Example files for reading Raman data (BWSpec spectrometer)

We publish in this folder a few example files that show how to read data files saved by the BWSpec 4.11_1 software with default settings (.txt files).

The examples are written both in Python (tested with Python 3.11.1 on Windows 10) and Matlab (tested with MATLAB 9.13 (R2022b) Update 3 and with GNU Octave 7.3.0, both on Windows 10).

For both languages we provide

- A function that reads data from file, from two user-selected columns
- Two scripts which, with the help of the function, respectively plot the data and save the two selected columns

A test data file (Raman_spectrum_file.txt) accompanies the software.

To use the programs, please copy all of the files (including the data file) in the same directory and execute the scripts.

The "read" file is the function that reads the files. "plot" and "save" are scripts. The "plot" script generates a plot of the data in the user-selected columns. The "save" script saves on disk a file containing only the selected columns; the file to which we save takes the same name as the original file, appended with the string "_columns".

The programs are provided **without guarantees**. In particular, we recommend to make a copy of your data files before using them as input of these programs.

A detailed description of the functions, aimed at users who do not have (yet) experience with either Python or Matlab but have at least some experience with programming, follows the table with the program list. The scripts have a few lines of comments that we hope are enough for users to understand how they work.

Program list

Language	File name	File purpose
Python	read_BWSpec_data_file.py	Function – reads data from the BWSpec file
Python	plot_BWSpec_Raman_spectrum.py	Script – plots data
Python	save_BWSpec_Raman_spectrum_columns.py	Script – saves selected columns to disk
Matlab	read_BWSpec_data_file.m	Function – reads data from the BWSpec file
Matlab	plot_BWSpec_Raman_spectrum.m	Script – plots data
Matlab	save_BWSpec_Raman_spectrum_columns.m	Script – saves selected columns to disk

Detailed description of the functions

The "read" programs are functions which read the data file, placing the data values in a two-column array which is the call return.

The functions scan the file line-by-line till they meet the header line. Once they have met that, they read line-by-line the rest of the data, parse the line using the ";" character as column separator, and write the contents of the user-selected columns to a row of the array.

Python - read_BWSpec_data_file.py

The program opens the data file with a `with` block. The Python `with` block is a so-called "context manager"; it "manages" the allocation and release of a resource, allowing the user to work with the resource without having to write the code that manages it. In our case the resource is a file; the `with` statement opens the file and allows Python to close it after the block has finished execution regardless of the way the program is exiting the block (i.e. Python closes the file even if the program meets an error condition).

In the file we look line-by-line for the data header line with the command `readline()`, which reads the current line and advances to the next; `readline()` itself is executed in a `while` block, which checks when we meet the header line. Since the number of lines before the data header may vary (see BWSpec software manual), we do need to test for the data header and cannot count lines to determine where data starts.

Once we have met the header line, we keep reading the file line-by-line, this time with a `for` loop, parsing each line; the parser (`line_split = line.split(";")`) splits the line at the semicolumn characters into a list of strings. We then pick the columns we want from the list of strings to save them into an array (prepared in advance), using the command

```
spectrum[p,:] = [float(line_split[q])
                 for q in [abscissa_column, spectrum_column]]
```

The syntax in the above code, concentrating on its essential features in the form of an example, is

```
newList = [process(q) for q in list_of_qs]
```

It is a Python construct called "list comprehension", which allows generating a list (`newList` in our example) starting from another—input—list (`list_of_qs`) by applying a function (`process`) to all elements of `list_of_qs` and collecting the results in a list of the same size as the input list.

After the `for` loop has finished, exhausting the lines in the file, the array `spectrum` contains the data columns we requested and we return it as the return value of the function.

read_BWSpec_data_file.py - Notes

The function checks that the header exists; if the reading loop does not find a header, the function exits with an error. On the other hand, there are no checks on the correctness of the format of each data line, neither on the presence of 2048 data lines; we are assuming the spectrometer software does save files correctly and that users submit to the program files saved by the spectrometer software.

The reading of the file and storing of the data to an array could have been achieved with more synthetic Python constructs.

E.g., we could have defined a line-parsing function (`parse`) that extracts the requested columns and could have applied it to all data lines of the file by executing

```
spectrum = [parse (line) for line in f_spectrum]
```

after finding the header; after which one could also convert `spectrum`, which is now a list of lists, to an array of the `numpy` class.

Another possibility is the straightforward `loadtxt` function of the NumPy package (loadtxt help page), that for the BWSpec files still must be combined with a data header search.

We decided to keep the program explicit in the interest of users which may be starting with programming.

Matlab (and Octave) - read_BWSpec_data_file.m

The Matlab function follows the same logic as the Python function: scan the file for the header line, then read line by line, parse each line and save the user-selected columns in an array. Finally, return the array.

An important difference with the Python code is that in Matlab there are no context managers, e.g. there is no `with` statement. Because of this, we need to open and close the file ourselves, taking care of closing the file before exiting the program for all ways of exiting the program. In particular, we need to close the file before issuing errors.

A potentially interesting read for the management of open files in Matlab is a post by Loren Shure on her "Loren on the art of Matlab" blog; the author of this function does not know which is the best approach for obtaining bug-free code while working with files in Matlab. In this function we kept the code simple and we issue an `fclose` command before issuing our errors.

read_BWSpec_data_file.m - Notes

Important differences between the Matlab code and the Python code:

- Matlab's `fgetl` strips newlines at the end of the line (fgetl help page), while Python's `readline` does not; the comparison between the header line and the line read from the file needs to keep that into account. For newlines in Python Input and output tutorial; see also the documentation on the `newline` input argument of `open` for details on newline characters in the documentation of the `open` function

- The `strsplit` command of Matlab collapses consecutive delimiters by default (strsplit help page), while the `split` method of Python, when used with an argument (e.g. our `line.split(";")`), does not (str.split documentation). We override Matlab's default: in this way we would obtain the correct number of columns even if there was some empty column, i.e. two consectuive semicolumns

- Matlab does not have list comprehension, and, although it has a function for array mapping (with the same effect), we have chosen to build each line of the spectrum array explicitly as

```
spectrum(p,:) = [str2double(line_list(abscissa_column))
str2double(line_list(spectrum_column))];
```

One could have written the Matlab code by taking advantage of Matlab functions that "do work for the user", as for example `readlines` that reads all of the lines in a text file into an array of strings and `find`, which can be used to return the first occurrence of an element in an array (and therefore to look for the header line in our data file). We would have thus obtained more concise code. Yet again, as in the Python code, we chose to keep the program explicit in the interest of users which may be starting with programming.

In doing so, we wrote a program that can be run on both Matlab and Octave. When aiming for concise code, one should keep in mind that compatibility between Matlab and Octave isn't complete, so one should write for either one or the other of the programming languages.