Extensions to the Feedback Systems Web Laboratory Client Prototype

Sriganesh Lokanathan

AUP Final Report

MIT EECS

July 21, 2004

Supervisor: Dr. Kent Lundberg

To Mahju

Acknowledgments

I would like to thank my supervisor Dr. Kent Lundberg for his patient, abundant and insightful guidance during this project.

I would also like to thank David Zych whose work for the MIT Microelectronics Weblab served as a framework for the Feedback Systems Weblab Client.

Thanks is also due to Gerardo Viedma for his work on the Weblab Client prototype structure, and to Brian Williams who created the original graphing applet to produce Bode, Nichols and Nyquist Plots.

Contents

Chapter 1

Introduction

Remote access to laboratory equipment is already being implemented at MIT via the iLab initiative[1], a project under the iCampus initiative here at MIT. The project has already demonstrated success in maximizing usage of expensive equipment while maintaining the safety of equipment use and maintenance and maximizing the learning experience for students, while allowing them 24 hour access opportunities.

The current development effort to build a Weblab system for the MIT Feedback Systems class (henceforth referred to as Weblab), has leveraged quite a bit of the tools and framework provided by the Weblab system that was developed for MIT's Microelectronics Devices and Circuits class [1]. The current Weblab prototype consists of the of the extensions and re-implementation work done by Gerardo Viedma [2] to adapt this original system to the Feedback Systems Weblab as well as the graphing tools developed by Brian Williams [3].

This paper describes the extensions that were done to the prototype Weblab client to enhance its results-handling functionality as well as to enhance the graphics capabilities of the prototype Weblab Client.

[1] http://ilab.mit.edu

Chapter 2

Weblab Client Extensions

While some of the extensions that were done were for data handling and communication, the bulk of the extensions were to increase the versatility and functionality of the graphical engine.

The Weblab Client was augmented with functionality to allow the user to export the results as well as to be able to save their experiment setups for reuse. In extending the graphical engine, the primary objectives were to enhance the user's experience while extending the versatility of the graphing functionalities. With this in mind, changes were made to handle multiple graphs with automatic axis resolution. Also all plotted graphs can now be saved locally by the user as JPEG image files. In addition, usability was enhanced by giving the user direct access to manipulate a drawn graph by the implementation of a right-click activated popup menu.

2.1 Implementation of client side validation checks

In order to reduce the burden on the lab server, some client side validation was added to the input before the experiment specification is submitted by the client. The decision was made to implement only limited validation checks to the client applet, so that the applet jar file size could be kept to a minimum.

The validation checks added were simply range checks on the input data. While generating an experiment specification for submission to the Lab Server, each numerical input was validated against its corresponding ranges as specified by the Lab Configuration. Should such range errors exist, the user is prompted to change the incorrect inputs such that they conform to their respective pre-set range limits.

2.2 Exporting experiment results

The results received by the lab client always consist of an XML data stream containing three data vectors, namely Frequency (in Hz), Magnitude (in dB) and Phase (in degrees). This data is used by the client to generate the required graphs. The decision was made to extend the use of this result data set by allowing the user to export the raw data into local files. The idea was that the user could then import this data into some mathematical or graphical package.

Two relevant formats for the exported results data were as a Comma Separated Value (CSV) file or as a Matlab file (*.m files). The existing client framework that we used already had some embedded code to export the data as CSV files. I used this as a guide to re-implement the "Export Results" functionality to allow for data export as an ASCII-format Matlab file as well as a CSV file. Even the original existing code was re-implemented because of the extension of the graphical package to allow plotting of multiple result sets[2].

2.3 Implementation of Setup Storage

The iLab infrastructure allows lab clients to save and load experimental setups (i.e. the users data inputs for a particular experiment) on the Service Broker server [4]. The client applet framework that we used already had some code to implement this functionality for the 6.012 Lab Experiments [1]. The code used for the communication was reused mostly unchanged. However, the *ExperimentSpecification* unmarshaller[3] class was re-written. The new unmarshaller code for the loading the setups was based mainly on the code written earlier for unmarshalling the Lab Configuration [2].

In addition, functionality was also added for the user to save and load setups on the local machine. This extra feature adds a layer of redundancy so that the client (in this case the user) is not dependant solely on the Service Broker.

