Proceedings of ME 701 & Computer Applications in Mechanical Engineering
ME 701 Fall 2015
Dec 11, 2015, Manhattan, KS, USA

ME 701 / Fall 2015

DYNAMIC SYSTEM & CONTROLLER DESIGN/ANALYSIS PACKAGE FOR PYTHON

Derek Black
Kansas State University
Manhattan, Kansas, USA

ABSTRACT

Control design and analysis is a growing field in industry, especially with the advent of modern robotics and a need to control them. However, for control engineers in industry and academia, there are currently few options for doing analysis of dynamic systems without the aid of expensive software tools such as MATLAB. There does exist free software to do some control analysis (such as Python's Scipy module), but the modules themselves are extremely lacking, especially for students throughout the world who are taught control analysis on MATLAB This leaves much to be desired in terms of a free controls package. The purpose of this paper and project is the development of a python package that acts like MATLAB's control package, including a SISO like GUI for controller design in python. The user of this module will find MATLAB like features implemented in python. Likewise, users will find the migration of their MATLAB's '.m' scripts to python to be an easy process.

INTRODUCTION

This python package consists of two files, controls.py, and sisotool.py. While the package does consist of multiple python files, only one need be imported for use, namely, controls.py. The controls.py module contains the main features, including: the ability to generate a step response, bode plot, zero/pole/gain transfer function, polynomial transfer function, function to determine the overshoot and settling time of a response. The sisotool.py file contains the SISO toolbox GUI. It is dynamically linked with the controls.py script. This GUI can be run by itself, or through a simple import of the controls.py script. These features will be explored further throughout the paper.

A brief introduction to control theory will also be introduced at the start of this paper for readers who have not been exposed to the field, however the majority of this paper will be devoted to the features and implementation of this python module. A tutorial on how to use this package for control design will be presented at the end of this paper.

NOMENCLATURE

.m	Mathlab script extension
.py	Python script extension
SISO	Single Input, Single Output
GUI	Graphical User Interface
K	Gain of dynamic system
p	Pole of dynamic system
z	Zero of dynamic system
ζ	Damping Ratio
ω_n	Natural Frequency
%OS	Percent Overshoot
IDE	Integrated Development Environment
Ts	Settling Time
PID	Partial-Integral-Derivative
PI-Lead	Partial-Integral-Lead
PD	Partial-Derivative

CONTROL THEORY: A BRIEF INTRODUCTION

Control theory is a branch of engineering and mathematics that deals with the behavior of dynamic systems from either forced or natural inputs and how the behavior is changed based on the feedback of the system. Typically, the plant, or dynamic system, is controlled by means of a controller or compensator that manipulates the system to produce a desired output [1]. In order to analyze both the plant and the controller, many mathematical tools, such as the use of a Bode plot, root locus plot, or the step response, can be utilized. Typically, dynamic systems are not looked at in the time domain, as working with these equations is exceedingly tedious. Instead, engineers typically transform these systems into the frequency domain by the use of the Laplace Transform. Once transformed, these systems are typically put into what is known as a transfer function, which is a ratio of the system's output over input [1]. Some common transfer functions for 1st and 2nd order systems can be seen by equations 1 and 2.

1st Order System: $G(s) = \frac{K}{s+p}$

2nd Order System: $G(s) = \frac{\omega_n^2}{s^2+2\zeta\omega_n+\omega_n^2}$

The interesting part of second order systems is their behavior to a changing ζ. Three import cases arise from this change in ζ [3].

1. $\zeta > 1$ Over Damped
2. $\zeta = 0$ Critically Damped
3. $0 < \zeta < 1$ Under Damped

Below is a plot of a 2nd order system, with a varying ζ. The controls module controls.py, and the step() command were used to generate this response.

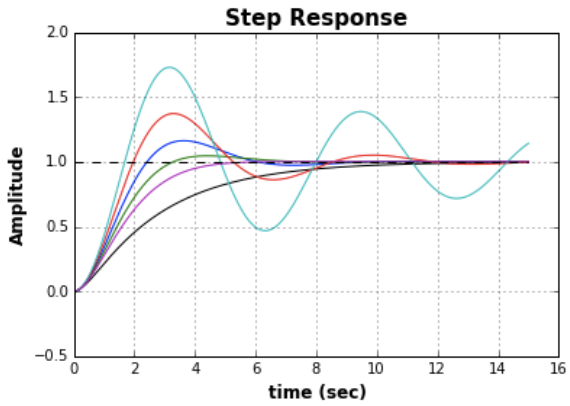


Figure 1: Step response to varying zeta.

Also of particular interest to a controls engineer is the step response's overshoot and settling time. Overshoot is the height of the first peak of the unit step response, which is usually expressed as a percentage of the steady state (from figure 1, the steady state in this case is 1.0). Settling time is the time it takes for the signal's oscillations to die down and stay in a range of 2% of the steady state value. These characteristics are important when designing a controller for a dynamic system, as the controls engineer will typically have design constraints that need to be implemented for control of a particular dynamic system. Such design constraints could be a %OS of 10% and a settling time of 1.5 s for example. These constraints highly depend on the system that is analyzed and what type of output the system will produce [2].

