SPECTR-O-MATIC
PROGRAMING USER GUIDE

Version 2.0
P.H. Lambrev, 17 July 2017

TABLE OF CONTENTS

OVERVIEW

The ***Spectr-O-Matic*** toolbox for MATLAB® is designed to organize, visualize and perform calculations with spectroscopic data. This user guide shows how to program scripts in MATLAB that use Spectr-O-Matic's data classes and functions. The Spectr-O-Matic toolbox contains an interactive app as well. Check the Spectr-O-Matic App Guide for help with using the interactive app.

What is a spectrum

Generally speaking, a spectrum is a set of two related number arrays, *X* and *Y*:

$$Y = f(X)$$

X is the spectral axis, typically representing frequency, wavelength, or time. Y is the magnitude – a measured spectrally dependent quantity, for example absorbance.

Spectr-O-Matic stores spectra as objects of class *specdata*. A *specdata* object contains one column of *X* values, one column of *Y* values, and a number of additional *properties*, storing related metadata:

ID – a name identifying the spectrum,
ExpID – group (experiment) identifier,
XType, XUnit – type of the *X* axis (e.g. 'Wavelength') and measurement units (e.g. 'nm')
YType, YUnit – type and units of the Y values,
DateTime – date and time of the measurement.

A separate class, *specdata2D,* is designed for two-way spectroscopy data, such that *Y* is measured as a function of two independent variables, *X* and *T*:

$$Y = f(X,T)$$

where *T* is a second axis, that could represent time (in time-resolved spectroscopy) or any other quantity. The main difference between *specdata* and *specdata2D* is that the *Y* values in *specdata* are a single column of numbers (*vector*), whereas in *specdata2D* the *Y* values are a two-dimensional *matrix* with a number of rows equal to the number of *X* values and a number of columns equal to the number of *T* values.

How to access data in a spectrum

In MATLAB, as in any programming language, objects stored in memory can be addressed and manipulated by their unique identifier (name). Let a spectrum be stored in memory by the name *SP*. Its properties can be accessed using dot-notation: *SP.X* and *SP.Y* will return the array of *X* and *Y* values, respectively, *SP.DateTime* will return the date and time of the measurement, etc.

The simplicity and elegance of *Spectr-O-Matic* lies in the fact that spectra stored as objects can be used in mathematical expressions like *c = 2*a* or *c = a + b*, just as if they were simple numbers. In the latter expression, if *a* and *b* are spectra of class *specdata*, the result *c* will be the summed spectrum.

General workflow

A sequence of *Spectr-O-Matic* operations can be written as a MATLAB script and executed or modified later as many times as needed. A typical script implements the following workflow:

1) Load data into *Spectr-O-Matic* objects
2) Perform operations – grouping, sorting, transformations, calculations
3) Plot the results as figures / Save the results to disk

LOAD DATA

Create specdata objects from workspace variables

specdata objects can be created using data stored in MATLAB variables:

```
X = 1:10;
Y = [0 0.5 2 5.5 9 9 5.5 2 0.5 0];
ID = 'spectrum1';
S1 = specdata(X, Y, ID);
```

The object *S*1 will be a spectrum with *X* values 1, 2, 3, … to 10, *Y* values 0, 0.5, 2, … and *ID* property containing the string literal 'spectrum1'.

Additional properties can be assigned to the spectrum using name/value pairs:

```
S1 = specdata(X, Y, 'spectrum1', 'XType', 'Time', 'XUnit', 's');
```

Now the spectrum has *XType* and *XUnit* properties equal to 'time' and 's', respectively.

You can also use a struct or table variable containing all data and properties:

```
s1 = struct('X', X, 'Y', Y, 'ID', ID);
S1 = specdata(s1);
```

Collections of spectra can be created in this way from struct arrays or tables with multiple rows:

```
S1 = specdata([s1 s2]);
```

Load data from a text file

It is practical to create *specdata* objects from data stored in a text (ASCII) file. This is done with commands of this kind:

```
S1 = specdata.load('data_1.txt');
S2 = specdata.load('data_2.csv', 'Delimiter', ';')
```

The command will create a *specdata* object *S*1, which has *X* and *Y* values read from the first two columns of the text file 'data_1.txt'. The *ID* and *DateTime* properties in this case are automatically assigned from the file's name and time stamp, and the *ExpID* property contains the source folder.

Load multiple files

A single command can load more than one file (spectrum) into a specdata object:

```
s = specdata.load({'data_1.txt','data_2.txt'});
```

The command reads the files 'data_1.txt' and 'data_2.txt'. Note that the list of files is enclosed in braces { } (in MATLAB this is called a cell array of strings).

