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 1 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 1—1 (MATLAB starten)

Sie können sich als Studierende der Albert-Ludwigs-Universität Freiburg über Ihre universitäre Email-Adresse kostenlos eine (persönliche) MATLAB-Lizenz erwerben und MATLAB auf Ihrem privaten Rechner installieren. Alle dazu notwendigen Informationen finden Sie auf den Seiten des Rechenzentrums[1].

Aufgabe 1—2 (Grundrechenarten und -Regeln)

Hinweis: Die kompakte Schreibweise der nachfolgenden Listings wurde durch Eingabe des Befehls „`format compact`“ erzeugt, um Platz zu sparen. „>>“ repräsentiert die Eingabe in der MATLAB-Kommandozeile.

Listing 1: Grundrechenarten in MATLAB – Ein- und Ausgabe auf der Kommandozeile

```
>> 1+5*3
ans =
    16
>> 3^2*5+2
ans =
    47
>> 1+3-5*2+3/5
ans =
   -5.4000
>> (5+7)*(5-7)
ans =
   -24
>>
```

Beachten Sie, dass Sie bei MATLAB den Multiplikationsoperator immer explizit ausschreiben müssen.

[1] `https://www.rz.uni-freiburg.de/services/beschaffung/software/matlab-landeslizenz`

Aufgabe 1—3 (Vektor- und Matrizenmultiplikation)

Hinweis: Beachten Sie, dass MATLAB für Vektoren und Matrizen den Multiplikationsoperator „$*$“ immer als Matrixmultiplikation auffasst. Deshalb hängt das Ergebnis entscheidend davon ab, welche Struktur die beiden zu multiplizierenden Vektoren bzw. Matrizen haben.

Zunächst werden die beiden Vektoren definiert. Beide Vektoren sind als Zeilenvektoren (eine Zeile, je drei Spalten) definiert. Beachten Sie, dass Sie die Elemente der Vektoren entweder durch Leerzeichen oder durch Kommata voneinander trennen können. Sie können für die bessere Übersichtlichkeit ggf. auch beide kombinieren. MATLAB ist es egal, ob Sie ein oder mehrere Leerzeichen angeben.

Listing 2: Definition der Vektoren in MATLAB

```
>> a = [1 2 3];
>> b = [-7,5,2];
```

Für das **Skalarprodukt** $c = \boldsymbol{a}\cdot\boldsymbol{b}$ der beiden Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$ stehen Ihnen in MATLAB zwei Möglichkeiten zur Verfügung, zumal das Skalarprodukt auch als Spezialfall der Matrizenmultiplikation aufgefasst werden kann. Beide Möglichkeiten sind nachfolgend gezeigt. Das Hochkomma (`'`) transponiert den Vektor. Beachten Sie, dass dieses Zeichen formal den Vektor bzw. die Matrix adjungiert. Für das reine Transponieren müssten Sie den Operator `.'` verwenden. Für reelle Vektoren und Matrizen besteht allerdings mathematisch kein Unterschied.

Wie Sie sehen, bringt die Verwendung des Befehls `dot` (mindestens) zwei Vorteile mit sich: Zum einen müssen Sie sich nicht darum kümmern, ob Sie Zeilen- oder Spaltenvektoren haben, solange die Zahl der Elemente identisch ist. Zum anderen ist aus dem Quellcode sofort ersichtlich, welche Operation durchgeführt wird.

Listing 3: Skalarprodukt zweier Vektoren in MATLAB

```
>> a*b'
ans =
     9
>> c = dot(a,b)
c =
     9
>>
```

Für die Berechnung des **Kreuzproduktes** $\boldsymbol{d} = \boldsymbol{a}\times\boldsymbol{b}$ der beiden Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$ steht Ihnen in MATLAB der Befehl `cross` zur Verfügung.

Listing 4: Kreuzprodukt zweier Vektoren in MATLAB

```
>> d = cross(a,b)
d =
   -11   -23    19
>>
```

Wie Sie wissen, erzeugt das Kreuzprodukt einen Vektor, der senkrecht auf den beiden multiplizierten Vektoren steht. Die Definition des Skalarprodukts, $\boldsymbol{a}\cdot\boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}|\cos\sphericalangle(\boldsymbol{a},\boldsymbol{b})$, lässt sich sehr einfach nutzen, um das zu bestätigen: Das Skalarprodukt der betreffenden Vektoren muss jeweils Null zurückgeben, da der Kosinus von 90° ja gerade Null ist.