The process of saving a setup involves creating a valid experimental specification and then writing that to an XML file (if the setup is stored on the local machine) or transmitting it for remote storage in the Service Broker's database. Thus, the process of saving a setup also runs the inputs through the client-side validation checks prior to saving.

One layer of checking that was added, was to make sure that the setup that the user tries to load is valid for the latest experiment (i.e. Lab Configuration) that the Weblab client is configured for. This was necessary to prevent users from accidentally trying to run older experimental setups on a Lab Configuration that didn't support it.

[2] Refer to Section 2.4.3 Exporting and Viewing Graph Data.

[3] An unmarshaller class is a class that governs the process of de-serializing XML data into Java content trees. The new *ExperimentSpecification* class also validates the XML data to make sure it is properly formatted before de-serializing the XML data into *Input* objects.

2.4 Extensions to the Graphical package

2.4.1 Frequency Display.

The existing applet's graphing package was only taking in the Magnitude and Phase data and distributing them evenly over an assumed range of frequencies. This was appropriate for the original graphing applet [3] whose purpose was different. However, we needed to plot the received frequency data. The *FreqResponse* class was re-implemented to store the actual received values instead of generating a fixed frequency spread everytime.

2.4.2 Plotting multiple graphs

The existing applet allowed the user to plot only one set of results at a time. It didn't have any functionality to allow the user to compare multiple generated graphs with each other. It was decided to extend the applet to allow for this functionality.

A careful examination of the code implemented for the original graphing applet [3] for the graphing, revealed that the graphing framework he created could easily be extended to handle multiple graphs. I implemented a few new methods to this framework, which allowed the client to retain the existing graph data when executing the experiment with a new set of input values. Williams' graphing framework [3] consisted of a *LinearSystemHolder* object, which holds a vector of systems (i.e. Frequency Response and/or Step Response). The *LinearSystemHolder* class was extended with some new methods to selectively delete a system and to retrieve information about a particular system. This information could be the Frequency, Magnitude and Phase data and the color of a specific system. The *LinearSystemHolder* object could also be queried to return the index of a system with a particular color[4].

A new interface *GraphColors* was also created in the *graphutils* package to hold two constant arrays. One was a *Color* array (*COLOR_SELECTION)* to hold all the possible graph colors and the other was a *String* array (*COLOR_NAME*) to hold the names of the colors. The length of the arrays (which had to be equal) determines how many graphs can be drawn. If in future the need arises to allow the user to be able to plot more graphs, the only change required is to add a new color to the *COLOR_SELECTION* array and enter its corresponding name (in its corresponding index position) in the *COLOR_NAME* array.

[4] Refer to Section 2.4.6 Direct Graph Manipulation

In addition, the local object that holds the results data (which is called *FreqResponse*[5]) was extended to assign a unique color to each graph. Whenever *LinearSystemHolder* instantiates a new *FreqResponse* object it assigns it a color by cycling through the list of colors and assigning the first available un-used color

Additional code was also written so that the applet had a mechanism through which the user could change the mode under which the graphing occurred. The two modes were "Always Replace" (also the default mode) and "Always Add". These changes were implemented in the *graphicalUI* package to make use of the new methods implemented to the *LinearSystemHolder* class.

2.4.3 Exporting and Viewing Graph Data

Once the applet was extended to allow multiple graphs to be displayed, some of the earlier functionality had to be re-implemented. In particular the “Export Results” and “View Results” functions had to be re-implemented so that the user first chooses the graph whose data set he wishes to export or view.

In re-implementing these functions, I could no longer directly read from a *Results* object, since this only contained the last set of results returned by the Lab Server. Instead, the results data is now read off the *FreqResponse* objects that are instantiated for each Graph.

2.4.4 Importing saved results data

Since the applet had the ability to export the experiment results into a Matlab or CSV file, a logical extension was to add the functionality to also import the data from saved results files. A new *DataImport* class was created to handle this functionality with a new DataImportException class as well. The *DataImport* class can easily be extended to handle other file formats by just writing a new method for the new file format.

Extra code was also added to the *MainFrame* and *ResultsPanel* classes under the *graphicalUI* package to handle this new functionality.

[5] The name maybe a bit misleading since the underlying graphing package was imported from Williams’ original applet [3].

2.4.5 Variable Axis

The underlying graphical framework [3] had some limitations when it was integrated into our client. One limitation was that all the graphs had a fixed axis. The grid consisted of three vertical and three horizontal lines. Since the panel on which the graph was drawn was used entirely to display the graph, it meant there were no labels for the edges of the graph grid.

