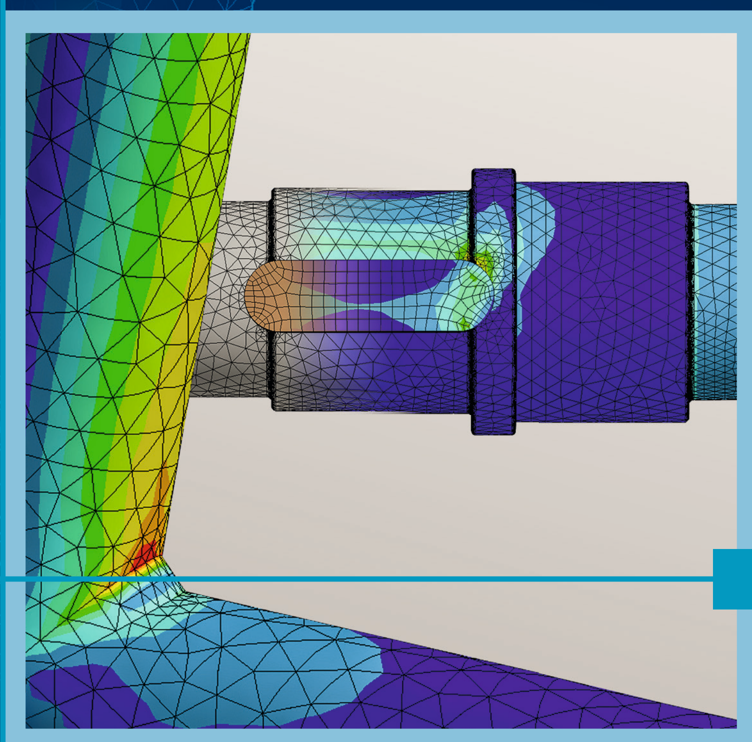
Jörg Frochte

Finite-Elemente-Methode

Eine praxisbezogene Einführung mit GNU Octave/MATLAB



2., aktualisierte und erweiterte Auflage

HANSER

Frochte

Finite-Elemente-Methode

Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial. Geben Sie dazu einfach diesen Code ein:

plus-zp69e-kel43

plus.hanser-fachbuch.de

Bleiben Sie auf dem Laufenden!

Hanser Newsletter informieren Sie regelmäßig über neue Bücher und Termine aus den verschiedenen Bereichen der Technik. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter

www.hanser-fachbuch.de/newsletter

Jörg Frochte

Finite-Elemente-Methode

Eine praxisbezogene Einführung mit GNU Octave/MATLAB

2., aktualisierte und erweiterte Auflage

HANSER

Autor:

Prof. Dr. rer. nat. Jörg Frochte
Hochschule Bochum
Arbeitsgruppe Angewandte Informatik und Mathematik

Alle in diesem Buch enthaltenen Informationen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt geprüft und getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor(en, Herausgeber) und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Weise aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.
Ebenso wenig übernehmen Autor(en, Herausgeber) und Verlag die Gewähr dafür, dass die beschriebenen Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

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.

Dieses Werk ist urheberrechtlich geschützt.
Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, sind vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2021 Carl Hanser Verlag München
Internet: www.hanser-fachbuch.de

Lektorat: Dipl.-Ing. Natalia Silakova-Herzberg
Herstellung: Anne Kurth
Covergestaltung: Max Kostopoulos
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Titelbild: © stock.adobe.com/Leandro
Satz: Jörg Frochte
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN 978-3-446-46915-0
E-Book-ISBN 978-3-446-46967-9

Inhalt

Einleitung 9

1 GNU Octave und MATLAB in a Nutshell 12
1.1 GNU Octave und MATLAB 12
1.2 Arbeiten mit Matrizen und Vektoren 14
1.3 Skripte und Funktionen schreiben 19
1.4 Elementare Kontrollstrukturen und Vektorisierung 23
1.5 Logische Ausdrücke, Zugriffe und Suchen 27
1.6 Plotten und Visualisieren 29
1.7 Daten importieren und exportieren 35

2 Motivation, Modellbildung und Anwendungsbeispiele 37
2.1 Die Wärmeleitungsgleichung 39
2.2 Elektro- und Magnetostatik 48
2.3 Transportphänomene mit Konvektion und Stoffabbau 54
2.4 Fishers-Gleichung: Populationsmodell mit beschränktem Wachstum ... 57
2.5 Klassifikation von partiellen Differentialgleichungen 58

3 Finite Elemente in 1D 61
3.1 Funktionen approximieren und numerisch integrieren 61
3.2 Variationsformulierung elliptischer Randwertprobleme 70
3.3 Ritz-Galerkin-Verfahren für elliptische Randwertprobleme . 82
3.4 Implementierung in 1D mit linearen Elementen 86
3.5 Elemente höherer Ordnung 99
3.6 Praxisbeispiel: Wärmeleitung in einem homogenen Stab 110

4 Finite Elemente in 2D 114
4.1 Variationsformulierung und Galerkin-Verfahren 114
4.2 Assemblierung und Implementierung 124
4.3 Ausblick auf hierarchische Basen, Elemente höherer Ordnung und isoparametrische Elemente ... 145

4.4 Fehlerabschätzungen und Konvergenzverhalten 150
4.5 Kondition, iterative Löser und Vorkonditionierung 157
4.6 Praxisbeispiel: Heizen mit offener Tür 166

5 Gemischte Randwerte und Gitterdatenstrukturen 171
5.1 Gmsh als Gittergenerator 172
5.2 Gitter-Datenaufbereitung und -struktur 178
5.3 Implementierung von gemischten Randwert-Problemen 191

6 Fehlerschätzer und Gitteranpassungen 202
6.1 Gradientenrekonstruktion und Z^2-Fehlerindikator 203
6.2 Algorithmus zur Gitterverfeinerung 210
6.3 Ausblick: weitere Fehlerschätzer und Fehlerindikatoren 222
6.4 Praxisbeispiel: E-Feld um Kondensatorplatten 225

7 BDF-Verfahren für zeitabhängige Modelle 230
7.1 Vertikale Linienmethode 230
7.2 Steife Probleme und BDF-Mehrschrittverfahren 234
7.3 Fehlerabschätzung für parabolische Differentialgleichungen 240
7.4 Algorithmische Umsetzung und Implementierung 241
7.5 Adaptivität in der Zeit und Schrittweitensteuerung 249
7.6 Praxisbeispiel: FEM-Modell als Strecke eines Regelkreises 258

8 Konvektionsdominierte Gleichungen 264
8.1 Stromliniendiffusion 265
8.2 Assemblierung der zusätzlichen Terme 270
8.3 Numerische Experimente zur Konvergenz und Stabilität 277
8.4 Praxisbeispiel: Schadstofftransport im Wasser 285

9 Nichtlineare Modelle 293
9.1 Ansatz über Fixpunkt- bzw. Picard-Iteration 294
9.2 Praxisbeispiel 1: Populationsmodell mittels Fishers-Gleichung 298
9.3 Praxisbeispiel 2: Magnetostatik mit nichtlinearer Permeabilität 304

10 Navier-Stokes-Gleichungen und Projektionsverfahren 316
10.1 Navier-Stokes-Gleichungen in der Strömungsmechanik 316
10.2 Stokes-Gleichung 321
10.3 Stattelpunkt-Probleme und die Inf-Sup-Bedingung 322

10.4 Das Taylor-Hood-Element ... 325
10.5 Implizites Projektionsverfahren nach Chorin ... 325
10.6 Driven-Cavity-Problem und Beispielimplementierung ... 327
10.7 Fluss um einen Zylinder als Übungsproblem ... 337

11 Ausblicke auf moderne Lösungsverfahren ... 342
11.1 Mehrgitterverfahren ... 342
11.2 Domain-Decomposition und Parallelisierung ... 344

Literatur ... 353

Index ... 359

Einleitung

„Eine Unterweisung mag äußerst tiefgründig sein, aber wenn sie für eine bestimmte Person nicht geeignet ist oder ihr nicht entspricht, was nützt sie dann?“

Aus „Die vier edlen Wahrheiten“ von Tendzin Gyatsho (Dalai Lama)

Ich bin vor Jahren auf dieses Zitat gestoßen und seitdem versuche ich mich immer danach zu richten, wenn ich etwas vermitteln oder erklären möchte. Man könnte es auch volkstümlicher ausdrücken und davon sprechen, dass viele Wege nach Rom führen. Bei der Methode der finiten Elemente (FEM) gibt es zwei sehr gut ausgebaute Wege – fast schon Schnellstraßen: Einmal einen Weg, den ich in wenigen Sätzen abhandeln möchte. Dieser besteht quasi aus Handbüchern, wie man eine spezielle FEM-Software, z. B. für lineare Elastizität, bedient und was zu beachten ist. Dieser Ansatz mag als Handbuch für eine Software taugen, die man i. d. R. auch teuer bezahlt hat und bei der man Anrecht auf eine ordentliche Dokumentation haben sollte. Ansonsten ist es jedoch nach meiner Ansicht ein Weg ins Nirgendwo, wenn man solch eine Methode wie die Finite-Elemente-Methode kennenlernen möchte. Das Ziel – besonders, aber nicht nur in einer akademischen Ausbildung – sollte es immer sein, sich Wissen anzueignen, das nicht nach einem Software-Update großflächig entwertet wird. Ein anderer gut ausgebauter Weg ist geprägt von hoher formaler Eleganz und dem Dreiklang aus Definition, Satz und Beweis. Was jedoch für die einen – meist Mathematiker oder Menschen, die ähnlich denken und lernen – eine Schnellstraße ist, ist für andere eine steinige Route, die oft ohne echten Erkenntniszuwachs endet. Damit haben wir einen für alle untauglichen Weg und einen, der für einige ein perfekt ausgebauter Initiationsritus ist und für andere zu oft mit einem frustrierten Abbruch endet. Ich hoffe, Ihnen mit diesem Buch einen dritten Weg zur FEM besser auszubauen, der sich stärker auf Algorithmen und das Ausprobieren dieser Algorithmen am Computer stützt.