Instead of providing a list of all files to load, it is simpler to use wild-card characters, such as *:

```
u = specdata.load('data*.txt');
```

The command will read all files in the current folder with names starting with 'data' and ending with '.txt'. To read all TXT files at once, write '*.txt' instead. In both cases, provided that such files exist and can be read, the

commands will create object *arrays*, i.e. *s* and *u* will contain more than one spectrum. MATLAB expressions containing *s* or *u* will operate on **all** spectra in the arrays, which makes *Spectr-O-Matic* a very powerful tool (see **Working with object arrays**).

Set additional properties

When loading *specdata* objects, additional properties (metadata) can be passed as name/value arguments to the *specdata* or *specdata.load* functions:

```
data = specdata.load('*.dat', 'XType', 'Frequency', ...
      'YType', 'Power', 'YUnit', 'mW');
```

The command will load all files with extension '.dat' into object array *data* and will set the *XType* property to 'Frequency', the *YType* property to 'Power', and the *YUnit* property to 'mW'. Note that all string literals in MATLAB are enclosed in 'single quotes'.

Another way to do the same is to first define all necessary properties in a *struct* object and then pass this *struct* to the *specdata.load* function:

```
prp = struct('XType', 'Frequency', ...
             'XUnit', 'Hz', ...
             'YType', 'Power', ...
             'YUnit', 'mW');
data = specdata.load('*.dat', prp);
```

The advantage is that the struct *prp* can be reused.

Load two-way data

Two-way (time-resolved) spectroscopy data can be loaded in a similar way, except in this case the *specdata2D* class is used, two independent variables must be supplied (*X* and *T*) and the *Y* values are not a single column but a matrix with rows corresponding to *X* values and columns corresponding to *T* values:

```
X = 1:10;
T = 1:2:60;
Y = load('2Ddata.txt');
mydata = specdata2D(X,T,Y,'MyData','TUnit','seconds');
```

In this example the *X* and *T* axis values of the 2D dataset *mydata* are created in MATLAB and the actual measured data are read from a text file '2Ddata.txt'. The file 2Ddata.txt must contain 10 rows and 30 columns; otherwise an error message will be generated when creating the *specdata2D* object.

For further information on what file types can be read, consult the ***specdata.load*** and ***specdata2D.load*** reference documentation.

OPERATE WITH SPECTRA

Perform arithmetic operations

Once the data are loaded into appropriate *specdata* or *specdata2D* objects, these objects can be used to perform calculations, such as arithmetic operations. A fundamental concept when performing any type of operations with *Spectr-O-Matic* is to create a **new** object entity by performing operations on one or more **old** objects:

```
c = a + b;
f = d/2;
x = sum([u, v, w]);
```

The new objects *c*, *f* and *x* are created by performing operations on *a, b, d, u, v*, and *w*. The original objects are not modified. If you want to **modify** an existing object entity instead, simply place its name on the left side of the '=' sign:

```
a = a + b;
```

The expression will calculate the sum of *a* and *b* and the result will be stored in *a*, effectively replacing the contents of *a.*

The four basic arithmetic operations addition, subtraction, multiplication and division can be performed on *Spectr-O-Matic* objects with the respective binary operators +, -, /, and *. In the expression

```
c = a + b;
```

c is the result, + is the operator sign, *a* is the left operand and *b* is the right operand. When either of the operands *a* or *b* is a *Spectr-O-Matic* object, the result *c* is also a *Spectr-O-Matic* object. The operation is always performed on the *Y* values of the operands. The rest of the properties are transferred without change from the left operand *a* to the result *c*.

Either operand of binary arithmetic operations can be a real number, rather than a *Spectr-O-Matic* object. For example the expression `b = 5; c = a - b` will subtract 5 from each *Y* value in *a* and the expression `a = 10*a` will multiply *a* by 10.

As mentioned in the previous section, the operands can be arrays of numbers or spectra, in which case the operations will be performed on every number/spectrum in the array.

Object arrays

Multiple *specdata* (and *specdata2D*) objects can be combined in object arrays and then accessed under the same MATLAB identifier. One way of creating an array is by reading several data files at once as shown in **Creating specdata objects**. Existing objects can be combined into arrays using MATLAB's array operator [ ]. The command

```
x = [u v w];
```

creates an array *x* with three elements, which will be copies of *u*, *v* and *w*. In this expression any or all of the operands *u, v* and *w* can be object arrays themselves. In such case they will be *concatenated* (appended one after another).

