere Matlab-Anwendung.

Dabei handelt es sich um eine forSchleife, die wir erst im nächsten Kapitel kennenlernen. Sie bewirkt, dass die Berechnung bis zur n-ten Fibonacci-Zahl durchgeführt wird. Wenn die Funktion ihre Berechnungen abgeschlossen hat wird der f an das Skript zurückgegeben und dort auf die Variable fib überschrieben. Diese wird anschließend vom Skript über das Command Window ausgegeben und geplottet, vgl Figures 3.55 und 3.56. Was hier auch offensichtlich wird ist, dass die OutputVariable in der Funktion nicht den selben Namen tragen muss wie die Variable, auf die sie im Skript überschrieben wird. Letzterer kann beliebig gewählt werden.

Hinweis: Das File sollte möglichst immer den gleichen Namen tragen, wie die Funktion selbst, um Probleme und Unübersichtlichkeiten zu vermeiden. Zu weiteren Infos bezüglich Funktionen werden die (Internet-)Hilfe und das Tutorium empfohlen. Sollte man nicht wissen, was Fibonacci-Zahlen sind, ist das für die SBWL nicht weiter tragisch. Interessierte finden im Internet genügend Material zu diesen spannenden Zahlen.

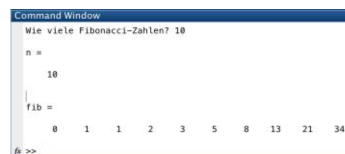

Figure 3.55: Ausgabe der ersten 10 Fibonacci-Zahlen

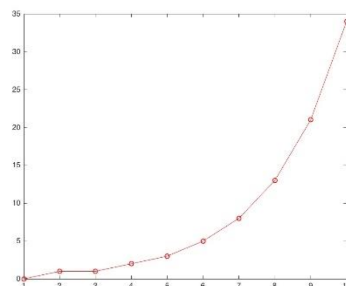

Figure 3.56: Plot der ersten 10 Fibonacci-Zahlen

3.6 Verzweigungen

Oft weiß man beim Starten eines Programmes noch nicht, in welche Richtung es an manchen Stellen des Programms gehen soll. z.B.: ist ein errechneter Wert unter oder über einer gewissen vorgegebenen Schranke. In solchen Fällen benötigt man sogenannte Verzweigungen. Die mit Abstand gängigste ist die if-Verzweigung.

3.6.1 If-Verzweigung

Die meisten Problemstellungen benötigen zu ihrer Lösung eine if-Verzweigung. Ähnlich wie die Funktion "WENN()“ in Excel können auch hier Werte abgefragt werden und danach in Abhängigkeit dieser Werte unterschiedliche Schritte vom Programm vorgenommen werden. Hierfür sind einige logische Operatoren notwendig, die in der folgenden Tabelle zu finden sind. In Figure 3.57 sieht man dann, wie die Syntax einer If-Verzweigung funktioniert.

Operator	Bedeutung
==	ist gleich
<	kleiner als
<=	kleiner gleich
>	größer als
>=	größer gleich
||	oder
&&	und

Table 3.1: Logische Operatoren in Matlab

```
1    %If-Verzweigung
2
3    random1 = rand();
4
5    if random1 > 0.5
6        disp('Zahl groeßer als 0.5')      %Anweisung, falls Zufallszahl größer als 0.5 ist
7    else
8        disp('Zahl kleiner als 0.5')      %Anweisung, falls Zufallszahl kleiner als 0.5 ist
9    end
```