Mich selbst hat die Computersimulation schon immer sehr fasziniert und wenn es um kontinuierliche Fragestellungen aus Natur- und Ingenieurwissenschaft geht, kommt man irgendwann auf partielle Differentialgleichungen, egal ob es Maxwell-Gleichungen, Wärmeleitung, Elastizität, Populationsdynamik oder Strömungsberechnungen sind. Damit hat man bzgl. der Anwender gerade die Gebiete der Elektrotechnik, des Maschinenbaus, der Physik und der Biologie gestreift. Da alles nicht ohne Algorithmen und Software geht, sitzt die Informatik – besonders die Ingenieurinformatik – noch halb mit im Boot.

All diesen Anwendern möchte ich mit diesem Buch einen weiteren Weg zur Simulation partieller Differentialgleichungen mit der Finite-Elemente-Methode anbieten. Der Ansatz besteht darin, den Dreiklang aus Definition, Satz und Beweis gegen einen eher algorithmischen Zugang zu tauschen. In den meisten Büchern steht die Theorie im Vordergrund und die Implementierung bildet ein Randthema – in diesem Buch wird dagegen das Verhältnis entsprechend umgedreht. Für einen Mathematiker führt dieser Weg ggf. zu einem besseren Verständnis der Software, die er – meist im Studium – schreibt oder benutzt. Für ihn fehlen in dieser Darstellung Erkenntnisse und Denkschritte, um die Methode mit dem Ziel zu durchdringen, die Methoden selbst weiter zu bringen und ihre Theorie möglichst tief zu erfassen.

Ich glaube also, dass dieser dritte Zugang über Algorithmen, also u. a. eigenes Experimentieren und Ausprobieren am Computer, vielen weiterhilft. Außerdem fehlen in vielen Büchern hinreichend detaillierte Anmerkungen zur Umsetzung in einer Programmiersprache. Hier kann dieses Buch auch für Mathematik-Studierende interessant sein, weil es ausführlicher in der algorithmischen Umsetzung ist. Wer stattdessen oder parallel eine fundierte Einführung in die Theorie der finiten Elemente wünscht, dem würde ich eines der folgenden Bücher empfehlen: [Hac05], [Bra03] oder [KA00].

Warum ist es richtig, sich mit den Grundlagen einer Methode, die man einsetzt, über die Bedienung der Software hinaus vertraut zu machen? Mit einer kleinen Anekdote möchte Ihnen noch etwas mehr Motivation mit auf den Weg geben: Wie schnell doch einmal etwas schiefgehen kann, wenn man sich auf eine Software als Blackbox verlässt, zeigt der Fall der Bohrplattform *Sleipner A* aus dem Jahr 1991. Deren Betonstruktur brach in sich zusammen und die Plattform versank. Die Ursache lag in der durchgeführten Analyse mittels der Finite-Elemente-Methode. Die damals verfügbare Hardware gab nur eine sehr grobe Auflösung des 3D-Problems wieder. Darüber hinaus wurden mehrere Parameter geschätzt. Wie Sie im Laufe des Buches sicherlich noch merken werden, hat man es oft mit sinnvollen Intervallen für gewisse Größen statt genauen Werten zu tun. Dazu kommen vernachlässigte Effekte etc. im Modell. Die Modelle, die einer Simulation zugrunde liegen, sind sehr wichtig, weshalb ich im Buch auch versuche, diesen besonders für Ingenieure und Naturwissenschaftler wichtigen Aspekt etwas herauszuarbeiten. Das passiert besonders in den Kapiteln mit Praxisbeispielen. Im Fall der Bohrplattform kam es in Folge dieser Schätzungen und der groben Triangulierung dazu, dass die auftretenden Scherspannungen unterschätzt wurden. Die tatsächlich auftretenden Scherspannungen waren deutlich größer als die aus der FEM-Berechnung. Im Nachhinein stellte sich heraus, dass man von einem Fehler von ca. 47 % ausgehen musste. Die Ingenieure haben damals einer Software mit dem darin umgesetzten Modell quasi blind vertraut. Im Anschluss rechnete der norwegische Energiekonzern Statoil das Ganze mit angepasster Softwarelösung und kritischeren Ingenieuren noch einmal durch. Dann hat es auch geklappt und *Sleipner A* ist seit 1993 nun im Betrieb. Wer gerne mehr zu den Details der damaligen Vorgänge lesen möchte, kann das z. B. hier tun: [JR94], [SVC97].

Als Sprache für die algorithmische Umsetzung habe ich hier die Schnittmenge der Sprachen von GNU Octave und MATLAB® gewählt. Den Schwerpunkt möchte ich dabei auf Octave legen, was seit der Version 3.8 ein eigenes integriertes GUI mitbringt und als freie und kostenlose Alternative für Studierende und Akademiker attraktiver wird. Wer MATLAB hat, kann jedoch alle Beispiele und Listings aus diesem Buch selbstverständlich auch nutzen.

Die Listings sind kompakt gehalten und vollständig angegeben. Sie können sich darüber hinaus die Listings ebenso wie die Geometrien von meiner Webseite *www.joerg.frochte.de* herunterladen. Die Codebeispiele dürfen unter der MIT-Lizenz frei verwendet werden. Code ohne Fehler ist wirklich sehr selten und ich befürchte auch der aus diesem Buch macht keine Ausnahme. Wenn Sie einen Fehler finden, weisen Sie mich doch bitte unter joerg@frochte.de darauf hin und ich korrigiere das zeitnah.

Ein kurzer Hinweis zur Notation

Auf eine optische Unterscheidung von Vektoren durch Fettdruck oder Pfeile wird verzichtet, da dies in der Software und den Listings auch nicht möglich wäre. Vielmehr ist es sinnvoll, sich die Natur einer Variablen aus dem Kontext zu erschließen bzw. diese vorher klar zu definieren.

Organisation des Buches inklusive Hinweisen für Dozenten

Das Buch beginnt mit einem Kick-Start-Kapitel zur Programmiersprache. Es beinhaltet das Minimum an Ausrüstung, das man für dieses Buch braucht, aber auch nicht mehr. Eine vollständigere Einführung findet man z. B. in dem Buch [ABRW14]. Wer sich mit Octave bzw. MATLAB schon auskennt, kann das Kapitel sicherlich überspringen. Das zweite Kapitel illustriert, woher die Anwendungsfälle kommen und was ihren Modellcharakter ausmacht. Danach beginnt der FEM-Teil des Buches.

Ich habe mich dazu entschlossen, mit der Darstellung in nur einer Raumdimension zu beginnen. Dadurch kann man die grundsätzlichen Prinzipien leichter verstehen, und auch die Listings sind übersichtlicher und kompakter. Durch diesen Ansatz kommen auch Menschen, die noch keine Erfahrungen mit GNU Octave oder MATLAB haben, schnell zu Erfolgen und ihren ersten Lösungen. Hat man diese Grundideen verstanden, hangeln wir uns die Dimensionen hinauf, wodurch die algorithmische Umsetzung komplexer wird, man sich jedoch mehr auf diese konzentrieren kann, weil man FEM im Grundsatz ja verstanden hat.

Wer das Buch in einer Lehrveranstaltung als Dozent einsetzen will, wird im Allgemeinen kürzen müssen. Je nach Vorkenntnissen der Teilnehmer sind hier die ersten beiden Kapitel sicherlich der erste Ansatzpunkt, da die Teilnehmer z. B. MATLAB-Kenntnisse mitbringen oder eben keine Einführung in die Modelle ihrer Fachwissenschaft benötigen.

