2013

Integrated Analysis Platform of Brain Wave Data Implementation and Testing Results

Team Dec13-17

EE/CprE/SE Senior Design

12/1/2013

Table of Contents

Implementation

Implementation Block Diagram

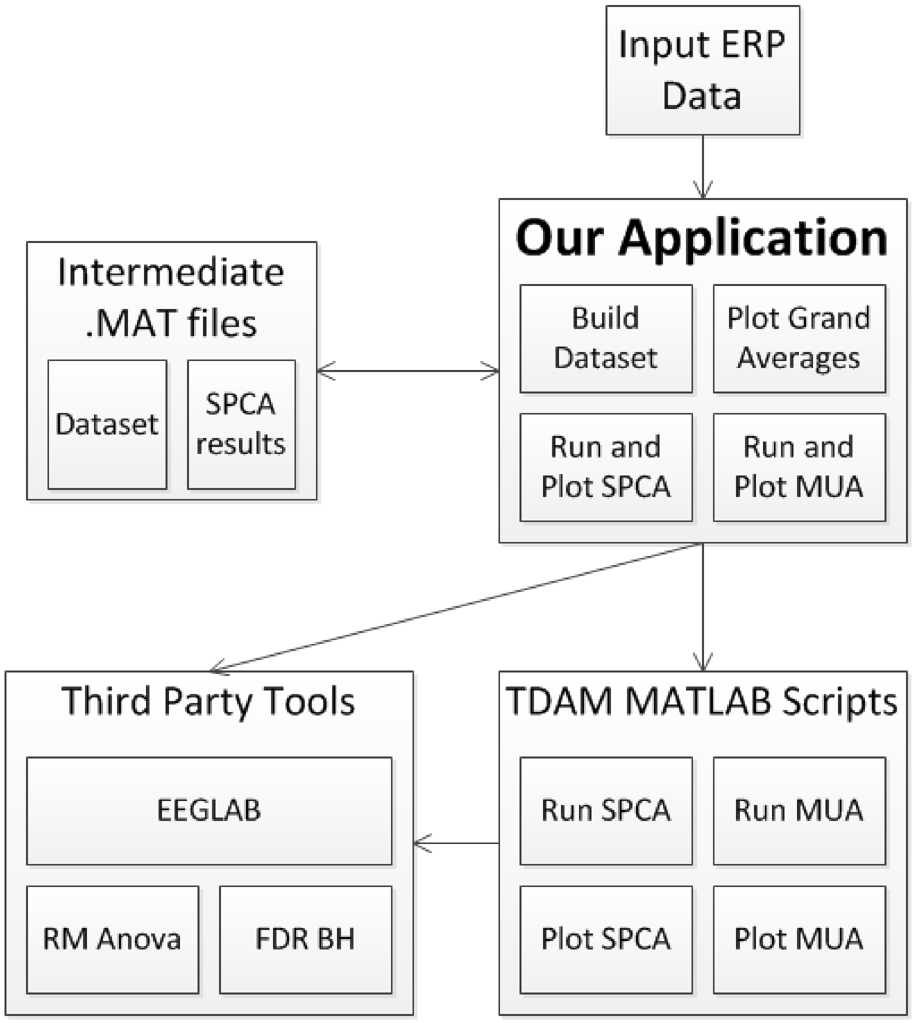


Figure 1.1: Block Diagram of our Implementation

This block diagram represents our complete solution. Our application required refactoring existing TDAM legacy scripts into reusable Matlab functions. Some of these scripts required to communicate with third-party tools, which our application also utilized.

Our application leveraged the TDAM source, third party tools, and our own source to input raw ERP data, store analytic results in intermediate .mat files, and interpret these results as ERP's and topographies, which are displayed to the user on demand.

All of these operations are now done at a click of a button, unlike how they were originally generated through a mixture of repetitive file structuring tasks and matlab command line calls.

Results Generated

The results generated by our application come in several forms: Grand-Averaged electrode locations, Spatial PCA results (in raw form, promax rotation, and varimax rotation), and Mass Univariate Analysis Results. A brief description of each type of result is shown below.

Grand-Averaged Electrode Locations

The first type of results our application generates are the Grand-Averaged Electrode locations. Upon building a dataset, the researcher will generally check this result to validate they have the correct input data. This result draws a 2D map at each electrode location. Each electrode displays its electrode name, and the grand-averaged ERP response for each subject and condition.

