Institut für Physikalische Chemie

**Methodenkurs „Anwendungen von Mathematica und Matlab in der Physikalischen Chemie“ im Wintersemester 2016/2017**

Prof. Dr. Stefan Weber, Dr. Till Biskup

— Lösungen zum Aufgabenblatt 3 zum Teil „Matlab“ vom 15.02.2017 —

---

## Vorbemerkung

Die nachfolgend vorgestellten Lösungen wurden (unter MATLAB 2014a) getestet und sollten funktionieren. In den meisten Fällen gibt es aber mehr als eine Lösung, und die hier vorgestellte ist mit großer Wahrscheinlichkeit nicht die eleganteste oder beste und auf keinen Fall die „allein glückselig machende“.

Grundsätzlich gilt: Versuchen Sie immer, den Quellcode anderer Menschen in jedem Detail zu verstehen, bevor Sie ihn einsetzen. Nur so lernen Sie für sich selbst etwas dazu.

## **Aufgabe 3—1** (Präzision numerischer Datentypen)

Reelle Zahlen lassen sich aus nachvollziehbaren Gründen numerisch nicht exakt in Computersystemen darstellen. Schon eine beliebige rationale Zahl lässt sich numerisch nur durch eine unendliche Dezimalbruchentwicklung darstellen, das gleiche gilt für *jede* irrationale Zahl. Die Folge davon ist, dass zur exakten numerischen Darstellung dieser Zahlen unendlich viele Dezimalstellen benötigt würden, was sowohl den verfügbaren Speicherbereich als auch die verfügbare Rechenzeit (beide sind endlich) überschreitet.

Ein Ausweg ist die symbolische Rechnung in Computeralgebrasystemen (wie z.B. Mathematica) und die anschließende Konvertierung eines symbolischen Ausdrucks in einen numerischen Ausdruck mit (nahezu) beliebiger endlicher Genauigkeit. Allerdings lassen sich bei weitem nicht alle mathematischen Probleme symbolisch lösen, schon gar nicht mit den verfügbaren Computeralgebrasystemen. Hier bleibt als einzige Alternative die numerische Berechnung mit der angesprochenen endlichen Genauigkeit.

MATLAB ist darauf optimiert, dass die numerische Genauigkeit des verwendeten Datentyps für Gleitkommazahlen selten zum Problem wird, und das ist auch eine der großen Stärken des Programms. Trotzdem kann man bewusst und gezielt Fehler durch die endliche Genauigkeit der Darstellung reeller Zahlen hervorrufen und sich auf diesem Weg des grundsätzlich vorhandenen Problems anschaulich bewusst werden.

Wie wir wissen, gilt $\sin(\pi) = 0$. Da die Konstante `pi` in MATLAB allerdings nicht exakt der Kreiszahl $\pi$ entspricht, liefert uns MATLAB ein anderes Ergebnis.

**Listing 1: Demonstration der endlichen numerischen Genauigkeit in MATLAB**

```
>> sin(pi) == 0
ans =
     0
>> sin(pi) < eps(1)
ans =
     1
>>
```