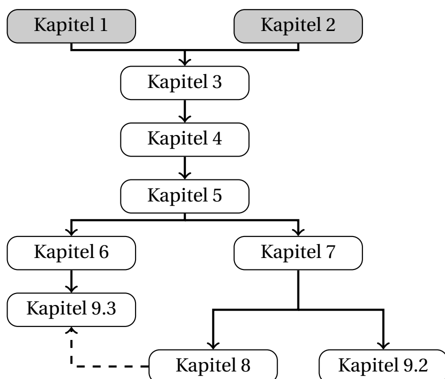


Abbildung 1 Schematischer Aufbau des Buches

Die Abbildung 1 stellt den schematischen Aufbau des Buches dar. Im Kapitel 4 kann bei Bedarf auf Abschnitt 4.3 und 4.5 verzichtet werden. Nach Kapitel 5 können leicht verschiedene Schwerpunkte gesetzt werden und aus den folgenden Kapiteln entsprechend den Interessen der Teilnehmer ausgewählt werden. Die gestrichelte Linie zwischen Kapitel 8 und 9.3 steht dabei für den Einsatz einer einzelnen Octave- bzw. MATLAB-Funktion aus Kapitel 8, die man als Dozent jedoch leicht ersetzen kann. Das in der zweiten Auflage neue Kapitel 10 benötigt beinah alles was zuvor behandelt wurde.

Danksagung

Das Buch hat bestimmt Fehler und Stilblüten, jedoch hoffentlich in jeder Auflage weniger und das besonders Dank der Mithilfe meiner Frau Barbara und dem Lektorat und Korrektorat des Hanser Verlages, für die ich mich an dieser Stelle besonders bedanken möchte. Daneben möchte ich noch Herbert Schmidt, Claudia Frohn-Schauf, Peter Gerwinski, Patrick Bouillon, Christof Kaufmann, Ana Belén Martínez Torres, Michael Knorrenschild, Patrick Bosselmann und Markus Lemmen danken, die mir helfend zur Seite gestanden haben. Meinen Sohn bitte ich hingegen um Verzeihung für die Ausflüchte, mit denen sich sein Vater manchmal in den Keller verzog, um dieses Buch fertigzustellen, anstatt mit ihm zu spielen. Ich hoffe – wie wahrscheinlich jeder Autor – mit diesem Buch hilfreich zu sein. Wenn Sie Fehler gefunden oder noch Verbesserungsvorschläge haben, senden Sie diese doch einfach an joerg@frochte.de.

1 GNU Octave und MATLAB in a Nutshell

In diesem Kapitel gebe ich Ihnen eine Art Kickstart in GNU Octave, also nur die wirklich grundlegenden Techniken vermitteln. Andere Befehle und Techniken werden wir direkt einüben, wenn wir sie in den einzelnen Kapiteln brauchen. Auf keinen Fall ist dieses Kapitel als vollständige Einführung in die Software zu verstehen, da dies ein eigenes Buch füllen würde. Es sollte aber ausreichen, um Personen, die über elementare Programmierkenntnisse in anderen Sprachen verfügen, die Arbeit mit diesem Buch zu erlauben. Die Berechnungen, Beispiele und Screenshots stammen, soweit nicht anders angegeben, von GNU Octave 4.0. Sie finden diese Software für alle gängigen Betriebssysteme, u. a. Windows und Linux, kostenlos unter der folgenden Adresse: *http://www.gnu.org/software/octave/*.

1.1 GNU Octave und MATLAB

MATLAB wird von *The MathWorks* entwickelt und dient zur Lösung und Visualisierung mathematischer und numerischer Fragestellungen in den Natur- und Ingenieurwissenschaften. Das primäre Hilfsmittel und gleichzeitig der Default-Variablentyp ist die Matrix. Daher kommt auch der Name: MATrix LABoratory.

Die Entwicklung von MATLAB startete Ende der 1970er Jahre an der Universität New Mexico, um Anwendern ohne Programmierkenntnisse in der Programmiersprache Fortran die Nutzung der Bibliotheken LINPACK und EISPACK zu ermöglichen. 1984 wurde dann *The MathWorks* gegründet mit MATLAB als Produkt. Heutzutage gibt es ein Grundpaket aus MATLAB und darauf aufbauenden Toolboxen. Die Student-Edition ist preislich sehr moderat ausgerichtet, die akademischen Versionen liegen schon darüber und für die kommerzielle Software sind die Kosten nicht mehr vernachlässigbar und erhöhen sich mit jeder verwendeten Toolbox.

Eine freie Software – sowohl im Sinne von *kostenlos* als auch im Sinne von *Open Source,* – die i. W. kompatibel zu MATLAB ist, ist GNU Octave. Die Entwicklung von Octave geht auf das Ende der 80-er und den Anfang der 90-er Jahre zurück. Heutzutage wird Octave von der Free Software Foundation als eines der „High Priority Free Software Projects“ gesehen.

Generell ist es wünschenswert, wo immer möglich, in der Wissenschaft auf Open Source Software zu setzen. Ein Grund ist, dass sich Wissenschaft immer darauf stützt, dass Experimente nachvollzogen und verifiziert werden können. Will eine Simulation – ggf. auch nur teilweise – an die Stelle des Experiments treten, so muss auch die Simulation für andere Wissenschaftler und Ingenieure verifizierbar sein. Nutzt man keine Open Source Software, so beschränkt man die Möglichkeiten zur Verifikation deutlich. Kein Externer kann nachvollziehen, wie die Daten innerhalb der Software verarbeitet werden und selbst auf dem zugänglichen Niveau der ent-

sprechenden Software schließt man alle Personen aus, die sich die Lizenz für das Programm nicht leisten können oder wollen. Ein lesenswertes Plädoyer wurde z. B. in der renommierten Zeitschrift „Nature“ im Jahr 2012 gebracht [IHGC12]. Aus Sympathie für diesen Ansatz werde ich den Schwerpunkt auf Octave legen.

Generell verwenden beide, MATLAB und Octave, für Vektor- und Matrizenoperationen optimierte **„Basic Linear Algebra Subprograms“ (BLAS)** Bibliotheken, die zu einer sehr schnellen Ausführung beitragen. Im Fall von Octave ist es die **„Automatically Tuned Linear Algebra Software“ (ATLAS)**. Beide Tools verwenden selbst Skriptsprachen, die interpretiert werden müssen. Das führt dazu, das z. B. `for`-Schleifen schlecht für die Performance sind. Das gilt besonders für GNU Octave, da MATLAB seit 2002 einen Just-in-time-Compiler besitzt, welcher immer weiterentwickelt wird und Programme zur Laufzeit in Maschinencode übersetzt. Dies führt zu deutlichen Verbesserungen bzgl. der Performance. Da aber beide Tools optimiert wurden, um mit Matrizen und Vektoren umzugehen, gibt es ein Gegenmittel, die Vektorisierung. So bezeichnet man das Ersetzen von Schleifen und eher skalarwertig organisiertem Code durch Matrix- und Vektor-Operationen.

Beide Umgebungen lassen sich über sogenannte MEX-Functions, z. B. durch C-Funktionen, erweitern. Diese Möglichkeit erlaubt es oft, zeitkritische Dinge, die man nicht vektorisieren kann, auszulagern oder Schnittstellen zu externen Programmen oder Hardware zu schaffen. Bei GNU Octave sei noch erwähnt, dass es direkt aus C++ heraus aufgerufen werden kann, was die Integration von Code in andere Umgebungen erleichtert.

Beim ersten Start wird sich GNU Octave für Sie etwa wie in Abbildung 1.1 gezeigt öffnen.

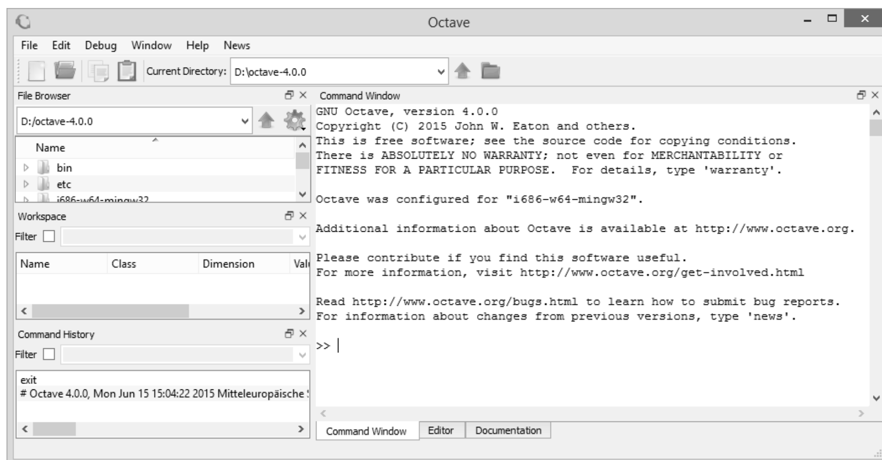


Abbildung 1.1 Screenshot von GNU Octave 4.0

Es kann sein, dass es sich bei Ihnen auf Deutsch öffnet. Ich möchte Ihnen vorschlagen, die Sprache auf US-Englisch zu ändern, da Sie bei der Suche nach Hilfe im Internet immer auf die Begriffe der englischen GUI stoßen werden. Der `File Browser` zeigt auf das aktuelle Arbeitsverzeichnis bzw. beim ersten Start auf das Verzeichnis des Programmes selbst. Ein sinnvolleres Arbeitsverzeichnis können Sie unter `Edit` → `Preferences` im Reiter `General` definieren. Hier können Sie auch, wie oben erwähnt, ggf. die Sprache der GUI ändern. Der `Workspace` enthält alle aktuell im Speicher befindlichen Variablen. Wir kommen gleich noch einmal darauf

zurück. Die `Command History` hingegen ist eine Komfortfunktion, mit der man sich schnell Befehle wiederholen kann, die man vor kurzem verwendet hat.