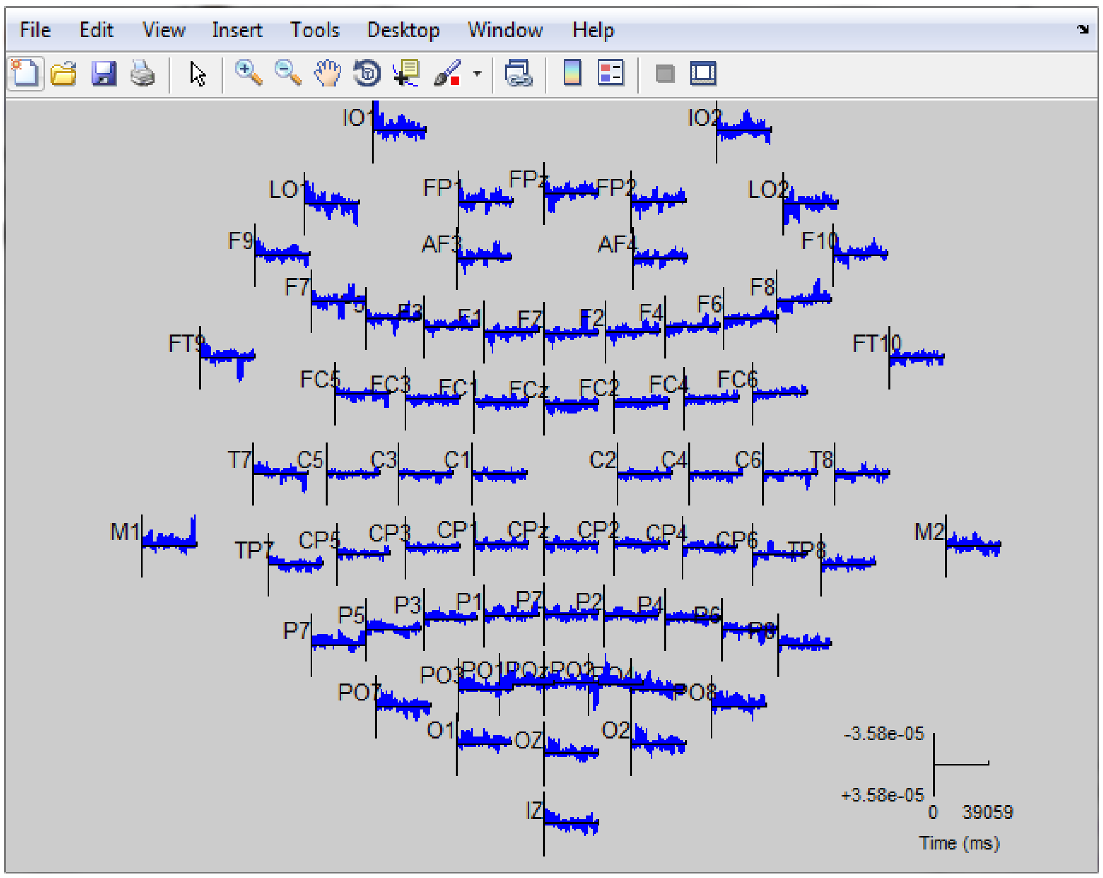


Figure 2.1: Grand-Averaged Electrode Locations

The researcher may also choose to zoom in on an electrode if they wish by double-clicking the electrode. The zoomed-in electrode ERP will appear in a new window.

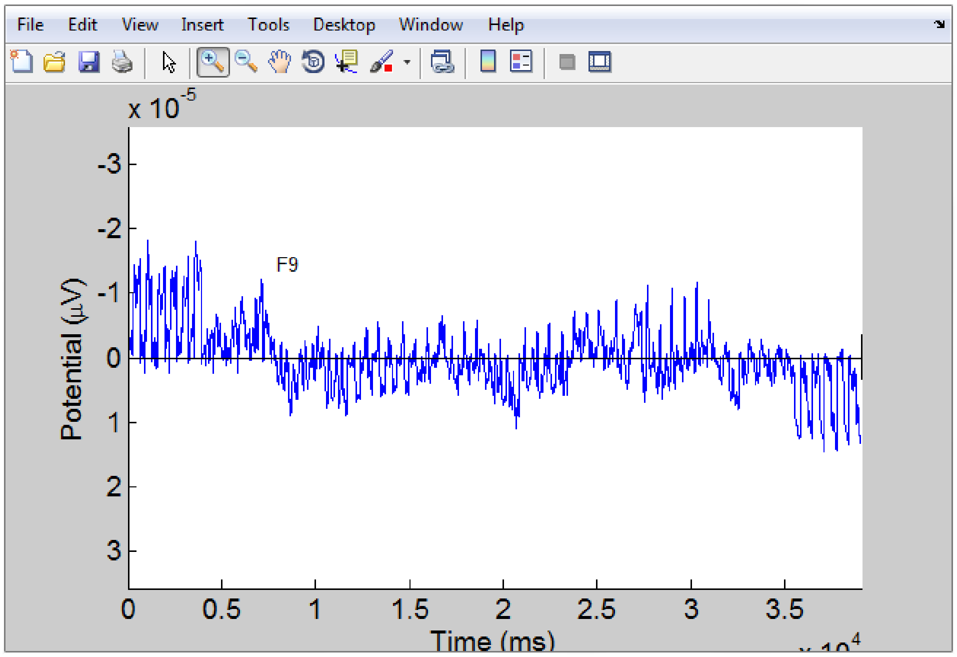


Figure 2.2: Zoomed-In Grand-Averaged Electrode

If the researcher approves of their input data, they will move on to Spatial Principal Component Analysis.