---

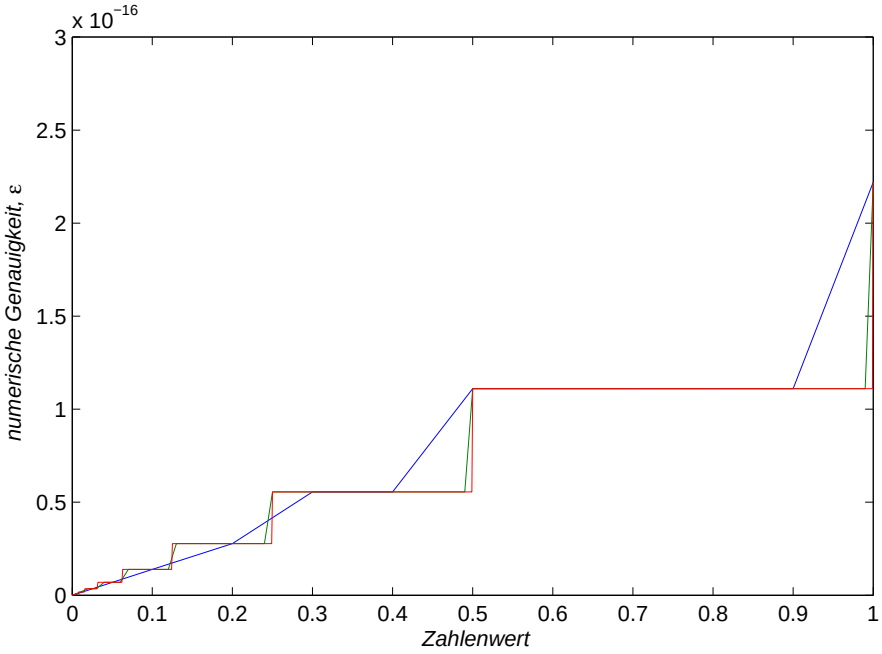


**Abbildung 1: Numerische Genauigkeit der Zahlen im Intervall [0,1].** Dargestellt ist die numerische Genauigkeit $\epsilon$ für die Zahlen im Intervall [0,1] in drei Schrittweiten: 0.1 (blau), 0.01 (grün) und 0.001 (rot). Für den zugrundeliegenden MATLAB-Quellcode vgl. Listing 2.

Der oben gezeigte MATLAB-Code zeigt anschaulich, warum es wegen der endlichen numerischen Genauigkeit oft keine gute Idee ist, über den Identitätsoperator Werte von Zahlen zu vergleichen, die auf unterschiedlichem Weg erzeugt wurden. Die Lösung ist hingegen, den erhaltenen Zahlenwert einer Berechnung daraufhin zu überprüfen, ob er kleiner als die numerische Genauigkeit $\epsilon$ ist.

Die numerische Genauigkeit $\epsilon$ hängt von der Größe der Zahl ab, die man gerade betrachtet. Das liegt daran, dass Mantisse und Exponent getrennt voneinander gespeichert werden.[1] Den jeweiligen Wert kann man in MATLAB mit dem Befehl `eps()` erhalten, der als Argument die betreffende Zahl akzeptiert.

Um anschaulich zu machen, dass die Genauigkeit von der Größe der betrachteten Zahl abhängt, kann man die numerische Genauigkeit von MATLAB für ein gegebenes Intervall grafisch darstellen. Das ist gemäß Aufgabenstellung für das Intervall $[0, 1]$ für drei unterschiedliche Schrittweiten (0,1, 0,01, 0,001) in Abb. 1 dargestellt. Die Abbildung wurde gemäß nachfolgendem Listing erzeugt.

**Listing 2: Grafische Darstellung der numerischen Genauigkeit in MATLAB**

```
x1 = [0:0.1:1];
x2 = [0:0.01:1];
x3 = [0:0.001:1];

plot(x1,eps(x1),x2,eps(x2),x3,eps(x3));
```

Was Sie aus der Abbildung entnehmen können, ist die Tatsache, dass die Speicherung von Gleitkommazahlen gemäß Norm IEEE 754 (mit 64 Bit pro Zahl) zu 15 signifikanten Nachkommastellen führt. Das erkennen Sie daran, dass $10^{-16} < \epsilon(1) < 10^{-15}$ ist.

[1] Wie Gleitkommazahlen gespeichert werden, hängt vom jeweils verwendeten Programm und System ab. MATLAB schafft hier insofern Klarheit, als dass es sich auf allen unterstützten Plattformen an die Norm IEEE 754 (ANSI/IEEE Std 754-1985) hält und pro Gleitkommazahl 64 Bit an Speicher belegt.

