Toy Model of pH Subtraction for FSCV Data

Vincent Toups

Nov 1, 2010

Abstract

This is a quick document describing the use of some tools for chemometrics in matlab.

0.1 Front Matter

This is a quick document describing the use of some tools for chemometrics in matlab. It should come with a set of library files in this directory under *matlab*. These library files should be loaded before running the examples. If you start Matlab in this directory, the file *startup.m* should execute automatically on startup. If it doesn't, you can execute *addpath(genpath(path-to-library-directory))* to load them. This document should have come in a zip file, which, when expanded, should create a directory. If you start Matlab in this directory, everything should "just work."

0.2 Generating CVs From Raw Data

The first step in analyzing data from a flow cell experiment is to generate a single CV for each trial from the flow cell. This process involves separating the scans which contain the sample from the background, estimating the background, removing it from the sample trials, and finally averaging some subset of the sample scans to produce the background subtracted CV. Because each of these steps involve some discretion, and because differences in each step produce different CVs at the end of the process, it makes sense to automate the process entirely.

Before the raw data can be analyzed by this library, it has to be exported from Tar Heel CV as a color data file, with care taken to make sure there is no background subtraction or averaging performed. This Matlab library will take care of all that itself. In order to export a raw data file, you should load the data into THCV, click Analyze Data, and set "#scans to average for background," "# of scans to average for CV," and "# of points to average for I vs T" to zero. Then click "Write out Color Plot." See Figure 1.

In the simplest case, you can now load Matlab and execute a single command to generate a CV.

```
filename = 'path-to-raw-file';
cv = autoCV(filename);
```

This just calls the function "autoCV" on the file. You can also load the file into Matlab, and then call the function on the data itself:

```
filename = 'path-to-raw-file';
data = load(filename);
cv = autoCV(data);
```

However, this is apt to provide less informative error messages during batch processing, because "autoCV" will not know which file has caused a possible error. To process a batch of files located in a single directory, you can write something like:

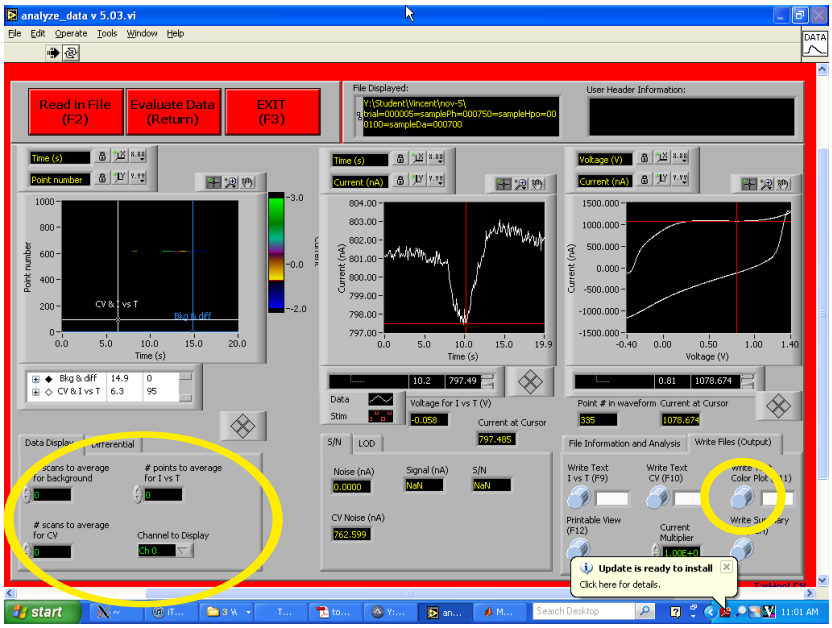

Figure 1:

```
filenames = ddirnames('/some-directory/'); % call a Vincent library
                                          % function to generate a list of all filenames.
for fi=1:length(filenames)
  filename=filenames{fi};
  cv = autoCV(data); % generate the CV.
  newName = strrep(filename,'.txt','-cv.txt'); % generate a new name
                                               % to save the cv to.
  save(newName, '-ascii', 'cv'); % save it.
end
```

0.3 How it Works

"AutoCV" is a function which does a few things. The first task, in broad terms, is to locate the scans in the data set corresponding to the presence of the sample in the flow cell. This process is automated based on the rather modest assumption that these scans will be different than the background scans (this assumption fails when a blank is injected, but then there is no way to tell the difference between sample and background). More formally speaking, each scan from the data set is treated as a vector, and then the Euclidian Distances between these vectors are used to find when the sample appears in the flow cell, and when it leaves. It does this by trying to find the locations for the entrance

and exit time which maximize the distances between putative "sample" scans and "background" scans.

Once we've found the division between background and sample trials, we can estimate the background and remove it. We'd like to be conservative about background estimation, so we pad the boundaries of the sample before finding background trials to estimate.