Spatial Principal Component Analysis (Spatial PCA) Results

The next type of results our application generates are the Spatial PCA Results. At this step, the researcher inputs the dataset, selects which electrodes they would like to include in the analysis, and what epoch (time period) they would like to include. The Spatial PCA computes the data and returns a paired ERP and a Topography for each Spatial Component it determines it must retain.

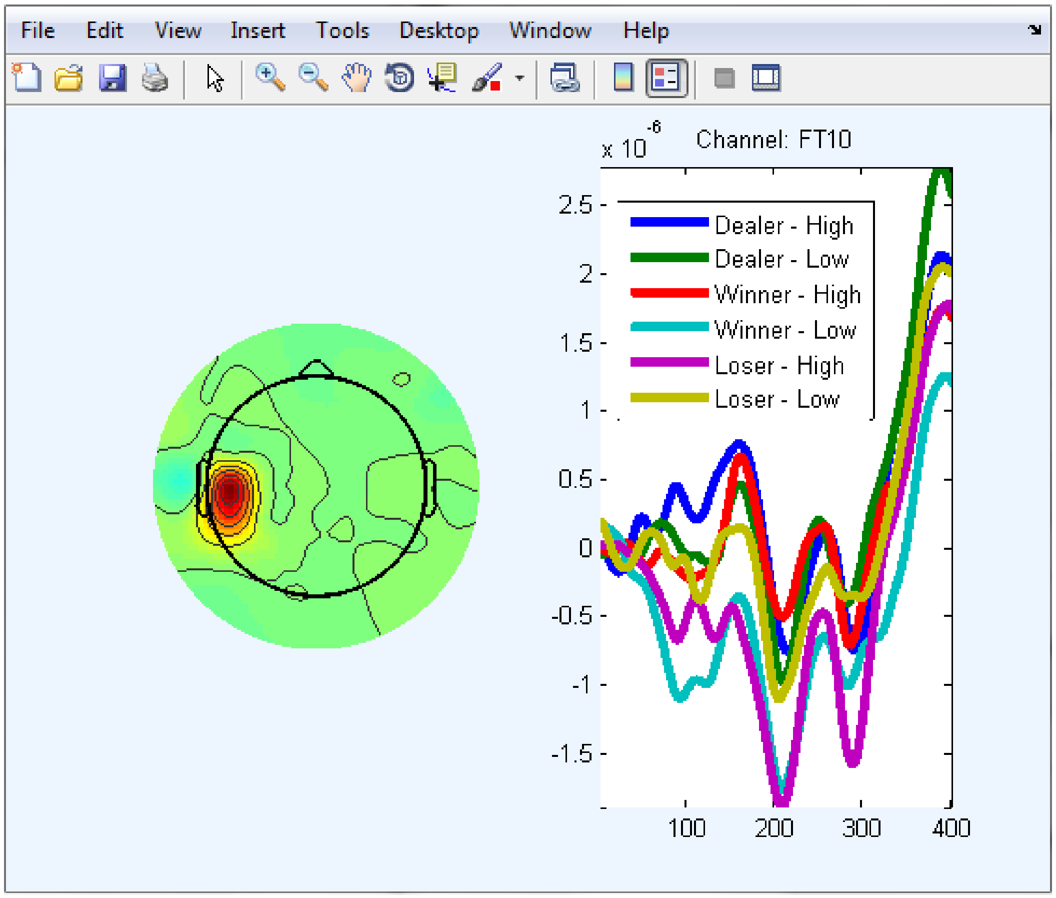


Figure 2.3: Spatial PCA Topography (left) and ERP (right) for a Spatial Component

After performing Spatial PCA and approving of the results, the researcher will move on to Mass Univariate Analysis.

Mass Univariate Analysis Results

The third type of results our application generates are the Mass Univariate Analysis Results. The Mass Univariate Analysis Results return an ERP for each Spatial Component from the Spatial PCA.

