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 5 zum Teil „Matlab“ vom 26.07.2016 —

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.

Alle für das Nachvollziehen der Lösungen der folgenden Aufgaben notwendigen Dateien werden Ihnen auf der zum Kurs gehörigen Internetseite[1] bereitgestellt. Alle Dateien befinden sich in einem einzigen ZIP-Archiv `daten.zip`, das Sie herunterladen und entpacken müssen. Am Besten entpacken Sie sich all diese Daten in ein eigenes Verzeichnis und arbeiten dann auch in diesem Verzeichnis mit MATLAB.

Aufgabe 5—1 (Einlesen von Daten aus einfachen Textdateien)

Die mit Abstand einfachste Variante des Datenimportes ist die MATLAB-Routine `load`. Auch wenn sie bei Textdateien auf solche Dateien beschränkt ist, die ausschließlich Zahlenkolonnen enthalten und eine identische Anzahl von Spalten für jede Zeile aufweisen, gibt es ausreichend viele Geräte, die diese primitivste Form des Datenexports unterstützen.

Hinweis: Das vielleicht größte Problem beim Export von Daten in dieser Form ist das vollständige Fehlen jeglicher weiterer Information über die Daten.

Listing 1: Einlesen einfacher Textdateien mit Daten

```
data = load('simple.txt');
```

Die Variable `data`, die hier als Rückgabewert für die Funktion `load` verwendet wurde, ist eine Matrix mit (in diesem Fall) 501 Zeilen und zwei Spalten. Die Dimension orientiert sich dabei strikt an dem Datenblock in der Textdatei. Voraussetzung ist deshalb auch, dass jede Zeile der Datei, die Daten beinhaltet, exakt gleich viele Spalten aufweist. Ansonsten kommt es zu einem Fehler, da MATLAB Probleme bekommt, die Daten der Matrix zuzuordnen.

Die Dimension der Daten können Sie entweder im Fenster „Workspace“, das alle momentan auf der MATLAB-Kommandozeile definierten Variablen mit Typ, Namen und Dimensionen anzeigt, einsehen, oder Sie verwenden den Befehl `size`, der Ihnen als Rückgabe die Dimensionen (in der Reihenfolge Zeilen, Spalten) zurückgibt. In unserem Fall sähe das Ergebnis wie folgt aus:

[1] `http://till-biskup.de/de/lehre/mathematica-matlab/ss2016/material/07/`

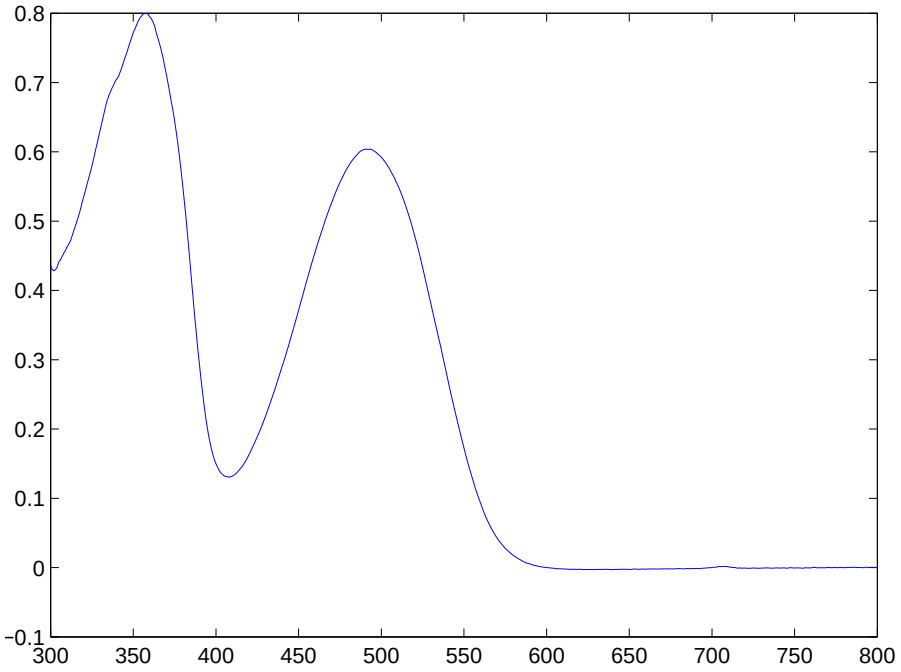

Abbildung 1: Grafische Auftragung der Daten aus der Datei `simple.txt`. Wie Sie anhand der Form der Daten und der Werte der x-Achse vielleicht schon vorsichtig spekuliert haben, handelt es sich tatsächlich um das Absorptionsspektrum eines Moleküls.

Listing 2: Ausgabe der Dimension einer Variablen

```
>> size(data)
ans =
   501     2
>>
```

Um nun die zweite Spalte der Daten gegen die erste Spalte zu plotten – was (berechtigt) davon ausgeht, dass die erste Spalte die Werte für die x-Achse enthält –, verwenden Sie den `plot`-Befehl und den Operator „`:`“, um jeweils eine ganze Spalte aus der Matrix `data` zu extrahieren:

Listing 3: Grafische Auftragung der zweiten Spalte der Variable `data` gegen ihre erste Spalte

```
plot(data(:,1),data(:,2))
```

Das Ergebnis dieser Darstellung ist in Abb. 1 dargestellt. Wie Sie anhand der Form der Daten und der Werte der x-Achse vielleicht schon vorsichtig spekuliert haben, handelt es sich tatsächlich um das Absorptionsspektrum eines Moleküls.

Da die Datendatei allerdings keine weiteren Hinweise darauf liefert, um welches Molekül es sich handeln könnte, und auch der Dateiname wenig aufschlussreich ist, wird es schwierig, ohne genaue Kenntnis der Herkunft der Datei weitere Aussagen über die Identität der untersuchten Probe zu machen. Hier sehen Sie eindrucksvoll das Problem genau dieser Exportformate: Sie sind zwar (mit sehr vielen Programmen) einfach einzulesen, aber sie liefern (mit Ausnahme evtl. des Dateinamens) keinerlei verwertbare Information.

Der Fairness halber sei dazu gesagt, dass diese Daten extra für diese Aufgabe aus einem Format, das minimal mehr Informationen über die Probe abspeichert, in das vorliegende Format konvertiert wurden.

Unterschätzen Sie aber nicht, wie viele Programme als einfache Exportmöglichkeit ein solches Dateiformat als reine Textdatei ohne jegliche weitere Informationen anbieten. Es liegt dann ganz in Ihrer Verantwortung, in sinnvoller Weise die notwendigen Informationen zu den Daten abzulegen.

Aufgabe 5—2 (Einlesen von Daten mit Kopfzeilen)

Zum Glück exportieren viele Geräte Dateien als Textdateien mit einem Minimum an Information am Anfang der Datei. Das erleichtert, wie oben angesprochen, ganz ungemein die Zuordnung der Daten.

Hier gehen wir für's Erste davon aus, dass Sie lediglich an den Daten interessiert sind und nicht an den Kopfzeilen. Dann können Sie – vorausgesetzt, die Zahl der Kopfzeilen ist immer konstant – mit der MATLAB-Routine `importdata` sehr weit kommen.

Öffnen Sie die Datei `daten-mit-header.txt` zunächst in einem Editor, z.B. dem MATLAB-Editor, und schauen Sie, wie viele Zeilen die Datei vor den eigentlichen Daten enthält.[2] Überprüfen Sie im gleichen Zuge, welche(s) Trennzeichen für die Trennung der Spalten verwendet wurde(n). Häufige Möglichkeiten sind hier das Komma oder Semikolon, ein oder mehrere Leerzeichen oder der Tabulator.

