SPECTR-O-MATIC
PROGRAMMING USER GUIDE

Version 2.4
P.H. Lambrev, 26 August 2022

TABLE OF CONTENTS

Overview 3

What is a spectrum 3

How to access data in a spectrum 3

General workflow 3

Load data 5

Create specdata objects from workspace variables 5

Load data from a text file 5

Load multiple files 5

Set additional properties 6

Load two-way data 6

Operate with spectra 7

Perform arithmetic operations 7

Object arrays 7

Other operations on spectra 8

SELECT AND ORGANIZE DATA 10

Select spectra by properties 10

Select spectra with find 10

Search in properties 10

Search for multiple strings 11

Create an index with findindex 11

Use logical operators 11

Search using the == operator 11

Use custom metadata 11

Add custom metadata 11

Access metadata 12

Delete custom metadata 12

Organize data Into groups 13

Create an index of categorical metadata 13
Extract metadata from spectra IDs 13
Create metadata from an existing table 14
Use the custom metadata as an index 14
Create a table of metadata and calculated data 14
Perform operations on groups of spectra using a categorical index 15

Plot results 16

Plot spectra 16

Plot spectra by groups 17
Plot using splitop 17
Plot using splitapply 17

Save data 18

Save MATLAB files 18

Export text files 18

Create tables from spectra 18
Create an X/Y table 18
Create a data table 19

Get further help 20

Quick Tutorials 20

Online documentation 20

Examples 20

Function Reference 21

Load data in *Spectr-O-Matic* 21

Get and set properties 21

Mathematical operations 21

Fitting and peak detection 22

Other operations on spectra 22

Search and indexing 22

Plotting 22

Save and export 23

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,

Metadata – additional custom metadata.

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 *A*. Its properties can be accessed using dot-notation: *A.X* and *A.Y* will return the array of *X* and *Y* values, respectively, *A.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

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 can be a cell array (as in the example above) or a string array.

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

```
S = 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:

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

The advantage is that the struct *Props* 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 all spectra in *A*.

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).

SELECT AND ORGANIZE DATA

The real power of Spectr-O-Matic comes with the ability to work with data in bulk. A specdata object can contain hundreds of different but related measurements. An effective analysis script will select the data of interest and perform appropriate operations on them. These operations can automatically be repeated on different groups of data, based on the metadata stored with each spectrum.

Select spectra by properties

Select spectra with find

If the spectra in an object are identifiable by name (ID) or another property, you can use the methods *find* and *findindex* to select and extract spectra. Suppose *Dat* contains a set of spectra of different sample and test conditions specified in the IDs property (or source filename):

```
F01 control pH5.dat
F02 control pH7.dat
F03 treated pH5.dat
F04 treated pH7.dat
…
```

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

```
Dat_control = dat.find('control');
```

This is equivalent to the command

```
Dat_control = dat([1,2]);
```

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

```
F01 control pH5.dat
F02 control pH7.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 'control'. In this case, the method searches for text in the *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):

```
Fl_control = Dat.find('ID', 'control', '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

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

```
pH_57 = dat.find('ID', {'pH5', 'pH7'});
```

The command will return spectra that match any of the words (either 'pH5' or 'pH7').

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('control');
```

The line creates the logical array *ix* = [1, 1, 0, 0,] meaning that the first two spectra match the search term.

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:

```
pH_57 = Dat.fi('pH5') | Dat.fi('pH7');   % OR
Fl_control = Dat.fi('control') & Dat.fi('YType','Fluorescence'); % AND
```

Search using the == operator

The == operator can also be used to select spectra. In this case the search term must match exactly the full property content.

```
Fl = Dat.YType=="Fluorescence";
```

Use custom metadata

Spectra can contain user-defined metadata to help organize and identify the data. The custom metadata are collected within the Metadata property of the specdata object.

Add custom metadata

The ***addmetadata*** (***addmd***) method adds a custom metadata property to spectrum:

```
S = S.addmd('Author',"Jane Doe");
```

The second argument ("Jane Doe") sets the value of the "Author" property. If S has N spectra, this should be an array of N values ["Value1","Value2,…].

Several properties can be added at ones as pairs of Name-Value arguments to the *addmd* function

Access metadata

To access the custom metadata field directly, use S.Metadata.(Name), e.g. S.Metadata.Author.

Delete custom metadata

To delete a metadata property, use the ***deletemetadata*** (***deletemd***) method:

```
S = S.deletemd('Author');
```

Calling *deletemd* without arguments will remove ALL custom metadata from all spectra:

ORGANIZE DATA INTO GROUPS

Create an index of categorical metadata

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.

The most important functions to create and work with categorical metadata are **metaindex** and **metatable (mt)**. metaindex assigns metadata to the spectra and metatable reads the spectra metadata and returns a the index table.

metaindex creates an index table if categorical metadata and assigns these metadata to spectra. The categorical index can be created from an existing table of metadata or created by searching for keywords in the IDs or other property.