Figure 3.57: Syntax einer if-Verzweigung

Nach dem *if* gibt man eine Bedingung ein. Wenn man nun in die nächste Zeile springt, rückt Matlab automatisch ein. Sollte es das nicht tun, ist das mittels Tabulator aus Gründen der Übersichtlichkeit selbst vorzunehmen. Ab jetzt schreibt man, was das Programm ausführen soll, wenn die Bedingung erfüllt ist. Mit der Eingabe von *else* kann man dem Programm noch mitteilen, was es zu tun hat, wenn die Bedingung nicht erfüllt ist. Die Verzweigung wird mit einem *end* beendet. Möchte man in einer if-Abfrage mehrere Zustände gleichzeitig abfragen, also den "&&" (und) oder "||" (oder) Operator verwenden, so sind diese in runde Klammern zu setzen. Weiters kann man mit der Abfrage *elseif* mehrere Verzweigungen mit verschiedenen Bedingungen erstellen, vgl Figure 3.58.

```
1     %If-Verzweigung
2
3 -   random1 = rand();
4 -   random2 = rand();
5
6 -   if (random1 > 0.5) && (random2 > 0.5)
7 -       disp('Beide Zahlen groesser als 0.5')       %Anweisung, falls beide Zufallszahlen größer als 0.5 sind
8 -   elseif (random1 < 0.5) && (random2 < 0.5)
9 -       disp('Beide Zahlen kleiner als 0.5')        %Anweisung, falls beide Zufallszahlen kleiner als 0.5 sind
10 -  else
11 -      disp('Eine Zahl groeßer, eine Zahl kleiner 0.5') %Anweisung, falls eine Zahl größer und eine kleiner 0.5 ist
12 -  end
```

Figure 3.58: If-Abfrage mit mehreren Verzweigungen

Hinweis: Bei mehreren if-Verzweigungen ist zu beachten, dass nur die erste Verzweigung, deren Bedingungen erfüllt sind, bearbeitet wird. Alle danach folgenden werden ignoriert.

3.7 Schleifen

Matlab wurde, wie der Name schon andeutet, vor allem für das Rechnen mit Matrizen konstruiert und funktioniert für diese schnell und effizient. Es gibt aber auch hier, wie in (fast) allen Programmiersprachen, das Konstrukt der Wiederholung. Sie ermöglicht die wiederholte Ausführung eines Anweisungsblocks, ohne Anweisungen wiederholt eintippen zu müssen. Dieses Konstrukt nennt sich "Schleifen". Sie sind äußerst praktisch und können auch in Verbindung mit Matrizen sehr gut eingesetzt werden. Allerdings sollte man (soweit möglich) zu verschachtelte Schleifen vermeiden, da Matlab Schleifen manchmal nicht effizient abarbeitet. Prinzipiell gibt es zwei Schleifentypen, die Zählschleife „for“ und die bedingte Schleife „while“. Diese beiden Schleifentypen werden im Folgenden näher beschrieben.

3.7.1 For-Schleife

In einer for-Schleife gibt man Matlab explizit an, wie oft ein gewisser Anweisungsblock wiederholt werden soll. Die Syntax dieser Schleife soll anhand eines Beispiels in Figures 3.60 und 3.61 erläutert werden.

```
1       %For-Schleife
2
3 -     clc
4
5 -     A = zeros(5,1);
6
7 -     for i=1:5
8 -         A(i)=2*i;
9 -     end
```

Figure 3.60: For-Schleife Code

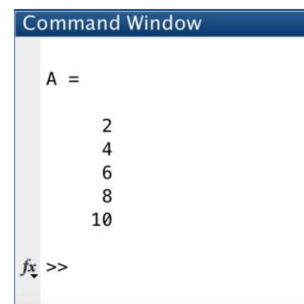

Figure 3.61: For-Schleife Ausgabe

Im linken Bild sehen wir den Code. Hier wird, nachdem ein nur mit Nullen befüllter Vektor *A* definiert wurde, die Schleife gestartet. Dies geschieht mit dem Wort *for*. In diesem Fall läuft die Schleife von 1 bis 5. Der Abstand zwischen zwei Schritten beträgt, sofern nichts anderes definiert wird, immer eins. Der Schleifenindex, hier *i*, nimmt also bei 1 beginnend in jedem Durchlauf den nächsten Wert an, in diesem Fall sind das die Werte 1, 2, 3, 4 bis 5. In einem Durchlauf bleibt der Schleifenindex *i* auf einem Wert und wird nach Abarbeitung des Anweisungsblocks um eins erhöht. Das geschieht so lange, bis der Endwert (hier 5) erreicht ist. Dann ist die Schleife beendet. Am Ende jeder Schleife ist ein *end* zu setzen! Die Ausgabe des neu konstruierten Vektors sieht man in Figure 3.61.

