Institut für Physikalische Chemie

Methodenkurs „Anwendungen von Mathematica und Matlab in der Physikalischen Chemie“ im Sommersemester 2016

Prof. Dr. Stefan Weber, Dr. Till Biskup

— Lösungen zum Aufgabenblatt 2 zum Teil „Matlab“ vom 25.07.2016 —

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 2—1 (Tastaturbelegung des MATLAB-Editors)

Sie finden die Einstellungen zur Tastaturbelegung unter „Preferences“ → „Keyboard“ → „Shortcuts“. Je nach Betriebssystem heißen die Einstellungen unterschiedlich. Sie wollen mit hoher Sicherheit nicht das „Emacs Default Set“ verwenden.

Aufgabe 2—2 (Funktionen des MATLAB-Editors)

Nachfolgend finden Sie den von Fehlern und Warnungen bereinigten Quellcode. Vergleichen Sie ihn mit dem ursprünglichen Quellcode von der Internetseite[1] des Methodenkurses.

Ein paar Anmerkungen zum Quellcode in seiner bereinigten Form:

- Wie Sie sehen, wurde zusätzlich noch die Einrückung automatisch vereinheitlicht. Sie finden diese Funktion im Matlab-Editor im Menü „Text“ → „Smart Indent“ bzw. als Schaltfläche in der Symbolleiste.
- In Zeile drei (Zeile zwei im originalen Quellcode) befand sich ein Fehler, der nicht vom Editor selbst bemerkt wurde, sondern erst durch die versuchte Ausführung auf der Matlab-Kommandozeile. Hier fehlte an einer Stelle der Punkt vor dem Multiplikationszeichen. Im vorliegenden Fall wollen Sie elementweise multiplizieren, da es sich bei „`x`“ um einen Vektor handelt. Hierfür benötigen Sie als Multiplikator „`.*`“ statt einfach nur „`*`“.
- Es wurden minimale Kommentare in den Quellcode eingefügt und die einzelnen, komplett unabhängigen Codeteile optisch durch eine Leerzeile voneinander getrennt.

  Auch wenn Kommentare eigentlich erst in der nächsten Lektion wirklich eingeführt werden, sei hier dieser Vorgriff erlaubt. In MATLAB wird alles, was hinter einem Prozentzeichen erscheint, als Kommentar aufgefasst und von MATLAB bei der Ausführung des Quellcodes ignoriert.

[1] `http://www.till-biskup.de/de/lehre/mathematica-matlab/ss2016/material/04/index`

- Wie Sie sehen können gibt es Befehle, die man sinnvoller Weise mit einem Semikolon abschließt, um die Ausgabe zu unterdrücken (z.B. die Definition der Vektoren in den ersten Zeilen), Befehle, bei denen es egal ist, ob sie mit Semikolon abgeschlossen werden (z.B. der Befehl `plot`), und Befehle, wo Ihnen (zumindest im hier vorliegenden Kontext) daran gelegen ist, die Ausgabe nicht zu unterdrücken und deshalb den Befehl gerade *nicht* mit einem Semikolon abzuschließen (Befehl `toc`) im obigen Beispiel.

- Einen Aspekt, den der Matlab-Editor nicht oder nur in seltenen Fällen bemängelt, sind mehrere durch Semikolon getrennte Definitionen oder Befehle in einer Zeile. Trotzdem erhöht es immens die Übersichtlichkeit, wenn Sie hier die entsprechenden Definitionen jeweils in einer eigenen Zeile vornehmen.

Listing 1: Fehlerbereinigter Quellcode

```
% First plot: Some trigonometric functions
x=1:0.1:40*pi;
y=sin(5*x).*sin(0.1*x).*cos(2*x);
plot(x,y)
xlabel('time');
ylabel('intensity');

% Second plot: Random numbers
tic;
figure();
maxvalue = 100;
set(gca,'XLim',[0 maxvalue]);
set(gca,'YLim',[0 1]);
hold on;
plot([0 maxvalue],[.5 .5])
for k=1:maxvalue
    value = rand;
    if value<0.5
        disp('Lower half');
    else
        disp('Upper half');
    end
    plot(k,value,'rx');
    pause(0.02);
end
hold off
toc

% Third plot: Lissajous
tic;
t = 0:0.01:2*pi;
x = 4 * cos(3*t);
y = 4 * cos(4*t+pi/2);
figure;
hold on;
for k=1:length(x)
    plot(x(k),y(k),'r.');
    pause(0.001);
end
toc
```