Objects are indexed in object arrays in the same way as numbers in number arrays. To reference a particular object, specify its index in parentheses ( ). The next command statement has the same result as the one above, although it is less efficient to write and execute:

```
x(1) = u; x(2) = v; x(3) = w;
```

Range of objects can be specified with the range operator ':' in expressions such as

```
B = A(1:5); C = A(6:end)
```

Most *Spectr-O-Matic* operations can be performed on single objects as well as object arrays. If *x* is an array, the statement `y = 2*x;` will multiply all spectra in *x* by 2 and the result *y* will also be an object array with the same number of spectra as *x*. Not that in this binary operation, the right operand is an object array and the left operand is a single number. *Spectr-O-Matic* will recycle that number to perform operations on all objects in the array. However, both operands can be arrays, as in the next example:

```
A = specdata.load({'dat1.txt', 'dat2.txt', 'dat3.txt'});
N = [1, 0.8, 1.5];
An = A / N;
```

Here *A* is an array created by loading three files. *N* is an array of three numbers, representing normalization factors. The last statement divides each spectrum in *A* by the corresponding number in *N*. The normalized spectra are stored in *An*.

When performing binary operations with two array operands, the number of elements in one array should be the same as in the other array. If one array has fewer elements then its elements will be reused. For example, if *b* is an array of two elements, then the result of the expression `a*b` will be an array [a_1b_1, a_2b_2, a_3b_1, a_4b_2, …].

Both operands can be arrays of spectra, or one operand can be an array and the other a single spectrum:

```
x = [u v w] - x0;
```

The spectrum *x*0 will be subtracted from each of the spectra in the left operand. Note that here an array was created directly in the expression where it was used.

Other operations on spectra

Most of the functionality of *Spectr-O-Matic* is implemented as *methods* of the classes *specdata* and *specdata2D*. There are methods that calculate properties, such as the number of datapoints in a spectrum (***dim***), methods to normalize (***norm***), integrate (***int***), differentiate (***diff***), smoothen (***smooth***) spectra and so on. The same concept applies as with arithmetic operations, i.e. a new object is created by performing operations using a given method. Methods are called either by using dot notation:

```
S = A.sum;
```

or by using function notation:

```
S = sum(A);
```

The two expressions are equivalent and will calculate the sum of *A*, assuming that *A* is an array of spectra.

Some methods return real numbers as a result instead of objects. For example the method dim returns the number of data points in a one-way spectrum, or the number of *X* and *T* values in a 2D dataset. The methods ***max*** and ***min*** return the global maximum and global minimum of each operand spectrum. The expression:

```
An = A / A.max;
```

will divide the spectrum *A* by its maximum. If *A* is an array, all spectra will be divided by their corresponding maxima. The statement is equivalent to:

```
An = A.norm;
```

The method ***Xind*** searches for a given *X* value and returns its index in the *X* array. A similar method ***Yx*** searches for an *X* value and returns the corresponding *Y* value. The statement

```
An = A / A.Yx(650);
```

will normalize the spectrum (or spectra) *A* at *X* = 650.

For *specdata2D* objects the method will return an array of all *Y* values that correspond to the given *X.* An equivalent method Yt returns the *Y* values for a given *T*.

In addition to methods operating on the *Y* data only, there are a few methods operating on the *X* (and *T*) axes of *Spectr-O-Matic* objects. The methods ***setxlim*** and ***settlim*** trim the spectra to a range of *X* (*T*) values:

```
Ablu = A.setxlim([420 480]);
Ared = A.setxlim([620 680]);
```

The result is a spectrum, in which any data lying outside the specified *X* range are discarded. Other methods operating on *X* are ***bin*** (box-car averaging), ***setx*** (replaces the *X* axis and interpolates the *Y* data to match the new axis), and ***merge*** (joins spectra together).

SEARCH AND INDEX

An experiment can contain many measurements (spectra) performed under various conditions. Rather than having different specdata objects for each experimental group, it is often much more efficient to have a single object containing all spectra and operate on selected spectra within the object as necessary. There are three ways to make selections from a collection of spectra:

1) ***find*** and ***findindex*** select spectra matching one or more keywords. This is the simplest way to make a selection.
2) ***autoindex*** scans all spectra for keywords and makes creates an index for each keyword
3) ***catindex*** creates an index with categorical variables. This is the most versatile method to group spectra into multiple categories. With a categorical index, operations on groups of related data are automatically, following a "group-split-apply" workflow.

The three methods can also be used in combination.

Select spectra with find and findindex

Select a subset with find

Using the methods *find* and *findindex* it is possible to perform search queries to select one or more spectra from collections of spectra. The *findindex* method returns a logical index of the spectra matching the search conditions and the *find* method outputs the actual spectra. Consider the following example. Suppose *dat* contains a set of ten spectra of different samples, test conditions, and measurement types, specified in the spectra IDs:

```
F01 sample1 pH2 UV.dat
F02 sample1 pH5 FL.dat
F03 sample1 pH5 UV.dat
F04 sample1 pH7 FL.dat
F05 sample1 pH7 UV.dat
F06 sample2 pH2 UV.dat
F07 sample2 pH5 FL.dat
F08 sample2 pH5 UV.dat
F09 sample2 pH7 FL.dat
F10 sample2 pH7 UV.dat
```

Using *find* we can select some of the spectra, for example only those which contain the word 'FL'

```
fl = dat.find('FL');
```

This is equivalent to the command

```
fl = dat([2,4,7,9]);
```

The result *fl* in both cases will be a specdata object containing four spectra:

```
F02 sample1 pH5 FL.dat
F04 sample1 pH7 FL.dat
F07 sample2 pH5 FL.dat
F09 sample2 pH7 FL.dat
```

It is a good practice to reference spectra by using search queries rather than by their absolute index in the data array. Using search queries will always return the correct spectra even if the contents of the data array have

changed – for example if data are added or deleted at a later time. It also creates readable code and helps to avoid errors.