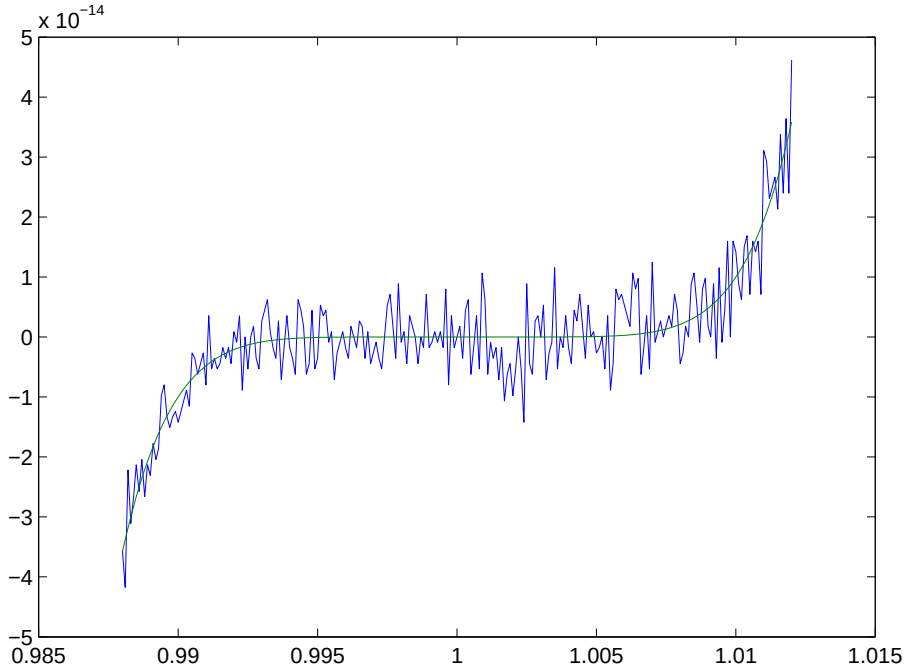


**Abbildung 2: Auswirkung der numerischen Genauigkeit auf die Berechnung eines Polynoms siebter Ordnung.** Dargestellt sind die Funktionen $f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ (blau) und $g(x) = (x-1)^7$ (rot). Das „Rauschen" der blauen Kurve ist die Auswirkung der endlichen numerischen Genauigkeit. Für den zugrundeliegenden MATLAB-Quellcode vgl. Listing 3.

Um das Ergebnis des Rundungsfehlers in MATLAB zu demonstrieren, eignet sich ein Polynom siebter Ordnung recht gut. Das Beispiel ist aus dem online auf den Seiten von MathWorks verfügbaren Buch „Numerical Computing with MATLAB" von Cleve Moler[2] entnommen. Zunächst werden die beiden Polynome

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$
$$g(x) = (x-1)^7$$

in MATLAB definiert und anschließend in derselben Abbildung für $x$ im Intervall $[0.988, 1.012]$ mit einer Schrittweite von 0.0001 dargestellt. Der Code ist in Listing 3 dargestellt, das Ergebnis in Abb. 2.

**Listing 3: Demonstration der Auswirkung der numerischen Genauigkeit in MATLAB**

```
x = [0.988:0.0001:1.012];
f = @(x)x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
g = @(x)(x-1).^7;

plot(x,f(x),x,g(x));
```

Was Sie sehen, ist das Resultat von Rundungsfehlern für das erste Polynom, $f(x)$. Der Grund hierfür ist, dass die Werte für $f(x)$ durch Summen und Differenzen von Zahlen mit einer maximalen Größe von $35 \cdot 1.012^4$ berechnet werden. Das hier verwendete Polynom $f(x)$ hat rein didaktische Bedeutung. Die Auswirkungen der endlichen numerischen Genauigkeit sind hingegen durchaus real, und je nach Anwendung sollte man sich dessen bewusst sein und ggf. sowohl die Ergebnisse überprüfen als auch geeignete Gegenmaßnahmen ergreifen, um Fehler zu vermeiden.[3]

[2] https://de.mathworks.com/moler/index_ncm.html
[3] Details erfahren Sie in Literatur und Vorlesungen zur Numerik.

