MATLAB for SMS

Lesson 2: Programming Essentials

Variables and Classes

Variables

Think of a variable as a tag that identifies a certain object.

To assign a value to a variable named "variable": `variable = value;`

To clear a particular variable: `clear variable`

To clear every variable in the current workspace: `clear, clear all`

It's good practice to name a variable so that its contents are reflected in its name. Bear in mind that names will be truncated at 63 characters.

Classes

A logical is a binary 0 or 1. Consider a logical 0/1 to mean true/false, on/off, yes/no, etc.

Double means double precision format. 'Double' will be equivalent to 'scalar'.

3-D arrays are mentioned here for their potential uses in handling image data. The third subscripted index could be called a frame or page index, considering that each 2-D matrix occupies a single index in the third dimension.

```
A(row,column,frame)
```

Strings:

Strings are text, i.e., sequences of alpha-numeric characters.

```
variable = 'string';
```

Each character in a string is indexed exactly as an element in an array. Non-numerical array operations will work on strings too. e.g., concatenation

```
['con' 'catenate'] → ans = concatenate (1x11 char)
['con ' 'catenate'] = ['con' ' catenate'] → ans = con catenate  (1x12 char)
```

Cell Arrays:

Each element in a cell array is called …. a cell. To create an empty cell (1 X 1 cell array):

```
C = {};
```

To create an empty *m* x *n* cell array:

```
C = cell(m,n);
```

Each cell in a cell array is array-indexed. A single cell will hold any class, even another cell array. Separate cells in the same cell array can hold different classes.

Use curly brackets on *either* (not *both*) side of an equation to assign a cell's contents:

```
C(1) = {[1 2 3 4]};
C{1} = [1 2 3 4];
```

To retrieve a particular <u>*cell*</u> in the cell array use parentheses:

```
C(1) → ans = {[1 2 3 4]};   (class = cell)
```

To retrieve the <u>*contents*</u> of a particular cell in the cell array use curly brackets:

```
C{1} → ans = [1 2 3 4];    (class = double)
```

<u>Data Structures:</u>

Data structures are very similar to cell arrays in that any class can be stored in a structure's fields, and that separate fields within the same structure can store different classes.

A structure indexes its fields either by array indexing or by a fieldname.

Create an empty structure and assign variable to the field `fieldname`.

```
S = struct;
S.fieldname = variable;
```

Structures can contain substructures. Substructures can be indexed like matrices, or can be assigned their own fieldnames.

```
S(index).fieldname
S.subS.fieldname
```

<u>Scripts and Functions</u>

Scripts

A script is a file containing a sequence of MATLAB commands that is not designed to take or return arguments.

Scripts operate in the workspace of the calling function. Variables existing in the workspace are available to a script, and those created or altered by the script will remain.

To run the m-file 'script.m', simply type `script` at the command prompt or in a line of another script.

Functions

Functions are scripts that are designed to receive input arguments and return output arguments.

Functions operate in a local workspace. Any variables used by a function must be created locally or passed as input arguments. Any output is specified as a return argument.

This workspace can be saved to a .mat file at the end of a function run with the `save` command.

To call a function:

```
[return_args] = function_name(input_args);
```

The first line of any m-file containing a function must be of the form:

```
function return_args = function_name(input_args)
```

Functions can contain sub-functions. Sub-functions must begin with a `function` line and finish with an `end` line, and the parent function containing the sub-function must also finish with an end line.   The structure is as follows:

```
function
  ...
    function
       ...
    end
  ...
end
```

Unless a function contains a sub-function, the `end` line requirement is optional.

Comments

Green print in m-files should to provide clarity. Comments are added in m-files via the use of %:

```
% A comment will appear in green in the MATLAB editor.  Use them without
% discrimination.
```

Loading and Saving

`cd` points MATLAB's current working directory to a specified directory. The directory name should be in string format.

```
cd('C:\Data')
```

Equivalently:

```
directory = 'C:\Data';
cd(directory)
```

To load a particular file in MATLAB's current directory: `load('filename.ext')`

Files that were saved in .mat format with content from the workspace will appear exactly as they were saved.

To save the entire workspace to MATLAB's current directory: `save('workspace.mat')`

To save a particular variable or list of variables to MATLAB's current directory:

```
save('filename.mat', 'variable1', 'variable2')
```

`if/else` and logical conditions

```
if condition == true
    perform these statements;
elseif different condition == true
    perform these statements;
else
    perform these statements;
end
```

Note that the == signifies elementwise logical equality.