Search in properties

In the previous examples, the *find* method was called with a single operand 'FL'. In this case, the method searches for text in the object *ID* property. To search within other properties, give the property name first and then the search text:

```
fl = dat.find('YType', 'Fluorescence');
```

The search can be executed with several property/value pairs and will return only those spectra which match ALL of the search criteria (the search criteria are combined with logical AND):

```
fl2 = dat.find('ID', 'sample2', 'YType', 'Fluorescence');
```

This command will return only those spectra whose ID contains 'sample2' AND whose *YType* property contains 'Fluorescence'. The function will only return spectra that match both search terms.

Search for multiple strings

It is possible to use more than one search term with the *find* method. A trivial way is to search twice in the same property:

```
fl2 = dat.find('ID', 'sample2', 'ID', 'FL');
```

As above, the command will return only those spectra whose IDs contain **ALL** search terms, in this case both words 'sample2' and 'FL'.

To search for spectra matching **ANY** of the provided search terms, write the possible matches as an array, as in the following example:

```
pH_l = dat.find({'pH2', 'pH5'});
```

or

```
pH_l = dat.find('ID', {'pH2', 'pH5'});
```

In the commands above, the search argument is {'pH2', 'pH5'}, i.e. a list of two terms. The commands will return spectra that match any of the words (either 'pH2' or 'pH5').

Create an index with findindex

The *fi* method (short for *findindex*) applies the same search criteria as *find* but instead of returning objects, it returns a logical array (0s or 1s) specifying which spectra match the search conditions:

```
ix = dat.fi('FL');
```

The line creates the logical array *ix* = [0, 1, 0, 1, 0, 0, 1, 0, 1, 0], i.e. it specifies that the 2nd, 4th, 7th and 9th element in *dat* match the search criteria.

The index *ix* can be used to reference the respective spectra in the original array *dat* and, for example, change their *YType* property:

```
dat(ix) = dat(ix).set('YType','Fluorescence');
```

Use logical operators

Any logical operator (AND – '&', OR – '|', NOT – '~') or combination of them can be used with the results of *fi* queries, like in the following examples:

```
ipH_l = dat.fi('pH2') | dat.fi('pH5');   % either pH2 or pH5
ipH_h = dat.fi('pH7');                   % pH7
iuv1 = dat.fi('UV') & dat.fi('sample1'); % UV and sample1
```

The index arrays *ipH_l, ipH_h,* and *iuv1* specify different subsets of spectra in *dat*. The index arrays themselves can be combined with logical operators and used to address the respective spectra in *dat*. For example the command

```
uvdiff = dat(iuv1 & ipH_l) – dat(iuv1 & ipH_h);
```

is equivalent to

```
uvdiff = dat([1 3]) – dat(5);
```

Create a keyword index with autoindex

In the previous section, the *findindex* method was used to create a set of index arrays that specify groups of data. Instead of manually typing findindex commands for every keyword, the ***autoindex*** method will automatically create a set of index arrays for a list of keywords. The simplest way to use the command is

```
ix = dat.autoindex;
```

The command will first scan the IDs of all spectra in *dat* and extract a list of keywords. Then it will create an index for every keyword found. The output *ix* is a structure containing the indices for all keywords:

According to the above example, the output *ix* will be a structure with contents:

```
    FL: [0 1 0 1 0 0 1 0 1 0]
    UV: [1 0 1 0 1 1 0 1 0 1]
   pH2: [1 0 0 0 0 1 0 0 0 0]
   pH5: [0 1 1 0 0 0 1 1 0 0]
   pH7: [0 0 0 1 1 0 0 0 1 1]
sample1: [1 1 1 1 1 0 0 0 0 0]
sample2: [0 0 0 0 0 1 1 1 1 1]
```

The indices can be then used as previously to select groups of spectra:

```
FL = dat(ix.FL);                    % FL spectra
pH_l = dat(ix.pH2 | ix.pH5);        % pH2 or pH5
pHdiff5 = dat(ix.pH5) – dat(ix.pH7); % pH5-pH7 difference
```

Note that if the contents of *dat* are changed so that spectra are added, deleted or rearranged, the index will no longer be valid. After any such operation, the *autoindex* command must be executed again to create an updated index.