**Aufgabe 3—2** (Formatierte Ausgabe von Zeichenketten)

MATLAB beherrscht die formatierte Ausgabe von Zeichenketten und lehnt sich bei der dazu verwendeten Syntax eng an die Programmiersprache C an. Die beiden Befehle zur formatierten Ausgabe von Zeichenketten sind `fprintf` für die Ausgabe in eine Datei (oder auf die Kommandozeile) und `sprintf` für die Ausgabe in eine Stringvariable. Sie können sich die Namen der Befehle entsprechend erklären: Der erste Buchstabe („f“ für *file*, „s“ für *string*) gibt an, wohin die formatierte Ausgabe geleitet wird, der letzte Buchstabe steht für *formatted*, also die formatierte Ausgabe.

Diese Möglichkeiten der formatierten Ausgabe von Zeichenketten eignen sich u.a. zur Erzeugung von Datentabellen, sowohl auf der MATLAB-Kommandozeile als auch in Textdateien. Der Aufruf des Befehls `fprintf`, der für die Bearbeitung dieser Aufgabe verwendet wird, ist in seiner Struktur immer gleich:[4]

**Listing 4: Allgemeine Syntax des Befehls `fprintf`**

```
fprintf(<fid>,'<formatstring>',<replacementVariable>)
```

Das erste Argument (`fid`) ist optional, es gibt an, in welche Datei geschrieben werden soll. Die MATLAB-Kommandozeile wird ebenfalls als Datei betrachtet und hat immer die Identifizierungsnummer 1. Wird hier kein Wert angegeben, erfolgt die Ausgabe auf die MATLAB-Kommandozeile.

Das zweite Argument ist eine Zeichenkette mit spezifischen Platzhaltern zur Formatierung. So steht z.B. `%f` für eine Gleitkommazahl, `%s` für eine Zeichenkette und `%i` für eine Ganzzahl. Die Formatierung kann durch zusätzliche Angaben (u.a. Gesamtstellen und Dezimalstellen für Gleitkommazahlen) noch erweitert werden. Darüber hinaus gibt es eine Reihe von Steuerzeichen, die u.a. einen Tabulator (`\t`) oder eine neue Zeile (`\n`) einfügen. Beachten Sie, dass Sie am Ende einer Zeichenkette zur Formatierung zwingend angeben müssen, wenn Sie einen Zeilenumbruch erzeugen möchten (was häufig/meist der Fall ist).[5]

Nach den Angaben zur Formatierung folgt eine Liste von Variablen, deren Inhalt entsprechend verwendet wird, um die Platzhalter (gemäß Formatierung) zu ersetzen. Die Liste der Variablen sollte genauso lang sein wie die Zahl der Platzhalter in der Formatierungsangabe.

Eine einfache Möglichkeit, die Aufgabe entsprechend zu bearbeiten, die beiden gefragten Vektoren zu erzeugen und sie als Tabelle auf der Kommandozeile auszugeben, ist in nachfolgendem Listing gezeigt. Es empfiehlt sich hier dringend, mit einer `for`-Schleife zu arbeiten und über jedes Element zu iterieren.

**Listing 5: Formatierte Ausgabe mit dem Befehl `fprintf`**

```
x=0:0.2:2; y=sin(x);
for line = 1:length(x)
    fprintf('%f %f\n',x(line),y(line));
end
```

Wie Sie sehen, wurde keine Datei für die Ausgabe angegeben, was gleichbedeutend mit einer Ausgabe auf der MATLAB-Kommandozeile ist.[6] Beide Zahlen wurden als Gleitkommazahl ausgegeben, getrennt von einem Leerzeichen, und jede Ausgabe durch einen Zeilenumbruch (`\n`) beendet.

[4] Für Details vgl. die MATLAB-Hilfe, die z.B. über `doc fprintf` aufgerufen werden kann.

[5] Der Grund, dass Befehle wie `fprintf` nicht automatisch für einen Zeilenumbruch sorgen, ist der, dass nur so Ausgaben in einer Zeile schrittweise zusammengesetzt werden können.

[6] Für Unix-Kenner: Die MATLAB-Kommandozeile ist als Standardausgabe (*standard output*, fid = 1) definiert, die Standard-Fehlerausgabe (*error output*, fid = 2) ist ebenfalls nutzbar und führt zur Ausgabe von rotem Text auf der MATLAB-Kommandozeile.