Da Sie dem Befehl `importdata` nur dann die Zahl der Kopfzeilen (als drittes Argument) übergeben können, wenn Sie gleichzeitig (als zweites Argument) das Trennzeichen für die Spaltentrennung angeben, ist es wichtig, auch letztere Information zu kennen. Wie Sie sich nach Inspektion der Datei `daten-mit-header.txt` in einem Texteditor Ihrer Wahl sicherlich überzeugen konnten, besteht der Dateikopf aus sechs Zeilen, und als Trenner für die Spalten der eigentlichen Daten kommen mehrere Leerzeichen zum Einsatz. Netterweise müssen wir MATLAB, genauer gesagt der Routine `importdata`, nicht die genaue Anzahl, sondern nur die Art der Trennzeichen angeben. Deshalb funktioniert der in nachfolgendem Listing präsentierte Aufruf:

Listing 4: Einlesen der Detei `daten-mit-header.txt` mit Hilfe des Befehls `importdata`

```
>> data = importdata('daten-mit-header.txt',' ',6)
data =
           data: [5000x2 double]
       textdata: {6x2 cell}
    colheaders: {'s'  'V'}
>>
```

Sie sehen, dass der Aufruf von `importdata` nicht etwa eine Matrix, sondern eine Struktur (*struct*) zurückgibt, und zwar mit den drei Feldern `data`, `textdata` und (in diesem Fall, aber nicht immer) `colheaders`. Das Feld `textdata` enthält – wenig überraschend – die sechs ersten Zeilen der Datei, allerdings als ein `cell array`. Wir hatten uns eingangs darauf geeinigt, im gegebenen Kontext lediglich an den eigentlichen Daten interessiert zu sein. Dieses Feld lieferte Ihnen allerdings die Möglichkeit, an die über die Daten hinausgehenden Informationen in der Datei bequem heranzukommen – so Sie wissen, wie diese Informationen strukturiert sind.

Wie Sie sich vielleicht noch aus der Präsentation zur Vorstellung der Datentypen von MATLAB erinnern, greift man auf Felder einer MATLAB-Struktur mit Hilfe des Punktes zu. Entsprechend müssen Sie den `plot`-Befehl zur Darstellung der Daten geringfügig anpassen, wenn Sie wieder die zweite Spalte des Feldes `data` der Variable `data` als Funktion der ersten darstellen wollen. Lassen Sie sich dabei nicht von der Tatsache verwirren, dass sowohl die MATLAB-Struktur, die der Rückgabe des Befehls `importdata`

[2] Tipp: Verwenden Sie einen Editor, der Ihnen Zeilennummern anzeigt. Bei dieser Datei ist das noch überschaubar, später werden Sie Beispiele sehen, wo Sie die Zahl der Kopfzeilen nicht „per Hand" bestimmen wollen, zumal das auch schnell fehleranfällig wird.

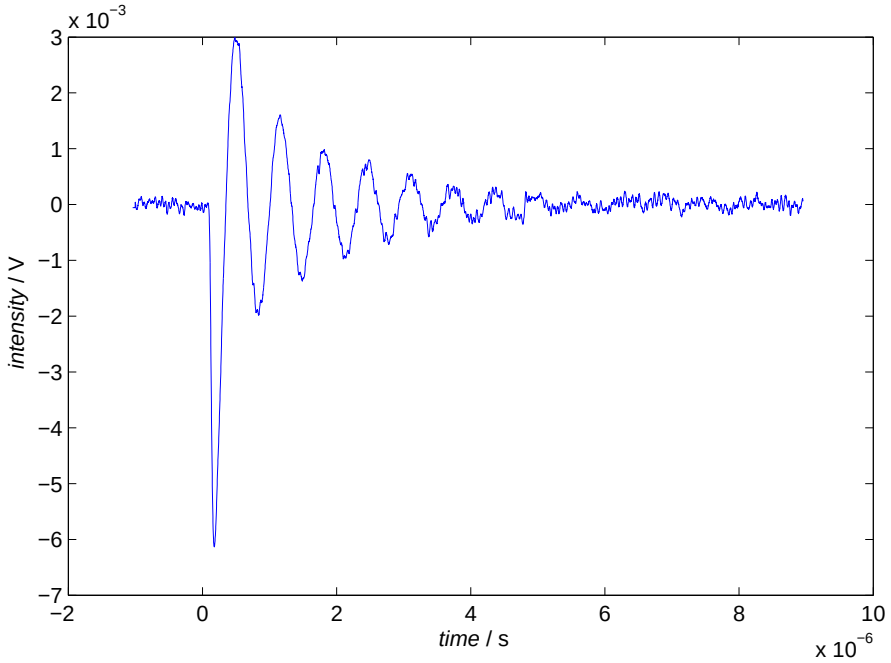

Abbildung 2: Grafische Auftragung der Daten aus der Datei `daten-mit-header.txt`. Wie Sie aus dem Beginn der Datei entnehmen können, handelt es sich hierbei um zeitaufgelöste EPR-Daten, genauer um eine einzelne Zeitspur (Intensität gegen Zeit). Vermessen wurde ein Einkristall von Pentacenmolekülen in einer *p*-Terphenyl-Matrix. Was Sie sehen, ist eine sogenannte transiente Nutation, die dadurch zustande kommt, dass der Magnetisierungsvektor der Elektronenspins in den Pentacenmolekülen um das (effektive) Magnetfeld präzediert. Der Zeitnullpunkt wird durch Lichtanregung der Probe (hier durch einen kurzen Laserpuls) definiert, die paramagnetische Spezies (ein Triplett-Zustand des Pentacens) wird dadurch erst erzeugt.

zugewiesen wurde, in unserem Falle `data` heißt als auch das Feld dieser Struktur, das die eigentlichen Daten enthält. Während Sie (innerhalb der Ihnen von MATLAB auferlegten Grenzen) frei in der Wahl des Namens der Variable sind, der Sie die Ausgabe von `importdata` zuweisen, sind die Feldnamen dieser Struktur immer dieselben, weil sie von `importdata` intern festgelegt werden.

Listing 5: Grafische Auftragung der zweiten Spalte der Variable `data.data` gegen ihre erste Spalte

```
plot(data.data(:,1),data.data(:,2))
```

Das Ergebnis dieser Darstellung ist in Abb. 2 dargestellt. Allerdings wurden hier noch Achsenbeschriftungen vorgenommen. Diese Achsenbeschriftungen können Sie über die folgenden Befehle erzeugen:

Listing 6: Einfache Achsenbeschriftungen in MATLAB

```
xlabel('{\it time} / s');
ylabel('{\it intensity} / V');
```

Beachten Sie die Verwendung von LaTeX-Steuerzeichen zur kursiven Darstellung der aufgetragenen Größen. Die geschweiften Klammern beschränken die Wirkung des Befehls `\it`, der für den kursiven Textsatz sorgt. Weitere Möglichkeiten, die Grafiken schöner und mehr nach Ihren Vorstellungen zu gestalten, werden wir uns in der nachfolgenden Lektion ansehen.

Die Information darüber, welche Größen hier überhaupt gegeneinander aufgetragen werden, bekommen Sie einerseits (aus den in der Datei angegeben Einheiten) direkt aus dem Datensatz, andererseits aus dem Hinweis, dass es sich um zeitaufgelöste EPR-Daten handelt.

Aufgabe 5—3 (Einlesen von Daten mit Komma als Dezimaltrenner)

Sobald ASCII-Textdateien statt des Punktes als Dezimaltrenner das Komma verwenden, wird das Einlesen in MATLAB etwas komplizierter, da Sie keine der bisher genannten MATLAB-Routinen (`load`, `importdata`) verwenden können. Hier sind Sie stattdessen vollständig auf das Schreiben einer eigenen Funktion und den Rückgriff auf „Low-level"-Routinen angewiesen. Nachfolgend sind als Beispiel eines solchen Datenformates die ersten Zeilen der hier einzulesenden Datei `uvvis.txt` gezeigt.