Die **Überprüfung des Kreuzproduktes** über die Orthogonalität von $\boldsymbol{d}$ bezüglich der beiden ursprünglichen Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$ lässt sich in MATLAB elegant wie folgt durchführen:

Listing 5: Überprüfung des Kreuzproduktes zweier Vektoren in MATLAB

```
>> dot(d,a)
ans =
     0
>> dot(d,b)
ans =
     0
>>
```

Wie schon bei den beiden möglichen Schreibweisen des Skalarproduktes in MATLAB weiter oben erwähnt, bietet die Verwendung des Befehls `dot` den Vorteil, dass man aus dem Code sofort auch ohne Kenntnis der Dimensionen der beteiligten Vektoren entnehmen kann, welche Operation hier auf den beiden Vektoren durchgeführt wird.

Dass das anderenfalls nicht möglich ist, zeigt Ihnen die Berechnung der beiden Produkte $\boldsymbol{a} \cdot \boldsymbol{b}^\top$ und $\boldsymbol{a}^\top \cdot \boldsymbol{b}$ mittels **Matrixmultiplikation** in MATLAB:

Listing 6: Inneres und äußeres Produkt zweier Vektoren in MATLAB

```
>> a*b'
ans =
     9
>> a'*b
ans =
    -7     5     2
   -14    10     4
   -21    15     6
>>
```

Im ersten Fall haben wir es mit dem Skalarprodukt (auch „inneres Produkt“ genannt) zu tun, im zweiteren Fall mit dem Matrixprodukt („äußeres Produkt“).

Der Versuch, die beiden Zeilenvektoren miteinander zu multiplizieren, führt zu folgender Fehlermeldung:

Listing 7: Fehlermeldung bei der Multiplikation zweier Zeilenvektoren in MATLAB

```
>> a*b
Error using  *
Inner matrix dimensions must agree.
>>
```

Der Grund dafür ist, dass das Matrixprodukt zweier Zeilen- oder Spaltenvektoren mathematisch nicht definiert ist.

Aufgabe 1—4 (Spezielle Matrizen)

MATLAB bringt eine Reihe von Funktionen mit, mit denen spezielle Matrizen erzeugt werden können. Einen Überblick finden Sie in der Dokumentation.

Die hier verwendeten speziellen Matrizen haben insbesondere aus Programmierersicht eine Bedeutung.

Eine **Matrix mit Zufallszahlen** wird in MATLAB über den Befehl `rand` erzeugt. Es handelt sich dabei um Pseudozufallszahlen (Gleitkommazahlen), die im Intervall $[0, 1]$ gleichverteilt sind. Weitere Befehle in MATLAB für Zufallszahlen sind `randi` und `randn`. Details dazu finden Sie in der Dokumentation.

Listing 8: Matrix mit Zufallszahlen in MATLAB

```
>> rand(3,5)
ans =
    0.8147    0.9134    0.2785    0.9649    0.9572
    0.9058    0.6324    0.5469    0.1576    0.4854
    0.1270    0.0975    0.9575    0.9706    0.8003
>>
```

Hinweis: Die Zahlen werden in Ihrem Fall selbstverständlich anders aussehen. Sie können allerdings einmal ausprobieren, was passiert, wenn Sie MATLAB neu starten und den Befehl erneut ausführen. Das liefert Ihnen Hinweise darauf, wann und wie der von MATLAB verwendete Pseudozufallszahlengenerator initialisiert wird.

Eine mögliche Verwendung von Zufallszahlen und entsprechend des Befehls `rand` in MATLAB ist das Hinzufügen von „Rauschen“ zu berechneten Datenpunkten. Das ist weniger zum Fälschen von Experimenten nützlich[2] als zum Test von Auswertungsroutinen, insbesondere hinsichtlich ihrer Robustheit gegenüber verrauschten Daten.

Genauso wie man eine Matrix (nahezu) beliebiger Dimensionen mit Pseudozufallszahlen erzeugen kann, erlaubt MATLAB über die beiden Befehle `zeros` und `ones` die Erzeugung von **Matrizen aus Nullen respektive Einsen**.

