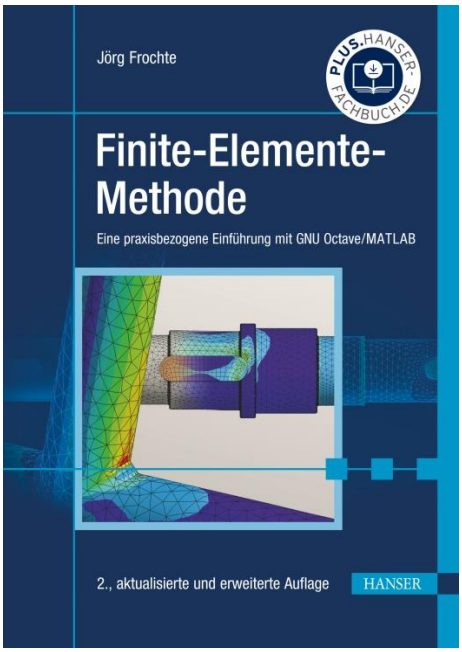
HANSER



Leseprobe

zu

Finite - Elemente - Methode

von Jörg Frochte

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

Weitere Informationen und Bestellungen unter

https://www.hanser-kundencenter.de/fachbuch/artikel/9783446469150

sowie im Buchhandel

Inhalt

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.

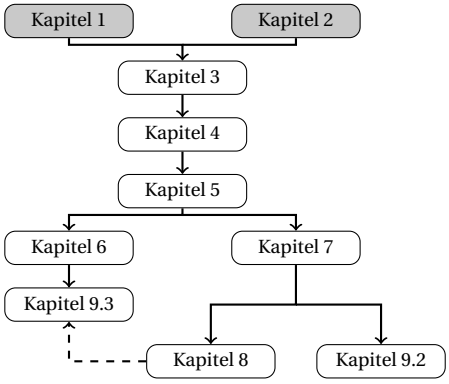


Abbildung 1 Schematischer Aufbau des Buches

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.

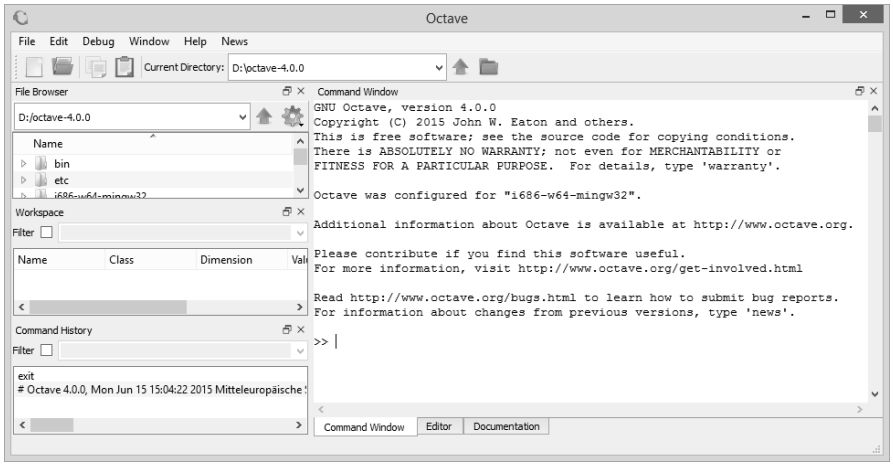


Abbildung 1.1 Screenshot von GNU Octave 4.0

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.

1.1000	1.2000	1.3000	1.4000
2.1000	2.2000	2.3000	2.4000
3.1000	3.2000	3.3000	3.4000

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 $Ax = b$.

Index