Es ginge an dieser Stelle zu weit, Ihnen im Detail zu erklären, was der Quellcode macht. Allerdings sollten Sie das Ergebnis ja gesehen haben, nachdem Sie die Fehler im Quellcode bereinigt hatten. Viele Aspekte werden wir in den nächsten Lektionen noch in mehr Detail behandeln. Hier war das Ziel, einen Quellcode zur Verfügung zu stellen, der sich sehr gut komprimieren lässt und an dem Sie die grundlegenden Fähigkeiten des MATLAB-Editors wie Syntaxhervorhebung, automatische Einrückung und automatische Codeüberprüfung an einem Beispiel selbst erleben können.

Aufgabe 2—3 (Gauß-Funktion)

Nachdem bereits im Rahmen von Aufgabenblatt 1 die Gauß-Funktion

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

als anonyme Funktion definiert wurde, ist die Umwandlung dieser anonymen Funktion in eine Funktion, die in einer separaten Datei ausgelagert ist, wenig aufwendig.

Listing 2: Funktion `gaussian.m`

```
function y = gaussian(x,sigma,mu)

y = 1/sqrt(2*pi*sigma^2).*exp(-(x-mu).^2./(2*sigma^2));

end
```

Die Funktion besitzt die auf dem Aufgabenblatt definierte Schnittstelle, `y` und `x` sind jeweils Vektoren und `sigma` und `mu` Skalare. Beachten Sie, dass Sie diese Funktion *zwingend* in einer Datei unter dem Namen `gaussian.m` speichern müssen. Das ist eine der Einschränkungen von MATLAB hinsichtlich der Definition von Funktionen in separaten Dateien. Es darf nur eine Funktion pro Datei definiert werden (Unterfunktionen sind hiervon ausgenommen, sie können aber nicht von außerhalb der Hauptfunktion aufgerufen werden). Darüber hinaus müssen Datei- und Funktionsname übereinstimmen.

Achten Sie darauf, innerhalb von Funktionen, die Sie in separaten Dateien definieren, alle Zeilen jeweils mit einem Semikolon zu beenden. Anderenfalls wird Ihnen das Ergebnis des jeweiligen Ausdrucks auf der MATLAB-Kommandozeile ausgegeben. Das kann für die Fehlersuche nützlich sein, wird aber zurecht vom MATLAB-Editor mit einer Warnung quittiert, die Sie ernst nehmen sollten. Spätestens bei Projekten, die aus mehr als ein paar Funktionen bestehen, die sich gegenseitig aufrufen, gleicht ansonsten die Suche nach dem vergessenen Semikolon der berühmten Suche nach der Nadel im Heuhaufen – zumal es nicht trivial ist, automatisiert nach einem entsprechenden Zeilenende ohne Semikolon zu suchen, und MATLAB Ihnen keinerlei Information darüber gibt, aus welcher Funktion die Ausgabe auf der Kommandozeile kommt.

Noch eine letzte Anmerkung zur oben definierten Funktion: Zugegeben gibt es keinen Unterschied im Aufruf, wenn Sie die Funktion direkt als anonyme Funktion auf der Kommandozeile definieren, wie Sie das bei der Bearbeitung des vorangegangenen Aufgabenblattes getan haben.

Der große Vorteil der Definition als Funktion in einer entsprechenden Datei ist die viel bessere Wiederverwendbarkeit. Dazu muss der Ordner, in dem sich die Datei befindet, lediglich zum MATLAB-Suchpfad hinzugefügt werden. Ein weiterer Vorteil ist die Dokumentierbarkeit der Funktion. Darauf werden wir später noch einmal zurückkommen.

Die nächste Aufgabe widmet sich einer Funktion, die hinreichend komplex ist, so dass man sie nur noch schwer auf der Kommandozeile definieren kann.