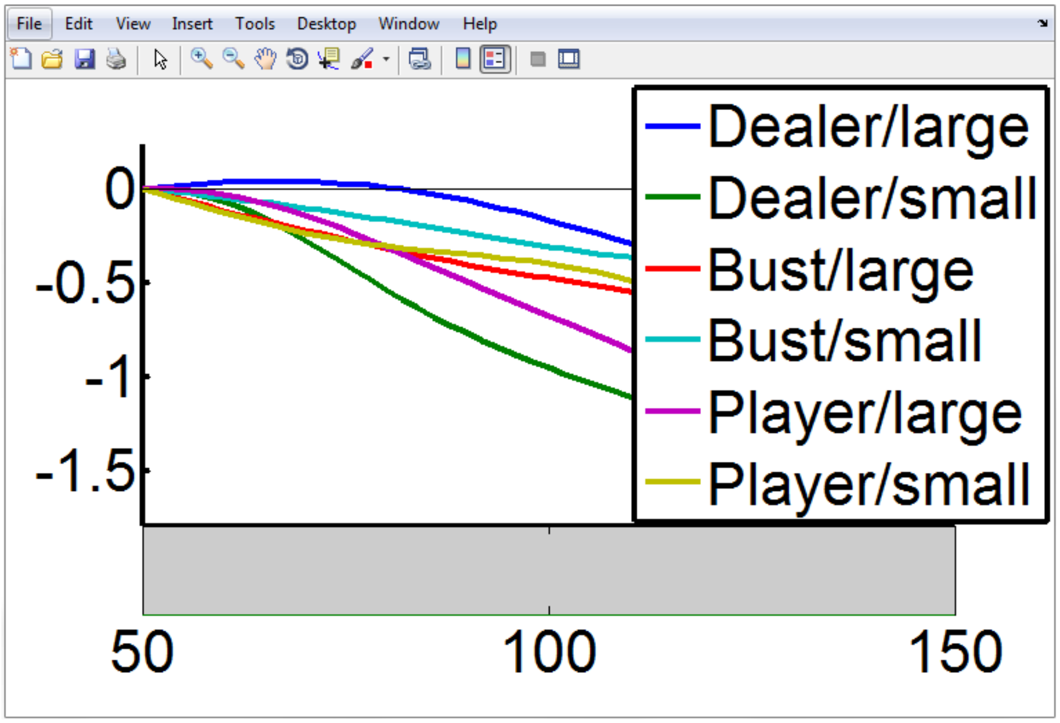


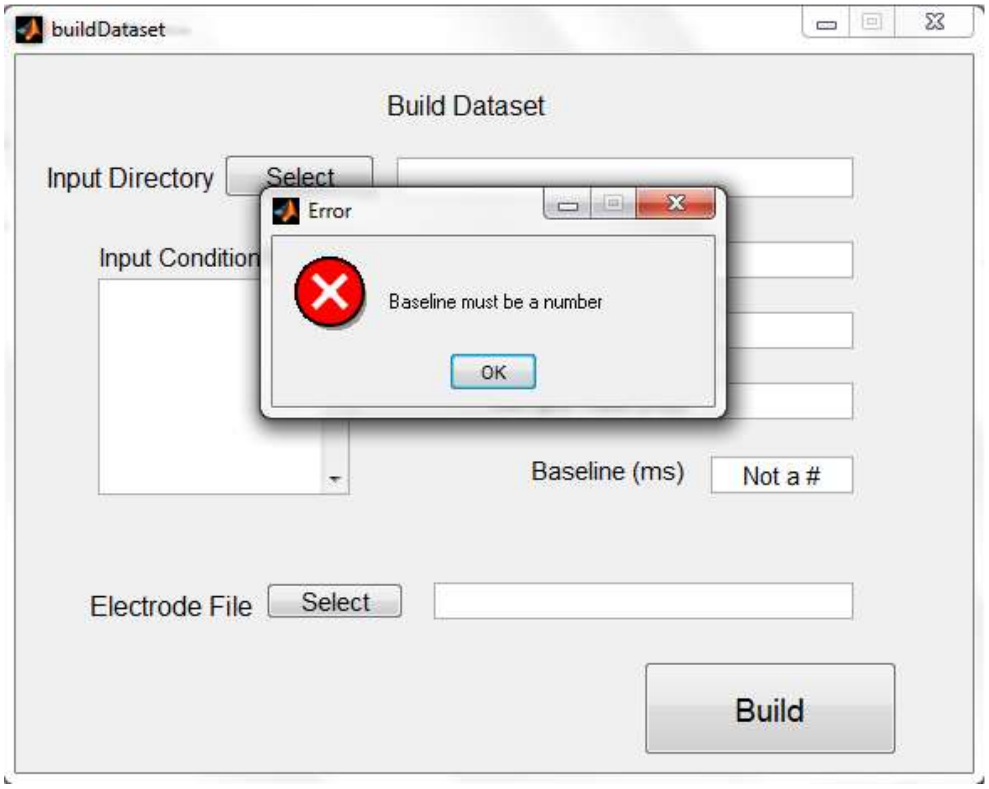
Figure 2.4: Mass Univariate ERP (right) for a Spatial Component

Error Checking

In our solution, we implemented various types of error checking. A direct request from our client was to make the solution do extensive error checking. Previous similar products they had used lacked a lot of this, which led to frustration.

Input Value Validation

One type of error checking we implemented was input value validation. Every input field in our solution checks that the input value is the correct format (integer, string, etc). An example is shown below.



