TCLab State Observer

David Brown, Lindsay Farrell, Niijor May, and Ashley Young

May 4, 2018

1 Problem Statement

TCLab is a device and Python coding system that can be used in process control classrooms to simulate basic control processes and functions, such as feedback and feedforward loops, the process of overshooting and damping effects. This is done by controlling the power input to two heaters on an Arduino device, and measuring the individual temperatures at thermocouples attached to the heat stacks. As of now, TCLab can only measure the temperature at the thermocouple, which does not give as accurate of a temperature reading as it possibly could for the entire device. This is due to the fact that it is measuring the temperature of the device at its heat source as opposed to providing a more accurate temperature reading of the full heat stack. In order to be able to estimate the temperature above the thermocouple, resulting in a better temperature reading of the full heat stack, the code for TCLab needs to be improved.

This will be done by using a Luenberger state observer for linear discrete time systems. The overall goal of the observer is to provide a temperature estimate for the entire TCLab system, rather than just the thermocouple. The technological challenge of this project is that there is only one thermocouple for the heat stack, so the observer is highly dependent on the model of the system.

2 Theoretical Development

2.1 Modeling Equations

The modeling equations for this project originate from the GitHub repository for CBE 30338. The development of the model is based on the energy balances for the TCLab heater and the TCLab sensor. These initial equations are written in terms of the deviation from the ambient temperature.

$$C_p{}^H \left[\frac{dT_H{}'}{dt}\right] = P_1 u - U_a T_H{}' + U_c(T_S{}' - T_H{}') \tag{1}$$

$$C_p{}^S \left[\frac{dT_S{}'}{dt}\right] = U_c(T_H{}' - T_S{}') \tag{2}$$

In this model, $T_H{}'$ and $T_S{}'$ refer to the temperature deviation from the ambient temperature for the heater and sensor, respectively. The coefficient U_a represents the heat transfer coefficient between the heater and the ambient. This model also includes the heat transfer coefficient between the heater and the sensor, U_c. This system of first order equations (Eq. 1 and Eq. 2) can be rewritten into matrix form.

$$\underbrace{\frac{d}{dt}\begin{bmatrix} T_H{}' \\ T_S{}' \end{bmatrix}}_{dx} = \underbrace{\begin{bmatrix} -\dfrac{U_a + U_c}{C_p{}^H} & \dfrac{U_c}{C_p{}^H} \\ \dfrac{U_c}{C_p{}^S} & -\dfrac{U_c}{C_p{}^S} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} T_H{}' \\ T_S{}' \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} P_1 \\ 0 \end{bmatrix}}_{B} u \tag{3}$$

2.2 Core Assumptions

As was stated in the problem statement, a technological challenge in this project is that the thermocouple is measuring the temperature of the device at its heat source as opposed to providing a more accurate temperature reading of the full heat stack. Because of this, we are working with estimates instead of concrete precise numbers. A core assumption stemming from this is that the data we collect is still accurate enough to predict the heat capacities and heat transfer coefficients that we are deriving from it.

It is unlikely that the same Arduino device will be used for all of the data we collect throughout the project. Another assumption that must be made because of this is that the readings from one device to another are consistent and the fact that different devices are used throughout the project makes no difference to the accuracy of the data.

2.3 Preliminary Calculations

The initial modeling equations can be manipulated to be in the form of a second order differential equation in terms of ${T_S}'$. This process starts with the derivative of Eq 2.

$$\frac{d^2{T_S}'}{dt^2} = \frac{U_c}{{C_p}^S}\left[\frac{d{T_H}'}{dt} - \frac{d{T_S}'}{dt}\right] \tag{4}$$

Rearranging and substituting for $d{T_H}'/dt$, the differential equation is almost entirely in terms of ${T_S}'$.

$$\frac{d^2{T_S}'}{dt^2} + \frac{U_c}{{C_p}^S}\frac{d{T_S}'}{dt} - \left(\frac{U_c}{{C_p}^S}\right)^2 {T_S}' = \left(\frac{U_c}{{C_p}^S}\right)\left[\left(\frac{U_c - U_a}{{C_p}^H}\right){T_H}' + P_1 u\right] \tag{5}$$

Substituting for ${T_H}'$ and rearranging gives the second order differential equation for ${T_S}'$ in terms of heat transfer coefficients and heat capacities.

