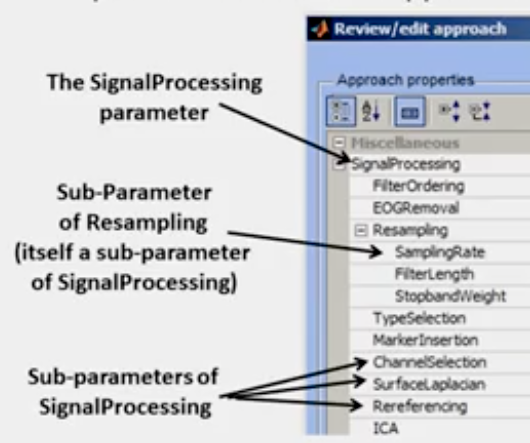
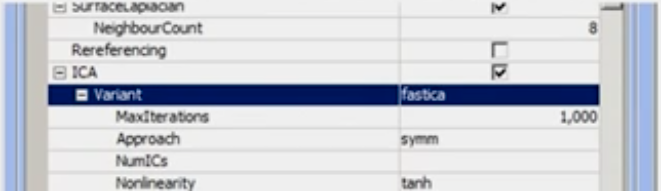
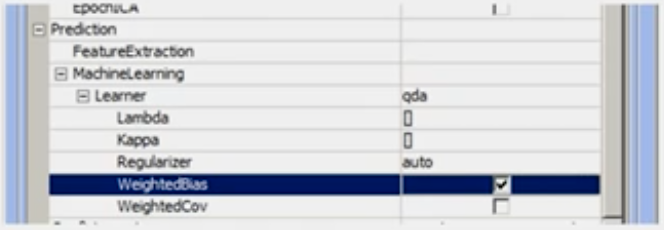
BCILAB Scripting and Plugins

1. Lecture 9.1 – Prerequisites
   1. Finding the Right Functions
      1. There is a scriptable function for every GUI command
      2. For documentation on script functions see Help menu of type doc function\_name or help function\_name
      3. Most functions have a brief summary, documentation for all input arguments, and code examples
         1. Every function has default values except for the data
         2. Code samples maybe present too!
      4. Some functions have paper references, some have cross-references
   2. Calling Syntax
      1. Most functions take their arguments in the order in which they are listed in the documentation, and some can *alternatively* [be] called with all parameters passed in as name-value pairs (using the same names as in the help text, in CamelCase)
      2. If in doubt, pass them in by name – less chance of getting the order wrong, etc.
      3. It is usually a bad idea to try and mix positional and name-value arguments in one call – don’t do it unless that’s the default way to call the function
      4. **Example**:  
         bci\_train(mydata,myapproach)  
         bci\_train(‘Data’,mydata,’Approach’,myapproach)
         1. Fundamentally when you are calling a function like above most of them can take the arguments in the order that they show up in the documentation.
         2. Some of the functions, in particular the high level functions and the user written plugins, can also be called with name-value pairs.
   3. Loading Data
      1. A data set (no matter what file format) is loaded using the function io\_loadset()
      2. It is almost always enough [to] pass in just the file name, as in the example:  
         data = io\_loadset(‘/somepath/somefile.xyz’)
2. Lecture 9.2 - Defining an Approach
   1. Defining a new Approach
      1. Defining an approach is the most complex area in scripting because a data structure must be constructed
      2. Since an approach is a particular instance of a BCI paradigm (used with custom parameters), an approach definition consists of:
         1. The name of the paradigm
         2. Optionally a list of arguments for the paradigm’s calibrate() function
      3. The default way to specify an approach is as a cell array whose first element is the name of the paradigm and whose remaining elements are arguments to its calibrate() function
         1. There are always defaults to the problem
      4. Example  
         appr = {‘CSP’,’SignalProcessing’,…,’FeatureExtraction’,…};
         1. This says I want to use the CSP paradigm
   2. Approach Parameters
      1. The parameters are a list of name-value pairs
      2. Important: The argument of an approach are not passed in a long ‘flat’ list, but they are organized in a hierarchy, i.e. some parameters have *named sub-parameters*
      3. Example:  
         app = {‘CSP’,’Prediction’,{’MachineLearning’,…}};
         1. ‘Prediction’ is a “top-level” parameter
         2. ‘MachineLearning’ is a sub-parameter of ‘Prediction’
      4. Which parameter names a BCI paradigm exposes is the business of the BSI paradigm
      5. However, practically all of them adhere to a uniform scheme of 2 top-level parameter names:
         1. ‘SignalProcessing’ is a top-level parameter that determines the signal processing stages that shall be used
         2. ‘Prediction’ is a top-level parameter that governs how the prediction function is being calibrated or applied
            1. Also includes the Machine Learning that gives rise to the prediction function
   3. Correspondence With The GUI
      1. There is a 1:1 correspondence between the hierarchy of parameters that are specified in scripts and the layout of the parameter tree in the approach definition GUI  
         
      2. Therefore: If in doubt about parameter names, look them up in the GUI
      3. It is also okay to look up the parameter names in the function documentation or code, but they can be nested in a hierarchy of functions calling each other
      4. Essentially, with this name-value parameter reference paradigm, remembering the exact names of the name portion will be tough. If you forget it, you can look it up using the GUI, or work through the top level functions down till you find the parameter nested in some set of functions calling each other.