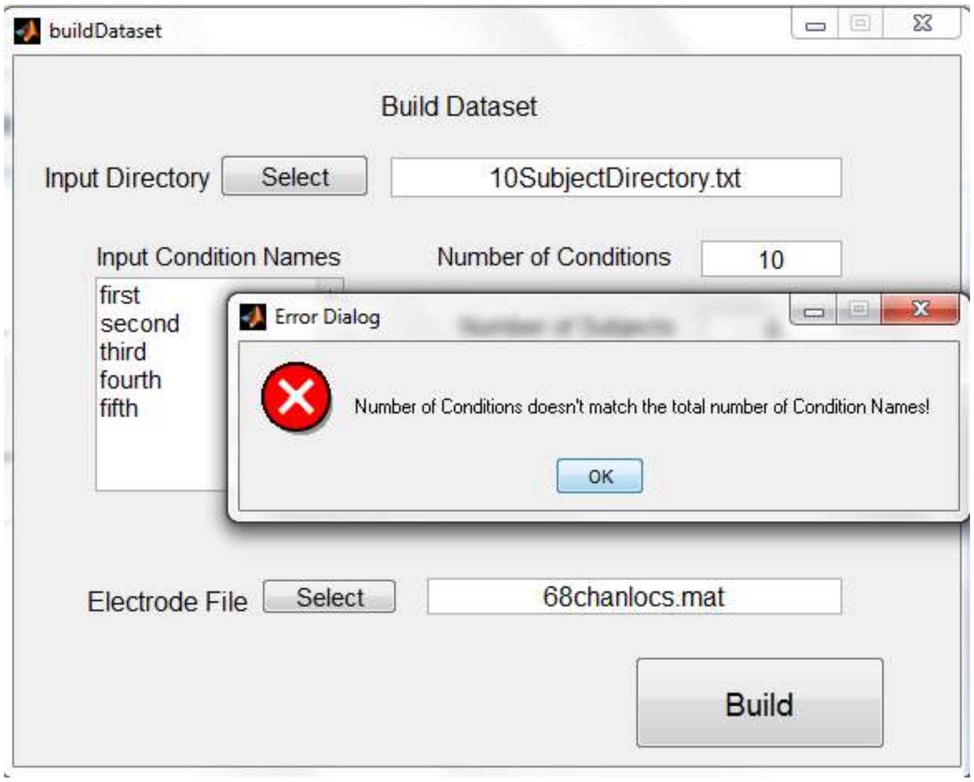
Baseline must be a number

Baseline (ms) Not a #

Figure 3.1: Error Checking for Baseline

Analysis Data Validation

Another type of error checking we did was analysis data validation. Every analysis our solution performs, contains error checking for correct input parameters before doing any calculations. For instance, before building a dataset, our application makes sure the user has input the correct number of subjects and conditions, and these map correctly to the number of input files.



10SubjectDirectory.txt

10

first
second
third
fourth
fifth

Number of Conditions doesn't match the total number of Condition Names!

68chanlocs.mat

Figure 3.2: Error Checking for Dataset Parameters

Object-Oriented Implementation

Matlab is a procedural language by nature, but has some support for object oriented programming. We left our application design, from Senior Design I, open so that we could decide whether or not to utilize Object-Oriented Matlab. Once we began experimenting with Matlab OO programming, we converted our project over to completely Object-Oriented code. Specifically, we have implemented a Model-View-Presenter (MVP) architecture into our application.

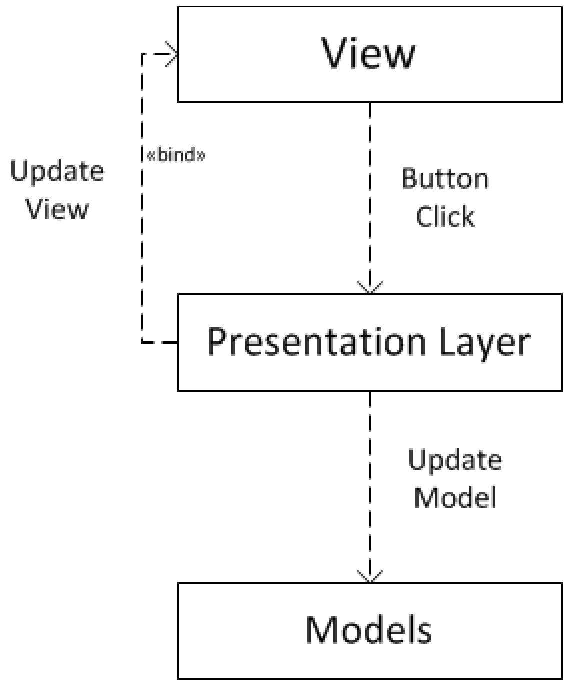
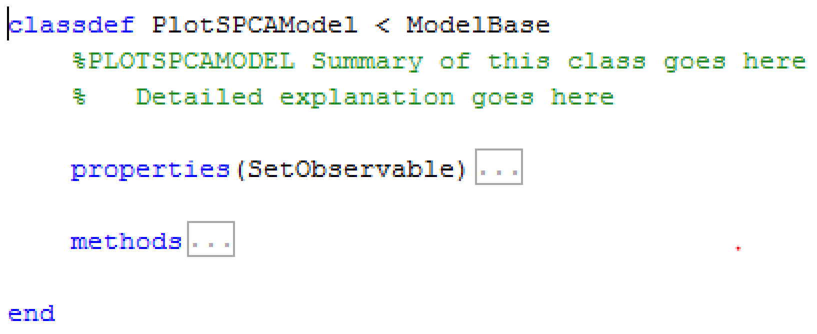