CONTROLS MODULE

The purpose of this module is not to replace MATLAB's control package, or for that matter, any paid control package available for individual use. The real purpose of this program is to have a controls package similar to MATLAB's interface, for use by hobbyists who do not have access to an expensive software license. The biggest feature of this package is the SISO toolbox GUI. The need of an open source controller designer is a necessity, and thus, more attention was given to its development. However, this package offers much more than SISO tools and provides a wide variety of other useful functions. In the next couple of sections, this paper will address some of these functions, and then an extensive review of the SISO toolbox will be uncovered.

TRANSFER FUNCTIONS

Transfer functions in the controls.py module work exactly the same as they would in MATLAB. The user inputs the polynomial of the numerator and denominator inside the 'tf()' command, and the resulting transfer function is displayed on screen. Let's look at a simple example of this function. Ipython will be used for the input and output, however any IDE can be used.

```
[In [1]: from controls import tf

[In [2]: T = tf([0,4,7.543],[1,4,6])

[In [3]: print T
 4s + 7.543
 -------------
 1s² + 4s + 6
```

Figure 2: Python output of transfer function T

Comparing controls.py's output to MATLAB, it is found that the functionality is nearly identical.

```
>> T = tf([0,4,7.543],[1,4,6])

Transfer function:
 4 s + 7.543
-------------
s^2 + 4 s + 6
```

Figure 3: Matlab output of transfer function T

ZERO/POLE/GAIN

The python controls module also has the functionality of outputting a transfer function in a zero/pole/gain format. Before introducing the details of this zpk function, each of these variables will be defined. The poles and zeros of the system are the roots of the characteristic equation of the denominator and numerator, respectively, of the transfer function [4]. In addition, the gain is defined as a proportional relationship between the input and output magnitudes [5].

Just like the transfer function example, the same holds true for the zpk() function. The user simply enters the zeros, poles, and gain of the dynamic system into the function, and python

outputs the resulting transfer function. Below is an example of this function.

```
In [5]: from controls import zpk

In [6]: T = zpk([-2],[-3,-6],10)

In [7]: print T
 10 ( s + 2 )
 -------------
 ( s + 3 )( s + 6 )
```

Figure 4: Python output of zpk transfer function

UNIT STEP RESPONSE

Step responses are an important tool in control design, especially for analysis type purposes. As a result, this controls package includes a unit step response function. While python does have a few libraries for stepping a response, they can be unintuitive and confusing to use. This is especially true if you come from a MATLAB type environment where it is as simple as typing step(transfer function) into the command line. These other packages also assume the engineer is familiar with Matplotlib and its plotting capabilities. The important thing to note is the purpose of this control module is not to program more, but to make control analysis simple. In figures 5 and 6, a step response is taken of a first and second order system, respectively.

1st Order Unit Step Response

```
In [4]: from controls import step, tf

In [5]: T = tf([1],[1,4])

In [6]: step(T)
```

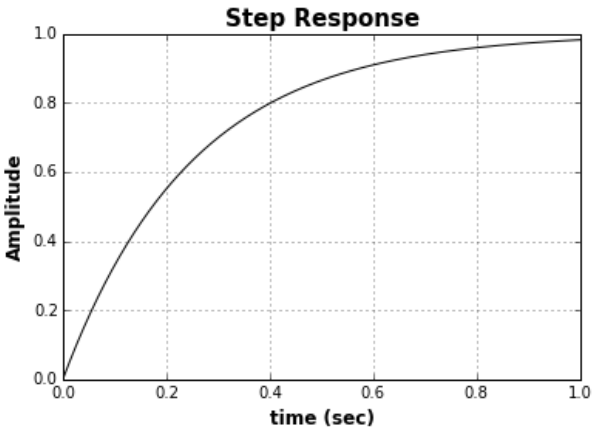


Figure 5: First Order Step Response

2nd Order Unit Step Response

```
In [1]: from controls import *

In [2]: T = tf([1,3],[1,2,6])

In [3]: print T
1s + 3
-------------
1s² + 2s + 6

In [4]: tfchar(T)
Out[4]: 'Overshoot =  24.5376148401 Settling Time =  4.0'

In [5]: step(T)
```

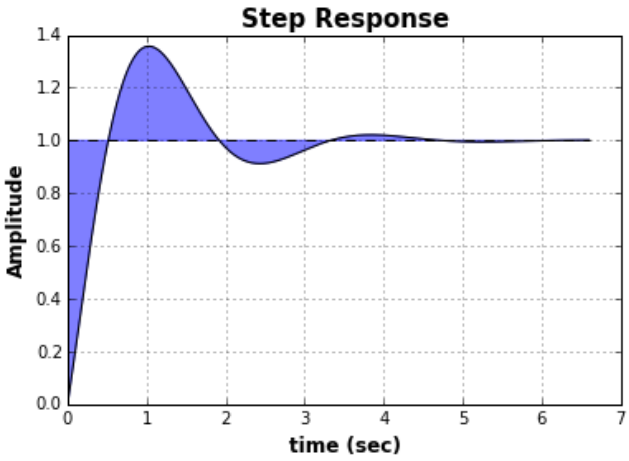


Figure 6: Second Order Step Response

It should be noted from figure 6, line 4, the function 'tfchar(T)' was used. It stands for transfer function characteristics. It simply calculates the %OS and settling time of the transfer function. MATLAB also has a similar function.