$${T_H}' = \left(\frac{U_c}{{C_p}^S}\right)\frac{d{T_S}'}{dt} + {T_S}' \tag{6}$$

$$\frac{d^2 {T_S}'}{d^2 t} + \frac{d{T_S}'}{dt}\left(\frac{U_c}{{C_p}^S} + \frac{U_a - U_c}{{C_p}^H}\right) + {T_S}'\left(\frac{U_c}{{C_p}^S}\frac{U_a - 2U_c}{{C_p}^S}\right) = \frac{U_c}{{C_p}^S}P_1 u \tag{7}$$

This can be compared to the step response data to provide initial estimates for the heat transfer coefficients and heat capacities.

$$\frac{d^2 {T_S}'}{dt^2} + 2\zeta\tau\frac{d{T_S}'}{dt} + \tau^2 {T_S}' = KU \tag{8}$$

Finally, the quantities K, ζ and τ can be determined by comparing the model against step data from the TCLab.

2.4 Observer Gain

The two components of the state observer are the physical system and the model. The functioning model, developed in the previous section, is in the form of linear, discrete time system. The equations for the implementation of the Luenberger state observer are as follows.

$$\hat{x}(k+1) = A\hat{x}(k) + L(y(k) - \hat{y}(k)) + Bu(k) \tag{9}$$

$$\hat{y}(k) = C\hat{x} \tag{10}$$

In these equations, k represents the current discrete time-step. The variables $\hat{x}$ and $\hat{y}$ represent the variables within the observer, compared to the variables of the actual system.

The observer gain is initially to be determined mainly by testing how the two elements in L effect the performance of the observer. The constraints for the gain can be determined by finding the eigenvalues of the matrix $A - LC$. For the system to be considered stable, the eigenvalues should be within the unit circle.

3 Results

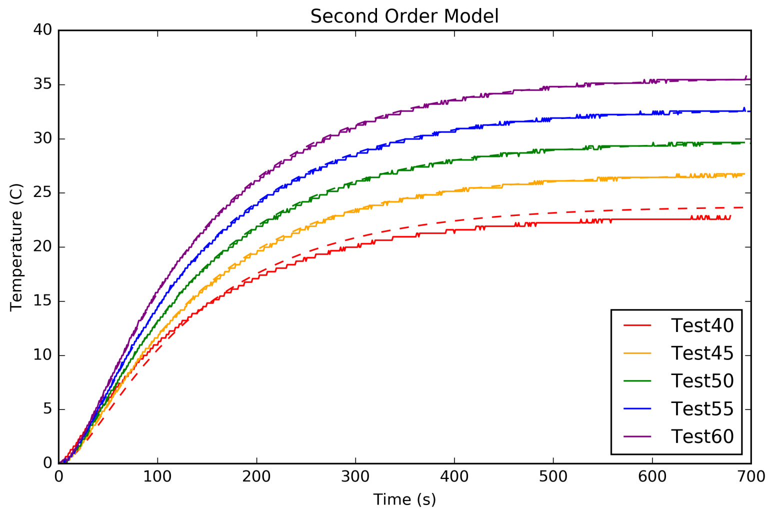

Figure 1: The step test data for various powers are plotted against the solution to the generic second order model.

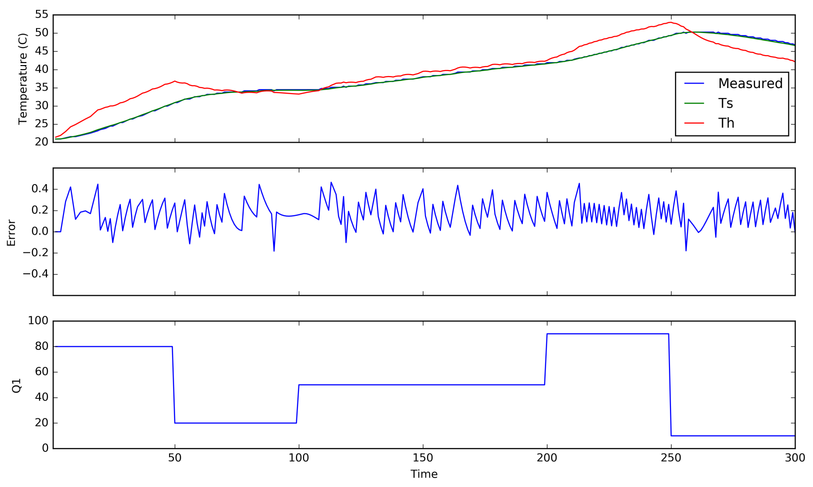

Figure 2: Estimated heater and sensor temperatures with varied heat power, $Q1$

4 Executable Element

The code for the observer module can be found in Appendix A. Furthermore, the complete set of step test data, code, and notebooks can be found in the following GitHub repository:

https://github.com/djbrown1/observer

5 Conclusions and Summary

The estimations provided by the state observer fit the expectations very well. The observer shows that the heater temperature responds to changes in heater power almost immediately, while the sensor component experiences a delay. This is the expected behavior, and reinforces the need for the observer to accurately predict the temperature.