Listing 7: Ausschnitt aus der einzulesenden Datei `uvvis.txt`

```
"Some flavin containing protein in some buffer - uvvis"
"Wavelength nm." "Abs."
300,000 1,4466
301,000 1,2401
302,000 1,0665
303,000 0,9128
```

Als Mensch erkennen Sie sofort, dass die Datei mit zwei Kopfzeilen beginnt, die minimale Informationen über die Probe und die beiden Spalten liefern, und dass dann die eigentlichen Daten in zwei Spalten folgen, jeweils mit dem Komma als Dezimaltrenner und einem Leerzeichen als Spaltentrenner. Das zeigt, wie gut wir als Menschen im Erkennen von Mustern sind. MATLAB bringt wie erwähnt keine Funktion mit, die das einfache Einlesen dieser Daten erlaubte.

Hinweis: Die Lösungen zum letzten Aufgabenblatt beinhalten bereits eine Funktion, die ziemlich genau das tut, was wir hier machen wollen (und sogar noch ein bisschen mehr). Hier wollen wir trotzdem noch einmal eine entsprechende Routine entwickeln und sie in einem entscheidenden Detail gegenüber der vom letzten Aufgabenblatt verbessern: der Robustheit gegenüber Problemen beim Einlesen der Datei.

Das Vorgehen beim Einlesen der Datei lässt sich in zwei Schritte aufteilen:

1. Einlesen der Inhalte der Datei
2. Verarbeitung der eingelesenen Inhalte

Diese Zweiteilung hat einen ernsthaften Hintergrund: Nur so ist es möglich, die eigentlich kritischen Operationen auf dem Dateisystem zu kapseln und entsprechend abzusichern. Darüber hinaus ließe sich so gegebenenfalls diese Funktionalität (Einlesen einer beliebigen Textdatei) in eine separate Funktion auslagern.

Das Kernstück des Einlesens einer Datei besteht im Wesentlichen aus den drei Befehlen `fopen` zum Öffnen der Datei, `fgetl` zum Einlesen einer Zeile und `fclose` zum Schließen der Datei sowie aus einer `while`-Schleife, die solange Zeilen aus der Datei liest, bis sie das Dateiende (`feof`) erreicht hat.

Listing 8: Kompakter Quellcode zum Einlesen einer Textdatei mit MATLAB

```
fid=fopen(filename);
line=cell(0);
k=1;
while ~feof(fid)
    line{k}=fgetl(fid);
    k=k+1;
end
fclose(fid);
```

Der Befehl `fopen` gibt Ihnen im Erfolgsfall eine Referenz auf die geöffnete Datei zurück (das ist in der Regel eine Ganzzahl größer zwei), im Falle des Misserfolgs ist das Resultat „−1“. In der nächsten Zeile wird der Variable `line` ein leeres `cell array` zugewiesen, um den MATLAB-Editor zufrieden zu stellen und um den Typ der Variablen sauber vorher zu definieren. Beachten Sie, dass Sie vor dem Einlesen der Textdatei nicht wissen können, wie viele Zeilen Sie einlesen werden. Deshalb wird im Folgenden auch eine `while`-Schleife anstelle einer `for`-Schleife verwendet. Die Laufvariable `k` sollte vor der Schleife definiert und initialisiert werden, sie wird in der Schleife (manuell) inkrementiert. Die Bedingung der `while`-Schleife besagt: „Solange noch nicht das Ende der Datei erreicht ist“. Das wird durch die Negation des Rückgabewertes des Befehls `feof` ausgedrückt, der, auf eine Dateireferenz (hier: `fid` für *file identifier*) angewandt, einen Booleschen Wert zurückgibt. „`eof`“ steht dabei für *end of file*. In der `while`-Schleife selbst wird die Datei Zeile für Zeile eingelesen und jeweils einem Element der Variable `line` zugewiesen sowie anschließend die Laufvariable manuell inkrementiert. Nach erfolgreicher Abarbeitung der Schleife (im Normalfall das Erreichen des Dateiendes) wird die geöffnete Datei geschlossen.

Wichtiger Hinweis: Auch wenn der oben vorgestellte Quellcode voll funktionstüchtig ist (und Sie ihm in dieser oder sehr ähnlicher Form oft begegnen werden), ist das *kein robuster Code* und sollte nicht in produktiver Umgebung eingesetzt werden. Der Grund ist einfach: Ein Dateisystem erlaubt nur eine beschränkte Zahl gleichzeitig geöffneter Dateien, darüber hinaus bekommen Sie je nach Betriebs- und Dateisystem Probleme, wenn Sie auf eine geöffnete Datei anderweitig zugreifen wollen. Kommt es nun zu einem (unvorhergesehenen) Ereignis und der obige Quellcode bricht die Verarbeitung ab, noch bevor die Datei über `fclose` wieder ordnungsgemäß geschlossen wurde, kann das zu ernsthaften Problemen führen, die im Zweifelsfall nur durch einen Neustart des Rechners behoben werden können.[3] Insbesondere haben Sie in der Regel keine Chance, an die Dateireferenz heranzukommen, da sie normalerweise nur innerhalb des lokalen Kontextes einer Funtion als Variable existiert und für Sie damit unerreichbar ist. Erratisches Schließen von Dateien (unter der Prämisse, dass die Dateireferenzen Ganzzahlen größer zwei sind) ist auch nicht zu empfehlen, da hier ungewollte Seiteneffekte fast schon zwangsläufig auftreten.

Abhilfe schafft die Verwendung einer `try-catch`-Struktur um die kritischen Operationen auf dem Dateisystem herum, die in ihrem `catch`-Block dafür sorgt, dass auch im Fehlerfall die Datei geschlossen wird.

Listing 9: Robuster Quellcode zum Einlesen einer Textdatei mit MATLAB

```
try
    fid=fopen(filename);
    line=cell(0);
    k=1;
    while ~feof(fid)
        line{k}=fgetl(fid);
        k=k+1;
    end
    fclose(fid);
catch
    fclose(fid);
end
```

Beachten Sie, dass Sie zwingend außerhalb des `catch`-Blockes, also entweder innerhalb des `try`-Blockes wie hier oder unterhalb der ganzen `try-catch`-Struktur, einen `fclose`-Befehl schreiben müssen, da der `catch`-Block nur dann ausgeführt wird, wenn bei der Ausführung des `try`-Blocks ein Fehler auftritt.

[3] Der Neustart eines Rechners mag lästig, aber unkritisch sein, solange es sich um einen Rechner handelt, auf dem nur Sie arbeiten. Ganz anders stellt sich die Situation dar, wenn es sich um einen Server handelt, auf dem viele andere Prozesse laufen. Da könnten Sie dann mitunter Ärger bekommen, wenn Sie durch unbedachte Programmierung einen Systemneustart notwendig machen.

Auch wenn dieser Quellcode schon wesentlich robuster ist und in der Regel größere Gefahr für Ihr Datei- und Betriebssystem abwendet, ließe er sich noch robuster gestalten, wenngleich diesmal eher im Hinblick auf die Benutzerfreundlichkeit. Stellen Sie sich vor, der Dateiname, der in der Variablen `filename` übergeben wird, existiere nicht. In diesem Fall gibt der Befehl `fopen` statt einer positiven Ganzzahl größer zwei den Wert -1 zurück. Diese Konvention kommt aus der Unix-Welt, eine negative Dateireferenz ist eine ungültige Dateireferenz und bedeutet, dass die Datei (aus welchem Grund auch immer) nicht geöffnet werden konnte. Sie könnten also direkt nach dem `fopen`-Befehl die erhaltene Dateireferenz daraufhin überprüfen und gegebenenfalls eine Fehlermeldung an den Nutzer zurückgeben und die weitere Bearbeitung abbrechen. In unserem Fall wird spätestens beim Zugriff auf die Datei mittels `feof` ein Fehler erzeugt, der zur Ausführung des `catch`-Blockes führt, der wiederum ebenfalls mit einem Fehler beendet wird, da `fclose` ebenfalls nicht auf ungültigen Dateireferenzen arbeiten kann. Sie sehen, es gibt also auch bei einer an sich fast trivialen Aufgabe noch sehr viele Möglichkeiten, den Quellcode robuster zu gestalten. Das führt allerdings über den hier gegebenen Kontext hinaus und soll daher nicht weiter vertieft werden. Interessierte seien an die MATLAB-Dokumentation verwiesen.