"AutoCV" can perform background subtraction in a variety of ways. It takes optional arguments which allow the user to customize how it performs background subtraction. If you open the function definition file, you'll see:

```
function [cv,aux] = autoCV(data,varargin)
% [cv, aux] = AUTOCV(DATA,VARARGIN) find a CV in the colorplot in DATA
%
% This function attempts to calculate a CV for a flow cell data set.
%  It isolates the sample from the background using differences in the
%  CVs in each column of DATA and then calculates the background
%  trend and subtracts it.
% It supports the following optional arguments which modify the default
%  behavior.
%
% OPTIONAL ARG            | Default Value
%    marginPct              = 2;
%      After the sample CVs are delineated from the
%      buffer CVs, pad the region by this percentage before
%      calculating the background.  Two here corresponds to
%      a 200% pad.
%    filename               = 'UNSPECIFIED';
%      Use this as the filename if raw data was passed in and
%      there is an error.
%    flattenArgs            = {};
%      Arguments to be passed to
%      flattenBackgroundPoly, if it is used to subtract the
%      background.
%    trialSelectionMethod   = 'peak';
%      Specifies the method used to select the trials use for the CV.
%      'peak' means that the maximum power CV in the sample region,
%      after background subtract, forms the center of the CV region.
%    aroundPeak             = 10;
%      Number of CVs around the trialSelected for the sample CV to
%      average to produce the final sample CV.
%    deFlickerData          =  True;
%      Specifies whether to deflicker the data before analysis.
%      This removes "pops and snaps" in the data before any processing.
%      Should be mostly harmless for data that doesn't
%      have this kind of noise.
```

```
%     deFlickerArgs          = {};
%       Arguments to pass to the deflicker function, in addition to
%       the default ones.
%     partitionArgs          = {};
%       Arguments to pass to the function which calculates the sample
%       and background trials.
%     biasTowardsPre         = 0;
%       Enables biasing towards early CVs in the data set as
%       background CVs.
%     useSimpleMean          = 0;
%       Disable polynomial background subtraction and simply use
%       a mean of the background trials as the background to subtract.
%     saveBackground         = 0;
%       Turn this on to save the background CVs, which you might
%       want to examine to detect long term trends in the data.
%     backgroundDirectory    = './autoBackgrounds/'
%       If you enable background CV saving, this is where they go.
%     autoSignFix            = True;
%       Attempt to automatically flip data that has been inverted
%       as a consequence of using UEI mode vs non UEI mode.
```

This shows the degree to which AutoCV can be customized. I've tried to set up defaults which work for most CVs. Optional arguments are passed in like standard Matlab functions, so if you want to enable simple mean background subtraction but disable deflickering you would call autoCV like this:

```
cv = autoCV('a-color-plot-file.txt', 'useSimpleMean', True,...
'deFlickerData',False);
```

AutoCV is pretty smart - I'd estimate it gets the CV right more than 80 percent of the time, but you should always double check the output before proceeding with further analysis steps.

0.4 In Vivo Data

Automatically extracting CVs from flow cell data is substantially easier than similar analysis of in-vivo data. The critical step in extracting a sample CV is finding the CVs in a recording which correspond to the sample and the ones which correspond to the background. In the flow cell, we know that the trials start with background CVs, at some point the sample enters the flow cell, after which we have sample CVs until the sample leaves the cell, at which point buffer should resume flowing. The job of selecting the sample CVs simplifies into selecting just two points: onset and offset. This is a problem domain which can be exhaustively searched (although that is not how autoCV works, in most cases). In Vivo, particularly for transients, there are no guarantees of single chemical events or rapid onsets or offsets. As a consequence, I've developed

alternative tools to deal with this case. Rather than trying to find a single CV in an in vivo set, it makes sense to find multiple candidates, and return a few of them for subsequent analysis.

At the moment, in vivo automatic extraction is more of an aid in finding CVs rather than a bulletproof method. You can extract CVs from in vivo color plots (exported without background subtraction) with the function "autoCVInVivo". As most of the lab is doing in vitro experiments, and this is alpha-level tech, that is probably all the useful documentation for this code at this time.

0.5 Performing PCR

Above you learned about extracting CVs from in vitro (or in vivo) data automatically. Usually the next step in our analysis pipeline is to calibrate a linear model which is able to take a CV and predict a concentration. This can be approached in several ways.

For single analyte runs, I've found its almost always possible to simply pick the peak associated with the analyte that has the highest variability with respect to concentration and perform a regression between concentration and just those points. Flow cell data tends to be of sufficiently low variability that there is more than enough information in a single voltage point to get very high r^2 values.