A blue fill was added to this 2nd order response in order to provide emphasis of the oscillations of the response, which can be toggled off if it undesirable to the user.

ROOT LOCUS / PZMAP

Root Locus is also an import tool in the controls engineer's toolbox. It is a simple tool to implement and can provide valuable insight to the engineer. I had attempted to complete a function to plot the root locus in this control module; however, due to time constraints and overall difficulty, was unable to complete it. Fortunately, my attempts at a root locus function did not go in vain. What this function did turn up is another useful MATLAB like function, namely pzmap(). Pzmap graphs the poles and zeros of a system on the real and imaginary axis. Typically this function is used for checking stability of a system. If one finds that the system contains poles or zeros in the right hand plane (positive roots), the system is unstable, and a controller would have to be developed to stabilize the system. For the sake of completeness, a graph of MATLABs root locus is shown to demonstrate what it does and what it looks like.

```
[In [7]: from controls import pzmap

[In [8]: print T
 1s + 4
 -------------
 5s² + 1s + 7

[In [9]: pzmap(T)
```

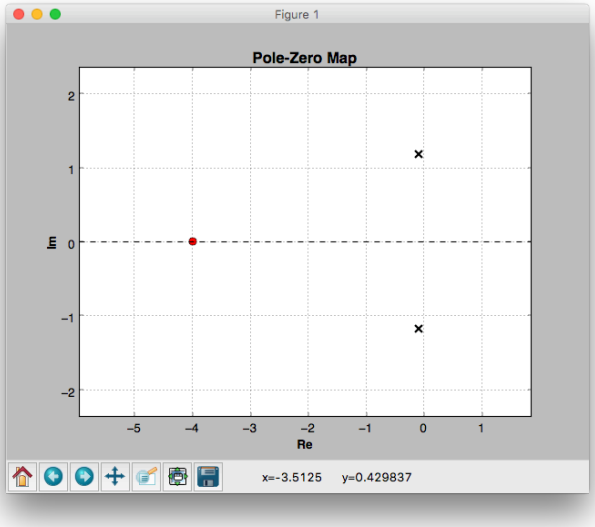


Figure 7: Controls.py PZ Map Plot

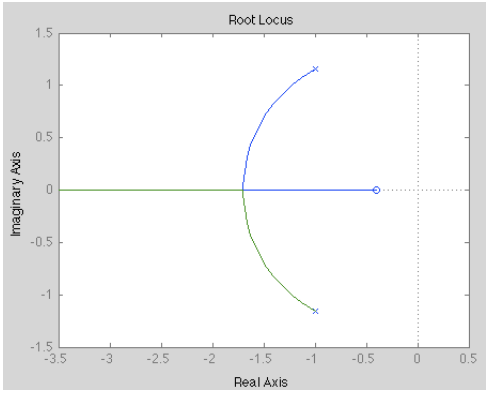


Figure 8: Matlab Root Locus Plot

SISO TOOLS AND PYQT4

The python controls module also has other useful features for signal processing. Controls.py can deploy a MATLAB like SISO tool environment for controller design. It includes a Bode plot, along with the step response and Pole-Zero based tuning methods. This SISO tool is a separate GUI developed in PyQt4. It is dynamically linked with the controls.py module. What this means is only one import statement is required in your python code, namely 'from controls import *', which gives you access to all of the controls functions, including SISO tools. To access the GUI, the user simply types 'sisotool()' into the python command line to access the toolbox.

DEVELOPMENT OF THE SISO GUI

The one interesting thing I found in my research was that there are not too many computer-based solutions to controls problems (at least without the need of fancy features like symbolic solvers and such). Many of the algorithms developed were my own doing, and I plan on presenting my derivations in the next section.

CLOSED LOOP FEEDBACK CONTROL

My first task in developing a SISO controller designer was being able to step a closed loop system with a plant and controller.