Conditions must be logical arguments. A numerical relation like A < 5 will return true (logical 1) for A < 5 and false (logical 0) for A >= 5.

Avoid numerical equality conditions (i.e., A == 5). In short, a number is converted to binary when stored. Conversion results in small, but occasionally nonzero roundoff error. If conversion of A has nonzero error, the A == 5 condition returns false.

`is*` logical operations

The is* operators ask MATLAB yes/no or true/false types of questions. They return logical 0 (false/no) and logical 1 (true/yes). They are useful in constructing logical conditions.

Suppose that we want to handle the variable A differently if it's a cell array:

```
if iscell(A)
   perform cell ops;
end
```

`for` loops

Use a `for` loop to perform a repeating set of operations a predetermined number of times.

```
for loop_index = array_of_indices

    statements…

end
```

`while` loops

Use a `while` loop to perform a repeating set of operations until a logical condition is no longer true.

```
while condition == true

    statements…

end
```

Logical indexing and `find`

Logical indexing and `find` are operations that isolate certain elements of an array based on a logical relation.

Entering `A < 0` returns a logical array equivalent in size to A. Elements are logical 0 for those `A >= 0` and logical 1 for those `A < 0`.

Entering `A(A < 0)` returns a vector (ordered by linear index) containing those values of `A < 0` that return logical 1.

This is what is referred to as logical indexing. The logical array `A < 0` acts as a array of indices that contains only those elements of A that are negative.

Entering `A(A < 0) = 0;` sets those negative elements in A to zero. Notice this is equivalent to specifying `max(A,0)`.

`find` will return the linear indices of the elements that meet a specified logical relation. To return a vector index containing the indices of the elements of `A > 0`

```
index = find(A > 0);
```

`index` can then be used in subsequent operations. To set the negative elements of A to zero as above:

```
A(index) = 0;
```

The decision to use find or logical indexing is typically that of the programmer. The functional difference between the two operations is minimal.

Basic Operations Demo: Processing a FRET Trajectory

Importing and Loading

So, we've acquired some data and we want look at it. The first thing we'll need to do is load it. We've been provided with a function that converts from binary to a .mat, and we are grateful for such kindness.

.mat is the extension MATLAB assigns to its native data storage files. Files with a .mat extension will load directly to the MATLAB workspace without any extra effort. As we will see, they are very convenient for data storage when MATLAB will be the data's only analysis medium.

Binning up, `for` and `while` loops

Our signal was acquired with higher time resolution (spacing interval) than is optimal, so the first thing we want to do is bin up. This means we want to make a new vector with longer time intervals, whose elements contain the sum of all the elements in the signal vector on the longer time interval we've specified.

For example, we want to bin our signal from 1 ms to 10 ms.

We can use a `for` loop:

```
numbins = floor(numel(signal)/10);
signal = signal(1:10*numbins);
for bin = 1:numbins;
    index1 = 10*(bin-1) + 1;
    index2 = 10*bin;
    binned_up(bin) = sum(signal(index1:index2));
end
```

Or, we can use a `while` loop:

```
numbins = floor(numel(signal)/10);
signal = signal(1:10*numbins);
bin = 0;
while bin < numbins;
    bin = bin + 1;
    index1 = 10*(bin-1) + 1;
    index2 = 10*bin;
    binned_up(bin) = sum(signal(index1:index2));
end
```

Or we can use `sum` and `reshape`:

```
numbins = floor(numel(signal)/10);
signal = signal(1:10*numbins);
binned_up = sum(reshape(signal,10,numbins));
```

How fast are each of these MOs? Bracketing each function with `tic` and `toc` returns the duration of each method. For an acquired trajectory with 90,000 bins (90 seconds with 1 ms bins):

```
using for:  Elapsed time is 0.575073 seconds.
using while: Elapsed time is 0.572355 seconds.
using sum/reshape: Elapsed time is 0.001567 seconds.
```

Visualizing the trace, basic plots

The first thing we'll want to do with the binned trace is look at it. We collect two signals in a FRET experiment, acceptor counts and donor counts. The signals aren't independent of one another, so we'll want to look at both of them at the same time.