Der hier gezeigte Beispiel-Quellcode ist bewusst einfach gehalten, um das Grundprinzip aufzuzeigen. Sie werden anschließend noch Beispiele für weitere Formatierungsoptionen sehen. Die Ausgabe des obigen Listings 5 auf der MATLAB-Kommandozeile sieht wie nachfolgend gezeigt aus:

**Listing 6: Einfache formatierte Ausgabe einer Datentabelle**

```
0.000000 0.000000
0.200000 0.198669
0.400000 0.389418
0.600000 0.564642
0.800000 0.717356
1.000000 0.841471
1.200000 0.932039
1.400000 0.985450
1.600000 0.999574
1.800000 0.973848
2.000000 0.909297
```

Um die in der Aufgabenstellung gefragte formatierte Ausgabe der Datentabelle wie im Beispiel angegeben zu erzeugen, müssen Sie die Anzahl der Stellen der Gleitkommazahlen explizit kontrollieren. Dafür werden zwischen das Prozentzeichen und das `f` des Formatierungsstrings noch Angaben eingefügt, und zwar, durch einen Punkt voneinander getrennt, die Zahl der insgesamt anzuzeigenden Stellen (inkl. Dezimaltrennzeichen) und die Zahl der Nachkommastellen. Die Spaltenköpfe müssen natürlich nur einmal vor der Schleife ausgegeben werden.

**Listing 7: Formatierte Ausgabe einer Datentabelle mit Spaltenköpfen**

```
fprintf('%s\t%s\n','x','sin(x)');
for line = 1:length(x)
    fprintf('%3.1f\t%11.9f\n',x(line),y(line));
end
```

Noch ein paar Anmerkungen zum Quellcode:

- Sie hätten die Spaltenköpfe auch so angeben können, dass Sie die beiden Zeichenketten „`x`“ und „`sin(x)`“ direkt in die Zeichenkette zur Formatierung setzen: `fprintf('x\tsin(x)\n')`.

  Welche Variante Sie bevorzugen, ist in erster Linie eine Frage des persönlichen Geschmacks. Wollen Sie Zeichenketten ausgeben, die Sie in Variablen vordefiniert haben, bleibt Ihnen allerdings nur die hier angegebene Variante mit Platzhaltern und der Angabe von Variablen.

- Wenn Sie hinter den Tabulator ein Leerzeichen schreiben, wird dieses Leerzeichen zusätzlich ausgegeben. Darauf sollten Sie achten, wenn Sie die Ausgabe entsprechend ausrichten wollen.

### **Aufgabe 3—3** (Schleifen und Entscheidungsstrukturen)

Schleifen (`for`, `while`) und Entscheidungsstrukturen (`if`, `switch`) gehören zu den essentiellen Konstrukten fast jeder Programmiersprache. MATLAB bietet Ihnen zwei verschiedene Arten von Schleifen, die sowohl Gemeinsamkeiten als auch wichtige Unterschiede aufweisen.

Gemeinsam ist beiden Schleifen, `for` und `while`, dass die Bedingung zum Eintritt in die bzw. zur Fortführung der Schleife am Anfang der Schleife überprüft wird. Eine Schleife, die immer mindestens

einmal durchlaufen wird und erst am Ende des Durchlaufs die Bedingung überprüft, gibt es in MATLAB nicht, man kann ein solches Verhalten aber einfach erzeugen, indem man dafür sorgt, dass die Bedingung beim ersten Eintritt in die Schleife erfüllt ist.

Zu den Unterschieden: Bei einer `for`-Schleife ist (in der Regel) vorher bekannt, wie oft die Schleife durchlaufen wird. Die Laufvariable wird entsprechend automatisch bei jedem Durchlauf inkrementiert (oder dekrementiert), und zwar entweder um eins oder um die explizit angegebene Schrittweite. Für eine `while`-Schleife ist hingegen normalerweise nicht bekannt, wie oft sie durchlaufen wird. Darüber hinaus muss der Programmierer selbst dafür sorgen, dass eine eventuell notwendige Laufvariable inkrementiert oder dekrementiert (und vorher auf einen definierten Wert gesetzt) wird. Fehlerhafte Programmierung führt schnell zu einer sogenannten Endlosschleife, die in MATLAB durch die Tastenkombination `Strg+c` unterbrochen werden kann.