Man kann nun auch Schleifen erstellen, die nicht mit Schritten der Länge eins, sondern beliebiger Länge durchlaufen. Dafür ist einfach ein dritter Parameter einzuführen, der, durch Doppelpunkte getrennt, zwischen Start- und Endwert eingesetzt wird. Figures 3.62 und 3.63 zeigen die Syntax und das Ergebnis an einem Beispiel.

```
1      %For-Schleife
2
3 -    clc
4
5 -    A = zeros(5,1);
6
7 -    for i=1:2:5
8 -        A(i)=2*i;
9 -    end
```

Figure 3.62: For-Schleife Code

```
Command Window
A =

     2
     0
     6
     0
    10

fx >>
```

Figure 3.63: For-Schleife Ausgabe

Wie man an diesem Beispiel schnell sehen kann, beginnt der Schleifenindex *i* bei 1 zu laufen und soll dann in 2er-Schritten bis 5 gehen. Der Index durchläuft also die Werte 1, 3 und 5. Deshalb werden, wie man im zweiten Bild sehen kann, nur der 1., 3. und 5. Eintrag des Vektors verändert, die anderen bleiben unverändert auf 0.

Hinweis: Als Variablenname für den Schleifenindex kann grundsätzlich ein beliebiger Name verwendet werden. Meist verwendet man, je nachdem wie viele Schleifen man benötigt, Buchstaben ab *i*. Je nach Anwendung können aber sprechende Variablennamen für die Indizes besser geeignet sein. Hier ein paar Beispiele:

1. Schleife: *i*, 2. Schleife: *j*, 3. Schleife: *k*, usw.

oder

1. Schleife: *Jahr*, 2. Schleife: *Monat*, 3. Schleife: *Tag*, usw.

Man kann natürlich auch Schleifen ineinander verschachteln, wie in Figures 3.64 und 3.65 dargestellt wird:

```
1      %For-Schleife
2
3 -    clc
4
5 -    A = zeros(3,3);
6
7 -    for i=1:3
8 -        for j=1:3
9 -            A(i,j)=i+j;
10 -       end
11 -   end
```

Figure 3.64: Verschachtelte for-Schleifen

```
Command Window
A =

     2     3     4
     3     4     5
     4     5     6

fx >>
```

Figure 3.65: For-Schleife Ausgabe

Hier wird jedesmal, wenn der Schleifenindex der äußeren Schleife *i* um eins erhöht wird, die gesamte innere Schleife mit dem Index *j* abgearbeitet. So kann man leicht, wie in diesem Beispiel gezeigt, eine Matrix Element für Element abtasten und bearbeiten. In diesem Beispiel wird jedes Element mit der Summe seiner Indizes *i* und *j* überschrieben, z.B.: hat das Element *A*(2,2) den Wert 4, welcher 2+2 entspricht. Manchmal möchte man auch Schleifen an einer gewissen Stelle abbrechen lassen, bevor der eigentliche Endwert erreicht ist. Dafür eignet sich die *break* Funktion in Kombination mit eine if-Abfrage. So kann man in jedem Durchlauf abfragen, ist ein gewisses Ereignis eingetreten, wenn ja, dann *break*, also beende die Schleife, wenn nicht, mach weiter. Ein illustrierendes Beispiel findet sich in Figures 3.66 und 3.67.

```
1       %For-Schleife
2
3  -    clc
4
5  -    A = zeros(5,1);
6
7  -    for i=1:5
8  -        A(i)=2*i;
9  -        if A(i) > 5
10 -            break
11 -        end
12 -    end
```

Figure 3.66: Break-Funktion

```
Command Window

A =

     2
     4
     6
     0
     0

fx >>
```

Figure 3.67: Break Funktion Ausgabe

Hier wird nach jedem Überschreiben eines Vektorelements abgefragt, ob das Element größer als 5 ist. Nach dem dritten Durchlauf ist dies der Fall, da *V*(3) = 6. Damit wird die Schleife abgebrochen und das 4. und 5. Element des Vektors bleiben auf 0.