Listing 9: Matrix mit Nullen bzw. Einsen in MATLAB

```
>> zeros(3,5)
ans =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
>> ones(3,5)
ans =
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
>>
```

Eine große Bedeutung dieser beiden Befehle ist die elegante und einfache Vordefinition von Matrizen entsprechender Größe, was die Ausführung von Schleifen, die auf den Elementen dieser Matrizen arbeiten, immens beschleunigt. Das liegt daran, dass das Vergrößern einer Variable (bezüglich des Speicherplatzes, den sie belegt) ein auf Betriebssystemebene vergleichsweise aufwändiger Prozess ist, zumal zusätzlicher Speicher angefragt und zugewiesen werden muss. Bei jedem Schleifendurchlauf eine solche Operation durchzuführen ist also wesentlich zeitaufwändiger, als die Variable einmal vorweg in ihrer endgültigen Größe zu definieren.

Wie Sie später noch sehen werden, gibt MATLAB in seinem Editor eine entsprechende Warnung aus, wenn sich die Größe einer Variablen in einer Schleife mit jedem Durchlauf verändert:

[2] Da die von `rand` zurückgegeben Zahlen nur Pseudozufallszahlen und keine wirklich zufälligen Zahlen sind, ist – ggf. mit etwas Aufwand – nachweisbar, ob das Rauschen eines Vektors mit Datenpunkten mit einem Pseudozufallszahlengenerator erzeugt wurde oder reales experimentelles Rauschen ist. Wollen Sie wirklich Daten fälschen, sollten Sie also schlauer vorgehen.

The variable <variablename> appears to change size on every loop iteration.
Consider preallocating for speed.

Die Lösung dieses Problems liegt genau darin, die betreffende Variable vorher in entsprechender Größe zu definieren („*preallocate*“). Allerdings ist das nicht immer möglich, wie Sie ebenfalls noch sehen werden. In einem solchen Fall kann die Warnung im MATLAB-Editor auch unterdrückt werden.

Die **Einheitsmatrix** ist eine quadratische Matrix, auf deren Diagonale Einsen stehen und deren restliche Elemente Nullen sind. Die mathematische Bedeutung der Einheitsmatrix liegt darin, dass Sie beliebige Matrizen oder Vektoren entsprechend passender Dimensionen mit der Einheitsmatrix multiplizieren können und wieder die identische Matrix bzw. den identischen Vektor erhalten.

Der entsprechende Befehl in MATLAB lautet `eye`. Es handelt sich hier um ein Wortspiel, zumal „eye“ im Englischen so ausgesprochen wird wie „I“ (das große „i“) – die Einheitsmatrix wird im englischen Sprachraum häufig mit $\mathbb{I}$ abgekürzt. Anfangs unterschied MATLAB offensichtlich noch nicht zwischen Groß- und Kleinschreibung, und das „`i`“ war bereits für die Definition komplexer Zahlen reserviert.

Listing 10: Einheitsmatrix in MATLAB

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
>>
```

Letztlich ist die Einheitsmatrix eine Spezialform der **Diagonalmatrix**. Auch bei den Diagonalmatrizen handelt es sich um quadratische Matrizen, bei denen alle Elemente außer diejenigen der Hauptdiagonale Nullen sind.

In MATLAB gibt es eine denkbar einfache Möglichkeit, eine Diagonalmatrix mit quasi beliebigen Elementen auf der Diagonalen zu erzeugen: `diag`. Um also eine Diagonalmatrix zu erzeugen, deren Elemente durch den Vektor $\boldsymbol{x} = (1, 3, 5, 7, 6, 4, 2)$ vorgegeben sind, könnte man in MATLAB wie folgt vorgehen:

Listing 11: Diagonalmatrix mit vorgegebenen Elementen in MATLAB

```
>> diag([1,3,5,7,6,4,2])
ans =
     1     0     0     0     0     0     0
     0     3     0     0     0     0     0
     0     0     5     0     0     0     0
     0     0     0     7     0     0     0
     0     0     0     0     6     0     0
     0     0     0     0     0     4     0
     0     0     0     0     0     0     2
>>
```

Natürlich hätte man sich auch zunächst den Vektor $\boldsymbol{x}$ definieren können und dann den Befehl `diag(x)` aufrufen. Ebenfalls hätte man die einzelnen Elemente des Vektors statt durch Kommata auch durch (ein oder mehrere) Leerzeichen trennen können. Darüber hinaus ist es MATLAB egal, ob Sie dem Befehl `diag` einen Zeilen- oder Spaltenvektor übergeben. Das Ergebnis ist immer das gleiche.

Aufgabe 1—5 (Operationen auf Matrizen)

Aufgrund seines Ursprungs in der linearen Algebra bringt MATLAB eine ganze Reihe an Funktionen mit, die auf Matrizen arbeiten und die für die lineare Algebra wichtige Aspekte abdecken. Ein kleiner Teil davon wurde in dieser Aufgabe vorgestellt und benutzt.

Zunächst müssen die beiden Matrizen ***M*** und ***N*** definiert werden. In MATLAB werden Matrizen im Normalfall zeilenweise definiert. Dabei kommt die gleiche Definition zum Zuge wie für Vektoren, also mit eckigen Klammern und der Trennung der einzelnen Elemente einer Zeile mit Kommata oder Leerzeichen. Für die Trennung der Spalten findet das Semikolon Verwendung.

Listing 12: Definition zweidimensionaler Matrizen mit vorgegebenen Elementen in MATLAB