      5. Kothe recommends using the GUI
   4. Default Values
      1. Each parameter has a default value (unless it makes *absolutely no sense*), which can also be looked up in the GUI
         1. i.e. a default parameter will be enabled and filled with values by default
   5. Parameter Help
      1. Each parameter has a help text, which is also visible in the GUI panel (at the bottom)
   6. The ‘SignalProcessing’ Parameter
      1. Has one named sub-parameter for every single processing plugin that can be used (these are found automatically)
      2. The name under which a given signal processing plugin appears is up to the plugin – they declare this property at the beginning of their code (you can look it up there)
         1. The property that the programming named the parameter is defined using the name-value paradigm on the function called  
            declare\_properties(‘name’,’PluginName’,…
      3. The plugins that are listed under ‘SignalProcessing’ are in the directories:
         1. code/filters
            1. File names beginning with flt\_
         2. code/dataset\_editing
            1. File names beginning with set\_
      4. The value assigned to a sub-parameter (e.g., ‘FIRFilter’) that is presented by a function (e.g., flt\_fir.m) is by default a cell array of arguments to that function.
      5. The arguments can be passed in any format accepted by the function, but preferably that should again be passed as name-value pairs to avoid confusion.
   7. Configuring Signal Processing Stages
      1. Example:  
         app = {‘CSP’,’SignalProcessing’,…  
          {‘FIRFilter’,{‘Frequencies’,[7 8 14 15]}}};
         1. This example defines a CSP-based approach that uses a particular Frequencies value in its FIR filter
         2. The FIR filter is now also “enabled” if it was not before
         3. PS (…) in MATLAB is a line break used by programmers when a line of code is longer then the width of the window
   8. Disabling Signal Processing Stages
      1. It is sometimes useful to disable a parameter that is enabled by default: This can be written (by convention) as follows:  
         app={‘CSP’,’SignalProcessing’,{‘Resampling’,[]};
      2. Note that these are [] brackets – using {} accidentally would still enable the filter, but passes an empty argument list to it!
      3. Essentials
         1. If you want to disable a certain parameter, in this case above ‘Resampling’, you can pass in the empty numerical array
         2. Don’t pass in the cell array
   9. Shortcuts for the Impatient
      1. BCILAB has the unhealthy habit of allowing *short forms for most things* – I recommend to avoid them whenever possible but it helps recognizing them
         1. Unhealthy because it can be thought of as *dangerous* to allow the user access to functions in this manner because a user may act on a function in the wrong way, potentially causing unknown or unexpected behavior.
      2. The most salient short-cut form is when a parameter that has sub-parameters is not assigned a cell array of argument (like it should), but instead directly the value of the first sub-argument
      3. Example:  
         app = {‘CSP’,’SignalProcessing’,{‘Resampling’,200}};
         1. Here ‘200’ is assigned to the first sub-argument of the resampling filter
            1. In this function for example the first argument is the target sampling rate
   10. Multi-Option Parameters
       1. The last kind of parameter that deserves mention are multi-option parameters, which consists of a *selection* argument (a string) and for each possible value a different list of sub-arguments
       2. An example are the different alternative variants supported by the ICA filter: amica, infomax, etc., all of which have algorithm-specific sub-arguments
       3. In scripts, multi-option parameters are written just like the overall approach definition: as a cell array whose first element is the name of the selection followed by name-value pairs for this case
       4. For Example, the parameter called ‘Variant’ is set to ‘fastica’, and the ‘MaxIterations’ sub-parameter of Variant for the *fastica case* is set to 1000
          1. GUI Example:  
             
          2. MATLAB Example:  
             …,’Variant’,{‘fastica’,’MaxIterations’,1000,’Approach’,’symm’}
   11. Other Paradigm Parameters
       1. The other parameters behave in exactly the same ways
       2. Example:
          1. ‘MachineLearning’ is a sub-parameter of ‘Prediction’, it has a ‘Learner’ sub-parameter
          2. ‘Learner’ is a multi-option parameter with one case for each machine learning plugin (e.g., ‘lda’,’qda’,’logreg’,…)
          3. The sub-parameters of the respective case are those that are exposed by the respective plugin function (e.g. ml\_trainqda.m)
   12. Configuring the Machine Leaning Stage
       1. Thus, the following is a valid way to configure the machine learning function paradigm:  
          app = {‘CSP’,’Prediction’,{‘MachineLearning’,…  
           {‘Learner’,{‘qda’ ‘WeightedBias’,true}}}};
       2. It corresponds to the following GUI setting:  
          
   13. Shortcut for Multi-Options