3.7.2 While-Schleife

Oft kann es auch sinnvoll sein, eine while-Schleife anstatt einer for-Schleife zu verwenden. Insbesondere dann, wenn die Aufgabenstellung so lautet, dass das Durchlaufen einer Schleife bis zum Eintreten eines bestimmten Zustandes gefordert wird, man aber vor dem Starten der Schleife nicht sicher weiß, wann dieser Zustand eintreten wird bzw. wie oft die Schleife überhaupt laufen soll. Die Syntax ist hier ähnlich der if-Verzweigung. Es werden auch die selben logischen Operatoren benötigt, wie wir sie schon bei den if-Verzweigungen in Kapitel 3.6 kennengelernt haben. Figures 3.68 und 3.69 zeigen anhand eines simplen Beispiels, wie die Syntax einer while-Schleife in Matlab funktioniert.

```
1      %While-Schleife
2
3 -    clc
4
5 -    RandSum=0;
6
7 -    while RandSum<10
8 -        RandSum = RandSum + rand();
9 -    end
10
11 -   disp(RandSum)
```

Figure 3.68: Syntax While-Schleife

```
Command Window

RandSum =

   10.1732

fx >>
```

Figure 3.69: While-Schleife Ausgabe

Hier wird eine Zahl mit 0 initialisiert. Danach wird der Schleife der Befehl gegeben zu laufen, so lange die Zahl kleiner als 10 ist. In der Schleife wird mit jedem Durchlauf eine stetig gleichverteilte Zufallszahl zwischen 0 und 1 zu der Zahl hinzu addiert. Wenn die Schleife beendet ist, soll die Zahl ausgegeben werden. Wie wir im rechten Bild sehen können, ist die Zahl bei 10.1732 das erste Mal über den Wert 10 gerutscht.

3.8 Weitere Datentypen

In diesem Unterkapitel sollen kurz weitere Datentypen vorgestellt werden, die alternativ zu Variablen oder gewöhnlichen Matrizen verwendet werden können.

3.8.1 Datentyp Struktur (struct)

Bisher wurden Daten immer in einfachen Variablen als Text oder Matrix angelegt. Darüber hinaus gibt es in Matlab auch die Möglichkeit, den Datentyp *struct* (Struktur) zu verwenden. Vor allem für die Ablage von komplexen Daten kann eine Struktur als übersichtlichere Alternative dienen, verglichen mit der Verwendung von vielen einzelnen Variablen. Im Unterschied zu einfachen Variablen besteht eine Struktur aus mehreren „Unter-Variablen", die auch „member" genannt werden. Anhand eines Beispiels wird gezeigt, in welchen Fällen die Verwendung von Strukturen von Vorteil sein kann. Figure 3.70 zeigt anhand eines Beispiels, wie dieser Datentyp funktioniert.

```
Command Window

>> Car = struct('Company', 'VW', 'Type', 'Golf', 'Year', 2005, 'Colour', 'black', 'Price', 5000)

Car =

    Company: 'VW'
       Type: 'Golf'
       Year: 2005
     Colour: 'black'
      Price: 5000

fx >>
```

Figure 3.70: Der Datentyp Struktur

Mit Hilfe der Funktion struct(memberName1, text/value, memberName2, text/value,...) wurde nun eine Struktur definiert und unter dem Namen „Car" angelegt. In diesem Fall wurden die member „Company", „Model", „Year", „Colour" und "Price"angelegt.

Die member-Namen müssen zwar im Textformat übergeben werden, ihr Inhalt kann jedoch auch eine Zahl, eine Matrix oder wiederum eine Struktur sein. Hat man nun nicht nur ein Fahrzeug, „Car“, sondern mehrere, sagen wir drei Fahrzeuge, kann ein sogenanntes Struktur-Array für diese drei Fahrzeuge definiert werden. Figure 3.71 stellt dar, wie so ein Struktur-Array aussehen kann.

```
Command Window
>> Car(1)=struct('Company', 'VW', 'Type', 'Golf', 'Year', 2005, 'Colour', 'black', 'Price', 5000)
Car(2)=struct('Company', 'BMW', 'Type', 'X6', 'Year', 2014, 'Colour', 'blue', 'Price', 96000)
Car(3)=struct('Company', 'Audi', 'Type', 'A5', 'Year', 2007, 'Colour', 'silver', 'Price', 43000)
```