Conversely, an index created from one object array can be used with another array if there is a logical relationship between the elements of the two arrays. This is illustrated in the next example:

```
Gm = max(dat);        % get the global maximum for each spectrum
ix = dat.autoindex;   % create an index
Fm = P(ix.FL);        % access a subset of Gm using the index
```

ORGANIZE DATA BY CATEGORIES

Create categorical arrays of keywords

Often spectra can belong to one of a several mutually exclusive categories. In the **autoindex** example, spectra were recorded from two samples: *sample1*, *sample2*. Instead of having logical arrays for each category, it is more convenient to index the spectra with a categorical array instead. Use ***catfind*** to create a categorical array of keywords.

Let *dat* contain spectra with IDs:

```
sample1 rep1
sample1 rep2
sample1 rep3
sample2 rep1
sample2 rep2
reference1
```

Suppose we want to index the files for sample1 and sample2.

```
sample = catfind(dat, {'sample1', 'sample2'});
```

The command catfind searches for the keywords sample1 and sample2 in the spectra IDs. The result will be a 6-element categorical array:

```
>>disp(sample')
  sample1 sample1 sample1 sample2 sample2 <undefined>
```

The categorical array can then be used to make selections and perform operations on specific groups of spectra. The next line will select and average the spectra that belong to sample1:

```
avg1 = dat(sample=='sample1').mean;
```

Categorical index tables

A categorical index is a more powerful and flexible way to organize your data. Categorical index is a MATLAB table with rows for every spectrum and columns that describe the spectrum. Columns are typically categorical variables, but can be strings or numbers as well. Variables can be species, gender, age group, sample ID, pH, temperature, and so on. This way define logical relationships between spectra that can make operations on groups of spectra much easier to code.

There are two ways to create a categorical index. **catindex** creates an index by searching the spectra ID's for keywords that match a predefined list of variables. **indextable** matches the spectra ID's to an already existing table. See *Examples\catindex* for example data and code.

catindex – search for keywords to variables

catindex creates categorical index tables from spectra IDs. **catindex** works similarly to **catfind** but instead of a single categorical array for a single list of keywords (categories), it searches in several keyword lists (variables) and returns a collection of categorical arrays in the form of a table.

The first step is to define the variables and possible values (keywords) for each variable. For example, perhaps the sample pH is marked in the spectra file names (IDs) as 'pH2', 'pH5', 'pH7'. These keywords can be grouped as values of the variable 'pH'. List all variables, applicable to the experiment in a MATLAB struct:

```
vars.sample = {'sample1', 'sample2'};
vars.pH = {'pH2', 'pH5', 'pH7'};
vars.type = {'UV', 'FL'};
```

Each variable is a field containing a cell array of the possible values (categories) for that variable.

Next, execute ***catindex*** to create the index table:

```
catx = dat.catindex(vars)
```

The result *catx* is a categorical index table that looks like this:

```
catx =
                                sample     pH     type
                                _______    ___    ____
    F01 sample1 pH2 UV.dat      sample1    pH2    UV
    F02 sample1 pH5 FL.dat      sample1    pH5    FL
    F03 sample1 pH5 UV.dat      sample1    pH5    UV
    F04 sample1 pH7 FL.dat      sample1    pH7    FL
    F05 sample1 pH7 UV.dat      sample1    pH7    UV
    F06 sample2 pH2 UV.dat      sample2    pH2    UV
    F07 sample2 pH5 FL.dat      sample2    pH5    FL
    F08 sample2 pH5 UV.dat      sample2    pH5    UV
    F09 sample2 pH7 FL.dat      sample2    pH7    FL
    F10 sample2 pH7 UV.dat      sample2    pH7    UV
```

The index is a MATLAB table with rows for each spectrum and three columns – sample, pH and type. Each column is a categorical variable. It gives an overview of the collection of spectra and which categories they belong to.

Note that, to classify spectra according to the set of variables, **catindex** searches the spectra IDs for keywords that match the predefined values in the struct *vars*. It is not necessary that the keywords are always in the spectra IDs – **catindex** can be instructed to search in any list of strings. For example, to search in the Comments field, type

```
catx = catindex({dat.Comment}, vars);
```

This command uses the synthax `{dat.Comment}` to extract the contents of a data field (*Comment*) for all spectra in *dat*. The result is a cell array of strings.

indextable – match spectra to an existing table

Sometimes it is not practical to define all necessary information in the data file names, especially if the experiment contains many variables. It may be more efficient to externally create the table describing the measurements, for example in Excel. If such a table exists, *Spectr-O-Matic* can automatically use it to create a categorical index. For example, suppose that an Excel file named 'Workbook.xls' contains the file name in the first column and subsequent columns describe the data. To create a categorical index table, write:

```
catx = indextable({dat.ID}, 'Workbook.xls');
```