The first thing we need to do is construct a time vector whose elements are elapsed times between the start of the trace and the particular element.

To put it more simply, we want this: `t = bintime*(1:numel(signal));`

We can use this vector along with the plot function to look at our signals. If we type `plot(t,signal)` at the command prompt, a plot of our acceptor counts versus time appears.

We can add axis labels: `xlabel('Time (s)')` and `ylabel('Photons')`

We have two vectors we want to look at though. Suppose the acceptor signal is a vector called `acc_signal`, and likewise, the donor is `don_signal`. We can plot the acceptor and donor signals together, on the same set of axes, by supplying x and y vectors for both signals.

```
plot(t,acc_signal,t,don_signal)
```

If we'd like them to be different colors, we can specify that also. This will make the acceptor counts red and the donor counts blue:

```
plot(t,acc_signal,'r',t,don_signal,'b')
```

Selecting regions and applying corrections

We collect background and crosstalk portions of the trajectory to calculate corrections for the FRET portion. It is important to apply these corrections because ultimately we desire the time-dependent FRET efficiency, and this efficiency depends on the dye-emitted, rather than detected, acceptor and donor photon counts. Detected counts and dye-emitted counts are not equivalent, so efficiencies calculated from uncorrected signals will be obscured.

Separating regions:

Background, crosstalk, and FRET regions are selected manually from a figure window. The `ginput` function displays crosshairs for the mouse pointer; when the figure window is clicked, `ginput` returns the coordinates of the crosshairs at the time of the click. Coordinates are converted to linear indices that correspond to the start and end times of the region being selected.

Separating the regions is simply a matter of retrieving the portion of the signal vector that contains the particular region:

```
background = signal(background1:background2);
crosstalk = signal(crosstalk1:crosstalk2);
fret = signal(fret1:fret2);
```

Background correction:

Although there are numerous contributions to the background signal, we approximate background as an additive Poisson noise with nonzero mean. This means we can approximate its contribution at each step with the mean of the observed background signal, and correct for its contribution in the FRET region simply by subtracting its mean.

```
fret_bck_correct = fret - mean(background);
```

Crosstalk correction:

Crosstalk is the result of imperfect separation of donor and acceptor photons by the optical filter (dichroic). Particularly, an appreciable fraction of donor-emitted photons are detected on the acceptor detector. We quantify this fraction with the crosstalk fraction, x.

Any acceptor photons in the crosstalk region are considered to be donor-emitted. Here, `ba` and `bd` are the mean acceptor and donor background counts, and `acc_crosstalk` and `don_crosstalk` are the crosstalk portions of the acceptor and donor signal vectors, respectively.

It is first necessary to remove any photoblinks that occur during the crosstalk region. Using a fairly discriminating threshold of `bd + 3*sqrt(bd)`:

```
don_ct_blinks = find(don_crosstalk < bd + 3*sqrt(bd));
don_crosstalk(don_ct_blinks) = [];
acc_crosstalk(don_ct_blinks) = [];
```

We can use logical indexing if we prefer. Equivalently:

```
don_crosstalk(don_crosstalk < bd + 3*sqrt(bd)) = [];
acc_crosstalk(don_crosstalk < bd + 3*sqrt(bd)) = [];
```

We calculate the crosstalk fraction with the relation:

```
acc_x = acc_crosstalk - ba;
don_x = don_crosstalk - bd;
x = mean(acc_x ./ (acc_x + don_x));
```

Notice that this expression is simply the mean fraction of acceptor-detected, donor-emitted photons `acc_x` to the total number of detected, donor-emitted photons, (`acc_x + don_x`).

To correct the FRET portion of the detected acceptor and donor signals `acc_fret` and `don_fret`, we need to calculate the number of crosstalk counts `num_x`, i.e., donor-emitted acceptor-detected, at each bin in the FRET region (see the SI of Biophys. J. 98(1): 164-173 for more detail):

```
num_x = (x/(1-x))*(don_fret - bd);
```

Now that we've estimated the number of crosstalk counts at each bin in the FRET region, we recall that they represent donor-emitted photons, and appropriately, we subtract them from the acceptor signal and add them to the donor signal at each bin in the FRET region:

```
acc_fret_corrected = acc_fret - ba - num_x;
don_fret_corrected = don_fret - bd + num_x;
```

Notice that `acc_fret_corrected` and `don_fret_corrected` are shown such that both background and crosstalk correction are being applied simultaneously. They are estimates of the actual number of dye-emitted photons detected during each bin in the FRET region.