The axis has now been extended to allow for variable axis ranges while at the same time allowing for more grid lines. When drawing the axis, the minimum and maximum points (which make up the edges of the graph) are calculated for each axis by examining the minimum and maximum values of all the different graphs that are to be drawn.

Based on the minimum and maximum values of the actual graph data (i.e. frequency, magnitude and phase) the minimum and maximum points (i.e. edges) of the axis are calculated by rounding off these values to the nearest "regular" value. The "regular" minimum and maximum value for an axis is calculated differently depending on whether the variable is frequency, magnitude or phase.

<u>Frequency</u>

The minimum frequency among all the graph data is determined and then rounded down such that it is the nearest power of 10. This value constitutes the minimum value of the frequency axis. Similarly, the maximum frequency among the graph data is rounded up to the nearest power of 10. This calculated value is the maximum value of the frequency axis.

<u>Magnitude</u>

Here the minimum and maximum axis values are determined by the range of the entire set of magnitude values that will be plotted. The following chart describes this approach. The minimum and maximum magnitude values are rounded down and up respectively to the nearest multiple of *RoundOffValue* which in turn is dependant on the range. The *RoundOffValue* is also the distance between any two grid lines on the magnitude axis.

Range (dB)	**RoundOffValue (dB)**
range<=10	1
10<range<=20	2
20<range<=40	5
40<range<=80	10
Range > 80	20

Phase

The minimum and maximum axis values are calculated in the same manner as that for magnitude however the *RoundOffValues* and the corresponding ranges that it applies to are different. Here too, the *RoundOffValue* is the distance between any two grid lines on the phase axis.

Range (degrees)	**RoundOffValue (degrees)**
range<=10	1
10<range<=45	5
45<range<=90	15
90<range<=180	30
range > 180	45

2.4.6 Direct Graph Manipulation

From a usability point of view, I decided that it would be convenient to the user to be able to directly manipulate a graph without having to go through the process of choosing a graph function and then having to choose the graph. I decided to implement a popup menu that could be obtained by right clicking the mouse over a graph. This menu allows the user to directly access all relevant functions related to a particular graph. These include exporting the graph results data, viewing the graph results data, deleting the graph and deleting all graphs.

The process of associating the popup menu with the right graph was done by first getting the color of the pixel under the mouse. If that color was one of the graph colors then a popup menu is created with that color which in turn uses it to calculate the appropriate index of the graph in the vector of systems.

In case that the color of the pixel directly under the mouse is not a graph color then it looks for the first graph color that may exist in the 8 pixels surrounding the central pixel (basically a 3x3 pixel matrix with the central pixel being the one the mouse is under). The 3x3 pixel matrix size was determined by testing how the mouse behaved. It is a large enough size so that the user doesn't have to precisely maneuver the mouse pointer over the exact graph pixel, while at the same time being not so large that it effects the resolution of choosing a specific graph if other graphs also have points which intersect the pixel matrix.

In order to utilize Java’s *PixelGrabber* class to resolve the color of a pixel, the implementation of how the graph was painted was changed. The existing applet would draw to a Java *Graphics* object, which was then finally painted on the panel. This was changed so that now it draws to a *BufferedImage* object, which is then painted on the screen. In doing this change I also realized that I could store a copy of the *BufferedImage* in memory instead of have to rebuild it every time the panel needs to be repainted, when another window intersects its viewing rectangle. This resulted in a slight increase in performance, with a new *BufferedImage* being calculated only when a graph is added or deleted.

In order to implement this functionality most of the classes under the *graphutils* package had to be changed. This was mainly so that the *MainFrame* object could be passed downward to the individual graphing classes from where the popup menu is instantiated.

2.4.7 Saving Graph Plots as JPEG Image Files

The main users of the Weblab client would be students utilizing it do class assignments. This meant that they would potentially have to hand in the graphs that were plotted in running the experiments. With this in mind, the Weblab client was extended with the functionality to save the plotted graphs (Bode Plot, Nichols Plot and/or Nyquist Plot) as JPEG images.