Nachdem die Datei also nun glücklich eingelesen und wieder geschlossen wurde, können wir uns der Verarbeitung der Inhalte widmen. Auch das besteht wiederum aus mehreren Schritten:

1. Ignorieren der Kopfzeilen
2. Ersetzen des Kommas als Dezimaltrenner durch einen Punkt
3. Konversion der Zeichenketten in Zahlen

Die einfachste Möglichkeit ist auch hier wieder die Verwendung einer Schleife, diesmal allerdings aufgrund der bekannten Anzahl von zu verarbeitenden Zeilen eine `for`-Schleife. Für die Ersetzung des Kommas als Dezimaltrennzeichen durch einen Punkt kann der Befehl `strrep` verwendet werden, für die anschließende Aufteilung der Zeichenkette und die Konversion in zwei Zahlen können Sie den Befehl `textscan` verwenden. Eine mögliche Realisierung dieser Schleife ist nachfolgend gezeigt:

Listing 10: Ersetzung von Komma durch Punkt und Umwandlung in numerische Daten

```
for k=1:length(line)
    data(k,:)=cell2mat(textscan(strrep(line{k},',','.'),'%f %f'));
end
```

Achten Sie zusätzlich darauf, dass Sie die ersten beiden eingelesenen Zeilen ignorieren müssen, da es sich dabei nicht um Daten handelt. Eine sehr einfache Variante ist die, die ersten beiden Elemente der Variable `line`, die die eingelesenen Zeilen enthält, zu löschen. In sinnvollem Quellcode sollten Sie die darin enthaltene Information vorher anderweitig ablegen. Eine etwas erweiterte Form des obigen Listings, das sowohl das Löschen der ersten beiden Elemente der Variable `line` (ein `cell array`) als auch die Vorbelegung der Matrix `data` mit Nullen beinhaltet, könnte wie folgt aussehen:

Listing 11: Ersetzung von Komma durch Punkt und Umwandlung in numerische Daten

```
line(1:2)=[];
data=zeros(length(line),2);
for k=1:length(line)
    data(k,:)=cell2mat(textscan(strrep(line{k},',','.'),'%f %f'));
end
```

Beachten Sie, dass Sie, wenn Sie die Elemente eines `cell arrays` löschen wollen, diesen Elementen den leeren Vektor „`[]`“ zuweisen müssen und dass Sie in diesem Fall nicht etwa dem Inhalt der jeweiligen Elemente des `cell arrays` diesen leeren Vektor zuweisen müssen, sondern den Elementen selbst. Deshalb die Verwendung runder statt geschweifter Klammern. Runde Klammern liefern Ihnen bei einem `cell array` die jeweiligen Felder (wiederum `cell arrays`), geschweifte Klammern die *Inhalte* der jeweiligen Felder – und damit potentiell von beliebigem Datentyp, da Sie bekanntlich in einem `cell array` jegliche Form von Datentypen ablegen können.

Eine Realisierungsmöglichkeit der Funktion `loadUVVisData`, die der in der Aufgabenstellung definierten Schnittstellenspezifikation entspricht, ist nachfolgend wiedergegeben.

Listing 12: Funktion zum Einlesen von UV/vis-Daten

```
function data = loadUVVisData(filename)
% LOADUVVISDATA Import UV/vis data from Shimadzu ASCII export.
%
% Usage
%   data = loadUVVisData(filename)
%
%   filename - string
%              name of the file to load
%
%   data     - matrix (nx2)
%              absorption data
%              column 1: wavelength, column 2: intensity

% Copyright (c) 2016, Till Biskup <till.biskup@physchem.uni-freiburg.de>
% 2016-07-31

% Number of header lines
nHeaderLines = 2;

% Read file contents
try
    fid=fopen(filename);
    line=cell(0);
    k=1;
    while ~feof(fid)
        line{k}=fgetl(fid);
        k=k+1;
    end
    fclose(fid);
catch
    fclose(fid);
end

% Ignore header lines
line(1:nHeaderLines)=[];

% Convert data
data=zeros(length(line),2);
for k=1:length(line)
    data(k,:)=cell2mat(textscan(strrep(line{k},',','.'),'%f %f'));
end

end
```

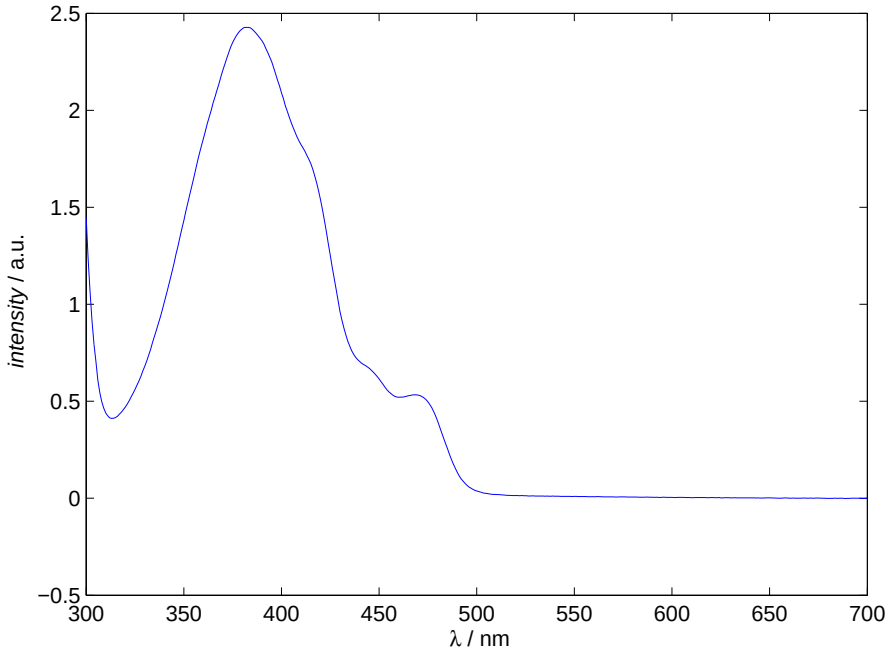

Abbildung 3: Grafische Auftragung der Daten aus der Datei `uvvis.txt`. Die Datei wurde mit der Routine `loadUVVisData` eingelesen, die Daten wie gewohnt mit Hilfe des `plot`-Befehls dargestellt. Wie Sie dem Dateikopf entnehmen konnten, handelt es sich um das Absorptionsspektrum eines Flavin enthaltenden Proteins. Die Strukturen bei ca. 440 und 470 nm sind charakteristische Schwingungsbanden des proteingebundenen, voll oxidierten Flavin-Kofaktors, allerdings wird das Spektrum durch das Signal eines zweiten Kofaktors mit einem Maximum der Absorptionsbande bei ca. 380 nm überlagert.

Nachdem Sie nun also erfolgreich die Funktion `loadUVVisData` implementiert haben, können Sie sie verwenden, um die Datei `uvvis.txt` einzulesen und deren Inhalte in gewohnter Weise darzustellen. Das Ergebnis sehen Sie in Abb. 3.