Im `Command Window` werden wir jetzt arbeiten. Hier werden Befehle eingetragen und direkt ausgeführt, z. B. `disp('Es kann losgehen')`. Wenn Sie diesen Befehl eintippen, wird der entsprechende Text ausgegeben. Wenn Sie den Befehl erneut ausführen wollen, reicht es, die Pfeiltasten hoch und herunter zu verwenden, um so durch die `Command History` zu spulen und von dort alte Eingaben auszuwählen.

Octave verwendet übrigens die gleichen primären GUI-Elemente – also `File Manager`, `Workspace`, `Command History` und `Command Window` – wie MATLAB. Sollten Sie also MATLAB verwenden, gilt alles analog.

Wenn Sie über den folgenden Abschnitt hinaus Hilfe suchen, gibt es zahlreiche Anlaufstellen:

- Die Eingabe `help name` liefert im `Command Window` die Kurzhilfe zu jeder Funktion
- MATLAB Online-Doku: *http://www.mathworks.com/help/techdoc/*
- Octave Online-Doku: *https://www.gnu.org/software/octave/doc/interpreter/*
- Community Portal von MATLAB: *http://de.mathworks.com/matlabcentral/*
- Bug-Reports und Diskussion zu Octave: *http://savannah.gnu.org/bugs/?group=octave*

1.2 Arbeiten mit Matrizen und Vektoren

Auf Grund der schon erwähnten fundamentalen Rolle von Matrizen und Vektoren in den verwendeten Tools starten wir auch mit diesem Aspekt.

1.2.1 Arbeiten auf der Konsole

Auf der Kommandozeile können Sie Befehle eingeben, die sich direkt auf die Variablen im Workspace auswirken. Beispielsweise

```
» b = 42
b = 42
```

Variablen, die noch nicht existieren, werden direkt erzeugt. Im Gegensatz zu Java und C ist es nicht nötig, diese vorher zu definieren oder zu deklarieren. Entsprechend finden wir im Workspace-Fenster jetzt die Variable b mit der Eigenschaft eine double-Variable zu sein und die Dimension 1x1 zu haben. Letzteres bedarf vielleicht einer kurzen Erklärung. Für Octave/-MATLAB ist alles erst einmal eine Matrix. Ein Skalar ist also eine 1x1-Matrix usw. Ohne GUI erhalten Sie den Workspace-Inhalt durch die Befehle `who` (Kurzinformation) und `whos` (ausführliche Information).

Das Default-Format für eine Ausgabe ist *short* ; das bedeutet, dass die Ausgabe – nicht die Berechnung – in Form einer 5-Stellen-Fixkommadarstellung erfolgt. Oft ist es sinnvoller, das Format auf *shortE*, also eine wissenschaftliche Gleitkomma-Darstellung zu ändern.

Kurzes Beispiel:

```
» pi/10
ans = 0.31416
» format shortE
» pi/10
ans = 3.1416E-001
» format longE
» pi/10
ans = 3.14159265358979E-001
```

Wie man sieht, kann die Software auch quasi als Taschenrechner verwendet werden. Die typischen mathematischen Funktionen sind völlig analog zu anderen Programmiersprachen. Das bedeutet insbesondere: Die trigonometrischen Funktionen wie Sinus, Tangens etc. erwarten das Argument (Winkel) im Bogenmaß, und `log` meint den natürlichen Logarithmus (ln). Wer gewohnt ist, mit Winkeln in Grad und Logarithmen zur Basis 10 zu rechnen, muss hier also umdenken und ggf. auf den Befehl `log10(x)` zurückgreifen.

1.2.2 Matrizen erzeugen und manipulieren

Da Octave primär auf der Manipulation von Matrizen und Vektoren basiert, ist der Funktionsumfang hier besonders groß und der Zugriff sehr komfortabel. Eine Matrix ist eine rechteckige Anordnung von Zahlen, z. B.

$$\begin{matrix} 1.1000 & 1.2000 & 1.3000 & 1.4000 \\ 2.1000 & 2.2000 & 2.3000 & 2.4000 \\ 3.1000 & 3.2000 & 3.3000 & 3.4000 \end{matrix}$$

Will man auf ein Element der Matrix zugreifen bzw. es benennen, gibt man die Zeile und die Spalte an:

	1	2	3	4
1	1.1000	1.2000	1.3000	1.4000
2	2.1000	2.2000	2.3000	2.4000
3	3.1000	3.2000	3.3000	3.4000

Es wird immer zuerst die Zeile und dann die Spalte angegeben. Entsprechend ist $A(2,3) = 2.3$.

In vielen Programmiersprachen, wie z. B. C und Java, beginnt die Nummerierung mit *0*, in MATLAB und Octave jedoch bei *1*!

Erzeugen kann man Matrizen auf sehr unterschiedliche Weise. Ich beschränke mich hier auf die vier für uns wichtigsten Varianten. Zum einen kann man eine Matrix mit flexiblen Werten direkt erzeugen mittels Zuweisung:

```
» A = [1 2 3.2; 0.1 2 1/5; 0.9 1 2]
A =
1.0000E+000 2.0000E+000 3.2000E+000
```

```
1.0000E-001 2.0000E+000 2.0000E-001
9.0000E-001 1.0000E+000 2.0000E+000
```

Die Zeilen werden mit einem Semikolon definiert und der Start sowie das Ende durch eine eckige Klammer. Einen Zeilenvektor erhalten Sie entsprechend wie folgt:

```
» x = [ 1 2 3 4]
x =

1.0000E+000 2.0000E+000 3.0000E+000 4.0000E+000
```

Wenn Sie einen Spaltenvektor benötigen, können Sie entweder viele Semikolons verwenden oder einfacher das Apostroph-Symbol `'`, das bei reellen Matrizen dem Transponieren entspricht:

```
» x = x'
x =

1.0000E+000
2.0000E+000
3.0000E+000
4.0000E+000
```

Neben dieser Möglichkeit gibt es noch die Option, mittels der Befehle `zeros` und `ones` Matrizen zu erzeugen, die ausschließlich mit Nullen bzw. Einsen vorinitialisiert sind. Darüber hinaus ist `eye` noch sehr nützlich, um eine Diagonalmatrix mit Einsen auf der Diagonalen zu erzeugen. Alle drei Befehle funktionieren so, dass sie entweder die Zeilen und Spalten als Option angeben können, z. B. `A = zeros(3,6)`, oder nur eine Zahl, wenn es sich um eine quadratische Matrix handeln soll, z. B. `A = eye(5)`.

Bei den finiten Elementen sind andere Speichertechniken für Matrizen wichtiger. Der Hintergrund ist, dass die oben erwähnten Matrizen voll besetzt oder dicht sind. Das bedeutet, dass wirklich jeder Eintrag gespeichert wird. Bei der FEM entstehen riesige Matrizen, jedoch sind hier fast alle Einträge null. Hier bietet es sich an, nur die von null verschiedenen Einträge zu speichern. Das hat zwei primäre Vorteile: Zum einen passen die Datenstrukturen für große Matrizen in den Speicher einer typischen Workstation und zum anderen wird die Performance bei wichtigen Operationen wie Matrix-Vektor-Multiplikationen verbessert. Bei einer voll besetzten Matrix müssten sehr viele Gleitkommaoperationen vom Typ $0 \cot a$ durchgeführt werden, obwohl diese natürlich nichts zu dem Ergebnis beitragen. Werden nur von null verschiedene Einträge gespeichert, werden auch nur die Operationen durchgeführt, die potenziell etwas zum Ergebnis beitragen können. Diese im FEM-Umfeld verwendeten Matrizen nennt man im Gegensatz zu den voll besetzten Matrizen (eng. *dense matrix*) dünn besetzte Matrizen – im Englischen und in der Dokumentation als *sparse,* also *spärlich* bezeichnet. Diese Datenstrukturen sind leider nicht in allen Operationen performant so z. B. ist das Erzeugen von Einträgen sehr aufwendig. Es gibt aber Befehle und Techniken, die Matrizen effizient aus zuvor gesammelten Operationen aufbauen können. Davon werden wir im Laufe des Buches noch Gebrauch machen.