Implementing the state observer as part of a PID controller in it's current state does not appear to be an effective solution. In its current state, the main benefit of the observer is in it's ability to reduce the effects of signal noise and quantization error. These two effects have the potential to improve the precision of a PID controller, though this is only noticeable on the order of tenths of degrees and would likely be negligible compared to the disturbance variables.

Future development would include changing the controller to be control the heater temperature, rather than the sensor temperature. The potential to implement a controller that takes both temperatures into account would be interesting to test, as this could allow for more precise control.

A Appendix: Code for Observer Module

```
class controller():
    """PID controller with a Luenberger observer."""
    def __init__(self,Kp, Ki, Kd, MV_bar=0, beta=1, gamma=0):
        from numpy import matrix

        """ Initialize the PID aspect of the controller """
        # Define the proportional, integral,
        # and derivative coefficients
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd

        # Define the reference value of MV
        self.MV_bar = MV_bar

        # Define the coefficients for setpoint weighting
        self.beta = beta
        self.gamma = gamma

        self.eD_prev = 0
        self.t_prev = 0

        # Initialize the PID variables
        self.P = 0
        self.I = 0
        self.D = 0

        """ Initialize the Luenberger observer
        aspect of the controller """
        # set the model parameters for the observer
        self.A = matrix([[-0.01546814,0.00639784],
                         [0.03924884,-0.03924884]])
        self.B = matrix([[5.71428571429e-3],[0]])
        self.C = matrix([[0,1]])
        self.L = matrix([[1],[1]])

        # Use the observer by default
        self.use_observer = True

        # Initialize the observer variable
        self.x = matrix([[0],[0]])

    def control(self, t, PV, SP, Q1):
        """PID control function."""

        # Apply observer
        PV = self.observer_func(PV, Q1, t)

        # Tracking
        self.I = self.MV - self.MV_bar - self.P - self.D

        # Setpoint Weighting
        eD = self.gamma*SP - PV
        eP = self.beta*SP - PV

        # PID Calculations
        self.P = self.Kp*eP
        self.I = self.I + self.Ki*(SP - PV)*(t - self.t_prev)
        self.D = self.Kd*(eD - self.eD_prev)/(t - self.t_prev)

        MV = self.MV_bar + self.P + self.I + self.D

        # Set values for next iteration
        self.eD_prev = eD
        self.t_prev = t
        # anti-reset windup
        self.MV = 0 if MV < 0 else 100 if MV > 100 else MV

        # return control value for manipulated variable
        return self.MV

    def observer_func(self, y, u, t):
        """Apply observer for the given model."""
        from numpy import matmul

        term1 = matmul(self.A,self.x)
        term2 = matmul(self.L,(y - matmul(self.C,self.x)))
        term3 = self.B*u

        d_est = term1 + term2 + term3

        self.x = self.x + (d_est*(t - self.t_prev))

        PV = matmul(self.C,self.x)[0,0]
        return PV


class observer():
    """Luenberger state observer"""
    def __init__(self, x_0, t_0):
        """ Initialize the coefficients
        and parameters for the observer """
        # set the model parameters for the observer
        self.A = matrix([[-0.01546814,0.00639784],
                         [0.03924884,-0.03924884]])
        self.B = matrix([[5.71428571429e-3],[0]])
        self.C = matrix([[0,1]])
        self.L = matrix([[1],[0.2]])

        self.x = x_0
        self.t_prev = t_0

    def observer_func(self, y, u, t):
        """Apply observer for the given model."""
        from numpy import matmul

        term1 = matmul(self.A,self.x)
        term2 = matmul(self.L,(y - matmul(self.C,self.x)))
        term3 = self.B*u

        d_est = term1 + term2 + term3

        self.x = self.x + (d_est*(t - self.t_prev))

        # Returns the current state of the TCLab
        return self.x
```

B Statement of Contributions

1. **David Brown:**

   Wrote the Python code for the Luenberger Observer and did further research necessary for writing

2. **Lindsay Farrell:**

   Provided writing support with all deliverables. Aided when possible and necessary for all other aspects.

3. **Niijor May:**

   Provided writing support with all deliverables. Aided when possible and necessary for all other aspects.

4. **Ashley Young:**

   Did the initial research, initial problem statement, helped with the other deliverable, and was the main organizer for the group.