Wie bereits weiter oben angesprochen ist auch diese Funktion noch nicht der Weisheit letzter Schluss. Insbesondere in Bezug auf die Bedienerfreundlichkeit ließen sich noch weitere Verbesserungen implementieren. Um Ihnen ein paar Ideen zu geben, deren Realisierung Sie sich dann ggf. selbst überlegen können:

- Überprüfung des übergebenen Dateinamens

  Existiert die Datei überhaupt, ist sie lesbar? Hier sollten hilfreiche Fehlermeldungen zurückgegeben werden, darüber hinaus sollte der Rückgabeparameter `data` im Falle unlesbarer Daten eine leere Matrix sein.

- Ergänzung der Dateiendung

  Oft haben Datendateien immer die gleiche Dateiendung, und nichts ist ärgerlicher, als wenn eine Importroutine ihren Dienst versagt, weil man versehentlich nur den Dateinamen ohne Endung eingegeben hat.

- Rückgabe des Kommentars als zweiter Parameter

  Wie Sie vielleicht bemerkt haben, enthält die erste Zeile der Datei einen vom Nutzer (Messenden) vergebenen Kommentar zur Probe. Es wäre durchaus praktisch, diesen (und ggf. auch die in der zweiten Zeile verfügbaren Informationen über die beiden Spalten) dem Nutzer der Importroutine in einem zweiten Rückgabeargument zur Verfügung zu stellen. Achten Sie darauf, ggf. die einschließenden Anführungszeichen zu löschen.

Eventuell führen diese Überlegungen auch dazu, analog zur Rückgabe des Befehls `importdata` statt einer numerischen Matrix eine Struktur zurückzugeben, die dann ihrerseits die Felder `data` und `colheaders` und ein weiteres sinnvoll benanntes Feld für den Kommentar zur Messung enthält.

Aufgabe 5—4 (Lesen und Schreiben von Textdateien)

Eine immer wiederkehrende Aufgabe, für die Matlab keine wirklich einfache Funktion bereitstellt, ist das schlichte Einlesen von Textdateien in ein `cell array` und das Schreiben beliebigen Textes (meist in einem `cell array` vorliegend) in eine Textdateien.

Nach der vorangegangenen ausführlichen Beschreibung des Vorgehens beim Einlesen von Textdateien nimmt sich diese Aufgabe vergleichsweise harmlos aus, zumal Sie für das Einlesen von Textdateien auf bereits existierende Quellcode-Abschnitte zurückgreifen können. Deshalb sei ohne große Kommentare nachfolgend eine mögliche Realisierung der Funktion `readTextFile` wiedergegeben.

Listing 13: Funktion zum Einlesen von Textdateien

```
function fileContents = readTextFile(filename)
% READTEXTFILE Read text file and return cell array with contents.
%
% Usage
%   fileContents = readTextFile(filename)
%
%   filename     - string
%                  name of the file to load
%
%   fileContents - cell array
%                  file contents line by line

% Copyright (c) 2016, Till Biskup <till.biskup@physchem.uni-freiburg.de>
% 2016-07-31

try
    fid=fopen(filename);
    fileContents=cell(0);
    k=1;
    while ~feof(fid)
        fileContents{k}=fgetl(fid);
        k=k+1;
    end
    fclose(fid);
catch
    fclose(fid);
end

end
```

Eine mögliche Alternative wäre die Verwendung der MATLAB-Funktion `fileread`, die eine Textdatei einliest und als Zeichenkette zurückgibt. Wollen Sie hier ebenfalls eine Ausgabe als `cell array` mit einem Element pro Zeile in der Datei erzeugen, müssen Sie die zurückgegebene Zeichenkette entsprechend nach das Zeilenende definierenden Steuercodes durchsuchen und dort entsprechend auftrennen.

Wenn Sie daran interessiert sind, wie MATLAB intern diese Aufgabe bewerkstelligt, können Sie sich den Quellcode der Funktion `fileread` ansehen. Am Einfachsten geht das über die Eingabe des Befehls

`edit fileread` auf der MATLAB-Kommandozeile. Sie sehen, dass die entscheidende Zeile ein Aufruf der eigentlich für Binärdateien verwendeten Funktion `fread` ist. Allerdings ist es `fread` egal, welchen Datentyp Sie angeben.

Listing 14: Ausschnitt aus der MATLAB-Funktion `fileread` zum Einlesen von Textdateien

```
out = fread(fid,'*char')';
```

Die Angabe von `*char` als Argument für `fread` bewirkt, dass alle Bytes der Datei als Zeichen eingelesen und zu einer Zeichenkette zusammengefügt werden. Eine alternative Implementierung der Routine `readTextFile` unter Verwendung der MATLAB-Funktion `fileread` könnte also in ihrem Kern wie folgt aussehen:

Listing 15: Alternative Implementierung von `readTextFile` unter Verwendung von `fileread`

```
fileContent = fileread(filename);
fileContents = strsplit(fileContent,'\n');
```

Sie können in diesem Fall auf jegliche `try-catch`-Strukturen verzichten, da das alles bereits intern in der Funktion `fileread` erledigt wird. Die hier eingesetzte Funktion `strsplit` teilt eine Zeichenkette entsprechend des als zweites Argument übergebenen Trennzeichens (hier der Steuercode für eine neue Zeile) auf und gibt ein `cell array` zurück. Welche der beiden Implementationsvarianten Sie bevorzugen, ist weitestgehend eine Frage des persönlichen Geschmacks. Im Rahmen der Aufgabenstellung ging es allerdings darum, Sie an die Verwendung der „Low-level"-Routinen `fopen`, `fclose` etc. zu gewöhnen.

Eine Alternative zu `fileread`, die Textdateien ähnlich komfortabel schreibt, liefert MATLAB nicht mit, hier müssen Sie also in jedem Fall selbst Hand anlegen.

Auch hier ist die generelle Struktur wieder identisch: Sie müssen zunächst eine Datei mit Hilfe des Befehls `fopen` öffnen, anschließend können Sie in die geöffnete Datei schreiben – vorausgesetzt, Sie haben die Datei auch für schreibenden Zugriff geöffnet. Ohne weitere Angabe von Parametern wird Ihnen `fopen` nämlich nur lesenden Zugriff gewähren, was ein sehr sinnvoller Sicherheitsmechanismus ist. Der „Trick" besteht in der Übergabe eines zweiten Arguments an `fopen`, in unserem Fall `w` oder `w+`. In beiden Fällen wird die angegebene Datei entweder geöffnet und komplett ihres Inhaltes entleert oder, sollte sie nicht existieren, neu erzeugt. Die zusätzliche Angabe von + öffnet die Datei für schreibenden *und* lesenden Zugriff. Weitere Dateiberechtigungen und ihre Bedeutung finden Sie in der MATLAB-Hilfe zum Befehl `fopen`.

Der eigentliche Schreibvorgang wird mit Hilfe des Befehls `fprintf` durchgeführt. Sie hatten diesen Befehl bereits im Kontext formatierter Ausgabe auf der Kommandozeile kennengelernt. Erinnern Sie sich, dass die Kommandozeile in MATLAB intern auch nichts anderes ist als eine Datei mit einer fest zugewiesenen Dateireferenz. Übergeben Sie `fprintf` als erstes Argument *keine* Dateireferenz, wird die Ausgabe auf die MATLAB-Kommandozeile geschrieben. Anderenfalls, wie im aktuellen Kontext, müssen Sie dem Befehl als erstes Argument eine (gültige) Dateireferenz übergeben. Achten Sie darauf, jede Zeile durch einen Zeilenumbruch (Steuerzeichen `\n`) zu beenden, da Sie ansonsten nicht das gewünschte Ergebnis erzielen.

Die eigentliche Schreibarbeit kann bequem über eine `for`-Schleife erledigt werden, da die Zahl der Elemente des zu schreibenden `cell arrays` ja bekannt ist.

Vergessen Sie nicht, nach erfolgtem Schreibvorgang die Datei wieder zu schließen. Das erfolgt wie gewohnt über den Befehl `fclose`, der seinerseits als Argument eine gültige Dateireferenz benötigt.