Figure 3.71: Beispiel für ein Struktur-Array

Hinter der Variable „Car“ verstecken sich somit drei unterschiedliche Informationssets. Möchte man nun beispielsweise die Farbe des Fahrzeuges „2“ ausgeben, so muss zuerst der Variablenname angegeben werden „Car“, anschließend der Index „(2)“ in runden Klammern und zuletzt, verbunden mit einem Punkt „.“, der memberName „Colour“, vgl. Figure 3.72.

```
Command Window
>> Car(2).Colour

ans =

blue

>>
```

Figure 3.72: Auslesen einer Information aus einem Struktur-Array

3.8.2 Datentyp Block (cell)

Blöcke sind beliebig-dimensionale Gebilde, die Variablen beliebigen Datentyps wie Texte, Zahlen, Vektoren, Matrizen oder sogar Strukturen als Elemente beinhalten kann. Auch sie dienen zur einfachen und übersichtlichen Ablage von großen Datenmengen. Ein Block kann mit geschweiften Klammern „*{}*“ oder mit der Funktion cell(Element1, Element2,...) ähnlich wie eine Matrix erstellt werden. In den folgenden Abbildungen wird mit kleinen Beispielen illustriert, wie man mit diesem Datentyp umgehen kann.

```
Command Window
>> Block1={1991, 'Student1', [1,2,3,4], Car(1); 1992, 'Student2', [3,1,4], Car(2)}

Block1 =

    [1991]    'Student1'    [1x4 double]    [1x1 struct]
    [1992]    'Student2'    [1x3 double]    [1x1 struct]

>>
```

Figure 3.73: Beispiel 1 für den Umgang mit dem Datentyp *cell*

```
Command Window
>> Block2 = cell([4,4])

Block2 =

    []    []    []    []
    []    []    []    []
    []    []    []    []
    []    []    []    []

>> Block2{2,2} = 'Hier steht ein Text'

Block2 =

    []                        []    []    []
    []    'Hier steht ein Text'     []    []
    []                        []    []    []
    []                        []    []    []

>> Block2{3,3} = 42

Block2 =

    []                        []      []    []
    []    'Hier steht ein Text'       []    []
    []                        []    [42]    []
    []                        []      []    []

fx >>
```

Figure 3.74: Beispiel 2 für den Umgang mit dem Datentyp *cell*

3.9 Graphen plotten

Da man sich in der SBWL Operations Research sehr oft mit Graphen auseinander setzen muss (darf), soll dieses Unterkapitel kurz erläutern, wie man in Matlab einen Graph visualisieren, also plotten kann. Dafür gibt es in Matlab (neben noch einigen weiteren) die Funktion "gplot()". Um mit dieser Funktion Graphen zu visualisieren muss man an sie zwei Variablen (Matrizen) übergeben. Die eine ist, wie von vielen wahrscheinlich schon vermutet, eine Adjazenzmatrix, welche die Dimension $n \times n$ haben muss, also quadratisch sein muss, wobei n die Anzahl der Knoten ist. Diese reicht aus um viele Eigenschaften eines Graphen festzustellen. Unter anderem, welche Knoten mit welchen verbunden sind. Jedoch kann mit dieser Information der Graph noch immer auf beliebig viele Weisen visualisiert werden, da ich ja nur die Information besitze, zwischen welchen Knoten Kanten existieren, jedoch nicht wo im Raum (Ebene) diese überhaupt liegen. Deswegen muss noch ein zweiter Parameter an die gplot-Funktion übergeben werden, und zwar eine Koordinaten-Matrix. Diese Matrix ist stets eine $n \times 2$ Matrix (für zwei-dimensionale Graphen), wobei n die Anzahl der Knoten angibt. Jede Zeile beinhaltet die x- und die y-Koordinate eines Knoten. Mit diesen beiden Matrizen kann die Funktion gplot() den Graphen nun visualisieren, also plotten, was im Folgenden an einem kurzen Beispiel näher erklärt werden soll. Wir wollen einen simplen Graphen bestehend aus drei Knoten plotten, der durch folgende Adjazenzmatrix beschrieben wird:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Die Knoten sollen folgende Koordinaten besitzen:

Knoten 1: (1,1)

Knoten 2: (2,2)

Knoten 3: (3,1)

Dies ist für Matlab in die Form einer Koordinaten-Matrix zu bringen, die intuitiv wie folgt ausschaut:

$$C = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 1 \end{pmatrix}$$

```
Command Window
A =

     0     1     1
     1     0     1
     1     1     0

>> C

C =

     1     1
     2     2
     3     1

>> gplot(A,C,'-bo')
>>
```