```
>> M = [8 1 3; 5 8 5; 5 6 4];
>> N = [2 0; 4 8];
```

Bei der Definition in einer Datei (Skript oder Funktion, in der nächsten Lektion dazu mehr) gibt es auch noch alternative Möglichkeiten der Definition einer Matrix, die näher an deren mathematischer Struktur ist. Als kleiner Vorgriff sei das nachfolgend am Beispiel der 3×3-Matrix ***M*** gezeigt.

Listing 13: Alternative Definition zweidimensionaler Matrizen mit vorgegebenen Elementen in MATLAB

```
M = [ ...
    8 1 3; ...
    5 8 5; ...
    5 6 4  ...
    ];
```

Beachten Sie die Verwendung dreier Punkte, um MATLAB zu zeigen, dass die Definition des Befehls in der nächsten Zeile fortgesetzt wird.

Für die Berechnung der **Diagonale**, der **Spur**, der **Determinante** und der **Eigenwerte einer Matrix** stellt MATLAB jeweils entsprechende Befehle zur Verfügung: `diag` für die Diagonale, `trace` für die Spur, `det` für die Determinante und `eig` bzw. `eigs` für die Eigenwerte.

Listing 14: Diagonale, Spur, Determinante und Eigenwerte einer Matrix in MATLAB

```
>> diag(M)
ans =
     8
     8
     4
>> trace(M)
ans =
    20
>> det(M)
ans =
   -9.0000
>> eig(M)
ans =
   14.6243
    5.4878
   -0.1121
>>
```

Beachten Sie, dass Sie über den Befehl `eig` nicht nur die Eigenwerte, sondern auch die zugehörigen **Eigenvektoren einer Matrix** erhalten, indem Sie den Befehl mit zwei Rückgabeparametern aufrufen:

Listing 15: Eigenwerte und Eigenvektoren einer Matrix in MATLAB

```
>> [eigenVectors,eigenValues]=eig(M)
eigenVectors =
   -0.3730   -0.6755   -0.2813
   -0.7217    0.6495   -0.3719
   -0.5831    0.3492    0.8846
eigenValues =
   14.6243         0         0
         0    5.4878         0
         0         0   -0.1121
>> diag(eigenValues)
ans =
   14.6243
    5.4878
   -0.1121
>>
```

Wie Sie sehen, ist das zweite Rückgabeargument (die Eigenwerte) ebenfalls eine Matrix, allerdings eine Diagonalmatrix, auf deren Diagonale die Eigenwerte stehen. Um auch beim Aufruf des Befehls `eig` wieder an die Eigenwerte als Vektor heranzukommen, können Sie sich der Funktion `diag` bedienen.

Die Eigenvektoren sind die Spalten der Matrix, die Sie als erstes Rückgabeargument des Befehls `eig` beim Aufruf mit zwei Rückgabeargumenten bekommen.

Auch für das **Kronecker-Tensorprodukt** (oder kurz Kronecker-Produkt), das normalerweise mit dem Symbol $\otimes$ abgekürzt wird, gibt es in MATLAB einen eigenen Befehl: `kron`.

Listing 16: Kronecker-Tensorprodukt zweier Matrizen in MATLAB