Figure 4.1: Our MVP Architecture

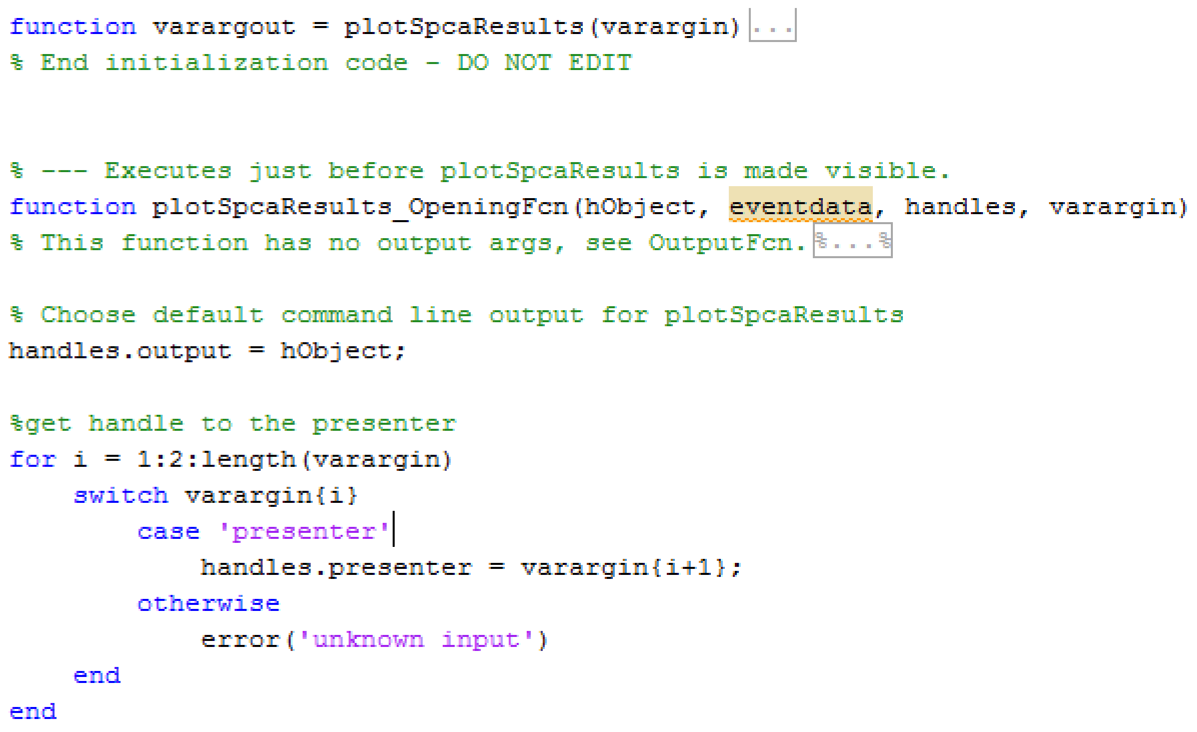
The *Model* classes contain all of the Matlab data structures and Analyses within the brain wave application. It inherits a *ModelBase* class, and has no notion of the view. It is completely business logic.



```
classdef PlotSPCAModel < ModelBase
    %PLOTSPCAMODEL Summary of this class goes here
    %   Detailed explanation goes here

    properties(SetObservable) ...

    methods ...

end
```

