MATLAB Youla Parameterization Toolkit

Timothy J. Wheeler

March 15, 2006

1 Introduction

Because this software is expected to evolve over the next few weeks, it would be very difficult to formally document every feature in great detail. By the time the course has ended, I hope to have a finalized version of the software that includes a clear and informative reference manual. However, in this version, I am attempting to use a more “agile” form of documentation. Within each MATLAB m-file, there is documentation on how that particular function works. This information is accessed using the MATLAB help command. For example, if the m-file is named cal.m, then typing help cal in the MATLAB Command Window will display the documentation embedded in cal.m. Thus, this manual is organized by task, and the documentation is referred to by such help commands.

2 Parameterizing the System

As shown in class, we use the Youla-parameterization to simplify the task of searching for stabilizing controller. Clearly, the parameterization is dependent upon the system architecture or “topology”. The two system topologies we have seen in class are shown in Figures 1 and 2.

![Figure 1: System topology with 1-by-1 controller (1DOF).](image1)

![Figure 2: System architecture with 1-by-2 controller (2DOF).](image2)

In the software, the topology shown in Figure 1 is referred to as ’1DOF’ and the topology shown in Figure 2 is ’2DOF’. Suppose we have chosen a plant \( P(s) \) in the diagrams) that we wish to control. Let the MATLAB variable \( \mathbf{P} \) be an LTI model of the plant (tf, ss, or zpk). To parameterize the system, we create a clsystem object that contains information about the plant, the system topology, and an optional name that will help us identify the variable in the future. The syntax is as follows:

\[ \text{sys} = \text{makesystem(plant, top, name)} \]
where top is a string ("1DOF" or "2DOF") and name is a string. Type help makesystem for more information. Once we have created a parameterization of our system, we can display its properties in the Command Window using the command

\[
display(sys)
\]

## 3 Specifying Outputs & Inputs of Interest

In addition to the system name, system topology, and controller dimensions, the `display` command also tells us the names of the system’s output and input signals. These names correspond to those shown in Figures 1 and 2. In our design objectives, we specify constraints on the closed-loop input-to-output response of the system for specific input-output pairs. To get the parameterization of our system for a particular input-output pair (say 'r' to 'y'), we use the command

\[
sysYR = sys('y', 'r')
\]

where `sysYR` is the new `clsystem` object that only parameterizes the map from 'r' to 'y'. Note that this command uses the MATLAB convention of specifying the output first. Suppose, for instance, that we want to constrain the response at output 'y' due to two different inputs (say 'r' and 'n'). We use the syntax

\[
sysYRN = sys('y', 'r', 'n')
\]

where we have simply put the input strings into a cell array. In general, we can specify any number of outputs and inputs if we use cell arrays. See `help clsystem/subsref` for more information.

## 4 Creating an Input Signal

We want to constrain the response of our system due to known inputs. Common examples are step inputs, random noise, and sinusoids. We assume that we have a 1-by-NT vector of time points `tInput`, at which the input is defined, and a NI-by-NT array `inputData` that contains the NI-by-1 input signal specified at each point of `tInput`. For example, to create a 10 second, one-dimensional unit step signal, we type

\[
tInput = linspace(0, 10, 100);
inputData = ones(1, 100);
\]

Now, to convert this “signal” into something our software can use, we create a timedomain object with the following command:

\[
myStep = timedomain(tInput, inputData, 'Unit Step')
\]

where the string 'Unit Step' is just a tag to identify the signal. The signal data is linearly interpolated between the times specified in `tInput`. See `help timedomain` for more information about creating timedomain objects.

## 5 Forcing the System with an Input

Recall that the variable `sysYR` parameterizes the map from input 'r' to output 'y'. In order to constrain the response at output 'y', we must specify which input should be applied at input 'r'. For example, we could force `sysYR` with the unit step input `myStep` created in the previous section. This is done using the * operator (see `help clsystem/mtimes`) as follows:

\[
timeRespYR = sysYR * myStep
\]

where `timeRespYR` is a timeresponse object (see `help timeresponse`). Note that if we created a parameterization of the map from two inputs to one output, such as in `sysYRN` above, we would have to apply a 2-by-1 input signal.
6 Constrained the Time Response