Dünn besetzte Matrizen können sehr einfach durch den Befehl `sparse` erzeugt werden. Zum einen kann man eine bestehende voll besetzte Matrix mittels `S = sparse(A)` in eine dünn besetzte konvertieren, die Rückkonvertierung funktioniert mittels des Befehls `full`. Zum anderen kann man mit `S = sparse(n,m)` eine Matrix direkt als dünn besetzte Matrix anlegen.

Beispiel:

```
» S=sparse(100,100);
» S(10,23)=1
S =
Compressed Column Sparse (rows = 100, cols = 100, nnz = 1 [0.01%])
(10, 23) -> 1
```

Octave teilt einem auch direkt mit, in welchem Format die dünn besetzte Matrix realisiert wurde. Hier gibt es tatsächlich eine recht große Auswahl, wobei hier eben das Compressed-Column-Format verwendet wurde. An dem Beispiel oben sieht man auch direkt, dass ein Semikolon dazu genutzt werden kann/sollte, die Ausgabe auf der Konsole zu unterdrücken. Bei größeren Operationen oder den Schleifen, die wir gleich kennenlernen werden, ist das unumgänglich.

Ein weiterer wichtiger Operator zur Definition von Vektoren ist der Doppelpunkt. Der Ausdruck

```
» x=[0:0.2:1]
x =
0.00000 0.20000 0.40000 0.60000 0.80000 1.00000
```

erzeugt einen Vektor mit festem Start- und Endpunkt und einer Schrittweite zur Inkrementierung. Die allgemeine Syntax dazu ist <Start>:<Increment>:<Ende> .

Zum Abschluss dieses Abschnittes schauen wir noch einmal darauf, welche weiteren Möglichkeiten wir haben, Matrizen zu manipulieren, außer nur – wie oben – elementweise auf Einträge zuzugreifen. Zunächst erzeugen wir uns dafür einmal eine Matrix mit einer schönen Struktur:

```
» A=magic(5)
A =
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9
```

Was es genau mit dieser besonderen Struktur auf sich hat, können Sie durch die Hilfefunktion mittels `help magic` erfahren. Will man auf einen rechteckigen Bereich dieser Matrix zugreifen, so ist dies über einen Index-Vektor möglich:

```
» A(2:3,4:5)
ans =
14 16
20 22
```

Der Bereich muss nicht unbedingt rechteckig sein, obwohl dies eine häufige Anwendung ist:

```
» A([2 3 4],[2 5])
ans =
5 16
6 22
12 3
```

Sehr angenehm ist die Möglichkeit, hier direkt ganze Gruppen von Einträgen zu verändern:

```
» A([2 3 4],[2 5])=ones(3,2)
A =
17 24 1 8 15
```

```
23 1 7 14 1
4 1 13 20 1
10 1 19 21 1
11 18 25 2 9
```

Bezüglich der Manipulation von Matrizen ist die Sprache also wirklich sehr komfortabel. Schauen wir uns das nun bzgl. der Rechenoperationen an.

1.2.3 Rechenoperationen

Ein großer Vorteil für unsere Arbeit ist, dass sich alle Funktionen wie Sinus, Kosinus etc. direkt auf Vektoren und Matrizen anwenden lassen. Nimmt man z. B. den oben definierten Vektor `x`, so kann man in einer Zeile direkt den zu jedem Eintrag zugehörigen Funktionswert berechnen:

```
» y=sin(x)
y =
0.00000 0.19867 0.38942 0.56464 0.71736 0.84147
```

Matrizen können direkt, wie man es auch normal notiert, mit den Operatoren `+`, `-` und `*` verknüpft werden. Der Operator `*` wird dabei im Kontext interpretiert, wie das folgende Beispiel zeigt:

```
» A=ones(2);
» 3*A
ans =
3 3
3 3

» A*[1 2; -1 0]
ans =
0 2
0 2
```

Diese Operationen entsprechen der traditionellen Notation. Es gibt diese Operatoren jedoch auch noch mit einem Punkt. Dieser Punkt signalisiert, dass die Operation elementweise ausgeführt werden soll. Beispiel zur Verdeutlichung des Unterschieds:

```
» [3 -1; 1 2]*[1 2; -1 0]
ans =
4 6
-1 2

» [3 -1; 1 2].*[1 2; -1 0]
ans =
3 -2
-1 0
```

Ihnen ist sicher aufgefallen, dass ich den Operator `\` bzw. `/` ausgespart habe. Ein Aspekt ist, dass beide Operatoren vorkommen, aber der eine, nämlich `\`, als *von links* gelesen wird und der andere *von rechts*. Ein einfaches Beispiel illustriert den Effekt: 8\6 ergibt 0.7500, während 8/6 1.3333 ergibt. Ein Punkt erzeugt wie oben ein elementweises Vorgehen. Ohne hingegen ist der \ der Aufruf eines direkten Lösers für lineare Gleichungssysteme $Ax = b$.

Beispiel:

```
» A=[1 2; 1 1]; b=[3 -1]';
» A\b
ans =
-5
4
```

Dabei müssen Sie sich als Nutzer keine Gedanken über die Auswahl des direkten Lösers machen. Das Programm entscheidet selbstständig aufgrund der Eigenschaften der Matrix. Der gleiche Operator kann daher auch für das Lösen der für uns wichtigen, dünn besetzten Gleichungssysteme verwendet werden; u. a. kommt hier als Fallback unter MATLAB und Octave der gleiche Löser, nämlich UMFPACK von Tim Davis, zum Einsatz; siehe *http://faculty.cse.tamu.edu/davis/research.html*. Auch bei Gleichungssystemen gibt es übrigens den Ansatz *von links* und *von rechts* lösen. Wie oben gezeigt, ist für uns der *Backslash*, oder wenn Sie so wollen der Rückschrägstrich, der intuitive Operator und die versehentliche Verwendung des Slash / eine nicht so seltene Fehlerquelle; kaum jemand, der noch nie mit einem solchen Tool gearbeitet hat, erwartet bei der Eingabe 8\6 als Antwort 0.7500!

1.3 Skripte und Funktionen schreiben

Das Command Window allein kann man schon produktiv einsetzen, aber es ist mehr der Ort, um komplexere Dinge zu starten und zu verknüpfen und nicht, um umfangreiche Algorithmen umzusetzen – geschweige denn zu debuggen. Dies wird mittels der m-Files umgesetzt. Diese entsprechen dem, was in anderen Programmiersprachen Programme und Funktionen sind. Ursprünglich war die von der Firma TheMathWorks ersonnene Sprache eine prozedurale Programmiersprache, der in jüngster Vergangenheit objektorientierte Möglichkeiten gegeben wurden. In Octave hat die Objektorientierung noch später Einzug gehalten und da der objektorientierte Ansatz bei unseren Fragestellungen bei weitem nicht zwingend ist, werden wir ihn hier als Teil der Sprache auch nicht ausführen.

Unsere m-Files bestehen im Wesentlichen aus einer Folge von Befehlen. Wir beginnen zunächst mit den Skripten. Skripte sind am ehesten verwandt mit Programmen. Ihre Variablen werden im Hauptspeicher angelegt und nach dem Ende des Skriptes nicht automatisch freigegeben. Die Befehle im Skript werden von oben nach unten durchlaufen und dabei ausgeführt. Skripte haben dabei weder Eingabe- noch Ausgabeparameter, sondern arbeiten mit dem, was im Speicher ist, und hinterlassen ihn nach getaner Arbeit verändert. Ein Skript erlaubt es also primär, Eingaben, die man auch im Command Window tätigen könnte, wiederverwertbar zu machen und dabei auch die Fehlersuche zu erleichtern. Schauen wir uns das zunächst an einem kleinen Beispiel an:

Zunächst müssen Sie in das Verzeichnis gehen, im dem Sie Ihre m-Files ablegen; z. B. so: `cd MatlabCodes/`. Legen Sie nun ein File `beispielskript.m` über die GUI mit dem Menüpunkt `File -> New -> New Script` an. Nun öffnet sich ein leeres Fenster, in dem Sie Ihr Skript, z. B. das aus dem nächsten Screenshot in Abbildung 1.2, eingeben können.

Kommentare werden in MATLAB durch ein Prozentzeichen % eingeleitet. Octave sieht eigentlich Kommentare mit # bzw. ## vor, versteht jedoch auch den MATLAB-Ansatz. Für eine platt-

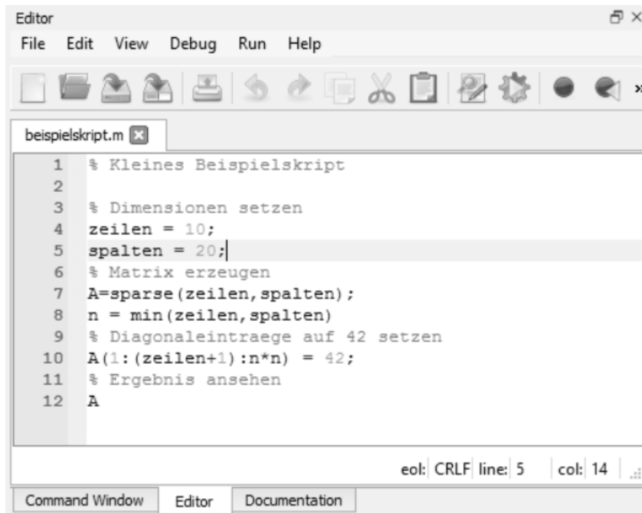


Abbildung 1.2 Screenshot Editor mit Mini-Skript (GNU Octave 4.0)

formunabhängige Entwicklung sind jedoch Kommentare mit % sehr empfohlen. Nutzt man die Octave-Variante, kommt es bei Kooperationen mit MATLAB-Usern mit Sicherheit zu Problemen. Entsprechend leiten wir im Buch Kommentare immer mit % ein. Das Skript oben funktioniert übrigens durch die Tatsache, dass man die Elemente einer Matrix auch durch einen einzigen Index ansprechen kann. Hierbei wird die Matrix so interpretiert, als wenn alle Spalten übereinandergestapelt in einem langen Vektor stünden.