Eine vollständige Implementierung der Funktion `writeTextFile` könnte wie folgt aussehen:

Listing 16: Funktion zum Schreiben von Textdateien

```
function writeTextFile(filename,fileContents)
% WRITETEXTFILE Write contents of cell array to text file.
%
% Usage
%   writeTextFile(filename,fileContents)
%
%   filename     - string
%                  name of the file to write to
%
%   fileContents - cell array
%                  file contents line by line

% Copyright (c) 2016, Till Biskup <till.biskup@physchem.uni-freiburg.de>
% 2016-07-31

try
    fid=fopen(filename,'w+');
    for line=1:length(fileContents)
        fprintf(fid,'%s\n',fileContents{line});
    end
    fclose(fid);
catch
    fclose(fid);
end

end
```

Einen einfachen Test der beiden implementierten Routinen zum Lesen und Schreiben von Textdateien könnten Sie wie folgt durchführen:

Listing 17: Testlauf der beiden Funktionen `readTextFile` und `writeTextFile`

```
fileContents = readTextFile('readTextFile.m');
writeTextFile('testoutput.txt',fileContents);
```

Wenn Sie nun (z.B. im MATLAB-Editor) die gerade erzeugte Datei `testoutput.txt` mit der Funktion `readTextFile.m` vergleichen, sollten beide (bis auf die nicht vorhandene Syntaxhervorhebung der ersteren Datei) identisch aussehen.

Aufgabe 5—5 (Einfaches Parsen von Textdateien)

Mit dem neu gewonnenen Wissen ob der diversen Möglichkeiten, Textdateien in MATLAB einzulesen, können Sie nun den nächsten Schritt gehen und versuchen, eine Textdatei nicht nur einzulesen, sondern für eine spezifische Art von Textdateien einen Analysevorgang anzuschließen, der die Inhalte der Datei so aufteilt, dass sich die Semantik einem Programm erschließt und daraufhin entsprechende Aktionen ausgeführt werden können. Diesen Vorgang des Zerlegens einer Datei in Einzelteile und die Erschließung von deren Bedeutung wird mit dem aus dem Englischen kommenden Begriff „parsen“ beschrieben. Der Ursprung des Wortes liegt im Lateinischen (*pars* = Teil).

Die Bruker-PAR-Datei, die Sie nachfolgend einlesen und parsen sollen, beinhaltet Parameter zu EPR-Messungen und wird vom Messprogramm automatisch geschrieben. Die eigentlichen Messdaten liegen in

einer korrespondierenden Datei gleichen Namens mit der Endung `SPC`. Deren Import wird in der nächsten Aufgabe im Mittelpunkt stehen.

Eine Schwierigkeit mit dem Bruker-Dateiformat bestehend aus `PAR`- und `SPC`-Dateien ist das, dass es letztlich zwei unterschiedliche Formate gibt, je nachdem, welches Spektrometer zur Datenaufnahme verwendet wurde, und dass diese beiden Formate sich unglücklicherweise in der Art, wie die Binärdaten in der `SPC`-Datei abgelegt werden, fundamental unterscheiden. Es gibt allerdings eine Möglichkeit, die beiden Dateiformate zu unterscheiden, auch wenn die `PAR`-Dateien auf den ersten Blick identisch aussehen: Der Unterschied liegt in den verwendeten Steuerzeichen für die Zeilenenden.[4]

Um zwischen den beiden Möglichkeiten der Steuerzeichen für die Zeilenenden unterscheiden zu können, empfiehlt sich die Verwendung der Funktion `fileread` und die anschließende Aufteilung der eingelesenen Zeichenkette mittels `strsplit`, wie bereits bei einer vorangegangenen Aufgabe gezeigt. Wollten Sie nach den beiden Möglichkeiten (EMX und ESP stehen hier für die beiden Bruker-Spektrometermodelle, die die unterschiedlichen Binärformate schreiben) unterscheiden, könnte der Quellcode wie folgt aussehen:

Listing 18: Mögliche Unterscheidung der beiden Bruker-Dateiformate beim Einlesen anhand der Zeilenenden

```
fileContent = fileread(filename);
lines = strsplit(fileContent,'\n');
if length(lines) == 1
    lines = strsplit(fileContent,'\r');
    fileType = 'EMX';
else
    fileType = 'ESP';
end
```

Nachdem Sie diese Klippe erfolgreich umschifft und zusätzliche Information gewonnen haben, können Sie sich dem eigentlichen Parsen zuwenden. Das Ziel ist, Zeile für Zeile die eingelesenen Dateiinhalte abzuarbeiten und dabei Schlüssel-Wert-Paare zu erzeugen, deren Schlüssel jeweils die Kürzel aus zwei oder drei Buchstaben am Anfang sind und die Werte der nachfolgende Rest der Zeile.

Zur Ablage und vor allem für den einfachen Zugriff auf Schlüssel-Wert-Paare eignen sich Strukturen (MATLAB: `struct`), da hier der Zugriff auf die jeweiligen Werte durch Angabe des Schlüssels erfolgt. Auf diese Weise können Sie ihren Programmen so etwas wie ein „Verständnis“ der Inhalte beibringen.

Eine einfache Variante, die `PAR`-Datei nach dem erfolgreichen Einlesen zeilenweise zu verarbeiten und Schlüssel-Wert-Paare zu erzeugen, die einer Struktur `parameters` hinzugefügt werden, ist nachfolgend wiedergegeben:

Listing 19: Zeilenweises Parsen und Generierung von Schlüssel-Wert-Paaren als Felder einer MATLAB-Struktur

```
parameters = struct();
for line = 1:length(lines)
    [key,value] = strtok(lines{line});
    parameters.(strtrim(key)) = strtrim(value);
end
```

Natürlich handelt es sich dabei erst einmal um ein Beispiel, das die generelle Vorgehensweise aufzeigt. Um unangenehmen Überraschungen beim Einlesen der `PAR`-Datei vorzubeugen, sind noch zusätzliche Details zu berücksichtigen, die später noch thematisiert werden.

[4] Für Sie sind im gegebenen Kontext diese Details zur Bearbeitung der Aufgabe unwichtig, andererseits erhalten Sie so einen Einblick in real auftretende Probleme und mögliche Lösungsansätze.

Zur Erläuterung des Quellcodes in Listing 19: Zunächst wird eine leere Struktur `parameters` erzeugt. Anschließend wird in der `for`-Schleife über jede aus der Datei eingelesene Zeile iteriert. Der Befehl `strtok` zerlegt eine Zeichenkette am ersten Leerraum (Leerzeichen, Tabulator, ..., engl. *whitespace*) und gibt zwei Zeichenketten als Parameter zurück, die hier gemäß ihrer späteren Verwendung `key` und `value` genannt werden. Da diese beiden Zeichenketten ggf. noch am Anfang und/oder Ende Leerraum aufweisen, sorgt der Befehl `strtrim` für die Entfernung dieses Leerraums, da anderenfalls zumindest beim Feldnamen für die Struktur Probleme aufträten. Die Feldnamen unterliegen den gleichen Einschränkungen wie alle anderen Variablen- und Funktionsnamen in MATLAB. Die Zuweisung des Feldnamens der Struktur durch runde Klammern, in denen eine Variable steht, die letztlich zu einer Zeichenkette evaluiert wird, ist in dieser Form eine Besonderheit von MATLAB. Die Alternative wäre ein Aufruf der Funktion `setfield`. Für Details sei auf die MATLAB-Hilfe verwiesen.

Ein Problem, das beim Parsen der eingelesenen `PAR`-Datei auftreten kann, ist die leere Zeile am Ende der Datei, die zu einer leeren Zeichenkette als letztes Element des zu parsenden `cell arrays` führt. Diese leere Zeichenkette führt zu zwei ebenfalls leeren Zeichenketten als Rückgabewerte des Befehls `strtok`, was dann bei der Zuweisung der Feldnamen zur Struktur zu einem Fehler führt, da Feldnamen (u.a.) aus mindestens einem Zeichen bestehen müssen.