This functionality was implemented by including a sub-menu in the Graph Menu in the *MainFrame* class, which allowed the user to choose which graph plot to save. This action resulted in a call to the *getGraphImage* method in the *ResultsPanel* class, which in turn calls the corresponding *getGraphImage* method in the *OutputGraphs* class (in the *graphutils* package). The *getGraphImage* method in the *OutputGraphs* class generates and returns a *BufferedImage* of the *Component* that displays the plot the user was trying to save. The calling method in the MainFrame class then encodes this *BufferedImage* object with a JPEG-Encoder and writes the contents to a .jpeg file.

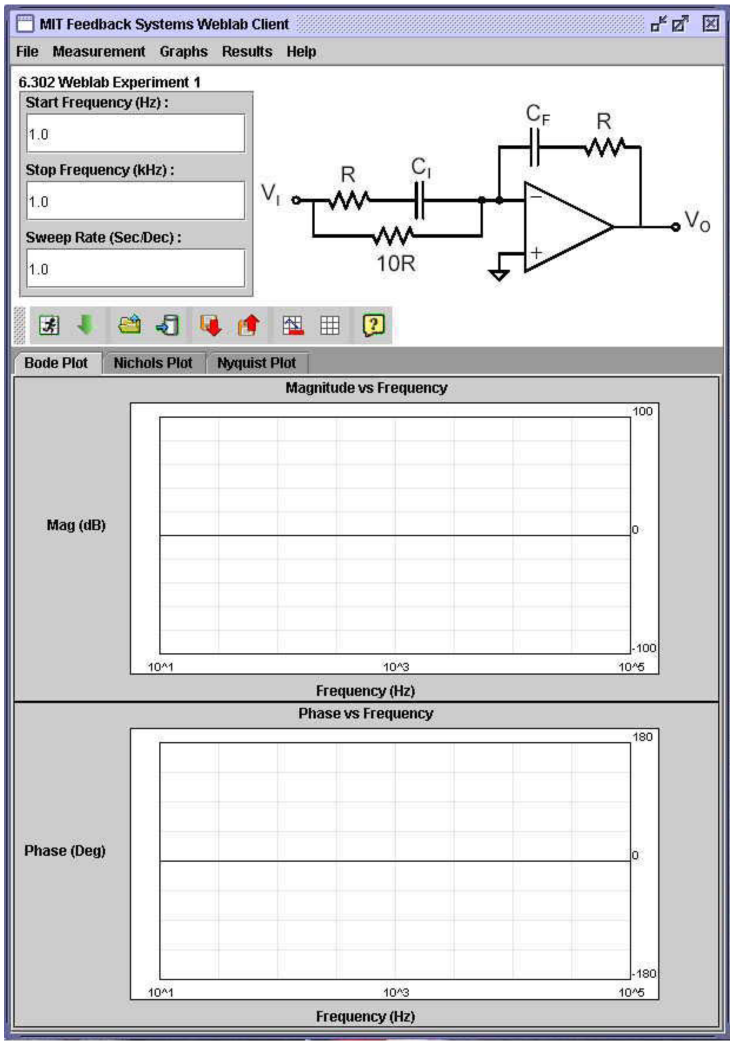
Appendix A

A Typical Usage Scenario of the Weblab Client

When the user launches the client by logging into the Service Broker, the applet retrieves the latest Lab Configuration and configures the client to accept the relevant inputs from the user.

The applet consists of the two sections. The top section contains the input fields and a display of the relevant lab circuit diagram for that lab experiment. The bottom section contains the graph panels which will display the graphs.

A typical user scenario is described below assuming that the applet has loaded and displayed some Lab Configuration.



6.302 Weblab Experiment 1

Start Frequency (Hz) : 1.0

Stop Frequency (kHz) : 1.0

Sweep Rate (Sec/Dec) : 1.0

Fig A.1: The Weblab Client after it is launched.

1. The user fills in the input fields and presses the “Execute” button.

The applet first does some trivial validation checks. Any errors in the data is notified and the user is prompted to correct the appropriate entry. Once the data has been successfully validated, an Experiment Specification is created and submitted to the Lab Server (via the Service Broker). A progress bar is displayed telling the user where his job is currently on the server and the time it will take to return the results.

2. Once the applet receives results from the Lab Server, it is at once used to plot the relevant graphs, which are displayed.