Sie können das Skript nun durch Drücken auf das Symbol *Save File and Run* speichern und anschließend sofort ausführen. Allerdings werden Sie die meisten Ausgaben und Fehlermeldungen so nicht sehen. Es ist also sinnvoll, ins Command Window durch den Reiter unten zurückzuwechseln. Ich gehe in der Regel so vor, dass ich das Skript mit einer Tastaturkombination – Default ist Strg-S – speichere und direkt auf das Command Window wechsle. Dort können Sie es durch die Eingabe des Dateinamens, im Beispiel `beispielskript`, das Skript starten. Hierbei darf das `.m` nicht angegeben werden.

Sowohl MATLAB als auch Octave suchen ihre Funktionen und Skripte immer zuerst im Arbeitsverzeichnis und dann im Suchpfad. Liegt eine Datei weder im Suchpfad noch im Arbeitsverzeichnis, wird sie nicht gefunden!

Man kann auch Skripte debuggen, aber wir machen das hier am typischeren Beispiel von Funktionen. Diese erzeugen wir nun einmal mit einem alternativen Weg, nämlich über das Command Window mittels z. B. `edit polar2kartesische.m`. Octave-Nutzer sollen sich nicht wundern: Hier öffnet sich – aus vermutlich ideologischen Gründen – sofort ein Template, welches die Funktion quasi mit der GPL versehen möchte. Die GNU General Public License (GPL) ist die von der Free Software Foundation favorisierte Software-Lizenz. Falls Sie das Thema interessiert, finden Sie hier [Ins05] weitere Informationen dazu. Wenn Sie diesen Wunsch teilen,

können Sie es stehen lassen; wenn nicht, ersetzen Sie es durch die Lizenz, die Ihnen am besten passt. Es gibt keine Verpflichtung, m-Files unter einer speziellen Lizenz zu veröffentlichen. Dies können Sie z. B. auch in der Octave FAQ: *http://wiki.octave.org/FAQ* unter *If I write code using Octave do I have to release it under the GPL?* nachlesen. Da wir MATLAB und Octave gleichermaßen mit unseren Skripten und Funktionen bedienen wollen, empfiehlt es sich, generell die Templates der beiden Tools eher zu ignorieren, da man sich sonst ggf. Eigenarten einer speziellen Plattform einhandelt. Als Beispiel seien die Kommentarzeilen mit # in Octave genannt, die inkompatibel zu MATLAB sind; außerdem will Octave Funktionen gerne MATLAB-inkompatibel beenden etc. Löschen Sie also am besten alles, was Sie vorfinden. Anschließend tragen Sie Folgendes ein:

```
1  function [x y] = polar2kartesische (angle, radius)
2  % Diese Funktion rechnet vom kartesischen Koordinatensystem in ein
3  % Polarkoordinatensystem mit gleichem Ursprung um.
4  % Den Winkel bitte im Bogenmass angeben.
5
6  % Kommentare nach der Leerzeile werden nicht mehr als Hilfe ausgegeben
7  x=radius*cos(angle);
8  y=radius*sin(angle);
```

Zunächst funktioniert das mit MATLAB kompatible Hilfesystem so, dass alle Kommentare bis zur ersten Leerzeile, also in unserem Beispiel bis inklusive Zeile 4, als Hilfetext ausgegeben werden, wenn der Benutzer die Hilfe wie folgt anfordert:

```
» help polar2kartesische
```

Alle anderen Kommentare im Anschluss daran sind ausschließlich – wie in eigentlich allen Programmiersprachen – ein Mittel der Dokumentation und zur Verbesserung der Lesbarkeit.

Wenn der Benutzer sich die Hilfe erfolgreich durchgelesen hat, kann er die Funktion wie folgt aufrufen:

```
» [x y] = polar2kartesische (pi/2, 3)
x = 1.8369e-016
y = 3
```

Das Ergebnis gibt die üblichen Probleme im Bereich der Gleitkommaarithmetik wieder.

Da wir den Hintergrund der Algorithmen im Fließtext des Buches diskutieren, verzichte ich weitgehend darauf, Kommentare im Buch abzudrucken. Dass dies für echten Programmcode keine Vorbildfunktion haben soll, muss sicherlich nicht erwähnt werden. Ohne Dokumentation steigt der Hang zu nicht wartbarem Code inklusive steigender Fehlerzahl an.

Fehler an sich treten natürlich während der Entwicklung immer auf und sowohl MATLAB als auch Octave besitzen einen eingebauten graphischen Debugger. Bei Octave ist er wie in der Abbildung 1.3 dargestellt in den Editor integriert.

Sie können hier mit dem roten Punkt-Symbol die von anderen Tools sicherlich bekannten *Breakpoints* setzen. An einem Breakpoint wird das Programm angehalten und Sie können es z. B. mit der Step-Taste schrittweise durchlaufen. Dabei lassen sich die Änderungen der Variablenwerte z. B. im Workspace beobachten. Es besteht jedoch auch die Möglichkeit, direkt im Command Window Eingaben durchzuführen, um sich Variablen zu visualisieren. Wie man das tut, dazu kommen wir später noch.

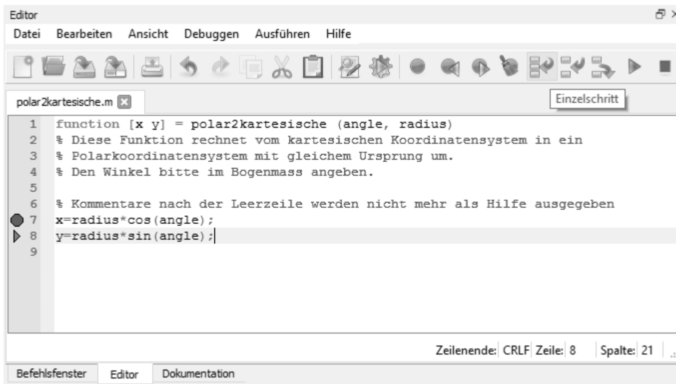


Abbildung 1.3 Screenshot: Editor im Debug-Modus (GNU Octave 4.0)

Ein Tipp: Man kann sich, wenn man Funktionen schreibt, die vektorwertige Eingaben verarbeiten, nie ganz sicher sein, ob der Benutzer einen Vektor als Spalten- oder Zeilenvektor übergibt. Oft führt genau das zu Problemen. Mit `x = x(:)'` wird der Vektor x immer in einen Zeilenvektor umgewandelt und mit `x = x(:)` immer in einen Spaltenvektor. Machen Sie das immer zu Beginn der Funktion.

Ein anderer Aspekt ist, dass man manchmal eine Funktion an eine andere Funktion übergeben möchte. In einer Sprache wie C würde man das mit einem Zeiger machen und in Octave oder MATLAB nutzt man einen sogenannten **Function-Handle**.

Zum Erzeugen eines *Function-Handles* benutzt man das @-Symbol. Es gibt zwei Arten, *Function Handles* einzusetzen, einmal als Referenz auf eine existierende Funktion in einem m-File und einmal als anonymes Handle, bei dem quasi en passant eine Funktion erzeugt wird. Als Beispiel betrachten wir die folgenden Zeilen:

```
» myfunc = @(x,y) 1-x^2-y^2;
» myfunc(4,3)
ans = -24
```

In der ersten Zeile wird eine Funktion erzeugt, die es in einem m-File gibt, die wir jedoch als Funktion über den Handle ansprechen können. Ebenso häufig nimmt man diesen Ansatz jedoch auch, um auf Funktionen in m-Files zu referenzieren. Betrachten wir z. B. die Funktion

```
1  function [ m ] = logarithmischerMittelwert( a,b )
2    m = (b - a)/(log(b) - log(a));
```

Um einen Handle auf diese Funktion zu erzeugen, schreiben wir einfach

```
mittelwert = @logarithmischerMittelwert
```

Nun steht `mittelwert` symbolisch für die verlinkte Funktion und kann analog genutzt werden, also z. B. :

```
» mittelwert(8,4)
```

Dadurch kann einer anderen Funktion ein solcher Handle mit dem Namen `mittelwert` übergeben werden, der je nach Anwendungsfall auf den logarithmischen, den geometrischen oder den arithmetischen Mittelwert zeigt.