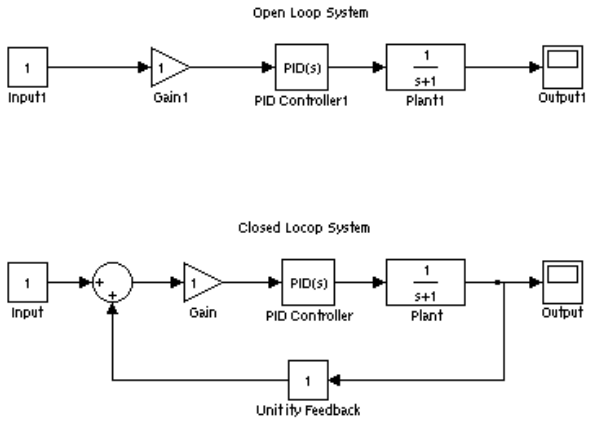


Figure 9: Closed & Open Loop Systems

From figure 9, an open loop system and closed loop system (with negative unity feedback) are presented. An open loop system is simply the Controller and Plant multiplied together. Open loop systems are fast; however, because they do not have feedback, they are very rarely reliable systems. On the other hand, closed loop systems have feedback, and as a result, can reduce the error signal between the input and output to eventually reach some steady state value. Closing an open loop is easy by typically using the following equation:

Eq. 1 $$CLTF = \frac{Plant\ Transfer\ Function}{1+Plant\ Transfer\ Function}$$

However, the CLTF can only be used with the plant and not with a controller. In order for the function to be applied to the controller, the CLTF equation becomes:

Eq. 2 $$CLTF = \frac{CP}{1+CP}$$

where P will represent the plant transfer function, and C, the controller transfer function.

Getting the control signal and being able to step it should be as simple as using equation 2 above. However, that is not the case. Instead it can be shown by example using MATLAB. Note: I will be using the same designed controller and plant transfer function presented in the tutorial in the proceeding section. Look at figure 19 to see what the actual step response looks like and compare it to figure 10. Here are the plant and control transfer functions, respectively.

Eq. 3 $$P(s) = \frac{s+4}{5s^2+s+7}$$

Eq. 4 $$C(s) = \frac{10(s+3)(s+35)}{s}$$

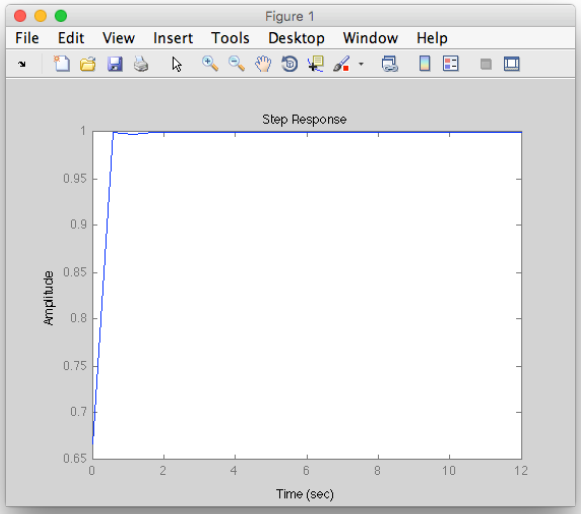


Figure 10: Step response of CLTF of P and C

Clearly, this is not the same response. MATLAB has the advantage of having symbolic solvers and other useful tools. The reason this doesn't work is the equation for the CLTF ends up having many pole-zero cancellations and other general simplifications. Obviously, these tools are not at my disposal in python, unless I wanted to install several other python libraries. The reason I did not use other python libraries is the fact I wanted anyone to be able to use this without the hassle of having library incompatibility issues. In order to tackle this problem, I employed my own derivation and is why there will be no reference attached to this derivation. This may be the same equation MATLAB uses to compute their feedback; however, I do not have access to their algorithms, so I cannot say for certain if this derivation is unique. Using equation 2, it is possible for me to derive the closed loop feedback, reduced polynomial transfer function:

$$\text{Let} \quad P = \frac{N_p}{D_p} \quad \text{and} \quad C = \frac{N_c}{D_c}$$

where N and D are the numerator and denominator polynomials of the respective transfer functions P and C. Substituting these values into equation 2, the following result was obtained:

Eq. 5 $$CLTF = \frac{\frac{N_pN_c}{D_pD_c}}{\frac{D_pD_c+N_pN_c}{D_pD_c}}$$

Simplifying this equation, I found that the reduced closed loop polynomial transfer function is:

Eq. 6 $$CLTF = \frac{N_pN_c}{D_pD_c+N_pN_c}$$

Using this simple equation, it is now possible to find the feedback of the system. A simple import of Numpy's function convolve makes this extremely difficult problem extremely easy. This is what is used in SISO tool's step function for control design. It should be mentioned SISO's step command is different from the controls package step, as it includes the controller in the response.

SISO TOOLBOX FEATURES & DEVELOPMENT

This toolbox was primarily developed in PyQt4. Because of this, the user should be expected to have PyQt installed on their system for this package to work, at least for the SISO tool. Qt Designer was also used to develop the template of the GUI; however, the use of Qt Designer for this GUI was limited, and most of the coding was done myself.

This toolbox employs many features. The ability to step the plant response (similar to the step() command in the controls module). The 'Add Controller' feature, as seen in figure 12, employs the equation developed in the preceding section. It steps the controlled response. Likewise, there is an adaptive Bode plot that changes with the controller. Bode will not be explained in this paper, but is a useful tool in control design and determining stability of a system.

Other features include many useful plotting options. Examples including: %OS, settling time, %2 error band, and step fill.

1. %OS: When enabled, calculates the % Overshoot of the system and draws lines to the peak.