Figure 3.75: Graph-Plotting Code

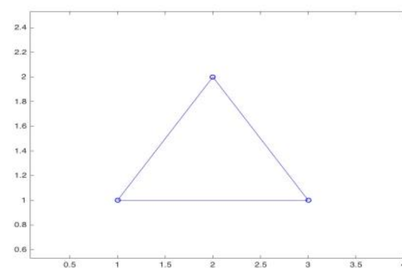

Figure 3.76: Visualisierter Graph

Wenn man diese zwei Matrizen nun an die Funktion gplot übergibt, wie es in Figure 3.75 gezeigt wird, wirft Matlab den in Figure 3.76 abgebildeten Graphen aus.

Hinweis: Es würde genügen, nur die zwei Parameter A und C an die Funktion zu übergeben. Die zusätzliche Eingabe ’-bo’ ist eine Anweisung, die bei vielen plot-Funktionen in Matlab gleich funktioniert und hier nur der besseren Visualisierung des Graphen dient. Es steht simpel gesprochen für "verbinde (-) die Knoten/Punkte mit einer blauen (b) Linie und markiere die Knoten mit Kreisen (o)". An dieser Stelle sei wiederum für weiterführende Informationen auf das Tutorium und die Internet-Hilfe verwiesen. Dieses Beispiel mag noch recht simpel anmuten, doch wenn man die Knoten und Kantenzahl ansteigen lässt, können die Graphen rasch sehr unübersichtlich wirken, wie die folgenden Abbildungen zeigen.

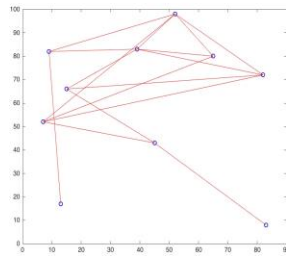

Figure 3.77: 10 Knoten, 16 Kanten

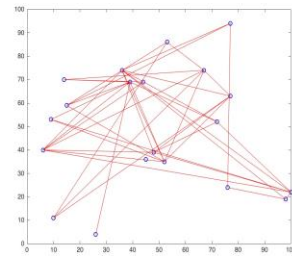

Figure 3.78: 20 Knoten, 41 Kanten

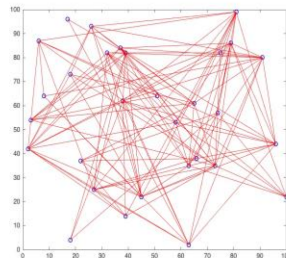

Figure 3.79: 30 Knoten, 122 Kanten

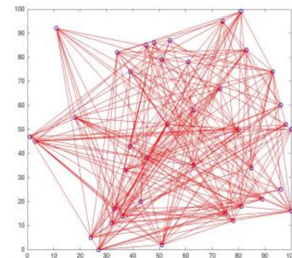

Figure 3.80: 40 Knoten, 242 Kanten

3.10 Debugging (Fehlersuche)

Debugging ist das Diagnostizieren und Auffinden von Fehlern in Computerprogrammen. Matlab hat ein sehr gutes und intuitives Debugging-System, das im Folgenden kurz vorgestellt werden soll. Während der Entwicklungsphase Ihres Programmes können zwei Arten von Problemen auftreten, in denen man allein durch mehrmalige Kontrolle ihres Codes nicht auf die Fehlerquelle stößt. Im besseren Fall wird Ihnen eine Fehlermeldung ausgegeben, in der sowohl die Position Ihres Fehlers steht (z.B. Zeile 7, Spalte 1), als auch die Art Ihres Fehlers (z.B. Sie haben das „end“ am Ende einer if-, switch-Verzweigung oder einer for-, while-Schleife vergessen), vgl. Figure 3.81.

```
Command Window
Error: File: untitled2.m Line: 7 Column: 1
At least one END is missing: the statement may begin here.

fx >>
```