Was an der Funktion oben natürlich noch unschön ist, ist, dass der Fall $a = b = 0$ nicht abgefangen wird. Dazu benötigt man Kontrollstrukturen, mit denen wir uns im nächsten Abschnitt beschäftigen.

1.4 Elementare Kontrollstrukturen und Vektorisierung

Für die Programmierung stehen uns die klassischen Kontrollstrukturen zur Verfügung. Die dabei am häufigsten benötigten sind:

- `if`-Verzweigung
- Zählschleife mittels `for`
- kopfgesteuerte Schleife mittels `while`

Für diese Schleifen sind oft die Befehle `length` und `size` wichtig, um die Dimension von Matrizen und Vektoren zu erfragen. Wir betrachten nun zu jeder dieser Kontrollstrukturen ein Beispiel, das gleichzeitig auf den Aspekt der **Vektorisierung** hinweisen soll, der bei Octave quasi lebenswichtig ist.

```
1  DoF=1000;
2  % ----------------------- Ansatz 1 -----------------------
3  A=sparse(DoF,DoF);
4  tic
5  for c = 1:DoF
6    for r = 1:DoF
7      if r == c
8        A(r,c) = 2;
9      elseif abs(r-c) == 1
10       A(r,c) = -1;
11     else
12       A(r,c) = 0;
13     end
14   end
15 end
16 time1=toc
17 % ----------------------- Ansatz 2 -----------------------
18 B=sparse(DoF,DoF);
19 tic
20 B(1,1)=2;
21 for i=2:DoF
22   B(i-1,i) = -1; B(i,i-1) = -1; B(i,i)=2;
23 end
24 time2=toc
25 % ----------------------- Ansatz 3 -----------------------
```

```
26 C=sparse(DoF,DoF);
27 tic
28 C(1:(DoF+1):DoF*DoF) = 2; C(DoF+1:DoF+1:DoF*DoF)=-1; C(2:DoF+1:DoF*DoF)=-1;
29 time3=toc
30 % ----------------------- Ansatz 4 -----------------------
31 tic
32 index1 = 2:DoF; index2 = 1:DoF-1; index3 = 1:DoF;
33 minus1 = -1*ones(size(index1)); two = 2*ones(size(index3));
34 x = [index1 index2 index3]; y = [index2 index1 index3];
35 value = [minus1 minus1 two];
36 C=sparse(x,y,value);
37 time4=toc
```

Wie man sieht, werden == und = analog zu den meisten Programmiersprachen, wie z. B. C, verwendet. Der Operator == dient dem Vergleich und = bewirkt eine Zuweisung.

Im Listing wird auf vier Arten eine Tridiagonalmatrix erzeugt. Solche Matrizen treten insbesondere bei 1D FEM sehr häufig auf und werden dort auch sehr groß.

Der erste Ansatz von Zeile 3 bis 16 demonstriert die Nutzung von `for` und `if` in dieser Programmiersprache. Wie man sieht, findet die Strukturierung über ein `end` am Ende der Kontrollstruktur statt. Die Zählvariable einer `for`-Schleife durchläuft alle Einträge eines Vektors, der mit einem Gleichheitszeichen mit der Zählvariablen verknüpft wird. Im Allgemeinen wird dieser durch `start:inkrementierung:ende` gebildet. Der Ansatz als solcher ist natürlich völlig abwegig, weil hierdurch ohne Not eine quadratische Komplexität erzeugt wird – bei einer $n \times n$-Matrix müssen alle Einträge durchlaufen werden. Das zeigt sich durch unsere Performance-Messung mit `tic` und `toc`. Mit diesen Befehlen wird die Ausführungszeit gemessen. Will man hingegen die CPU-Zeit zur Performance-Messung nutzen, so könnte man statt `tic` einfach `t = cputime;` schreiben und dann das `toc` jeweils durch `zeit = cputime-t;` ersetzen. Beide Ansätze haben Vor- und Nachteile. Die Ausführungszeit ist die Zeit, die Sie auf eine Antwort des Programmes warten, und daher oft ein legitimer Wert, den man senken möchte. Andererseits kann dieser Wert deutlich gestört werden, wenn parallel andere Programme ausgeführt werden, die ggf. unser Programm kurzzeitig vom Prozessor verdrängen. Die CPU-Zeit muss man hingegen bei Multicore-Systemen interpretieren lernen, denn wenn Teile eines Programmes parallelisiert werden, erhöht sich diese CPU-Zeit ggf. auf einen höheren Wert als die Ausführungszeit. Immerhin wurde Zeit auf mindestens zwei Prozessorkernen verbracht.

Wenn ich auf meinem Rechner das obige Programm mit 1000 Freiheitsgraden (Degrees of Freedom – DoF) ausführe, dann bekomme ich durch Octave bzw. MATLAB folgende Resultate aus Tabelle 1.1.

Tabelle 1.1 Ergebnisse auf beiden Plattformen bei DoF = 1000

	time1	time2	time3	time4
Octave	10.4930 s	0.0220 s	0.0000 s	0.0010 s
MATLAB	1.2058 s	0.0042 s	0.0032 s	0.0019 s

Dass der erste Ansatz, wie schon oben angedeutet, völlig inperformant ist, sollte uns nicht wundern. Durch die beiden Schleifen laufen wir alle Werte einer quadratischen Matrix ab und führen dann noch jeweils eine zusätzliche Operation – die if-Abfrage – aus. Das alleine erklärt aber nicht die so schlechten Werte. Die Sprache wird in Octave interpretiert. Dadurch sind

Schleifen, die in jedem Durchlauf Befehl für Befehl interpretiert werden, extrem zeitaufwendig. In MATLAB wird dieser Effekt schon seit Jahren durch einen eingebauten Just-In-Time-Compiler gemindert, sodass sich hier ein Geschwindigkeitsvorteil von ca. dem Faktor 10 gegenüber Octave ergibt. Octave selber verfügt über einen experimentellen JIT-Compiler, den man aber noch per Hand einkompilieren (vgl. [EBHW21] Abschnitt 19.5) muss. Schaut man jedoch einmal auf die Ansätze 3 (Zeile 26 – 29) und 4 (Zeile 31 – 37), so erkennt man, dass auch mit JIT-Compiler der Ansatz 4 deutlich schneller erscheint. Der Ansatz 3 war natürlich auch auf Octave nicht in Nullzeit fertig, aber die Ausführungszeit lag unter der Messgenauigkeit.

Der Zugriff auf die Elemente in den Ansätzen 3 und 4 ist nicht unbedingt direkt intuitiv. Malen Sie sich mal auf einem karierten Papier z. B. eine 10×10-Matrix auf und vollziehen Sie die Zugriffsstrategien auf die Elemente nach.

Leider gibt es für den Ansatz 3 eine Grenze bzgl. der DoF. Diese Grenze liegt bei DoF = 46340. Hintergrund ist, dass Octave für Windows und Linux als fertiger Download nur mit 32 Bit-Integer-Zugriff ausgeliefert wird. Dieser Integer-Zugriff endet bei 2 147 483 647, was jedoch bei quadratischem Verlauf wie bei uns eben schon mit DoF=46340 erreicht ist. Man kann Octave experimentell mit einem anderen Zugriff compilieren (vgl. [EBHW21], Abschnitt *E.3 Compiling Octave with 64-bit Indexing*), aber darauf möchte ich hier nicht eingehen.

Die einzige sichere und performante Variante ist Nr. 4. Ein Test mit 100 000 Unbekannten liefert – wie man in Tabelle 1.2 erkennen kann – riesige Faktoren zwischen dem Ansatz über eine Schleife (Nr. 2) und der vektorisierten Variante Nr. 4 mittels `sparse`. Diese macht sich die Tatsache zunutze, dass man die Zeilen und Spalten sowie die zugehörigen Werte vorab in Vektoren sammeln und diese dann direkt zur Erzeugung einer Sparse-Matrix übergeben kann. Betrachten wir hier ein kurzes Beispiel, wie das funktioniert:

```
» i=[1 2 3 1 2];
» j=[1 3 2 1 3];
» wert=[1 1 1 1 1];
» A=sparse(i,j,wert,4,5)
A =
(1,1) 2
(3,2) 1
(2,3) 2
```

Als Wert wird nur 1 hinterlegt. Würde der Befehl dies als Zuweisung interpretieren, also `A(2,3) = 1`, so hätten wir als Ergebnis nur Einsen in unserer Matrix. Es wird aber im Sinne eines Aufaddierens interpretiert, also `A(2,3) = A(2,3)+1`. Durch die Verwendung dieses Befehls verringert man deutlich die schon erwähnten Performance-Probleme beim Erzeugen von Einträgen in den Matrizen. Normalerweise wird die so erzeugte Matrix nur so groß, wie es der größte Eintrag in i und j vorgibt. Die letzten beiden Parameter sorgen für eine von uns festgelegte Größe.