The command will read 'Workbook.xls' and look up the first column for elements that match the IDs in the object *dat*. It will then create the table *catx*, with rows corresponding to the data in *dat*. Note that 'Workbook.xls' may contain more entries *dat* but all spectra in *dat* need to have a corresponding entry in 'Workbook.xls' for the command to work properly.

indextable can also accept an existing MATLAB table, as in this example:

```
wbk = readtable('Workbook.xls', 'ReadRowNames', true);
catx = indextable({dat.ID}, wbk);
```

Use the categorical index to select data

A simple way to use the categorical index table is to make selections on the data:

```
dat(catx.sample=='pH2').plot;
s1 = catx.type=='UV' & catx.sample=='sample1';
s2 = catx.type=='UV' & catx.sample=='sample2';
dif = dat(s1) - dat(s2);
```

Use the categorical index as a table

The categorical index table can be modified, for example adding or deleting columns. The variables (columns) do not need to be categorical but can also contain numerical data, such as the results of some calculations on the spectra:

```
catx.Smax = dat.max;
catx.Area = dat.int;
catx.S663 = dat.Yx(663) - dat.Yx(750);
```

The lines above will create three new variables with corresponding values for each spectrum. This data table can be exported, filtered, subjected to statistical analysis, etc.

You can add all or selected data properties to the table using the method ***proptable*** (or its shortcut ***pt***):

```
ptab = dat.proptable({'ID', 'DateTime'});
exptab = [ptab, catx];
writetable(exptab, 'Data.xls');
```

The first line creates a table with the spectra IDs and time stamps as columns. The second line combines that table with the table *catx*. The last line exports the combined table as an Excel file.

The categorical index can be combined with the actual measurement data as well. See **Create a data table**.

For more information see MATLAB's documentation on using tables.

Perform operations on groups of spectra using a categorical index

Using a categorical index table it is possible to split spectra into logical groups and apply operations to each group, i.e. applying a *split-apply-combine* workflow. This is achieved by the methods ***splitop*** and ***splitbinop***. A more general approach is using the MATLAB functions ***findgroups*** and ***splitapply***.

For example **splitop** can be used to automatically average spectra that belong to the same group:

```
avg = dat.splitop(@mean, catx)
```

@mean is a reference to the function to be applied, *catx* is the categorical index for the collection *dat*. The command will find spectra that belong the same category group – i.e. that have the same values for all variables in *catx* – and calculate their average.

splitbinop performs per-category binary operations on two subsets of spectra:

```
dif = dat.splitbinop(@minus, catx.group=='test', ...
```

```
        catx.group=='control', catx, {'age', 'gender'})
```

The command will first split the data in *dat* into two groups – *'test'* and *'control'*. Then it will search for data files between the two groups that belong to the same *'age'* and *'gender'* category. If matching pairs are found, the differences *test–control* will be calculated.

See **Plot using splitapply** for an example of using MATLAB functions **findgroups** and **splitapply**.

PLOT RESULTS

Plot spectra

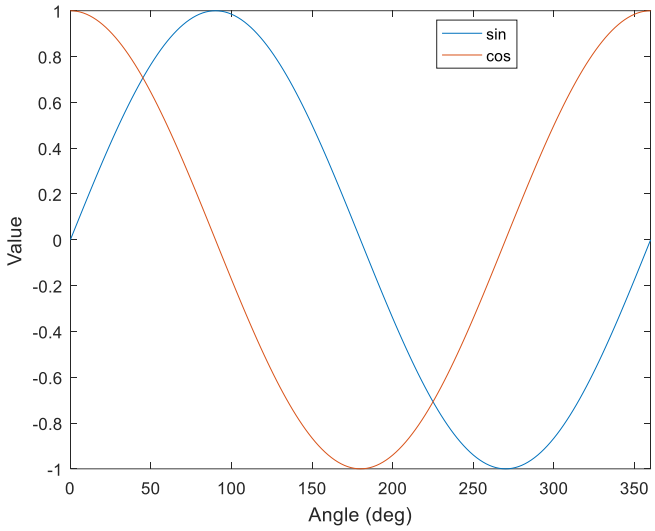
The simplest way to visualize the data in *specdata* objects is to use the ***plot*** method:

```
figure; plot(dat)
```

The *figure* command creates a new empty figure window, and the *plot* command draws the spectra in *dat* on the figure. Here is an example that plots the trigonometric sine and cosine functions:

```
x = 0:2:360; ysin = sind(x); ycos = cosd(x);
l = struct('XType','Angle','XUnit','deg','YType','Value');
trig(1) = specdata(x, ysin, 'sin', l);
trig(2) = specdata(x, ycos, 'cos', l);
figure; plot(trig)
```

Executing the script will produce a figure like this:



The spectra (*Y* values vs *X* values) are plotted in different colours. The axis labels are set by the *XType*, *XUnit*, *YType*, and *YUnit* properties and the figure legend is set by the *ID* properties. Note that MATLAB accepts LaTeX codes for formatting the legend text, which may lead to unwanted formatting if your IDs contain special characters. In such cases, call the *plot* method with parameter ('notex').