Figure 3.81: Fehlermeldung von Matlab

Im zweiten, schlechteren Fall haben Sie Ihr Programm zwar nach bestem Gewissen und ohne Fehlermeldung fertiggestellt, jedoch gibt Ihnen das Programm eine nicht in sich schlüssige Ausgabe zurück. Nun haben Sie entweder den falschen Ansatz zur Lösung Ihrer Aufgabenstellung verwendet oder Sie haben einen „versteckten“ Fehler im Programm. Sehr anfällig für Fehler dieser Art sind dabei Schleifen und Verzweigungen, vor allem, wenn sie stark verschachtelt werden. Sollten Sie nun entweder auf eine Fehlermeldung stoßen, die sie nicht deuten können, oder einen logischen Fehler im Programm vermuten, so kann Ihnen die Debugging-Funktion von Matlab großen Nutzen erweisen. Zum Starten des Debugg-Modus muss zunächst im Programm ein sogenannter "Breakpoint" gesetzt werden. Dafür setzten Sie den Cursor im Programm an eine Stelle, die Ihrer Meinung nach noch vor der Stelle liegt, in der Sie Ihren Fehler vermuten. Wie Sie einen "Breakpoint" setzen sehen Sie in den Figures 3.82 und 3.83.

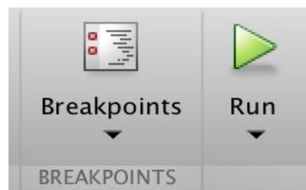

Figure 3.82: Der Breakpoint-Button

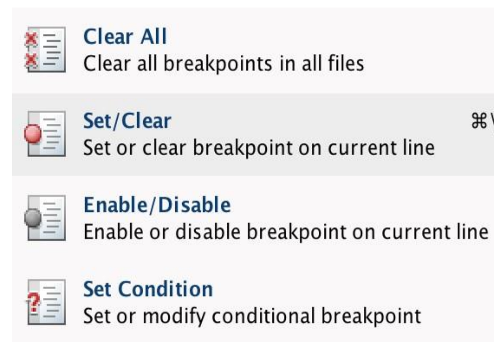

Figure 3.83: Das Breakpoint-Menü

Links neben dem "Run"-Button findet sich der "Breakpoint"-Button. Wenn man diesen anklickt öffnet sich das "Breakpoint"-Menü, zu sehen in Figure 3.83. Hier können Sie einige Befehle die Breakpoints betreffend ausführen, unter anderem den zweiten von oben "Set". Wenn Sie diesen Befehl anklicken, wird ein Breakpoint an den Anfang der Zeile im Programm gesetzt, in der der Cursor im Moment steht. Dies wird danach durch einen roten Punkt neben der Zeilenzahl angezeigt, vgl. Figure 3.84.

```
1      %While-Schleife
2
3 -    clc
4
5 -    RandSum=0;
6
7 ●    while RandSum<10
8 -        RandSum = RandSum + rand();
9
10 -   end
11 -   disp(RandSum)
```

Figure 3.84: Breakpoint im Programm

```
1      %While-Schleife
2
3 -    clc
4
5 -    RandSum=0;
6
7 ●➔   while RandSum<10
8 -        RandSum = RandSum + rand();
9
10 -   end
11 -   disp(RandSum)
```

Figure 3.85: Debugg-Modus

Wenn Sie dann das Programm starten, läuft es bis zu dem gesetzten Breakpoint durch. Danach hält es an und deutet das durch einen grünen Pfeil am linken Rand des Editors neben der Zeilenzahl, wie man in Figure 3.85 sehen kann. Jetzt befinden Sie sich im Debugg-Modus. In der oberen Befehlsleiste öffnet sich nun eine neue Auswahl an Buttons, mit denen man sich im Debugg-Modus "bewegen" kann, siehe Figure 3.86.