Die nachfolgenden Beispiele zeigen Ihnen, dass Schleifen ineinander verschachtelt werden können und innerhalb von Schleifen beliebige andere Befehle auftauchen dürfen, so z.B. Entscheidungsstrukturen.

Eine Einheitsmatrix ist, wie Sie wissen, dadurch definiert, dass alle Elemente außerhalb der Hauptdiagonalen Null sind und auf der Hauptdiagonalen ausschließlich Einsen stehen. Darüber hinaus handelt es sich um eine quadratische Matrix. Eine solche Matrix lässt sich einfach durch zwei ineinander verschachtelte `for`-Schleifen und in der inneren dieser Schleifen eine Entscheidungsstruktur erzeugen. Eine Realisierungsmöglichkeit der Funktion `unityMatrix` ist nachfolgend gezeigt.

**Listing 8: Mögliche Realisierung der Funktion `unityMatrix.m`**

```
1  function result = unityMatrix(dimension)
2
3  result = zeros(dimension);
4
5  for row=1:dimension
6      for col=1:dimension
7          if row==col
8              result(row,col) = 1;
9          end
10     end
11 end
```

In ähnlicher Weise lässt sich auch die Funktion `diagonalMatrix` realisieren, die natürlich ebenfalls Einheitsmatrizen erzeugen kann, wenn man ihr einen Vektor aus lauter Einsen übergibt.

**Listing 9: Mögliche Realisierung der Funktion `diagonalMatrix.m`**

```
1  function result = diagonalMatrix(diagonalElements)
2
3  result = zeros(length(diagonalElements));
4
5  for row=1:length(diagonalElements)
6      for col=1:length(diagonalElements)
7          if row==col
8              result(row,col) = diagonalElements(row);
9          end
10     end
11 end
```

In beiden Fällen wird die Matrix mit Hilfe des Befehls `zeros` in der richtigen Größe vordefiniert und mit Nullen befüllt. übergibt man diesem Befehl nur ein Argument, erzeugt er eine quadratische Matrix. Die Entscheidungsstruktur in der inneren `for`-Schleife muss also nur noch die Bedingung abprüfen, dass Zeile und Spalte identisch sind (die Bedingung für die Hauptdiagonale), und entsprechend das jeweilige Element ändern: entweder durch Eins ersetzen oder durch das Element des Vektors mit Diagonalelementen.

**Hinweis:** Die hier aufgeführten Beispiele dienen lediglich dazu, mit Schleifen bekannt zu werden. Die Verwendung von Schleifen für die elementweise Operation auf Matrizen sollte in MATLAB wann immer möglich vermieden werden, da die alternativen, durch spezielle Funktionen bereitgestellten Matrixoperationen teilweise um ein Vielfaches schneller ausgeführt werden als die Variante mit Schleifen direkt in MATLAB. (Das liegt u.a. daran, dass die Matrixoperationen oftmals direkt auf in Fortran programmierte LAPACK- und BLAS-Routinen zurückgreifen.)

### **Aufgabe 3—4** (Entscheidungsstrukturen)

Ein typischer Einsatzort von Entscheidungsstrukturen mit mehr als zwei Zweigen sind (optionale) Parameter einer Funktion, die innerhalb der Funktion als Schalter fungieren und zu unterschiedlichen Ergebnissen führen. Nachfolgend sind zwei Realisierungen einer hypothetischen Funktion `saveFigure` wiedergegeben, die abhängig von einem Parameter `size` unterschiedlichen Code ausführen. Dabei wird einmal eine `if`-Struktur und einmal eine `switch-case`-Anweisung verwendet.