You can select which spectra from the array to plot, e.g. to plot only the sine function from the above example, type

```
plot(trig(1))
```

or

```
plot(trig.find('sin'))
```

Conversely, you can combine different objects to plot together on the same figure, like

```
plot([a b c])
```

There is no general plot method available for *specdata2D* objects. Two-way datasets can be plotted as a colour surface using the ***plot2D*** method. Spectra (*Y* vs *X*) for selected *T* values are plotted with the ***plotxy*** method and time traces (*Y* vs *T*) for selected *X* values are plotted with the ***plotty*** method.

Plot spectra by groups

Plot using splitop

If the data are classified in categories with a categorical index, the command **splitop** can automatically plot the spectra grouped by category. The command line looks like this:

```
dat.splitop(@plotf, catx, {'sample', 'type'});
```

The command will first group all spectra into groups based on the "sample" and "type" variables and then execute the helper function **plotf** with each group. The result will be a set of figures with spectra belonging to the same group (sample and type).

plotf contains commands to create a new figure file, plot the data and any additional figure customization commands. You can copy the file plotf.m to your data folder and modify as required by the specific project.

plotf can also write the specific group (sample, type) in the figure title. To be able to do this, add the operand 'PassGroupVars':

```
dat.splitop(@plotf, catx, {'sample', 'type'}, [], 'PassGroupVars');
```

By default, the legend text will contain each spectrum ID. To use a custom legend, the easiest way is to replace the ID field with the custom text. The next example will write the sample pH in the legend:

```
ph = cellstr(catx.pH);
dat.set('ID',ph).splitop(@plotf, catx, {'sample', 'type'}, ...
       [], 'PassGroupVars');
```

The first line creates a cell array of strings from the categorical table column *catx.pH*. The second line replaces the data IDs with *ph* and plots the data grouped by sample and type. The figure titles will state the sample and type, and the legends will state the pH.

Plot using splitapply

Another, more flexible method is to use MATLAB's built-in functions **findgroups** and **splitapply**.

The next example splits the data into groups by *Sample* and then plots spectra in different figures per group. A custom function *plt* plots the data in a new figure and adds a title (from x.*Sample*) and a legend (x.*Species*).

```
g = findgroups(x.Sample);
splitapply(@plt, dat, x.Sample, x.Species, g)

function plt(d, t, l)
    figure; plot(d);
    title(sprintf('Sample: %s', t(1))); legend(cellstr(l))
end
```

Note that you can place the function definition at the end of your script in MATLAB 2016b or later. In earlier MATLAB versions, it must be in a separate file *plt.m*.

SAVE DATA

You may want to save the results of your calculations to a disk file for two main reasons. One is to use the intermediate results for later processing in MATLAB. Another reason is to export your data so that you can work with them in other applications, such as Excel™ or Origin™.

Save MATLAB files

To save data for later use in MATLAB, use the built-in *save* command:

```
save datafile dat
```

where *datafile* is the name of the disk file (it will have a .mat extension) and *dat* is the name of the variable(s) to be saved. When needed later, the data can be loaded again with the command

```
load datafile
```

It is good practice to load external data into *Spectr-O-Matic* objects once and immediately save them as .mat files for later use. The advantage is that .mat files are more compact and loading them is much faster than using the *Spectr-O-Matic load* methods that can take a long time to process large datasets.

Export text files

If you want to use your MATLAB results in another application, you can export the data as ASCII files instead of .mat files. Appending the parameter '-ascii' to MATLAB's built-in *save* command will create an ASCII text file. This way you can export numeric tables but not *Spectr-O-Matic* objects.

To export specdata objects as text files, use the specdata.save method:

```
dat.save('MyData.txt');
```

If *dat* is an array of spectra, the command will create a tab-delimited text file with *X* values in the first column. Note that this is only possible if all spectra have the same *X* axis values.

Create tables from spectra

Data in specdata objects can be converted to tables that can be used later in MATLAB, saved as files or copied to another application such as Excel™. Two types of tables can be created –

- X/Y table - contains X in rows and Y in columns. Does not include properties (metadata). Only works for 1D spectra with equal X axes.
- data table – spectra in rows, data and metadata (properties) in columns. Works with any kind of spectra.