0.5.1 A note on file naming conventions

People in the lab may have noticed I tend to use crazy long file names. This is for a great reason: it makes writing scripts to manipulate and analyze data much, much easier. Typically, my filenames look something like this:

```
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=001000=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=000500=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=000500=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=000500=course=001=trial=0
bufferPh=740=samplePh=000748=sampleHPO=000160=sampleDA=000500=course=001=trial=0
```

The critical thing to notice here is that these file names tell you everything you want to know about each file in a simple to parse format. This library includes code to transform a file name in the format above into a matlab structure which can be used programmatically.

```
>> dsf('cvs/trial=000005=sampleHpo=000100=sampleDa=000125=samplePh=000748=newRan
```

```
dsf('cvs/trial=000005=sampleHpo=000100=sampleDa=000125=samplePh=000748=newRank=4

ans =

        trial: 5
    sampleHpo: 100
     sampleDa: 125
     samplePh: 748
      newRank: 4
```

"dsf" stands for "destructure file". It is a very simple function which simply splits a file name (after discarding the directory) into a pieces at the "=" signs, and then converts those pieces into name/value pairs, finally loading those into a structure, which it returns.

Suppose you wanted to get all the files in a directory which corresponding to trials with only changes in pH. You could write a script like this in Matlab, assuming you files had the right names.

```
filenames = ddirnames('./cvs/'); % get all the files in a directory
                                 % called 'cvs'
phOnlyFiles = {}; % initialize an empty list for holding the pH filenames
for fi=1:length(filenames) % loop over all the files
  filename=filenames{fi};
  fileInfo =  dsf(filename);
  if fileInfo.sampleDa == 0 && fileInfo.sampleHpo == 0
    phOnlyFiles{end+1} = filename;
  end
end
```

It is extremely common to take subsets of your data like this, and so the library has a lot of code to make these kinds of operations easy to write. For instance, I would code the above like this:

```
queryFunctions; % load a set of functions to help organize filenames.

phOnlyTest = fAnd(mkMatcher('sampleDa',0),...
                  mkMatcher('sampleHpo',0));
% mkMatcher means "make matcher".  It returns a function which accepts
% a file name and which returns true only when the field specified
% is the value specified
% fAnd takes two functions and returns a new function which is true
% only when both of the input functions are true on the inputs to the
% new function.
% So the above code reads as:
%  create a function called pHOnlyTest which accepts a filename as
%  input and only returns true when that file has zero for sampleDa
```

```
%  and zero for sampleHpo.

phOnlyFiles = filt( phOnlyTest, ddirnames('./cvs/'));
% this line of code means"
% filter the list of files returned by ddirnames('./cvs/')
% return only those files which pass "phOnlyTest"
```

Without comments, the above is substantially shorter than the other version. Less code means fewer opportunities for mistakes!

Anyway, you can use this library with any files you want. Chemometrics and CV extraction don't depend on specific filename conventions. But if you want to do data analysis in Matlab more easily, choosing a good convention for filenames is pretty helpful.

0.6 Performing Standard PCR

Performing PCR is simple once you've examined the above examples. Here is a complete listing for a PCR in one of my directories:

```
    allFiles = ddirnames(['./cvs/']);  % get the cv names
    notOutlier = fAnd(@(x) dsf(x,[],'newRank',0)<4,...
                      @(x) dsf(x,[],'black',0)==0); % remove outliers
                           % and blacklisted data
    trainingFiles = unique(filt(notOutlier,[filt(daTraining,allFiles);
                        filt(hpoTraining,allFiles);
                        filt(phTraining,allFiles)])); % select
                           % training data
    testingFiles = unique(filt(notOutlier,filt(testing, allFiles)));
% select testing data

    clear pcrStruct; % make sure the structure containing pcr data is clear
    for ti=1:length(trainingFiles)
    trainingFile=trainingFiles{ti};
    props = dsf(trainingFile);
    pcrStruct(ti).data = load(trainingFile);
    pcrStruct(ti).da = dsf(trainingFile,[],'sampleDA');
    pcrStruct(ti).hpo = dsf(trainingFile,[],'sampleHPO');
    pcrStruct(ti).ph = dsf(trainingFile,[],'samplePh');
    end
% the above fills the pcr struct.

    testingData = loadFiles(testingFiles);
% load the data for testing.
    [models,aux] = doPCR(pcrStruct, testingData, ccm);
% perform PCR
```

The function doPCR returns a struct of models for each of the analytes indicated in the pcrStruct (in our example, da, hpo and ph). Models.da, for instance, contains the predicted and actual Dopamine concentrations in an array. The order corresponds to the training and then the testing data passed in, in their orders.

You can plot the results for the training data like so.

```
das = map(dag, allFiles);   % dag is a function defined in
                            % queryFunctions which gets the recorded
                            % dopamine concentration from a file
                            % name.  Map applies a function to a list
                            % of things and returns a list of the
                            % results.  So the above line returns a
                            % list of all the prepared dopamine concentrations.
nTr = 1:length(trainingFiles);
plotWithRedundantXs(das(nTr), models.da.predicted(nTr));
% plotWithRedundantXs makes a scatter plot of the data where Xs may be repeated.
```

Further documentation is forthcoming, but an intrepid analyst could start from these examples and go!