Aufgabe 2—4 (Euler-Drehmatrix)

Die Bedeutung von Koordinatentransformationen in der Physikalischen Chemie (und weit darüber hinaus) kann gar nicht überschätzt werden. Von der großen Zahl unterschiedlicher äquivalenter Konventionen für Drehungen beschränken wir uns hier auf die in der Magnetresonanz verbreitete $zy'z''$-Konvention, bei der zunächst um die z-Achse, dann um die (neue) y'-Achse und anschließend wieder um die (dann aktuelle) z''-Achse gedreht wird. Die Euler-Drehmatrix $\boldsymbol{M}_{zy'z''}$, die Sie als MATLAB-Funktion implementieren sollten, sieht also wie folgt aus:

$$\boldsymbol{M}_{zy'z''} = \begin{pmatrix} -\sin\alpha\sin\gamma + \cos\alpha\cos\beta\cos\gamma & \cos\alpha\sin\gamma + \sin\alpha\cos\beta\cos\gamma & -\sin\beta\cos\gamma \\ -\sin\alpha\cos\gamma - \cos\alpha\cos\beta\sin\gamma & \cos\alpha\cos\gamma - \sin\alpha\cos\beta\sin\gamma & \sin\beta\sin\gamma \\ \cos\alpha\sin\beta & \sin\alpha\sin\beta & \cos\beta \end{pmatrix}$$

Diese mathematische Struktur lässt sich direkt in MATLAB umsetzen – ein schönes Beispiel dafür, dass MATLAB gerade an solchen Stellen aufgrund seiner Sprachstruktur für intuitiven und lesbaren Quellcode sorgt.

Listing 3: Funktion `rotationMatrix.m`

```
function M = rotationMatrix(alpha,beta,gamma)

M = [...
    -sin(alpha)*sin(gamma)+cos(alpha)*cos(beta)*cos(gamma) ...
    cos(alpha)*sin(gamma)+sin(alpha)*cos(beta)*cos(gamma) ...
    -sin(beta)*cos(gamma) ; ...
    %
    -sin(alpha)*cos(gamma)-cos(alpha)*cos(beta)*sin(gamma) ...
    cos(alpha)*cos(gamma)-sin(alpha)*cos(beta)*sin(gamma) ...
    sin(beta)*sin(gamma) ; ...
    %
    cos(alpha)*sin(beta) ...
    sin(alpha)*sin(beta) ...
    cos(beta) ...
    ];

end
```

Wie bei der Spezifikation der Schnittstelle der Funktion in der Aufgabenstellung gefordert, sind die drei Euler-Winkel `alpha`, `beta` und `gamma` jeweils Skalare, der Rückgabeparameter `M` eine Matrix der Dimension 3×3.

Die spezielle Eigenschaft dieser Euler-Drehmatrizen, dass die Transponierte identisch mit ihrer inversen ist (man bezeichnet diese Matrizen auch als orthogonale Matrizen), lässt sich in MATLAB relativ einfach überprüfen:

Listing 4: Überprüfung der Orthogonalität der Euler-Drehmatrix

```
>> M = rotationMatrix(pi/2,pi,0);
>> isequal(inv(M),transpose(M))
ans =
     1
>>
```

Beachten Sie, dass aufgrund der endlichen numerischen Genauigkeit von MATLAB und der numerischen Berechnung jedes einzelnen Elementes der Drehmatrix durch Produkte trigonometrischer Funktionen diese Form der Überprüfung nicht für jeden beliebigen Satz von Euler-Winkeln möglich ist. Das ist nur dann möglich, wenn Sie (mit einem Computeralgebrasystem wie z.B. Mathematica) symbolisch rechnen. Der (vielleicht naive) Ansatz, sich durch die Funktion `rand` drei beliebige Euler-Winkel zu generieren und die Identität auf diese Weise zu überprüfen, wird also in aller Regel nicht zum gewünschten Ergebnis führen.

Die Verwendung des Befehls `isequal` anstelle des Identitätsoperators == in obigem Listing ist der Tatsache geschuldet, dass der Identitätsoperator für jedes Element der beiden Matrizen die Gleichheit überprüft und entsprechend eine Matrix gleicher Dimension mit Booleschen Werten (1 = *true*, 0 = *false*) zurückgibt.

Eine Möglichkeit, die Ausführung der Funktion `rotationMatrix` zu beschleunigen, liegt darin, die Ergebnisse der Evaluierung der beiden trigonometrischen Funktionen für die drei Euler-Winkel am Anfang der Funktion jeweils in einer Variable zu speichern und bei der eigentlichen Definition der Drehmatrix dann nur noch auf diese Variablen zuzugreifen. Dadurch können Sie die Anzahl der Aufrufe der trigonometrischen Funktionen von 29 auf sechs reduzieren, was einer Verringerung um knapp einen Faktor fünf entspricht. Das bedeutet nicht zwangsläufig, dass Ihre Funktion dann fünfmal so schnell ist, aber es wird die Ausführung der Funktion auf jeden Fall beschleunigen.