```
>> kron(M,N)
ans =
    16     0     2     0     6     0
    32    64     4     8    12    24
    10     0    16     0    10     0
    20    40    32    64    20    40
    10     0    12     0     8     0
    20    40    24    48    16    32
>> kron(N,M)
ans =
    16     2     6     0     0     0
    10    16    10     0     0     0
    10    12     8     0     0     0
    32     4    12    64     8    24
    20    32    20    40    64    40
    20    24    16    40    48    32
>>
```

Zur Beachtung: Das Kronecker-Tensorprodukt ist nicht nur für quadratische Matrizen definiert. Allgemein ist das Kronecker-Tensorprodukt zweier Matrizen $\boldsymbol{A}$ und $\boldsymbol{B}$, $\boldsymbol{A} \otimes \boldsymbol{B}$, die Multiplikation von $\boldsymbol{B}$ an jedes Element von $\boldsymbol{A}$. Ist $\boldsymbol{A}$ eine $m \times n$-Matrix und $\boldsymbol{B}$ eine $p \times q$-Matrix, dann hat die resultierende Matrix die Dimension $mp \times nq$.

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & a_{12}\boldsymbol{B} & \cdots & a_{1n}\boldsymbol{B} \\ a_{21}\boldsymbol{B} & a_{22}\boldsymbol{B} & \cdots & a_{2n}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & a_{m2}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{bmatrix}$$

Anwendung findet das Kronecker-Produkt u.a. in der Theorie zur magnetischen Resonanz, wenn Sie einen Spin-Hamilton-Operator für mehr als einen Spin aufstellen wollen. Hier werden die Pauli-Spinmatrizen entsprechend über das Kronecker-Produkt multipliziert. Da sich mit jedem weiteren hinzugenommenen Spin die Dimension der resultierenden Matrix verdoppelt, können Sie schnell ersehen, warum es aus Gründen des Speicherplatzes immer noch extrem schwierig ist, viel mehr als zwölf Spins zu rechnen[3].

Aufgabe 1—6 (Komplexe Zahlen)

MATLAB beherrscht von Hause aus die Arbeit mit komplexen Zahlen. Zur Definition des Imaginärteils wird der Buchstabe `i` verwendet.[4] Die Definition der beiden gegebenen komplexen Zahlen z_1 und z_2 sieht in MATLAB also wie folgt aus:

Listing 17: Definition komplexer Zahlen in MATLAB

```
>> z1 = 11-3i
z1 =
  11.0000 - 3.0000i
>> z2 = sqrt(2)*(cos(pi/4)+i*sin(pi/4))
z2 =
   1.0000 + 1.0000i
>>
```

Beachten Sie, dass Sie im Normalfall das imaginäre i auch ohne Multiplikationsoperator direkt an die Zahl anschließen können. Lediglich in den Fällen, in denen das zu Kollisionen mit nachfolgenden Befehlen führt (wie im Beispiel der Definition der Zahl z_2), muss der Multiplikationsoperator explizit angegeben werden.

Die **Differenz der beiden komplexen Zahlen** z_1 und z_2, gefragt war nach $(z_2 - z_1)$, berechnet sich in MATLAB wie von reellen Zahlen gewohnt und intuitiv:

Listing 18: Differenz zweier komplexer Zahlen in MATLAB

```
>> z2-z1
ans =
 -10.0000 + 4.0000i
>>
```

Auch der Nachweis der in Aufgabenteil b gegebenen **Identität** lässt sich mit MATLAB in gewohnter Weise bewerkstelligen. Entweder Sie geben nur den Term auf der linken Seite des Gleichheitszeichens ein und vergleichen das Ergebnis mit der Aufgabenstellung, oder Sie verwenden den Identitätsoperator == und sehen am Booleschen Rückgabewert, dass die Beziehung korrekt ist.

[3] Das gäbe eine Matrix-Dimension von $2^{12} = 4096$ und entsprechend $4096^2 = 16777216$, also mehr als 16 Millionen Elemente der Matrix. Wenn jedes Element, wie in MATLAB, als Gleitkommazahl doppelter Genauigkeit gespeichert wird, verbraucht es 64 Bit, also 8 Byte. Sie kommen also auf einen Speicherbedarf für diese Matrix von mehr als 10 GB

[4] Da MATLAB auch im Bereich der Elektrotechnik weite Verbreitung findet, ist die dort übliche alternative Schreibweise mit dem Buchstaben `j` ebenfalls möglich und äquivalent.

Listing 19: Arithmetik komplexer Zahlen in MATLAB