Dieser Ansatz in unserem Test oben liefert von allen vorgestellten die beste Perfomance. Man erkennt auch, dass der JIT-Compiler für uns sicherlich eine nette Sache ist, wenn wir es einmal wirklich nicht schaffen, einen Programmabschnitt zu vektorisieren, wir jedoch immer eine Vektorisierung anstreben sollten – egal ob unter MATLAB oder unter Octave.

Tabelle 1.2 Vektorisierung vs. Schleifenansatz bei DoF = 100.000

Ansatz	Octave	MATLAB
Schleife	158.56 s	16.2388 s
vektorisiert mit sparse	0.0120 s	0.0257 s
Faktor (ca.)	13213	631

In der vektorisierten Form scheint Octave hier sogar etwas schneller zu sein. Da man aber oft einige nicht vektorisierte Abschnitte in einem Programm hat bzw. es mehr auf die Optimierung der verlinkten BLAS- und LAPACK - Bibliotheken ankommt, sind meiner Erfahrung nach bei einem hinreichend hohen Optimierungsgrad beide Plattformen i. W. gleich effektiv. Eine Vernachlässigung der Vektorisierung hat hingegen unter Octave dramatische Auswirkungen. Leider ist vektorisieren nicht leicht und manchmal auch mit Fehlern verbunden. Ich selbst gehe so vor, dass ich i. d. R. zunächst einen Algorithmus als Schleife programmiere und überprüfe, ob er für kleine Größenordnungen das richtige Ergebnis liefert. Dann versuche ich in der gleichen Funktion eine vektorisierte Variante zu erstellen und ziehe die durch beide Ansätze berechneten Größen voneinander ab. Kommt im Rahmen der Maschinengenauigkeit eine Null heraus und bin ich mit meinen Bemühungen zufrieden, kommentiere ich die Schleife aus und lasse diese als Teil der Dokumentation stehen. Tatsächlich ist vektorisierter Code nämlich oft schwerer zu lesen als eine Schleifenformulierung.

Wir haben also jetzt Schleifen und if-Abfragen kennengelernt und wissen, dass wir Schleifen, wenn immer möglich, vermeiden und durch einen vektorisierten Ansatz ersetzen sollten.

Ich möchte noch einmal kurz auf Zeile 36 in dem Listing auf Seite 24 eingehen. Man könnte denken, der Befehl `sparse` würde die Werte entsprechend den Einträgen in `x`,`y` und `value` setzen; also in etwa `C(x(i),y(i)) = value(i)`. Er macht aber etwas anderes, er addiert die Werte, also eher so etwas wie `C(x(i),y(i)) = C(x(i),y(i)) + value(i)`. Das ist für uns sehr nützlich und wir werden davon noch reichlich Gebrauch machen.

Der Befehl `sparse` in Zeile 36 setzt nicht einfach Elemente, er addiert sie auf. Das ist für uns sehr nützlich!

Es bleibt jetzt noch kurz die kopflastige `while`-Schleife zu erklären – eine fußlastige kennt die Sprache nicht.

```
1  diced = 0;
2  counter = 0;
3  while (diced<6)
4    counter = counter +1;
5    diced = randi(6);
6  end
7  fprintf('Es wurde %d mal gewuerfelt, bis eine 6 kam. \n', counter);
```

`randi` erzeugt eine zufällige Integerzahl von 1 bis zu einem maximalen Wert von hier 6, die Variante ohne *i*, also `rand` ist für uns auch noch wichtig. Diese erzeugt Gleitkommazahlen zwischen 0 und 1. Beide können auch verwendet werden, um direkte ganze Matrizen oder Vektoren von Zufallszahlen zu erzeugen. Es gibt noch die Möglichkeit, Strings per `disp` auszugeben, aber mit `fprintf` und `sprintf` sind die meisten Benutzer, die schon mal C gesehen haben, besser bedient. Diese Befehle sind in MATLAB analog zu C definiert und können leicht adaptiert werden. `fprintf` enthält den Formatstring, in welchem Text steht, ggf. unterbrochen von durch % eingeleitete Platzhalte für Variablen. Diese folgen als letztes Argument. Die häufigsten Formatanweisungen in unserem Kontext sind %d Platzhalter für eine ganze Zahl, %f für eine Gleitkommazahl und %e für eine Gleitkommazahl in wissenschaftlicher Notation. Diese Formatanweisungen können noch mit Angaben zur Ausgabegenauigkeit versehen werden. So sollen bei %4.2f mindestens vier Ziffern und maximal zwei Nachkommastellen ausgegeben werden. Wenn eine Zahl kürzer ist als vier Ziffern, so werden die fehlenden durch Leerzeichen aufgefüllt. Längere Zahlen, wie z. B. 123456.789 werden hingegen vor dem Komma nicht angeschnitten.

1.5 Logische Ausdrücke, Zugriffe und Suchen

Es existieren zahlreiche Befehle bzw. **logische Operatoren** um boolesche Ausdrücke zu bilden, ein Teil davon wird in Tabelle 1.3 dargestellt.

Tabelle 1.3 Logische Operatoren

Kürzel	Befehl	Bedeutung
~	not(A)	Negation
&	and(A,B)	und
|	or(A,B)	oder
	xor(A,B)	exklusives oder
	all(A)	jedes Element wahr
	any(A)	mindestens ein Element wahr
	find(A)	Indizes wahrer Elemente

Fast alle Vergleiche wie <= etc. sind völlig analog zu Programmiersprachen wie C, mit einer Ausnahme:

Es gibt in dem Bereich zwei *Fallen* für Leute, die von anderen Sprachen kommen; gerade weil fast alles identisch ist. Die Negation aber eben nicht. Hier muss ~ genutzt werden und nicht ! . Der andere Punkt ist die Verwendung von && und || . Diese sind hier rein skalaren Ausdrücken vorbehalten, während & und | auch für Matrizen und Vektoren verwendet werden können.

Was wir oft tun müssen, wenn wir mit Octave oder MATLAB FEM-Algorithmen umsetzen, ist, in Vektoren oder Listen spezielle Werte oder Verhältnisse zu finden und zu manipulieren. Eine Anwendung ist die Manipulation von Gitterdaten. Dazu will man z. B. wissen, wo überall ein Wert größer $\frac{1}{2}$ gesetzt wurde. Das ist ein Job für den Befehl `find`. Hier ein kleines Demonstrationsbeispiel:

```
1  DoF = 100000;
2  x = randi(DoF,1,3*DoF);
3  y = randi(DoF,1,3*DoF);
4  v = rand(1,3*DoF);
5  A = sparse(x,y,v,DoF,DoF);
6  [m,n] = find(A>0.5);
7  for i=1:length(m)
8    A(m(i),n(i))=42;
9  end
```

In diesem Beispiel füllen wir die Matrix zufällig mit im Durchschnitt drei Einträgen pro Zeile bzw. Spalte, jeweils mit einem zufälligen Wert zwischen 0 und 1. Anschließend sucht uns der Befehl `find` die Koordinaten derjenigen Einträge heraus, für die eine boolesche Operation wahr ist. Von Zeile 7 bis 9 werden diese mit einer Schleife durch den Wert 42 überschrieben.

Im Netz werden Sie ggf. auf Hinweise stoßen, dass man besser eine sogenannte logische Indizierung statt `find` nutzen sollte. Die Begründung liegt im Bereich der Performance. Ich möchte im FEM-Umfeld vorschlagen, in der Regel bei `find` zu bleiben und auch keine lineare Indizierung zu nutzen. Eine lineare Indizierung, also `index = find(A>0.5);` und dann `A(index) = 42;`, erscheint zunächst sehr performant. Für größere Werte von DoF, wie hier schon bei 100 000, bricht das Programm jedoch mit einer Fehlermeldung ab: `subscript indices must be either positive integers less than 2^31 or logicals`. Dasselbe gilt für logische Indizierungen wie

```
» A=rand(3);A(A>0.5)=42
A =
42.000000 42.000000 42.000000
42.000000 0.458060 42.000000
0.214011 0.057251 42.000000
```

Diese sind sehr performant für kleinere, voll besetzte Matrizen, aber nicht für große, dünn besetzte. Ein weiterer Aspekt besteht sicherlich darin, dass im Beispiel in Zeile 7 bis 9 eine Schleife steht und wir im letzten Abschnitt festgestellt haben, dass Schleifen doch eher schlecht wären. Das Problem ist, dass der Zugriff `A(m,n)`, wie wir in Abschnitt 1.2 gesehen haben, etwas ganz anderes bedeutet als die Schleife oben. Wenn keine lineare Indizierung möglich ist und das Format als Spalten und Zeilen vorliegt, ist die Operation anders oft nicht umzusetzen. Manchmal erreicht man schnellere Ergebnisse, indem man eine neue Sparse-Matrix durch `sparse` erzeugt und mit der vorhandenen durch eine Operation wie + kombiniert.