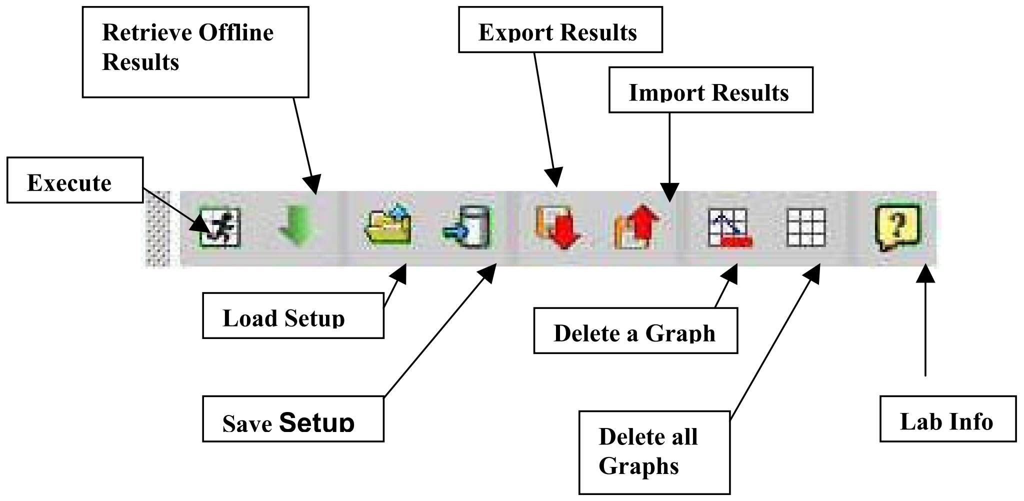


Fig A.2: The Toolbar

3. The user chooses to retrieve some offline results that may be needed for the lab experiment. To do this the user click on the “Retrieve Offline Results” button and is presented with a dialog allowing him to choose one from a predefined list of offline results.

4. The applet retrieves this offline data and plots it, clearing all previous graphs from the graph panels.

5. The user decides that he wants all his subsequent results to be plotted on top of this existing graph. So he changes the graph mode from “Always Replace” to "Always Add".

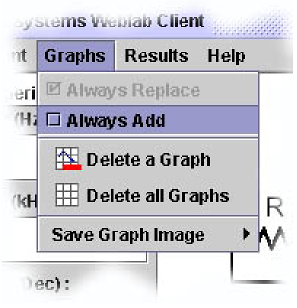


Fig A.3: The Graphs Menu

6. The user submits a new job with new parameters to the Lab Server.

7. Once the Lab Server returns the result, the relevant graphs are plotted on top of any existing graph. The new graph is displayed in a new color.

8. The user could repeat steps 6 and 7 twice more so that four different graphs are displayed.

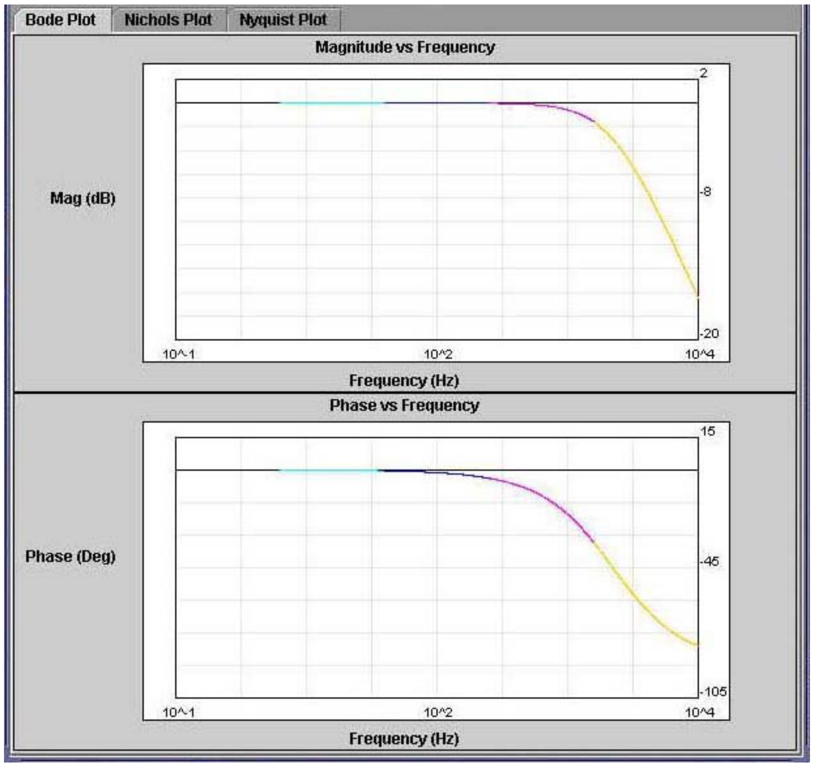


Fig A.4: The Bode Plot panel displaying four different graphs

9. Should the user try to submit yet another experiment, he will be unable to do so. The applet instead informs the user that the maximum number of graphs that could be displayed at any one time is 4 and that one or more current graphs should be removed, prior to submitting a new experiment.