```
>> ((1+i)/(1-i))^2
ans =
    -1
>> ((1+i)/(1-i))^2 == -1
ans =
     1
>>
```

Beachten Sie, dass der Boolesche Rückgabewert im zweiten Fall entweder 1 (*true*) oder 0 (*false*) ist. Details zu Booleschen Variablen erfahren Sie in einer späteren Lektion.

Auch zur Ausgabe des **Real- bzw. Imaginärteils einer komplexen Zahl** gibt es in MATLAB Befehle: `real` und `imag`. Entsprechend berechnet sich der Real- und Imaginärteil der Zahl $z_3 = (1 - \mathrm{i})^5$ in MATLAB:

Listing 20: Ausgabe des Real- bzw. Imaginärteils einer komplexen in MATLAB

```
>> z3=(1-i)^5
z3 =
  -4.0000 + 4.0000i
>> real(z3)
ans =
    -4
>> imag(z3)
ans =
     4
>>
```

Aufgabe 1—7 (Anonyme Funktionen)

Häufig ist es sehr nützlich, komplexere mathematische Formeln als „anonyme Funktionen“ in MATLAB zu definieren. Diese Funktionen können Sie in Abhängigkeit mehrerer Parameter definieren. Gefragt war in der Aufgabe nach der Gauß-Funktion

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

Eine entsprechende Definition dieser Funktion in MATLAB in Abhängigkeit der drei Parameter x, σ, μ könnte wie folgt aussehen:

Listing 21: Definition der Gauß-Funktion als anonyme Funktion in MATLAB

```
>> gaussian = @(x,mu,sigma)1/sqrt(2*pi*sigma^2).*exp(-(x-mu).^2/(2*sigma^2));
```

Wie Sie sehen, wird die Definition einer anonymen Funktion durch den „Klammeraffen“ (@), gefolgt von einem Paar runder Klammern mit der Liste durch Kommata getrennter Parameter, eingeleitet. Der Variablenname, dem Sie diese anonyme Funktion zuweisen, kann – im Rahmen der Einschränkungen von MATLAB für Variablennamen – frei gewählt werden. Es empfiehlt sich, sprechende („sinnvolle“) Namen zu verwenden.

Zur Beachtung: Da es sich bei x um die abhängige Variable handelt, also in MATLAB am Ende um einen Vektor, müssen Sie an den entsprechenden Stellen darauf achten, elementweise Operationen auszuführen.

Das gilt sowohl für die Multiplikation des Exponentialausdrucks mit dem Vorfaktor als auch für die elementweise Quadrierung des Zählers im Exponentialausdruck.

Weiterhin empfiehlt es sich, kurze, aber sprechende Namen für die Variablen zu verwenden. Schreiben Sie ggf., wie im obigen Beispiel, die griechischen Buchstaben als `mu` und `sigma` aus (und beachten Sie dabei, keine Sonderzeichen wie Umlaute zu verwenden, da MATLAB das nicht unterstützt).

Um eine Gauß-Kurve darzustellen – und sich gleichzeitig davon zu überzeugen, dass bei der Definition der anonymen Funktion in MATLAB nichts schief gelaufen ist –, verwenden Sie den Befehl `plot` in MATLAB. Dafür müssen Sie zunächst den Vektor $\boldsymbol{x}$ mit geeigneter Schrittweite definieren.

Listing 22: Darstellung der Gauß-Funktion, definiert als anonyme Funktion, in MATLAB

```
>> x=[-5:.1:5];
>> plot(x,gaussian(x,0,1))
```

Hier wurde der Vektor $\boldsymbol{x}$ im vorgegebenen Intervall $-5 \leq \boldsymbol{x} \leq 5$ mit einer Schrittweite von 0.1 definiert. Der `plot`-Befehl akzeptiert in seiner einfachen, hier verwendeten Form zwei Vektoren gleicher Länge ($\boldsymbol{x}$ und $\boldsymbol{y}$) und stellt den zweiten als Funktion des ersten dar.

Um nun alle drei Gauß-Kurven (für die drei Werte von σ) in einer Abbildung erscheinen zu lassen, gibt es mehrere Möglichkeiten. Entweder Sie übergeben dem `plot`-Befehl drei Paare von $\boldsymbol{x}$- und $\boldsymbol{y}$-Vektoren, oder aber Sie behelfen sich mit den Befehlen `hold on` und `hold off`. Der Hintergrund ist der, dass der Befehl `plot` in MATLAB immer die Inhalte des aktuellen Grafikfensters löscht und danach neu nur die ihm übergebenen Vektoren zeichnet.

Listing 23: Plot dreier Gauß-Kurven mit einem `plot`-Befehl in MATLAB