Der Hintergrund ist der, dass Funktionsaufrufe verglichen mit Variablenersetzungen sehr zeitintensiv sind. Das wird Ihnen in Ihrem momentanen Kontext nicht auffallen können. Wenn Sie sich aber vorstellen, dass Sie die Funktion `rotationMatrix` für die Simulation eines einzelnen Spektrums schnell einige hundertmal und öfter aufrufen müssen, um eine ausreichend feinkörnige Mittelung über alle möglichen Orientierungen eines anisotropen Systems zu erhalten, dann wird schnell klar, wie wichtig derlei Strategien zur Beschleunigung der Ausführung sein können.

Neben der (oben bereits beschriebenen) Überprüfung der Orthogonalität der Euler-Drehmatrix gibt es auch noch weitere einfache Möglichkeiten, sich von der Korrektheit der Definition der Euler-Drehmatrix in der Funktion `rotateMatrix` zu überzeugen. Unterschätzen Sie derlei Tests nicht, schon gar, wenn es sich um so grundlegende und wichtige Funktionen wie die Erzeugung von Drehmatrizen handelt, die Sie in der Regel einmal implementieren und dann sehr oft verwenden.

Ein erster Test wäre, sich einen der drei Einheitsvektoren des kartesischen Koordinatensystems zu definieren (bzw. ggf. auch alle drei), und diesen dann mit der Euler-Drehmatrix um *nur einen* Winkel und um genau 90° zu drehen. Der Vorteil dieser Strategie ist, dass man sich sofort überlegen kann, wie das Ergebnis dieser Drehung aussehen muss.

Da es bei der hier verwendeten $zy'z''$-Konvention keinen Unterschied zwischen der Drehung um z oder z'' gibt, wenn nur um *einen* Winkel gedreht wird, ist die Drehung für α und γ äquivalent und es bleiben nur zwei unterscheidbare Fälle. Nachfolgend ist das Ergebnis der Drehung des entlang der x-Achse liegenden Einheitsvektors um 90° entlang z gezeigt. Wie erwartet kommt der neue Einheitsvektor entlang der y-Achse zu liegen.

Listing 5: Überprüfung der Euler-Drehmatrix: Drehung eines Einheitsvektors

```
>> M = rotationMatrix(pi/2,0,0);
>> a = [1 0 0];
>> a*M
ans =
    0.0000    1.0000         0
>>
```

Wie Sie bei näherer Betrachtung der MATLAB-Ausgabe sehen, sind die ersten beiden Elemente des gedrehten Vektors nur innerhalb der numerischen Genauigkeit gleich Null bzw. Eins. Das dritte Element ist hingegen auch innerhalb der numerischen Genauigkeit exakt Null. Das liegt schlicht daran, dass die dritte Spalte der Euler-Drehmatrix nur von den Winkeln β und γ abhängt, die in obigem Beispiel exakt Null sind und deshalb nicht zur numerischen (Un-)Genauigkeit beitragen. Diese Grenzen der numerischen Genauigkeit machen es allerdings häufig schwierig, mithilfe des Identitätsoperators die Richtigkeit von Ergebnissen automatisiert zu überprüfen. Mehr Details zur numerischen Genauigkeit und dem Umgang damit erfahren Sie in der nächsten Lektion.

Ein weiterer Test wäre die automatische Generierung beliebiger Vektoren, die anschließende Drehung dieser Vektoren entlang einer Raumrichtung um 90° und die Überprüfung der korrekten Drehung mit Hilfe des Skalarproduktes. In die ursprüngliche Aufgabenstellung hatte sich hier allerdings ein Fehler eingeschlichen. Zwei beliebige Vektoren stehen nach einer solchen Drehung nur dann senkrecht aufeinander, wenn die Komponente des jeweiligen Vektors entlang der Drehachse verschwindet.

Ein Skript, das diese Einschränkung berücksichtigt und die Funktion `rotationMatrix` entsprechend überprüft, ist nachfolgend gezeigt.