2. Settling Time: This function determines the settling time of the system, that is, when the oscillations die out to 2% of the error (or steady state value). The steady state value is also known as the DC gain of the system.

3. Error: Draws red, horizontal error lines $\pm$ %2 of the DC gain. This helps the user visualize when and how fast the oscillations die out and reach a reasonable error band.

4. Step Fill: Simply fills the oscillations of the step response. This is a purely a visual aspect to this program. However, to me, it helps at looking at the actual graph and seeing the width and peaks of the oscillation.

The GUI also outputs the plant transfer function in a readable fashion. It also outputs the controller transfer function in zero/pole/gain format. The GUI includes many controller types, including a PID tuner, PD tuner, PI-Lead tuner, and, of course, a proportional tuner. All of these may be selected from the drop down menu. A menu bar was added to this program, but have still yet to be implemented.

HOW TO USE THIS PYTHON PACKAGE:

A GUIDED TUTORIAL

To demonstrate the usefulness of this python package, a quick tutorial will be outlined in this section. The example given in this tutorial will be hypothetical, meaning the transfer function defined is not bound to some specific system. However, constraints will be imposed to the system to give it depth. For this particular situation, our design constraints are:

$$T_s \leq 0.5\ seconds$$
$$\%OS \leq 5\%$$

With a settling time of less than 0.5 seconds, this guarantees a system that behaves very quickly. Along with the 5% overshoot, the hypothetical system will not diverge to far from the steady state value.

To begin this tutorial, open your favorite IDE, or simply use Python/iPython in the console. First, import the controls module into your IDE and define the transfer function using the tf() command. The transfer function to be used in this tutorial will be the equation below:

Eq.7 $$G(s) = \frac{s+4}{5s^2+s+7}$$

Next, open sisotool from your IDE by typing the command "sisotool()". This will invoke the GUI to open, where the controller for this plant can be developed.

```
In [2]: from controls import *

In [3]: sisotool()
```

Figure 11: SISO Toolbox GUI command

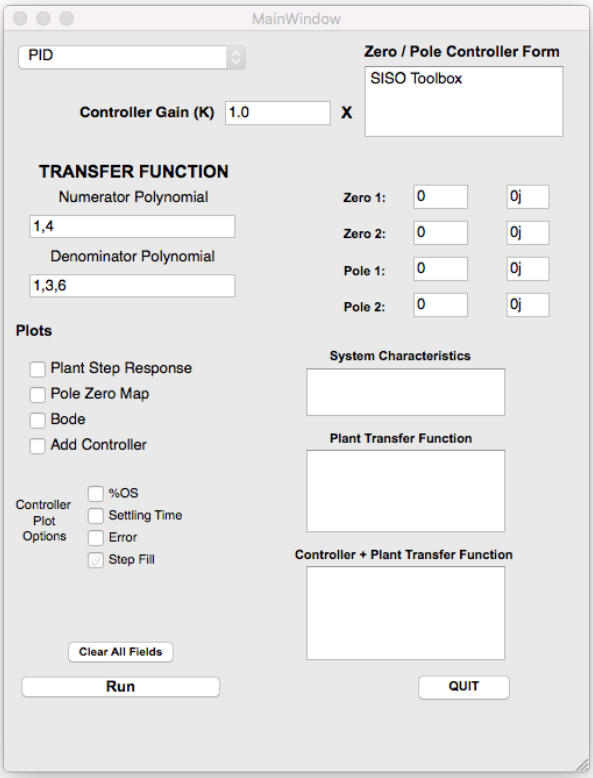
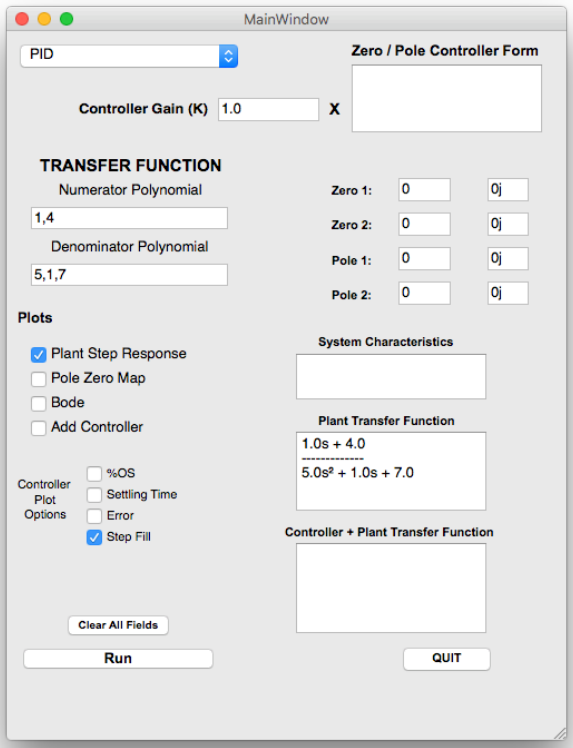


Figure 12: SISO toolbox GUI

From here, it is possible to now start designing the controller for this system. From figure 10, under the heading of 'Transfer Function', enter the coefficients of the numerator and denominator of the transfer function G(s) into the two input boxes. Then, under 'Plots' click 'Plant Step Response' and run the program. You should get a step response similar to the one in figure 11. (Note: Opening SISO tool is not necessary to plot this transfer function. You can first define G(s) using the tf() command, and then step the response using step() directly in your IDE.)