Es gibt mehrere Möglichkeiten, mit diesem Problem umzugehen. Eine Möglichkeit wäre, in Kenntnis der grundsätzlichen Struktur der `PAR`-Datei einfach vor der `for`-Schleife alle leeren Zeichenketten als Elemente im `cell array` zu löschen. Das kann über eine – zugegeben auf den ersten Blick recht schwer verständliche – einzeilige Anweisung erreicht werden:

Listing 20: Entfernen leerer Elemente aus einem `cell array`

```
lines(cellfun(@isempty,lines)) = [];
```

Hier wird über den Befehl `cellfun` ein Befehl auf alle Elemente des `cell arrays`, das als zweites Argument übergeben wird, angewendet und das Ergebnis zurückgegeben. Die Rückgabe ist hier ein Vektor mit Booleschen Ausdrücken, und zwar *false*, wenn das Element nicht leer ist, und *true*, wenn das Element leer ist. Nutzt man diesen Booleschen Vektor nun zur logischen Indizierung (daher der Name) für die Variable `lines` und spricht auf diesem Weg nur die Elemente an, die leer sind, können diese Elemente – wie früher bereits erwähnt – durch Zuweisung des leeren Vektors aus dem `cell array` entfernt werden.

Eine Alternative wäre, die Zuweisung des Schlüssel-Wert-Paares und damit die Erzeugung des entsprechenden Feldes in der Struktur `parameters` davon abhängig zu machen, dass die Variable `key`, die vom Befehl `strtok` zurückgegeben wird, nicht leer ist. Eine entsprechend erweiterte `for`-Schleife könnte wie folgt aussehen:

Listing 21: Zeilenweises Parsen und Abfangen leerer Feldnamen

```
parameters = struct();
for line = 1:length(lines)
    [key,value] = strtok(lines{line});
    if ~isempty(key)
        parameters.(strtrim(key)) = strtrim(value);
    end
end
```

Auch hier gilt wieder: Für welche der beiden Varianten Sie sich letztlich entscheiden, ist weitgehend eine Frage des Geschmacks und der persönlichen Vorlieben. Die zweite Variante ist im Allgemeinen robuster,

weil sie die Ursache des Fehlers direkt adressiert, erstere Variante funktioniert im gegebenen Kontext aber genauso gut.

Ein vollständiges Beispiel einer Funktion zum Einlesen und Parsen von PAR-Dateien ist nachfolgend gezeigt:

Listing 22: Funktion zum Parsen von PAR-Dateien

```
function [parameters,fileType] = parseBrukerPARFile(filename)
% PARSEBRUKERPARFILE Read Bruker PAR file and return struct with parameters
%
% Usage
%   parameters = parseBrukerPARFile(filename)
%   [parameters,fileType] = parseBrukerPARFile(filename)
%
%   filename   - string
%                name of the Bruker PAR file to parse
%
%   parameters - struct
%                structure containing key-value pairs from PAR file
%
%   fileType   - string, OPTIONAL
%                identifier specifying the type pf spectrometer used to
%                generate the PAR file
%                Possible values: EMX, ESP
%
% Note: The file type is determined by the newline character used in the
%       PAR file. This is a necessary information for loading the
%       corresponding binary SPC files, as they differ in their encoding.

% Copyright (c) 2016, Till Biskup <till.biskup@physchem.uni-freiburg.de>
% 2016-07-31

% Read file
fileContent = fileread(filename);

% Split lines, get type of PAR file by newline character
lines = strsplit(fileContent,'\n');
if length(lines) == 1
    lines = strsplit(fileContent,'\r');
    fileType = 'EMX';
else
    fileType = 'ESP';
end

% Parse lines and assign parameters struct
parameters = struct();
for line = 1:length(lines)
    [key,value] = strtok(lines{line});
    % Test for empty key
    if ~isempty(key)
        parameters.(strtrim(key)) = strtrim(value);
    end
end

end
```

Aufgabe 5—6 (Import von Bruker-EPR-Daten)

Nach all den Vorarbeiten steht der Entwicklung einer Funktion zum Einlesen von Bruker-Dateien, die mit einem Bruker EMX-Spektrometer erzeugt wurden, nichts mehr im Wege. Wie Sie aus der Aufgabenstellung entnehmen können, beschreibt die Spezifikation des Dateiformates das verwendete Binärformat in den `SPC`-Dateien als „*4 byte floating point*“. Darüber hinaus können Sie davon ausgehen, dass die Datei ausschließlich Binärdaten in diesem Format enthält, Sie also die Datei in einem Schwung einlesen können.

Zum Einlesen der Daten müssen folgende Schritte in genau dieser Reihenfolge abgearbeitet werden:

- Parsen der `PAR`-Datei

  Die Dimension der Daten können Sie den Feldern `RES` und `REY` entnehmen. Beachten Sie, dass das zweite genannte Feld nur existiert, wenn die Daten nicht eindimensional sind.

- Einlesen der `SPC`-Datei

  Da es sich um reine Binärdateien handelt, müssen auch hier wieder „Low-level“-Routinen bemüht werden. Das Vorgehen ist aber vergleichbar dem, was in vorangegangenen Aufgaben besprochen wurde.

- Verarbeiten der eingelesenen Binärdaten

  Je nach Dimension der Daten müssen die eingelesenen Daten noch in ihrer Dimension angepasst werden. Das ist im Fall der vorliegenden Daten zwar nicht notwendig, bei zweidimensionalen Daten aber durchaus.

Das Einlesen und Parsen der `PAR`-Datei wurde bereits in der letzten Aufgabe ausführlich besprochen und kann damit als erledigt betrachtet werden. Sie können in Ihrer Importroutine einfach auf die Funktion `parseBrukerPARFile` zurückgreifen.

Der Kern des Einlesens der Binärdatei (`SPC`-Datei) ist eine einzelne Zeile, die die Funktion `fread` mit den korrekten Parametern aufruft. Hier müssen Sie neben der Dateireferenz sowohl die Zahl der einzulesenden Werte als auch das Format angeben. Da Sie die gesamte Datei einlesen möchten, können Sie das Schlüsselwort `inf` (für unendlich, *infinity*) eingeben. Das Format können Sie aus der Spezifikation, „*4 byte floating point*“, ersehen. Das passende Schlüsselwort wäre also entweder `float` oder `real*4`. Mit diesen Informationen an der Hand können Sie die `SPC`-Datei wie folgt einlesen:

Listing 23: Einlesen einer Bruker-SPC-Datei (EMX-Format)

```
fid = fopen(filename);
data = fread(fid,inf,'float');
fclose(fid);
```

Da Sie alle Werte aus der Datei einlesen wollen, ist auch eine Schleife überflüssig und das eigentliche Einlesen passiert komplett in einer Zeile.

Im einfachsten Fall – dem Einlesen eindimensionaler Daten – könnten Sie dieses Ergebnis bereits darstellen. Allerdings sollten Sie sich nicht darauf verlassen, dass Sie immer nur eindimensionale Datensätze einlesen wollen, zumal Messungen in Abhängigkeit von der Mikrowellenleistung oder anderen zusätzlichen Parametern in der Praxis durchaus häufig auftauchen und das Spektrometer Ihnen erlaubt, diese Messungen automatisch durchzuführen. Das Resultat ist *ein* Datensatz mit zwei Dimensionen.

Abhilfe schafft hier der Vergleich des Parameters `RES` aus der `PAR`-Datei mit der Dimension der eingelesenen Daten und die Verwendung des MATLAB-Befehls `reshape` zur Anpassung der Dimensionen. Beachten Sie, dass Sie zum Vergleich des eingelesenen Parameters mit der Dimension diesen Parameter zunächst noch (mittels `str2double`) in eine Zahl umwandeln müssen.