Efficiencies and distributions, constructing and plotting histograms

After we correct the signal vectors, we want to look at the FRET efficiency. We calculate efficiency vs. time quite easily with elementwise division.

```
efficiency = acc_fret_corrected ./ (acc_fret_corrected + don_fret_corrected);
```

We'll want to look at the distribution of efficiencies, so we'll need to generate a histogram. To do this, we use `hist.`

To use `hist`, we either specify the number of bins we'd like the histogram to have, like this:

```
efficiency_hist = hist(efficiency,20);
```

Or we generate an x vector so that we can specify the bin locations and widths like this:

```
e = 0:0.02:1.1;
efficiency_hist = hist(efficiency,e);
```

It's often a good idea to normalize a histogram so that each bin contains a fractional occurrence rather than an absolute number of occurrences. We divide by the total number of occurrences:

```
efficiency_hist = efficiency_hist/sum(efficiency_hist);
```

We can plot a histogram in two ways. We can generate a histogram vector as we've been doing with `efficiency_hist`, or we can neglect the variable altogether. If we type `hist(efficiency,e)`, a figure appears.

Within scripts and functions, however, it is usually necessary to assign a variable name and plot the histogram later. We use the `bar` command. For example, typing `bar(e,efficiency_hist)` at the command prompt causes a figure that looks very similar (i.e., identical) to the last one, to appear. We can control the bar width and color easily:

```
figure
subplot 211; bar(e,efficiency_hist,1,'b')
subplot 212; bar(e,efficiency_hist,0.5,'r')
```

Filtering anomalous traces

Often, a trace will have some anomalous property, such as higher than average background counts on the acceptor channel. We can use the statistics of the data set to exclude this trace from contributing to our analysis.

The basic logic relies on evaluating the mean behavior of a certain property of this data set using statistics appropriate to a particular property. Whether or not the trace has high acceptor background is determined by the distribution of acceptor background across the data set assuming normal statistics.

These values will have a mean and a standard deviation associated with them, and anomalous values will deviate from the mean value with magnitude larger than some multiple of the standard deviation. An alternate formulation may be explained as something like, "independent values of X were declared anomalous at a 95% confidence interval."

If we were to tell MATLAB how to do this, it might look something like:

```
abs_D = abs(acc_bck - mean(acc_bck));
confidence_bound = multiple*std(acc_bck);
anomalous = abs_D > confidence_bound;
```

Properties by which traces are declared anomalous are high acceptor or donor background, low total photon counts, too short or too long, and anomalously high crosstalk (0.175 or higher, indicating possible contamination, among other things).

Saving output: `save`**, saving a workspace, saving particular variable, saving as structures**

So we've finished processing our trajectory and we need to save it. We have several options. We'll assume that we've already used `cd` to ensure our file is going to the right directory.

Our first option is saving our entire workspace. We do this by typing:

```
save('processed_trace.mat')
```

This will save the every variable currently in the workspace to a file named processed_trace.mat. When loaded to a cleared workspace, loading processed_trace.mat will restore our workspace to its current state. This is a nice trick for writing new functions, but not the best option for storing our data.

We can also specify only the variables we want to be in our saved file.

```
save('specific_variables.mat','variable1','variable2')
```

This syntax would save variable1 and variable2 in a file called specifc_variables.mat. Loading this file would load the two variables to the workspace. This option would be to our liking if we had a only few variables to save, but we have the raw trajectory, the binned trajectory, background, crosstalk, and FRET regions, crosstalk fractions, efficiencies, etc. This is not to mention that we have two of each of these, one for each acceptor and donor.

The best option, as it turns out, is a data structure. We create a data structure as we load the raw trace.

```
save_struct = struct;
```

Each time we create a new variable that needs to be saved, we store a copy of it in the data structure.

```
save_struct.new_variable_to_save = new_variable_to_save;
```

Each time we switch processing operations, we pass and return the data structure.

```
[output save_struct] = next_function(input,save_struct);
```

When we reach the final processing op, we don't have to worry about collecting all the variables we need to save. They are already in our structure. We include the following command and all of our saving is done. Equivalently:

```
save('save_struct.mat','save_struct');
save(save_struct);
```