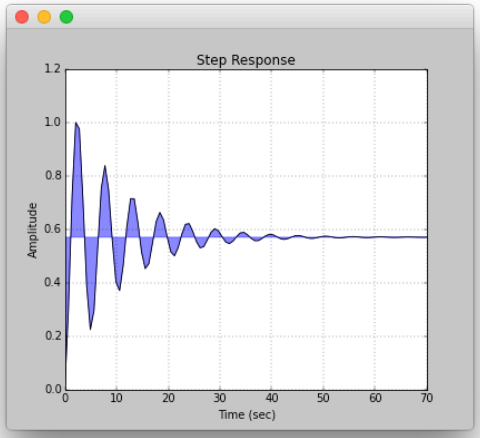


Figure 13: Step Response of G(s) and GUI.

Notice that G(s) is presented in a more readable fashion in the 'Plant Transfer Function' display box. Looking at figure 11, it should be clear that this response is not ideal. In fact, if the 'tfchar()' command from this python package is invoked, it can been seen that this plant has an overshoot of 77% and a settling time of 40 seconds (Figure 12).

```
[In [2]: T = tf([1,4],[5,1,7])

[In [3]: print tfchar(T)
 (76.60817351874944, 40.0)
```

Figure 14: tfchar(T) output

The next step is to start controller design. There are multiple options to choose from: Proportional, PD, PID, or PI-Lead controllers. Which option is chosen is completely up to the designer and his or her experience. However, the most common type of controller to choose in control design is the PID controller. But for the sake of this tutorial, lets pretend there is no good option to choose from at the start of this design process. To begin the design process of a controller for this plant, lets follow the following steps:

1. Design a Proportional Controller
2. Design a PD Controller
3. Design a PID Controller

While this process may seem arbitrary, there are rare instances where a controls engineer may not know of a controller to start with, and this series of steps can be followed, using the desired constraints.

Before steps one through three are carried out, make sure all the controller plot options are turned on so the characteristics of the controller can be better realized. Next, play with the controller gain a bit (proportional control), and see if the %OS and settling time can be improved. It may turn out that a simple proportional controller might suffice. Make sure the 'Add Controller' option is selected to see the response of the system.

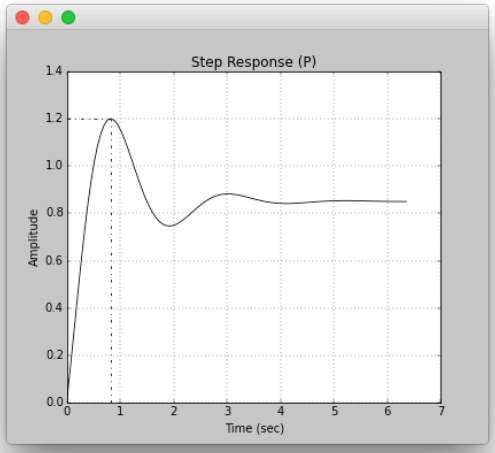


Figure 15: Step response of P Controller

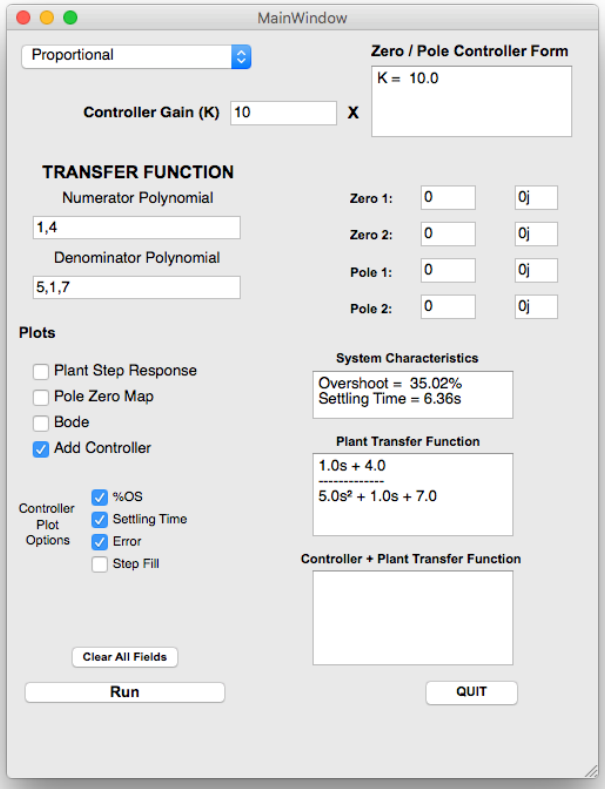


Figure 16: GUI settings

Surprisingly, using only a control gain of 10 actually did some good for the system. Looking at figure 13, we can see that the %OS dropped down to 35%, and the settling time got reduced to 6.35 seconds. This settling time almost meets the design constraints, however the %OS is still too high. Lets try to add a PD controller into the mix to see if we can improve this controller a bit. For the PD controller, we will be adding an extra zero to see how it affects the response. In order to utilize the PD controller, select PD from the drop down menu. It is urged that you also turn on all of the controller plot options so our system can be fine tuned even more so by visual inspection. The PD controller uses the user input box called 'Zero 1'. The gain can still be manipulated with the PD controller; however, since the gain of 10 helped improve the system drastically, it will be kept the same.