Extract metadata from spectra IDs

To create an index from keywords in the spectra IDs, the first step is to define the metadata variables and their possible values (keywords). 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 or string array of the possible values (categories) for that variable.

Next, call ***metaindex*** to create the index table and assign the metadata to the spectra.

```
Dat = Dat.metaindex(vars)
```

The modified object *Dat* will contain additional metadata properties 'sample', 'pH', and 'type'. They can be collected in a table with the command ***metatable***:

```
Idx = Dat.metatable;
```

```
Idx =
                                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, **metaindex** 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

```
Idx = catindex(Dat.get('Comment'), vars);
Dat = Dat.setmetadata(Idx);
```

The first line created a categorical index table by searching into the *Comment* property of *Dat*. The second line assigns the categorical index table as metadata into *Dat*.

Create metadata from 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:

```
Idx = 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:

```
T = readtable('Workbook.xls', 'ReadRowNames', true);
Idx = indextable([Dat.ID], T);
```

Again, it is strongly recommended to save the metadata table within the data variable:

```
Dat = Dat.setmetadata(Idx);
```

Use the custom metadata as an index

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

```
ix = Dat.mt.pH=='pH7';
Dat(ix).plot
```

The first line searches in the 'sample' field of the metadata and returns an index. This is used in the second line to plot the selected spectra. The index can of course be used directly:

```
Dat(Dat.mt.pH=='pH7') = [];
```

The line above deletes all spectra for which the 'pH' metadata property has the value 'pH7'.

Create a table of metadata and calculated data

To create a table of properties, including built-in and custom metadata, use the **proptable** (**pt**) method.

```
T = Dat.pt({'ID','DateTime','sample','pH'});
```

The table *T* is a convenient description of all spectra. Additional calculated properties can be added to it:

```
T.Max = Dat.max;
T.Area = Dat.trapz;
T.Y663 = Dat.Yx(663)-dat.Yx(750);
…
writetable(T, 'Parameters.xlsx');
```

The last line saves the table as an Excel file.

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

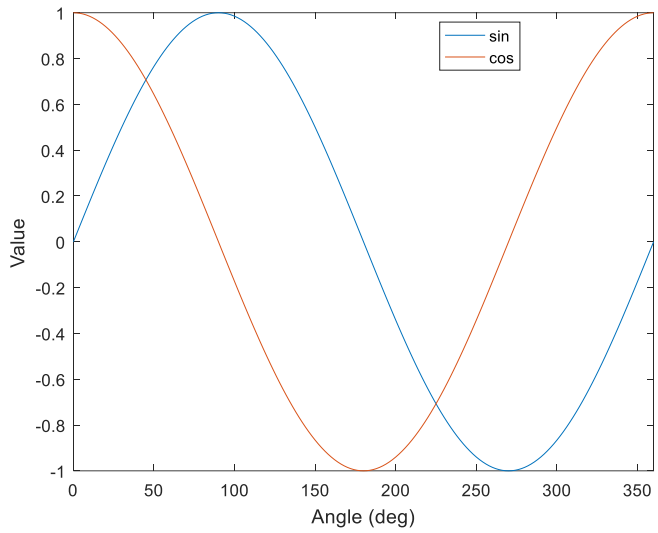
The simplest way to visualize the data in *specdata* objects is to use the ***plot*** method:

```
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

All data and metadata can be represented as a MATLAB table object with the function ***table*:**

```
T = table(Dat);
```

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/metaindex
```

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.

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

- `addmetadata, addmd` – add custom metadata
- `deletemetadata, deletemd` – delete custom metadata
- `dim` - number of data points in a spectrum
- `metatable, mt` – get all custom metadata as a table
- `peaks` - find local minima and maxima of Y
- `proptable, pt` - MATLAB table with properties
- `set` - set properties
- `setmetadata, setmd` – set custom metadata from a table
- `table` – convert a specdata object to a table
- `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
- `log, log10` - logarithm of Y
- `min, max` - minimum/maximum Y value
- `mean` - average spectra
- `norm` - normalize spectra
- `smooth` - smooth spectra
- `sum` - sum spectra
- `std` - standard deviation of spectra

- `stderr` - standard error of mean
- `trapz`, `cumtrapz` - integrated area, cumulative integral

Fitting and peak detection

- `islocalmin, islocalmax` – find local minima, maxima
- `fit` - fit model to spectra
- `fwhm` – peak full width at half-maximum
- `peakdecomp` - peak decomposition of spectra
- `peaks` – find peaks

Other operations on spectra

- `bin` - bin data (boxcar) along X
- `bint` - bin data (boxcar) along T
- `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 IDs
- `metaindex` – create a categorical index and assign as custom metadata
- `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
- `plotbygroup` – plot spectra by groups into separate figures
- `ploterror` – plot spectra with standard errors
- `plots, ploterrs` – plot spectra in new figure with formatting
- `plot2D` - surface plot of 2D spectra
- `plotty` - plot Y vs T
- `plotxy` - plot Y vs X

Save and export

- `save` - save spectra as a text file
- `saveh5` – save as HDF5 file