The time response of the system due to a given input can be constrained above and below. This is a convenient way to control transient behaviors, such as percent overshoot and rise time, as well as steady-state behaviors, such as steady-state error. As shown in class, these bounds are also functions of time. Thus, we use the \texttt{timedomain} function to create a bound as a piecewise-linear function of time. For example, suppose we have a vector of time points \texttt{tUpper} and a vector of bound magnitudes \texttt{boundUpper} with the same dimensions. We create the bound as follows:

\begin{verbatim}
upperBound = timedomain(tUpper, boundUpper,'UpperBound')
\end{verbatim}

Similarly, we could specify a lower bound at times \texttt{tLower} with magnitudes \texttt{boundLower} as follows

\begin{verbatim}
lowerBound = timedomain(tLower, boundLower,'LowerBound')
\end{verbatim}

To apply these constraints to our parameterized forced response \texttt{timeRespYR} we create a constraintfunc object. This object represents the constraint as a function of the Youla parameter \texttt{Q}. As discussed in class, we only evaluate the constraint at a finite number of time points. Suppose we have a vector of time points \texttt{tEvaluateAt}. We create the constraint function using the syntax

\begin{verbatim}
timeConst = constraintfunc('lowerBound <= timeRespYR <= upperBound', tEvaluateAt)
\end{verbatim}

where the first input is a string describing our constraint. Because we have specified a lower bound and an upper bound, the output \texttt{timeConst} is actually a 2-by-1 array of constraintfunc objects. See \texttt{help constraintfunc} for more information.

7 Frequency Response Magnitude of the System

Eventually, we will use our time domain constraints to formulate a linear program and search for a satisfactory controller; however, before we move on to that phase, we consider the task of constraining the frequency response magnitude of our system. To begin, we first construct an object that represents the frequency response magnitude of our system as a function of the Youla parameter \texttt{Q}. As in the time domain case, we are using the approximation that the frequency response magnitude will only be evaluated at a finite number of frequency points. Suppose we want to parameterize the frequency response of our system \texttt{sysYR} at the frequencies \texttt{fR}. We would use the syntax

\begin{verbatim}
freqRespYR = abs(sysYR, fR)
\end{verbatim}

where \texttt{abs} is an overloaded version of the normal \texttt{abs} function (see \texttt{help clsystem/abs}).

8 Constraining the Frequency Response

As in the time domain case, the bounds on the frequency response magnitude can be viewed as functions of frequency. Thus, we use the analogous \texttt{freqdomain} object to represent our frequency domain signal. For example, suppose we have a vector of frequencies \texttt{fBound} and a vector of frequency response magnitudes \texttt{boundMag} with the same dimensions. We create a frequency domain bound as follows:

\begin{verbatim}
freqBound = freqdomain(fBound, boundMag,'MyFrequencyBound')
\end{verbatim}

where the optional third input is just an identifying tag. See \texttt{help freqdomain} for more information. The interpolation of a frequency domain signal in between the specified frequencies is such that the bound will look piecewise-linear on a Bode (log-log) plot. To apply this constraint to our parameterized frequency response \texttt{freqRespYR} we create a constraintfunc object. Again, we select a finite vector of frequencies \texttt{fEvaluateAt} at which we want to impose our constraint. As shown in class, the bound on the magnitude of a complex number is nonlinear constraint that we approximate
with $N$ linear constraints. Given these data and our parameterized frequency response magnitude, we create the constraint function as follows:

$$\text{freqConst} = \text{constraintfunc('freqResYR <= freqBound', fEvaluate, N)}$$

See help constraintfunc for more information.

9 Constructing a Basis in “Q-space”

We know that the Youla parameterization allows use to search of the entire space of stable Qs. However, as discussed in class, we can only search over a finite set. Thus, we can either select Qs that have certain behaviors (see Homework 1, Problem 5) or we can select controllers from the “K-space” and convert them to “Q-space”. As we will see in the next section, the software automates this conversion process. To construct a Q basis, we simply take any stable LTI models and stack them into a cell array as follows:

$$\text{QBasis} = \{Q_1; Q_2; \ldots\}$$

Similarly, we take our controllers from the “K-space” and stack them into a cell array

$$\text{KBasis} = \{K_1; K_2; \ldots\}$$

Then the function getfeasible will take care of the rest.