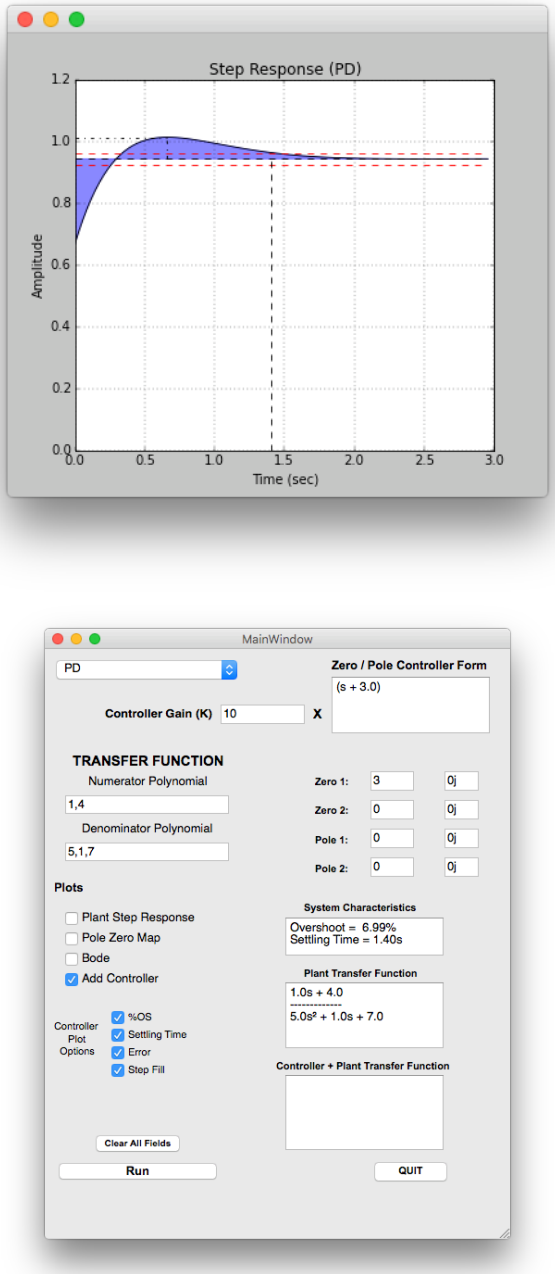
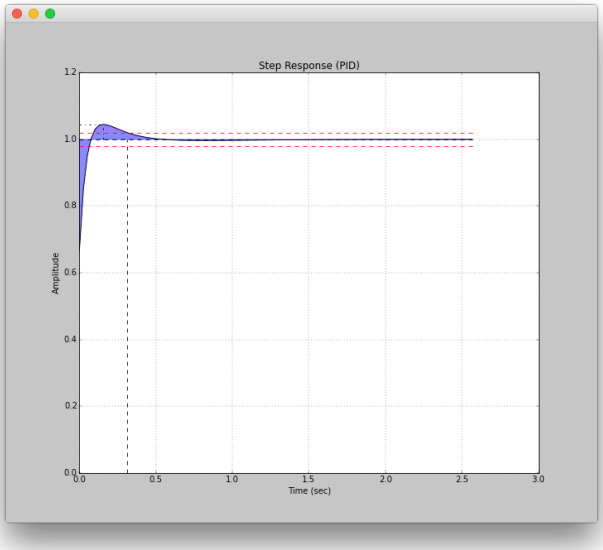


Figure 17: Step response & GUI of PD controller

It was decided to add a zero at -3. Just by adding this zero, it improved our system drastically (as can be seen from figure 14). With an %OS of 7% and settling time of 1.4 seconds, this controller is one step closer to meeting the design constraints. Here is the controller that has been developed this far.

$$C(s) = 10(s + 3)$$

However, more still needs to be done. Lets begin by changing the controller type to 'PID'. This will add another zero to our system, along with an integrator. By adding a zero further to the left (negative direction), it will result in dragging the response down, along with a reduction in settling time. Lets try adding a zero at -35.