Figure 4.2: Plot SPCA Model, containing only analyses and data

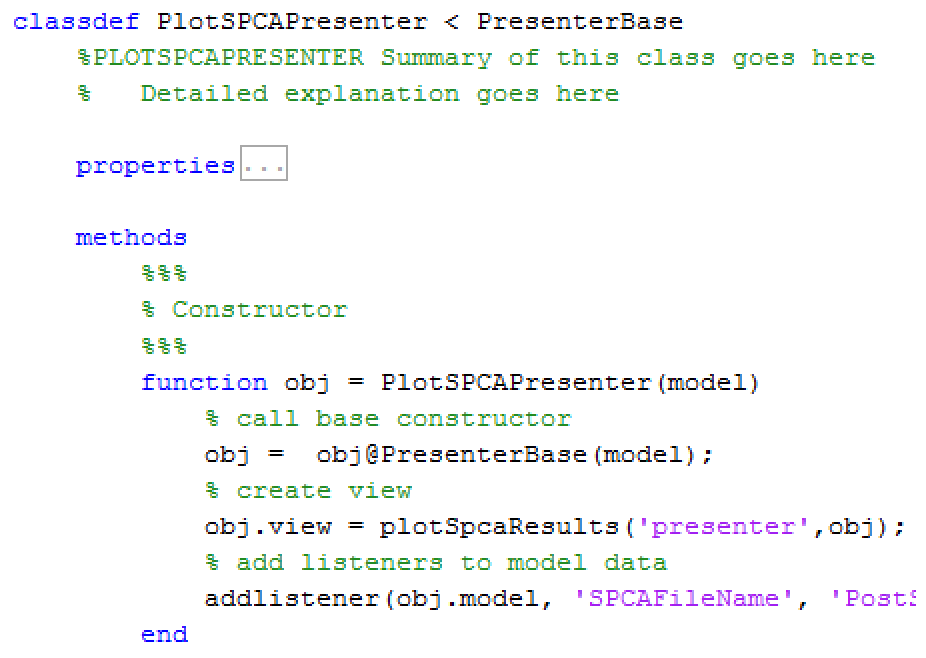
The View classes are the 2D forms you see, as Matlab .fig files, coupled with a Matlab GUI template, as .m source files. The Views reference a handle to the presenter on startup (OpeningFcn method is equivalent to the GUI’s constructor). When the user interacts with the *View*, it calls a method in the *Presenter*.



```
function varargout = plotSpcaResults(varargin)...
% End initialization code - DO NOT EDIT


% --- Executes just before plotSpcaResults is made visible.
function plotSpcaResults_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.%...%

% Choose default command line output for plotSpcaResults
handles.output = hObject;

%get handle to the presenter
for i = 1:2:length(varargin)
    switch varargin{i}
        case 'presenter'
            handles.presenter = varargin{i+1};
        otherwise
            error('unknown input')
    end
end
```

Figure 4.3: Plot SPCA View with OpeningFcn(), that binds to the Presenter

The *Presenter* classes separate the *Model* from the *View*. The *Presenter* decides what actions the *Model* performs in response to the *View*. The *Presenter* also listens to the *Model's* properties. When a *Model* property updates, the *Presenter* updates the *View* accordingly.



```
classdef PlotSPCAPresenter < PresenterBase
    %PLOTSPCAPRESENTER Summary of this class goes here
    %   Detailed explanation goes here

    properties...

    methods
        %%%
        % Constructor
        %%%
        function obj = PlotSPCAPresenter(model)
            % call base constructor
            obj =  obj@PresenterBase(model);
            % create view
            obj.view = plotSpcaResults('presenter',obj);
            % add listeners to model data
            addlistener(obj.model, 'SPCAFileName', 'Post
        end
```

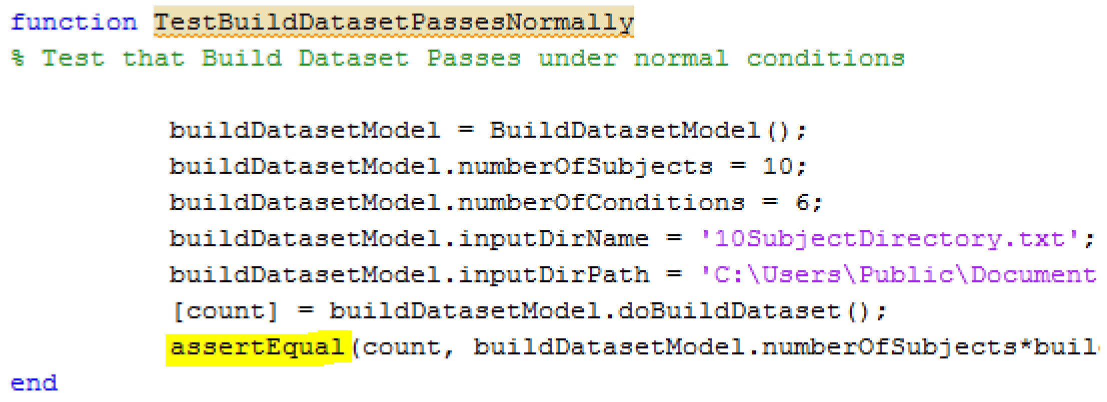
Figure 4.4: Plot SPCA Presenter with Constructor that listens to the Model

Testing Results

Unit Testing

In addition to the Matlab Framework, we used Matlab *xUnit* Testing Framework to perform Unit Testing on most of the methods we wrote. Every large analysis (Dataset, SPCA, MUA), we wrote at least three unit tests, one test for the standard use case, then at least two for alternative/exceptional cases.