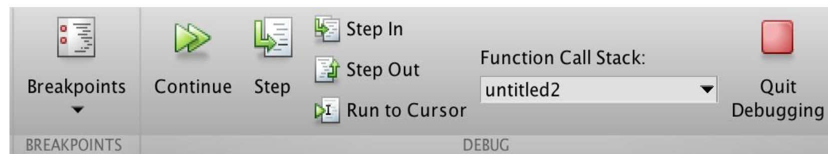

Figure 3.86: Das Debugg-Menü der Befehlsleiste

Mit dem Button "Step" können Sie sich nun Schritt für Schritt durch das Programm klicken und in jedem Schritt überprüfen, wo der Fehler liegen könnte. Sollten Sie im Debugg-Modus an eine Funktion kommen, können sie mit dem Button "Step In" in diese hineinsteigen und nachvollziehen, was dort passiert. Sollten Sie den Debugg-Modus beenden wollen, weil Sie zum Beispiel den Fehler gefunden haben, klicken Sie den "Quit Debugging"-Button. Der von Ihnen gesetzte Breakpoint befindet sich auch nach Beendigung des Debugg-Modus noch im Programm und wird dazu führen, dass Sie bei erneutem Starten des Programms wieder in den Debugg-Modus kommen. Sollten Sie den Breakpoint nicht mehr benötigen, so können Sie diesen löschen, indem Sie ihn entweder einfach anklicken, woraufhin dieser verschwindet, oder im "Breakpoints" Menü aus Figure 3.83 die ersten oder dritten Auswahl bestätigen und damit alle oder nur den ausgewählten Breakpoint wieder entfernen.

4 Übungs-Beispiele

1. Beispiel - Hello World

Schreiben Sie eine Abwandlung des Programms "Hello World". Der Benutzer soll auswählen können, ob er ein Freund, ein Feind oder ein Unbekannter ist. Je nachdem was der Benutzer auswählt, soll das Programm einen unterschiedlichen Gruß ausgeben.

2. Beispiel - Funktion

Schreiben Sie ein Programm, das auf eine einfache Funktion zugreift. Das Programm soll den Benutzer auffordern zwei Zahlen einzugeben. Diese zwei Zahlen übergibt das Programm an eine Funktion, welche die Summe dieser Zahlen bildet und diese zurück gibt. Das Programm soll die Summe inklusive Text ausgeben.

3. Beispiel - Funktion

Der Anwender wird zur Eingabe von zwei Zahlen aufgefordert. Anschließend soll folgendes im Command Window ausgegeben werden: Vier Werte (Summe, Differenz, Produkt, Quotient) als Spaltenvektor in absteigender Reihenfolge, die niedrigste Zahl dieser 4 Werte, der Text „ungerade“, falls die höchste Zahl ungerade ist und die die Aussage, ob die größte Zahl größer als 10 ist.

4. Beispiel - Graph

Schreiben Sie einen Algorithmus, der eine Inzidenz- zu einer Adjazenzmatrix umwandelt.
Zusatz: Schreiben Sie zusätzlich ein Programm, das eine Adjazenz- in eine Inzidenzmatrix überführt, vereinigen Sie diese beiden Programme und lassen Sie den Benutzer am Anfang auswählen, welche Matrixumwandlung er durchführen möchte.

5. Beispiel - Algorithmus von Prim

Programmieren Sie den Algorithmus von Prim mit entsprechender Textausgabe.

6. Beispiel - Algorithmus von Moore

Programmieren Sie den Algorithmus von Moore mit entsprechender Textausgabe.

7. Beispiel - Kreissuche

Programmieren Sie den Algorithmus, der einen beliebigen Graphen überprüft, ob dieser einen Kreis besitzt. Das alles wiederum mit entsprechender Textausgabe.

8. Beispiel - EDD

Programmieren Sie den EDD-Algorithmus mit entsprechender Textausgabe.

9. Beispiel - Simulation

Simulieren Sie 2 Strategien für den Umgang mit Motorenausfällen einer Maschine und entscheiden Sie anschließend, welche Strategie günstiger ist. Es handelt sich um eine Maschine mit 2 Motoren, Ausfallwahrscheinlichkeit der Motoren siehe Tabelle.
Motorreparatur: 50 GE, Motorwartung: 25 GE
Strategie 1: Reparatur bei Motorausfall
Strategie 2: Reparatur bei Motorausfall und Wartung des zweiten Motors
Simulationszeit: 50 Tage simulieren

Ausfallwahrscheinlichkeiten (abhängig von Tagen seit letzter Reparatur/Wartung):

Tage seit letzter Reparatur/Wartung	1	2	3	4	5	6	>6
Ausfallswahrscheinlichkeit	0.05	0.15	0.2	0.3	0.2	0.1	0

10. Beispiel - Branch and Bound

Programmieren Sie einen beliebigen Branch and Bound Algorithmus (anspruchsvoll).