Listing 6: Skript zum Test der Funktion `rotationMatrix.m`

```
v11 = [rand(1,2) 0];
v12 = [rand(1,2) 0];
v13 = [rand(1,2) 0];
v14 = [rand(1,2) 0];
v15 = [rand(1,2) 0];

v21 = [rand(1) 0 rand(1)];
v22 = [rand(1) 0 rand(1)];
v23 = [rand(1) 0 rand(1)];
v24 = [rand(1) 0 rand(1)];
v25 = [rand(1) 0 rand(1)];

M1 = rotationMatrix(pi/2,0,0);
M2 = rotationMatrix(0,pi/2,0);

dot(v11,v11*M1)
dot(v21,v21*M2)
dot(v12,v12*M1)
dot(v22,v22*M2)
dot(v13,v13*M1)
dot(v23,v23*M2)
dot(v14,v14*M1)
dot(v24,v24*M2)
dot(v15,v15*M1)
dot(v25,v25*M2)
```

Wenn Sie dieses Skript ausführen, wird Ihnen auffallen, dass Ihnen in den seltensten Fällen als Ergebnis des Skalarproduktes wirklich Null angezeigt wird. Meist sind die ausgegebenen Zahlen $\leq 2 \cdot 10^{-16}$. Dabei handelt es sich um die numerische Genauigkeit, mit der MATLAB rechnet. Entsprechend können Sie die Abfrage des Ergebnisses des Skalarproduktes auch nicht einfach dahingegend automatisieren, dass Sie die Identität des Ergebnisses mit Null abfragen.

Eine erweiterte Version des obigen Skriptes zum Test der Funktion `rotationMatrix`, das von diversen Konzepten wie Schleifen und der numerischen Genauigkeit von MATLAB Gebrauch macht, die erst in der

nachfolgenden Lektion im Detail behandelt werden, sei Ihnen an dieser Stelle trotzdem nicht vorenthalten, zumal Sie daran sehen können, dass Sie den Test der Funktion `rotationMatrix` durchaus recht elegant automatisieren können.

Listing 7: Funktion zum Test der Funktion `rotationMatrix.m`

```
function testRotationMatrix(numVectors)

M1 = rotationMatrix(pi/2,0,0);
M2 = rotationMatrix(0,pi/2,0);

for vector = 1:numVectors
    zVector = [rand(1,2),0];
    yVector = [rand(1) 0 rand(1)];
    testIdentity(zVector,M1)
    testIdentity(yVector,M2)
end

end

function result = testIdentity(vector,rotationMatrix)

result = (dot(vector,vector*rotationMatrix) < eps(1));

end
```

Diese Testfunktion basiert auf einer Reihe von Konzepten, die nachfolgend kurz erwähnt werden sollen:

- Verwendung einer `for`-Schleife zur eleganten Generierung der Vektoren und zum gleichzeitigen Abtesten der korrekten Drehung.

  Schleifen werden in der nachfolgenden Lektion eingeführt.

- Verwendung des Befehls `rand` zur Erzeugung von (Pseudo-)Zufallszahlen für die Elemente der Vektoren.

  Hierbei ist darauf zu achten, dass die Elemente entlang der jeweiligen Drehmatrix einen Wert von Null aufweisen müssen, da anderenfalls keine Drehung des Vektors um 90° durchgeführt wird.

- Verwendung einer Unterfunktion zum Test der Bedingung.

  Die Bedingung für eine korrekte Drehung ist, dass das Skalarprodukt verschwindet bzw. kleiner als die numerische Genauigkeit ist.

- Vergleich des Ergebnisses des Skalarprodukts mit der numerischen Genauigkeit.

  Wie bereits oben erwähnt, erlaubt die endliche numerische Genauigkeit nicht die einfache Überprüfung durch die Abfrage der Identität des Ergebnisses mit Null. Stattdessen muss abgefragt werden, ob das Ergebnis kleiner als die numerische Genauigkeit ist, was bedeutet, dass es nicht von Null unterschieden werden kann. Diese Abfrage wird mit Hilfe des Befehls `eps` (für ϵ) realisiert.

  Die numerische Genauigkeit wird detaillierter in der nächsten Lektion und auf dem nächsten Aufgabenblatt thematisiert.

- Übergabe der Anzahl an zu testenden Vektoren als Argument an die Testfunktion.

  Der Aufruf der Funktion erfolgt entsprechend, z.B.: `testRotationMatrix(5)`.