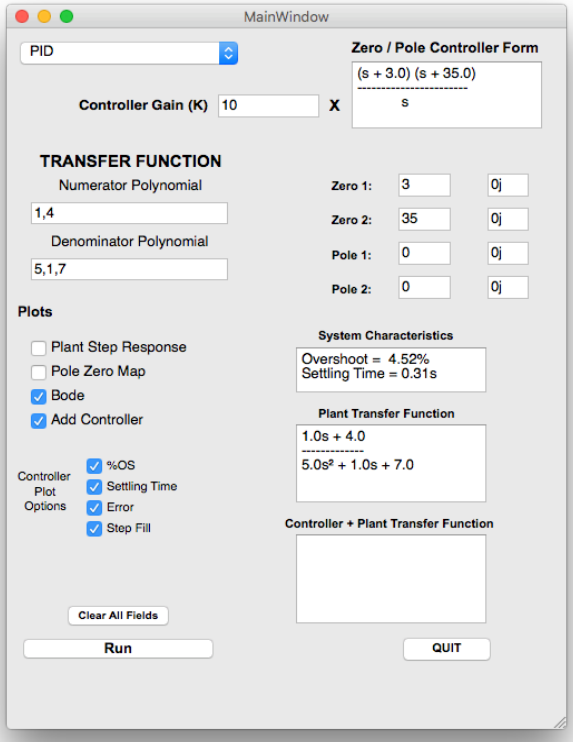


Figure 18: Response to PID controller

It looks like a PID controller did just the trick at controlling our system. If figure 15 is referenced, it can be seen that there is an %OS of 4.52% and a settling time of 0.31 seconds. The design criteria have been met. Now to address the burning question, does this program actually do what it's supposed to do? Lets take a look at MATLAB, using the same controller and plant.

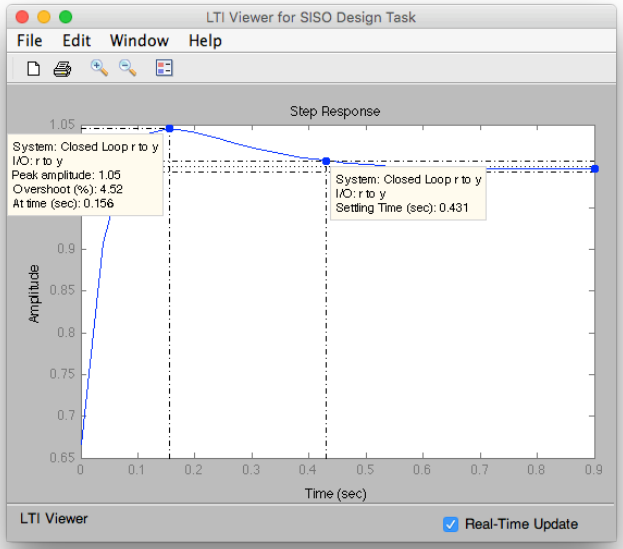


Figure 19: MATLAB control signal

MATLAB and control's SISO toolbox come to pretty close to the same results. It looks like this controls package can hold a light to MATLABs own SISO tool package. While it might not be as flashy or have as many features, it can be assured accurate results will be obtained just as readily as MATLAB could produce.

CONCLUSIONS

The purpose of this paper was to address an alternative open source package for control analysis and design to MATLAB's control package. Using the SISO tools in this controls module, it is demonstrated that the toolbox is adequate enough for a controls engineer to design controllers for their dynamic systems. The package contains several MATLAB like commands to aid control engineers in analyzing specific details of their systems. In addition, the package contains a valuable toolset to first semester controls students and hobbyists who wish to experiment outside of a lab.

FUTURE WORK & REMARKS

Automated PID tuner based on %OS and settling time.
As much as I would have loved to develop an algorithm for this, there was just simply not enough time to implement it. It was a feature I was really looking forward to do. As of right now, it is on the To-Do list.

Root Locus
Just as I stated in the section on Root Locus, there was just not enough time to complete this function. However I do plan on working on this more, as I wish to learn more about the algorithms behind the process.

SISO Toolbox
Most of the grunt work has been completed for this, however there were more controller types I wished to have implemented (If you look at figure 12, you will see a column of user input boxes filled with 0j; I was hoping to work with complex poles and zeros). I also never got around displaying the controller + plant transfer function.

The 'quit' and 'clear all fields' buttons do not work. However implementing them should be simple, and is on my To-Do list.

Transfer Function to State Space

The command tf2ss() in controls.py is still yet to be completed. However, it does work for 2^{nd} order systems, but anything higher will cause the function to produce incorrect or no output. I still plan on finishing this function. This idea was scraped early on because I found the SISO toolbox development a more exciting topic to work on. The algorithm still exists in the controls.py module, and can be implemented for 2^{nd} order systems.

ACKNOWLEDGMENTS

Dr. Jeremy Roberts – Kansas State University

REFERENCES

[1] https://en.wikipedia.org/wiki/Control_theory

[2] http://ocw.mit.edu/courses/mechanical-engineering/2-004-dynamics-and-control-ii-spring-2008/lecture-notes/lecture_21.pdf

[3] http://www.et.byu.edu/~tom/classes/436/ClassNotes/Class20(Second-Order).pdf

[4] http://web.mit.edu/2.14/www/Handouts/PoleZero.pdf

[5] https://en.wikibooks.org/wiki/Control_Systems/Gain