```
>> plot(x,gaussian(x,0,1),x,gaussian(x,0,2),x,gaussian(x,0,3))
```

Das Ergebnis des obigen Befehls sehen Sie in Abb. 1 dargestellt. Die Reihenfolge der Farben entspricht der Standard-Reihenfolge in MATLAB, von Blau über Grün nach Rot. Beachten Sie, dass die Kurven deshalb so glatt dargestellt wurden, weil der Vektor der x-Werte mit einer entsprechend feinen Schrittweite (0.1) definiert wurde.

Alternativ können Sie, wie angesprochen, auch mit den Befehlen `hold on` und `hold off` und drei unabhängigen `plot`-Befehlen arbeiten.

Listing 24: Plot dreier Gauß-Kurven mit drei `plot`-Befehlen in MATLAB

```
>> hold on;
>> plot(x,gaussian(x,0,1));
>> plot(x,gaussian(x,0,2));
>> plot(x,gaussian(x,0,3));
>> hold off;
```

Das Ergebnis unterscheidet sich leicht von dem in Abb. 1 dargestellten, da diesmal alle Linien in derselben Farbe (Blau) dargestellt werden. Das liegt daran, dass MATLAB ein automatisches Durchpermutieren der Linienfarben (und -Stile) nur dann durchführt, wenn Sie innerhalb eines Plot-Befehls mehrere Paare von Vektoren angeben. Um das zu umgehen, können Sie die Linienfarbe (und den Linienstil) manuell, z.B. durch einen dritten (optionalen) Parameter, angeben.

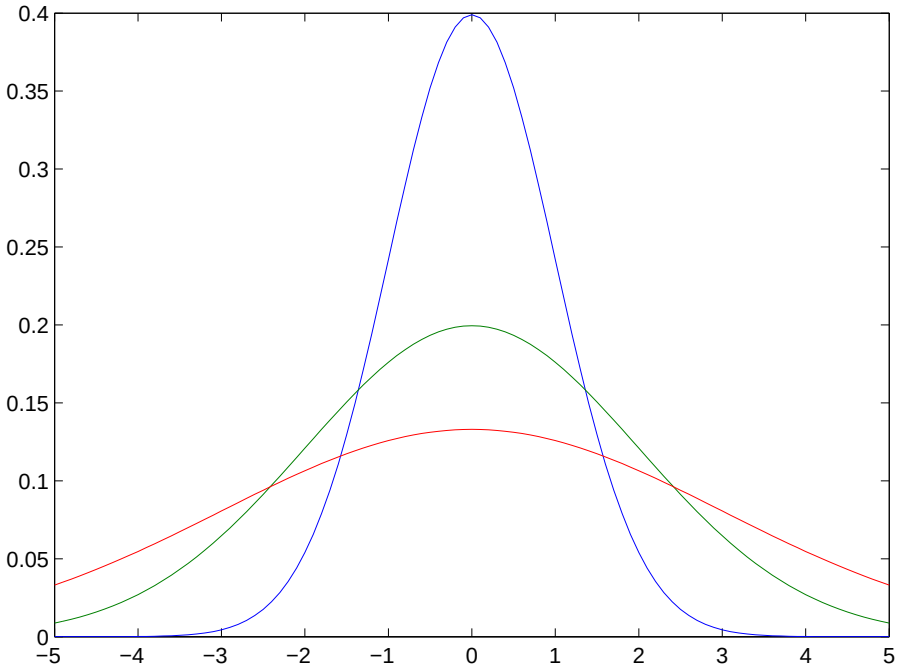

Abbildung 1: Ergebnis der Darstellung dreier Gauß-Kurven mit einem `plot`-Befehl. Dargestellt sind Gauß-Kurven für $\mu = 0$ und $\sigma = 1, 2, 3$. Die Reihenfolge der Farben entspricht der Standard-Reihenfolge in MATLAB, von Blau über Grün nach Rot. Beachten Sie, dass die Kurven deshalb so glatt dargestellt wurden, weil der Vektor der x-Werte mit einer entsprechend feinen Schrittweite (0.1) definiert wurde. Für den zugrundeliegenden MATLAB-Befehl vgl. Listing 23.

Details dazu erfahren Sie in einer späteren Lektion, wenn wir uns intensiver mit Abbildungen in MATLAB beschäftigen, hier sei aber schon einmal ein einfaches Beispiel gezeigt, das zur identischen Darstellung wie in Abb. 1 führt.

Listing 25: Plot dreier Gauß-Kurven mit drei `plot`-Befehlen in MATLAB