Create an X/Y table

The ***xymat*** method creates a matrix of X/Y values:

```
xy = dat.xymat;
```

The first column in *xy* contains the X values and subsequent columns contain the Y values for all spectra in *dat*. To copy the data to another application, double click on *tbl* in the workspace window.

The ***xytable*** method creates a similarly organized MATLAB table:

```
xy = dat.xytable;
```

The first column contains X and subsequent columns contain Y values. The column headers (variable names) contain the spectra ID's.

Note that the methods **xymat** and **xytable** work only if the spectra have identical X axes.

Create a data table

A data table is a MATLAB table object containing all data (X, Y, T) and metadata (properties such as XType, XUnit, …) belonging to a specdata or specdata2D array. The method ***datatable*** (or ***dt***) returns a data table:

```
T = dat.dt;
```

Rows in the table *T* represent spectra in *dat*. All properties and data are in the respective columns. The next example selects the first three rows of the table and the columns *ID*, *X*, *Y*. Not that *X* and *Y* are not single values but arrays.

```
>> T(1:3, {'ID','X','Y'})
ans =
      ID             X                Y
    _______    _____________    _____________
    'File1'    [801×1 double]    [801×1 double]
    'File2'    [801×1 double]    [801×1 double]
    'File3'    [801×1 double]    [801×1 double]
```

You can convert a data table back to a specdata array:

```
newdat = specdata(T);
```

GET FURTHER HELP

Quick Tutorials

A short video demo and quick animated tutorials are available online:

Spectr-O-Matic Introduction

Online documentation

An HTML version of this user guide is accessible from MATLAB's documentation. From the documentation home page, find the section "Supplemental software" and click "Spectr-O-Matic Toolbox". MATLAB's documentation also includes detailed reference help for all available functions. The function reference can be accessed either from the documentation browser, or from the command line:

```
doc Spectromatic
```

To access the documentation for a specific function directly, for example *catindex*, type

```
doc specdata/catindex
```

or click on the command in the Editor window and press F1.

Examples

A selection of examples can be found in the Spectromatic folder. Look for "Examples" and the subfolders therein.

Frequently used commands are collected in a file directly accessible from the MATLAB command window:

```
open spectromaticlib
```

The file is organized in sections. Use the "Go to" button in the toolbar (ribbon) to navigate to a particular section.

Contact

For more help, questions, comments or requests, write me at p.lambrev@gmail.com.

FUNCTION REFERENCE

This is a list of functions in *Spectr-O-Matic*. To get more detailed help, use the doc command:

```
doc Spectromatic
```

To go straight to a specific function help (e.g. load), type:

```
doc specdata/load
```

Load data in *Spectr-O-Matic*

- `specdata` - create specdata objects from MATLAB variables
- `specdata2D` - create specdata2D objects from MATLAB variables
- `load` - load data from text files
- `unstack` - unstack specdata array to specdata2D array

Get and set properties

- `datatable`, `dt` - MATLAB table with data and properties
- `dim` - number of data points in a spectrum
- `peaks` - find local minima and maxima of Y
- `proptable`, `pt` - MATLAB table with properties
- `set` - set properties
- `Yt`, `yatt` - get Y value at given T
- `Yx`, `yatx` - get Y value at given X
- `xind` - get indices of X values
- `xymat` - matrix of X/Y values
- `xytable` - table of X/Y values

Mathematical operations

- `+`, `-`, `*`, `/`, `^` - arithmetic operators
- `abs` - absolute values of Y
- `baseline` - subtract straight baseline
- `diff` - differentiate
- `int` - integrated area
- `log`, `log10` - logarithm of Y
- `min`, `max` - minimum/maximum Y value
- `mean` - average spectra
- `norm` - normalize spectra
- `smooth` - smooth spectra
- `sum` - sum spectra

Other operations on spectra

- `bin` - bin data (boxcar) along X
- `bint` - bin data (boxcar) along T

- `fit` - fit model to spectra
- `merge` - join spectra into one
- `sett` - set new T axis and interpolate data
- `setx` - set new X axis and interpolate data
- `settlim` - trim data to T range
- `setxlim` - trim data to X range
- `shiftx` - shift spectra along X axis
- `sort` - sort spectra in an array based on a given property

Search and indexing

- `autoindex` - create a keyword index from spectra ID's
- `catfind` - search for keywords and return a categorical array
- `catindex` - create a catalog index by searching for keywords
- `find` - find spectra by keyword
- `findindex`, fi - find indices of spectra by keyword
- `indextable` - match an index table to data
- `splitop` - split-apply operation on groups of spectra
- `splitbinop` - split-apply binary operation on groups of spectra

Plotting

- `markpeaks` - mark peak wavelengths on plot
- `plot` - plot spectra
- `plot2D` - surface plot of 2D spectra
- `plotty` - plot Y vs T
- `plotxy` - plot Y vs X

Save and export

- `save` - save as a text file