       1. Here is one last shortcut for today:  
          app = {‘CSP’, ‘Prediction’,{‘MachineLearning’,…  
           {‘Learner’,’qda’}}};
          1. Instead of at least {‘qda’}
3. Lecture 9.3 - All Other Steps
   1. Calibrating (“Training”) a Model
      1. A new BCI model is created using a previously loaded data set (the training set) and a previously defined approach
      2. This is done using the function bci\_train (the equivalent of the “Train new model…” dialog)
      3. Example:  
         raw = io\_loadset(‘imag.set’)  
         app = {‘SpecCSP’, … };  
         [loss.model,stats] = bci\_train(‘Data’,raw,’Approach’,app,…  
         ‘TargetMarkers’,{‘S 1’,’S 2’});
   2. Calibrating a Model
      1. The bci\_train function usually takes 3 inputs:
         1. The data
            1. ‘Data’ parameter
         2. The approach
            1. ‘Approach’ parameter
         3. The description of how event types map onto class labels (‘TargetMarkers’, same as in the GUI)
      2. The function returns three outputs
         1. The overall loss estimate (e.g. error rate)
            1. This is for whatever loss metric you choose
            2. i.e. if you used misclassification then it would be the misclassification rate and so on
            3. Just a single number
         2. The learned model
            1. The model structure or struct
         3. Statistics about the model and training process, including results of a cross-validation
            1. Any statistics the function managed to produce about how well the model is going to work
      3. The bci\_train function therefore not only returns a model but also produces estimates about the likely future performance
         1. If this is too slow, it can be disabled (in an extra parameter to bci\_train)
   3. Visualizing a Model
      1. Models are visualized using the function bci\_visualize
      2. Example:  
         bci\_visualize(mymodel)
         1. Some paradigms may not have a visualizer, it is completely up to the paradigm as to how to react to this function call
      3. This function can take extra arguments that are passed onto the responsible drawing function (but few drawing functions have arguments)
   4. Applying a Model to Test Data
      1. For *offline application* to test data, the function bci\_predict can be used – it applies the BCI model to each trial in the data and calculates loss statistics
         1. The big use case here is you have calibrated your model, you have used visualized it and you want to apply this offline to some new data
      2. Example:  
         [outputs, loss, stats] = …  
          bci\_predict{‘Data’,mydata,’Model’,mymodel};
      3. Note: the first output are the model’s predictions for each trial in the data
   5. Annotating Data with Continuous BCI Outputs
      1. The BCI output can be attached as an extra channel (or multiple channels, each representing the probability of class k) to a data set, using the function bci\_annotate
      2. Example:  
         newset = bci\_annotate(‘Data’,mydata,’Model’,mymodel)
      3. The last thing you can do offline is when you want to not only classify certain markers, maybe you don’t even have markers, maybe you just want to generate the time course of BCI outputs, the estimated condition or class label
   6. Reading Real-Time Data
      1. Real-time data can be acquired from a device and written into a named workspace variable using the online reader plugins
         1. run\_read\* functions
      2. Examples:  
         run\_readbiosemi(); # read from a BioSemi device  
         run\_readdataset(‘MatlabStream’,’mystream’,’Dataset’,myset);
      3. Some of the named run\_read\* will automatically check to see if there is an amplifier connected and if so in the background will open up the stream which streams from this \*
      4. If you want to test and see if your BCI is fast enough or if it crashes use run\_readdataset()
      5. The sudo online reader is run\_readdataset takes a dataset that you want to play back and generates stream in the workspace
   7. Sending Real-Time Outputs
      1. The outputs of a BCI model as applied to some stream(s) can be calculated in the background online and passed on to some destination – this is done using the online writer plugins
         1. run\_write\* functions
      2. These functions take usually the name of the model to use and the name(s) of the stream(s) to use
      3. Example:  
         run\_writevisualization(‘Model’,’mymodel’,…  
          ‘SourceStream’,’mystream’
         1. Note: the use of ‘ around the dynamic variable names!
            1. This takes the name of the model it should utilize, could use the struct too
         2. There are several for TCP and OSC and the very simple one to run a visualization
   8. Performing Bath Analyses
      1. Using bci\_batchtrain, a single approach can be efficiently applied to a list of data sets or file names
      2. Also multiple approaches can be applied to one or more data sets in an automated manner
      3. Can not just train models but also make predictions and evaluate losses on test data sets
   9. Taco