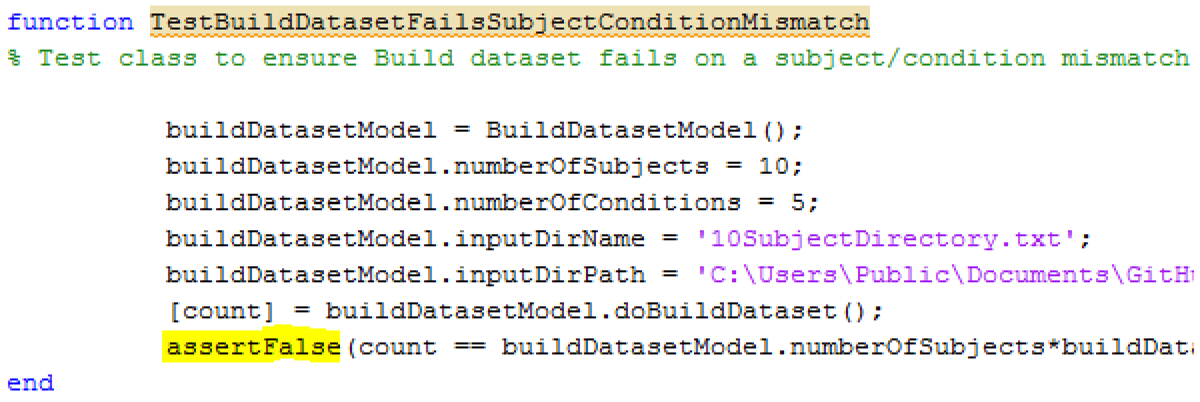
An example of one of our “standard use case” unit tests is shown below. This unit test checks the Build Dataset functionality under normal conditions. The *assertEqual* is a Matlab xUnit testing function, which passes if two values are equal.



```
function TestBuildDatasetPassesNormally
% Test that Build Dataset Passes under normal conditions

          buildDatasetModel = BuildDatasetModel();
          buildDatasetModel.numberOfSubjects = 10;
          buildDatasetModel.numberOfConditions = 6;
          buildDatasetModel.inputDirName = '10SubjectDirectory.txt';
          buildDatasetModel.inputDirPath = 'C:\Users\Public\Document
          [count] = buildDatasetModel.doBuildDataset();
          assertEqual(count, buildDatasetModel.numberOfSubjects*buil
end
```

Figure 5.1: Unit Test for the Standard Build Dataset Use Case

An example of one of our “alternate/exceptional” use case unit tests is shown below. This unit tests checks the Build Dataset functionality when the number of subjects and conditions doesn’t line up with the input files. The *assertFalse* is a Matlab xUnit testing function, which passes if the expression returns false.



```
function TestBuildDatasetFailsSubjectConditionMismatch
% Test class to ensure Build dataset fails on a subject/condition mismatch

          buildDatasetModel = BuildDatasetModel();
          buildDatasetModel.numberOfSubjects = 10;
          buildDatasetModel.numberOfConditions = 5;
          buildDatasetModel.inputDirName = '10SubjectDirectory.txt';
          buildDatasetModel.inputDirPath = 'C:\Users\Public\Documents\GitH
          [count] = buildDatasetModel.doBuildDataset();
          assertFalse(count == buildDatasetModel.numberOfSubjects*buildDat
end
```

Figure 5.2: Unit Test for the Alternate Build Dataset Use Case

White-Box and Black-Box Testing

In addition to Unit-Testing, each developer on Team 17 performs their own white-box testing to ensure each property, method, and class they submit to our repository is functions the way it is intended. White-box testing involves using several different use cases, and debugging step-by-step through each line of code they write.

Black-Box testing was performed by each developer in a similar fashion to white-box testing. The key difference is when the developer is testing; they test it at the application-level, where no interaction with code is involved. They do this to ensure usability, performance, and that all requirements are met for this piece of functionality.

When each developer committed a piece of code, they ensured that they have white-box tested over 70% of committed lines of code, and black-box tested for every applicable requirement. They ensured that all of their code performed as expected, and there are zero critical defects, zero major defects, and no known minor defects at the time of the commit.

Usability Testing

Team 17 met with our client multiple times throughout both semesters, in which we performed usability testing on both our Wireframe Mockups, and early versions of the software application. These were a mostly informal process, but changes made to the application were immediately reflected in new revisions of the design document, as well as in weekly reports. Our final usability testing session was performed on November 7th, in which our client reported satisfied with the existing application usability.

User Acceptance Testing

Team 17, in conjunction with Dr. West and his colleagues; have performed a two-part User Acceptance Testing. The first part was performed on November 7th, along with final Usability Testing. We tested whether the data and analyses running through the application were performing successfully, and whether the application was “acceptable” as a whole. Our client reported some minor bugs and feature requests, but had an overall positive outlook on the project.

Our second phase of User Acceptance Testing will be performed during Dead week (12/9-12/13). This will be the formal “Adopt/Not Adopt” decision by the client. The application is expected to be fully functional, with no critical issues. After this meeting, the entire application, with source code, will be given to the client for his indefinite use.