Listing 24: Anpassung der Dimensionen der Daten

```
if str2double(parameters.RES) < length(data)
    data = reshape(data,parameters.RES,[]);
end
```

Ein erstes vollständiges Beispiel einer Funktion zum Einlesen von Bruker-EMX-Spektren ist nachfolgend gezeigt. Beachten Sie, dass die Funktion in dieser Form nicht besonders nutzerfreundlich ist und außerdem wesentliche Informationen aus der `PAR`-Datei, u.a. über die x-Achse, (noch) nicht ausgewertet werden.

Listing 25: Funktion zum Einlesen von Bruker-EMX-Spektren

```
function data = importBrukerSPC(filename)
% IMPORTBRUKERSPC Read Bruker SPC file and return data
%
% Usage:
%   data = importBrukerSPC(filename)
%
%   filename - string
%              Base name of the file to read (EXCLUDING extension)
%
%   data     - matrix
%              EPR data read from binary file

% Copyright (c) 2016, Till Biskup
% 2016-08-01

parameters = parseBrukerPARFile([filename '.par']);

% Read binary data
try
    fid = fopen([filename '.spc']);
    data = fread(fid,inf,'float');
    fclose(fid);
catch
    fclose(fid);
end

% For multiple spectra in one file (e.g.: EMX, power sweep), reshape data
if str2double(parameters.RES) < length(data)
    data = reshape(data,parameters.RES,[]);
end

end
```

Wie Sie sehen, dürfen Sie dieser Funktion lediglich den Dateigrundnamen *ohne Endung* übergeben. Das ist nicht besonders nutzerfreundlich. Abhilfe schafft die Zerlegung des übergebenen Dateinamens und das anschließende Zusammensetzen mit den Funktionen `fileparts` und `fullfile` sowie der Test auf Existenz der Datei. Darüber hinaus sollen die notwendigen Informationen über die x-Achse aus der

PAR-Datei entnommen und eine entsprechende Achse generiert werden. Es folgt eine leicht modifizierte Variante der Funktion zum Einlesen von Bruker-EMX-Spektren, die diese Kriterien (minimal) erfüllt:

Listing 26: Funktion zum Einlesen von Bruker-EMX-Spektren

```
function data = importBrukerEMX(filename)
% IMPORTBRUKEREMX Read Bruker EMX file and return data
%
% Usage:
%   data = importBrukerEMX(filename)
%
%   filename - string
%              Name of the file to read
%
%   data     - matrix
%              EPR data read from binary file
%              column 1: axis; column 2-end: data

% Copyright (c) 2016, Till Biskup
% 2016-08-01

% Break file name into parts
[fpath,fname] = fileparts(filename);

% Parse PAR file
parfilename = fullfile(fpath,[fname '.par']);
if ~exist(parfilename,'file')
    parfilename = fullfile(fpath,[fname '.PAR']);
end
parameters = parseBrukerPARFile(parfilename);

% Read binary data
try
    fid = fopen(fullfile(fpath,[fname '.spc']));
    if fid == -1
        fopen(fullfile(fpath,[fname '.SPC']));
    end
    eprdata = fread(fid,inf,'float');
    fclose(fid);
catch
    fclose(fid);
end

% For multiple spectra in one file (e.g.: EMX, power sweep), reshape data
if str2double(parameters.RES) < length(eprdata)
    eprdata = reshape(eprdata,parameters.RES,[]);
end

% Get axis information from PAR file using GST (start) and GSI (width)
axis = linspace(str2double(parameters.GST),str2double(parameters.GSI),...
    str2double(parameters.RES));

% Create output as two-column matrix
data = [ axis' eprdata ];

end
```

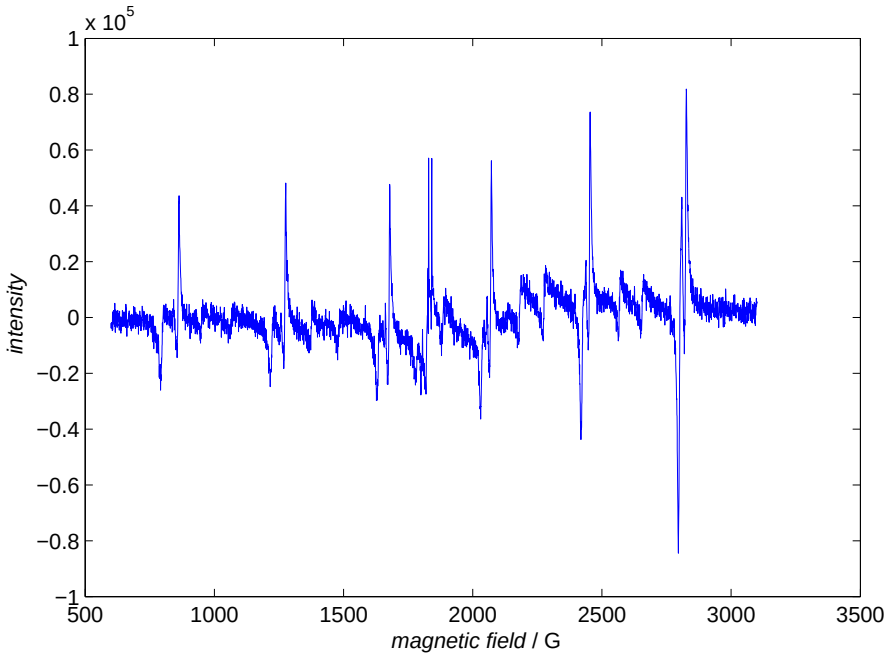

Abbildung 4: Grafische Auftragung der Daten aus der Datei `bruker.spc`. Die Datei wurde mit der Routine `importBrukerEMX` eingelesen, die Daten wie gewohnt mit Hilfe des `plot`-Befehls dargestellt. Die Einheit G (Gauß) des Magnetfeldes ist keine SI-Einheit, das wäre z.B. Millitesla (mT, 1 mT = 10 G). Allerdings speichern Bruker-Spektrometer nach wie vor die Magnetfeldachse in G. Was Sie sehen, ist das *cw*-EPR-Spektrum einer PVC-Probe.

Auch bei dieser Einleseroutine gibt es natürlich noch einige verbesserungswürdige Aspekte. Eine kleine Liste der Dinge, die man implementieren könnte:

- Umgang mit nicht existierenden Dateien

  Bislang überprüft die Routine nur, ob die Dateiendung groß oder klein geschrieben ist, aber nicht, ob Dateien mit dem Dateigrundnamen überhaupt existieren. In einem solchen Fall wäre es hilfreich, dem Anwender eine Fehlermeldung auszugeben.

- Vorbelegung des Rückgabearguments `data`

  Bislang wird die Routine einen Fehler produzieren, wenn z.B. die Datei nicht eingelesen werden konnte, da das Rückgabeargument `data` dann nicht definiert wurde. Abhilfe schafft die Initialisierung von `data` als leerer Vektor ganz am Anfang der Funktion.

- Format des Rückgabearguments

  In ihrer hier dargestellten Implementierung gibt die Einleseroutine in der ersten Spalte des Rückgabearguments `data` immer die Achse aus. Das ist bei zweidimensionalen Daten, die entsprechend auch mehr als eine Achse benötigen, nicht unbedingt die beste Lösung.

  Eine Möglichkeit wäre, statt einer Matrix eine Struktur zurückzugeben, die mehrere Felder enthält, eines für die Daten (als Matrix), weitere für die Achsen.

Trotzdem ist die Routine durchaus grundsätzlich nutzbar. Eine grafische Darstellung der mit ihrer Hilfe eingelesenen Daten findet sich in Abb. 4. Die Einheit der x-Achse könnten Sie darüber hinaus aus dem Parameter `JUN` der `PAR`-Datei entnehmen.