**Listing 10: Mögliche Realisierung der Funktion `saveFigure.m` mit `if`-Struktur**

```
function saveFigure(size)

if strcmp(size,'large')
    fprintf('Export figure in %s format.\n',size);
elseif strcmp(size,'medium')
    fprintf('Export figure in %s format.\n',size);
elseif strcmp(size,'small')
    fprintf('Export figure in %s format.\n',size);
else
    fprintf('WARNING: Unknown size "%s".\n',size);
end

end
```

**Listing 11: Mögliche Realisierung der Funktion `saveFigure.m` mit `switch-case`-Anweisung**

```
function saveFigure(size)

switch size
    case 'large'
        fprintf('Export figure in %s format.\n',size);
    case 'medium'
        fprintf('Export figure in %s format.\n',size);
    case 'small'
        fprintf('Export figure in %s format.\n',size);
    otherwise
        fprintf('WARNING: Unknown size "%s".\n',size);
end

end
```

Zugegeben ist diese Funktion in ihrem abgebildeten Zustand etwas langweilig, dient sie doch lediglich dem Vergleich der beiden Realisierungen von Entscheidungsmöglichkeiten und erfüllt keine produktiven Aufgaben. Dafür kann sie als leicht verständlicher Prototyp für eigene Entwicklungen dienen.

Wichtig anzumerken ist die Verwendung des Befehls `strcmp` zum Vergleich zweier Zeichenketten. Mit dem Identitätsoperator == kommen Sie hier nicht weiter, da er zeichenweise vergleicht und Fehler ausgibt, wenn die beiden Zeichenketten nicht die gleiche Länge haben.

Um dafür zu sorgen, dass die Abfrage nicht nach Groß- und Kleinschreibung unterscheidet, gibt es für beide Beispiele eine einfache Lösung. Im ersten Beispiel wird am Besten statt `strcmp` der Befehl `strcmpi` eingesetzt, der genau für diesen Zweck existiert, zwei Zeichenketten ohne Berücksichtigung der Groß- und Kleinschreibung zu vergleichen. Im zweiten Beispiel wäre es die einfachste Lösung, den Beginn der `switch`-Anweisung wie folgt umzuschreiben: `switch lower(size)`. Der Befehl `lower` wandelt alle Buchstaben einer ihm übergebenen Zeichenkette in Kleinbuchstaben um.

Eine elegante Variante, unter Verwendung der `switch-case`-Anweisung auch die jeweiligen Anfangsbuchstaben als Option zuzulassen, ist nachfolgend gezeigt.

**Listing 12: Mögliche weitere Realisierung der Funktion `saveFigure.m` mit `switch-case`-Anweisung**

```
1  function saveFigure(size)
2
3  switch lower(size)
4      case {'large','l'}
5          fprintf('Export figure in large format.\n');
6      case {'medium','m'}
7          fprintf('Export figure in medium format.\n');
8      case {'small','s'}
9          fprintf('Export figure in small format.\n');
10     otherwise
11         fprintf('WARNING: Unknown size "%s".\n',size);
12 end
13
14 end
```

Der „Trick“ besteht darin, den einzelnen `case`-Blöcken statt einer Zeichenkette ein `cell array` von Zeichenketten zu übergeben, die alle alternative Optionen darstellen.

Zuletzt noch zu den Grenzen der Verwendung von `switch-case`-Anweisungen. Im Gegensatz zu den Bedingungen in `if`-Strukturen erlauben `case`-Anweisungen keine Evaluierung Boolescher Ausdrücke. Wollten Sie also z.B. in Abhängigkeit der Anzahl der einer Funktion übergebenen Argumente (durch `nargin` repräsentiert) unterschiedliches Verhalten erzeugen und darüber hinaus für einen bestimmten Bereich den gleichen Codeblock ausführen, sind Sie auf `if`-Strukturen angewiesen. Ein Beispiel ist nachfolgend zur Verdeutlichung gezeigt:

**Listing 13: Beispiel einer `if`-Struktur, die nicht über `switch-case`-Anweisungen realisiert werden kann**

```
if nargin > 3
    % Do something
end
```