10 Solving the Linear Program

For now, we do not have an objective in our optimization. We simply wish to find any controller that satisfies our constraints, or if our constraints are too stringent, we want to know which ones are the most difficult to satisfy. The function getfeasible solves this problem (see help getfeasible). Suppose we have a number of constraints that we wish to satisfy, such as timeConst and freqConst constructed above. To combine these constraints, we simply stack them into column vector form

$$\text{allConst} = [\text{timeConst}; \text{freqConst}]$$

At this point, we have completely defined our optimization problem: we want to search over all linear combinations of the Qs in QBasis as well as those derived from KBasis to find a feasible Q which satisfies the constraints imposed by allConst. The corresponding MATLAB command is

$$[\text{Qfeas}, \text{newConst}] = \text{getfeasible(allConst, QBasis, KBasis)}$$

where the output Qfeas is our feasible controller, and the output newConst is an updated version of the array allConst (see the next section for a description of what “updated” means in this context).

11 Analyzing the Results

After we call getfeasible with our constraint functions and Q-basis, we are left with a feasible controller Qfeas and “updated” constraint functions newConst. If the problem is feasible, then newConst would contain all the same information as allConst as well as some new sensitivity information that the linear program solver produces. However, if the problem is infeasible, newConst contains the sensitivity information as well as a scalar slack variable. This slack variable is used by the optimization routine to multiplicatively relax the constraint until it can find a feasible Q. Thus, if the constraints had to be relaxed, Qfeas is feasible for newConst but not for allConst.

So far, we have not defined what sensitivity means or how to interpret the value of the slack variables. Because the linear program is solved in the less intuitive Q-space, we take the more casual
approach of interpreting slack and sensitivity graphically in a qualitative manner. To view these
data qualitatively, we use the following command

$$\texttt{qplot(newConst(1),’b’)}$$

where \texttt{newConst(1)} refers to the relaxed version of our first constraint, which in this case was the
lower bound on the time response. The second input ‘b’ is a MATLAB LineSpec like those used for
the default \texttt{plot} command (see \texttt{help plot}). If the constraint \texttt{newConst(1)} is relaxed (i.e. nonzero
slack variable), \texttt{qplot} will plot the constraint as a solid blue line and the relaxed version as a dashed
blue line. For concreteness, we assume \texttt{newConst(1)} is a time domain constraint function, as in our/example. If at any point in our finite time vector \texttt{tEvaluateAt} the optimization returns a nonzero
sensitivity, then \texttt{qplot} will draw a small red circle on the plot of the constraint function at that
point. Qualitatively, a circle with a larger radius indicates a more sensitive constraint than a circle
with a smaller radius. Similarly, for frequency domain constraint functions, \texttt{qplot} draws red circles
at the most sensitive frequencies. The \texttt{qplot} command can also display x- and y-data in the form
of double arrays, as well as timedomain and freqdomain objects (see \texttt{help qplot}). The example
script files show how \texttt{qplot} is used to display multiple objects on the same axes.

12 Using the Results

We can either use our feasible Q in “Q-space” or convert it to “K-space” and use it as our controller.
To use a Q directly, we use the special syntax

$$\texttt{result = obj.qcontrol(Q)}$$

where \texttt{obj} is either a clsystem, timeresponse, or freqresponse. If \texttt{obj} is a clsystem, \texttt{result} will be
a state-space model of the closed-loop system (see \texttt{help clsystem/subsref}). If \texttt{obj} is a timeresponse
or freqresponse, then \texttt{result} is a timedomain or freqdomain object, respectively (see \texttt{help
timedomain/subsref} and \texttt{help freqdomain/subsref}).

If we want to convert the feasible Q to the corresponding controller, we type

$$\texttt{Kfeas = k2q(sys,Qfeas)}$$

where \texttt{sys} is the clsystem object around which the optimization problem was built. For a more
concrete example of how to use the resulting \texttt{Kfeas}, suppose we want to plot the time response
of our system on the same axes as our time constraints. We assume that the first two entries of
\texttt{newConst} correspond to the upper and lower bounds on the time response \texttt{timeRespYR} created
above. To plot these bounds and the closed-loop system response on object, we use the command

$$\texttt{qplot(newConst(1:2),’b’,timeRespYR.kcontrol(Kfeas),’r’)}$$

where the constraints are plotted as blue lines and the response is red.