10. The user then right clicks his mouse over the appropriate graph and chooses to delete that graph from the popup menu.

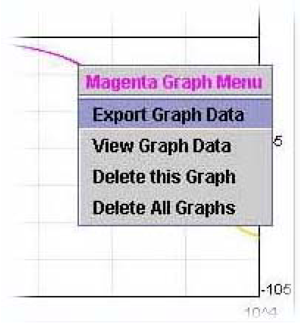


Fig A.5: The popup menu obtained by right clicking over a graph

11. The user chooses not to submit his new experiment specification but instead to save it so that he can submit it later. He clicks on the Save Setup button and from the subsequent dialog box chooses to save the setup on his local machine (the other option was to save it on the Server).
A Save File Dialog box is displayed allowing the user to save the setup on his local machine as an XML file.

12. The user decides that he would like to export his first experiment results. He either right clicks on the appropriate graph and chooses the Export Results menu item or directly clicks on the Export Results button. If he had clicked on the Export Results button, he has to first choose the appropriate graph. The Graphs are not named but color-coded so that he can identify which graph results he wants to save. Upon choosing a graph a Save File Dialog box is displayed. The user can choose to save the results either as a Comma Separated Value (CSV) file or as an ASCII Matlab text file. The user chooses a Matlab format and downloads the file.

13. The user then enters some new experimental parameters and submits the job to the Lab Server. Once the results are returned they are again plotted on top of the existing three graphs.

14. At this point, the user decides that he wants to start again and decides to clear the graph panel by clicking the Delete All Graphs button.

15. He first decides to load the experiment results that he had exported to a Matlab file. He Clicks on the Import Results button and loads the appropriate results file that he had saved earlier.

16. He then also decides to load the experimental setup that he had saved earlier. He does this by clicking on the Load Setup button. He then chooses the origin of this setup by choosing the "Load Setup from Local File" option. He chooses the appropriate XML file from the subsequent Load dialog and clicks enter.

17. The applet validates the experimental setup to make sure that it is a valid experimental setup for the current Lab Configuration. If it isn't then the user is notified of this. If the loaded setup is valid then the applet populates all the input fields with the values from the stored experimental setup.

18. The user submits this job to the Lab Server and once the results are returned, they are plotted on top of the last graph.

19. Satisfied with his results, the user exports the last graph results as a CSV file and saves it on his local machine.

20. The user chooses to save the Nichols Plot as a JPEG file by choosing Graphs → Save Graph Image → Save Nichols Plot. A Save Dialog box and allows the user to save the graph image as a JPEG image file.

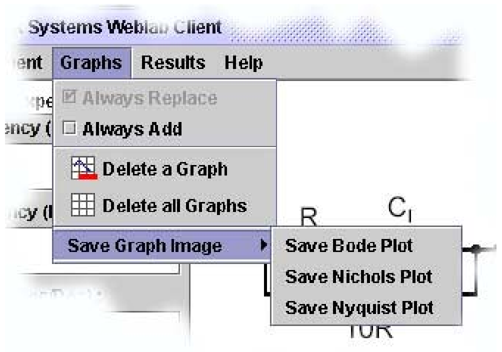


Fig A.6: The Save Graph Image sub-menu.

21. The user chooses to close the applet by choosing the Exit menu item from the File Menu.

Bibliography

[1] Zych, David. Microelectronics Devices and Circuits Weblab Client. http://weblab.mit.edu

[2] Viedma, Gerardo. Design and Implementation of a Feedback Systems Web Laboratory Prototype. Advanced Undergraduate Project, Massachusetts Institute of Technology, May 2004

[3] Williams, Brian. Educational Java Applet for Linear System Responses. http://web.mit.edu/6.302/www/pz. Advanced Undergraduate Project, Massachusetts Institute of Technology, May 2004

[4] Zych, David. Lab Client/Service Broker API