```
>> hold on;
>> plot(x,gaussian(x,0,1),'b-');
>> plot(x,gaussian(x,0,2),'g-');
>> plot(x,gaussian(x,0,3),'r-');
>> hold off;
```

Das dritte Argument des `plot`-Befehls ist in diesem Fall eine Zeichenkette, bestehend aus einem Buchstaben für die Definition der Farbe und einem Strich als Kürzel für den Linienstil (hier: durchgezogene Linie). Weitere Details in einer späteren Lektion oder in der MATLAB-Hilfe zum `plot`-Befehl.

Aufgabe 1—8 (Verwendung der MATLAB-Hilfe)

Beide Befehle, `str2num` und `str2double`, können in MATLAB dazu verwendet werden, Zeichenketten in Zahlen umzuwandeln. Der große Unterschied, der sich durch einen Blick in die MATLAB-Dokumentation (`doc str2num` bzw. `doc str2double`) herausfinden lässt, liegt im Vorgehen dieser beiden Befehle. Während für `str2double` die Zeichenketten weitestgehend schon so aussehen müssen wie Zahlen, verwendet `str2num` intern den Befehl `eval` (kurz für *evaluate*), der die Zeichenkette auf Funktions- bzw. Befehlsnamen hin untersucht und diese dann ggf. ausführt.

Der Vorteil von `str2num` gegenüber `str2double` ist, dass sich auch arithmetische Operationen entsprechend durchführen lassen. Der große Nachteil ist allerdings, dass die Evaluierung der Zeichenkette

von den jeweils in der lokalen Umgebung verfügbaren Befehlen und Funktionen abhängt – ggf. auch von vom Nutzer definierten Funktionen[5]. Das kann mitunter zu nicht reproduzierbarem Verhalten und schwer nachvollziehbaren Fehlern führen.

Darüber hinaus ist die Funktion `str2double` wesentlich schneller. Sie sollten also, wenn Sie keine triftigen Gründe haben, immer die Funktion `str2double` verwenden, z.B. wenn Sie – wie wir das in einer späteren Lektion noch machen werden – eine Textdatei mit Zahlenwerten Zeile für Zeile einlesen und die Zeichenketten dann in Zahlen umwandeln wollen.

Wie Sie im nachfolgenden kurzen Beispiel sehen können[6], gibt die Funktion `str2double` nicht etwa einen Fehler zurück, wenn die Umwandlung einer Zeichenkette in eine Zahl nicht funktioniert hat, sondern „`NaN`“, was für „*not a number*“ steht. Sie sollten also die Rückgabe von `str2double` immer darauf überprüfen, ob Sie auch tatsächlich eine Zahl erhalten haben, weil Sie ansonsten böse Überraschungen erleben können.

Listing 26: Beispiele für das unterschiedliche Verhalten von `str2double` und `str2num` in MATLAB

```
>> str2double('3')
ans =
     3
>> str2double('3+5i')
ans =
   3.0000 + 5.0000i
>> str2double('sin(3*pi/2)')
ans =
   NaN
>> str2num('sin(3*pi/2)')
ans =
    -1
>> a = 3;
>> str2num('sin(a*pi/2)')
ans =
     []
>>
```

Wie Sie im letzten Beispiel sehen können, gibt Ihnen – im Gegensatz zu `str2double` – die Funktion `str2num` im Fall einer fehlgeschlagenen Konversion nicht etwa `NaN` zurück, sondern einen leeren Vektor (der wiederum ein numerischer Datentyp, wenn auch ohne Inhalt, ist). Wenn Sie diese Funktion verwenden, müssten Sie die Rückgabe also immer darauf überprüfen, ob Sie einen leeren Vektor bekommen, und ggf. entsprechend handeln.

[5] Da der Befehl `eval` innerhalb der Funktion `str2num` aufgerufen wird, kann er nur auf Funktionen (und Variablen) zugreifen, die im lokalen Kontext der Funktion `str2num` verfügbar sind. Deshalb ist die Definition von inline-Funktionen auf der MATLAB-Kommandozeile kein Problem bei `str2num`, wohl aber Funktionen in eigenen Dateien (dazu mehr in der nächsten Lektion), die im MATLAB-Suchpfad auftauchen.

[6] Hinweis: Die Hochkommata um das Argument in den beiden Funktionen bedeuten, dass es sich um eine Zeichenkette (*string*) handelt.