SAND2021-12149C

Dakota and Pyomo for Closed and Open Box Controller Gain Tuning

Kyle R. Williams*[1], J. Justin Wilbanks*[1], Rachel Schlossman*[1], David Kozlowski[1], and Julie Parish[1]

Abstract— **Pyomo and Dakota are openly available software packages developed by Sandia National Labs. In this tutorial, methods for automating the optimization of controller parameters for a nonlinear cart-pole system are presented. Two approaches are described and demonstrated on the cart-pole example problem for tuning a linear quadratic regulator and also a partial feedback linearization controller. First the problem is formulated as a pseudospectral optimization problem under an open box methodology utilizing Pyomo, where the plant model is fully known to the optimizer. In the next approach, a black-box approach utilizing Dakota in concert with a MATLAB or Simulink plant model is discussed, where the plant model is unknown to the optimizer. A comparison of the two approaches provides the end user the advantages and shortcomings of each method in order to pick the right tool for their problem. We find that complex system models and objectives are easily incorporated in the Dakota-based approach with minimal setup time, while the Pyomo-based approach provides rapid solutions once the system model has been developed.**

I. INTRODUCTION

Control system engineering is largely concerned with the stabilization of a dynamic system of interest to bring about a desired response. Typically the control design process involves mathematically modeling the dynamic system, synthesizing a stabilizing control law, and tuning the free parameters through some means. In particular, control system tuning is becoming an increasingly difficult task due to the highly complex nature of modern systems as well as the multi-objective nature of most control tasks. As such, systematic methods to automate the tuning process are highly desirable.

For linear time invariant (LTI) systems, the MATLAB Control System Toolbox [1] provides a suite of tools for establishing guarantees of gain and phase margin, rise time, system overshoot, etc. The MATLAB Robust Control Toolbox [2] automates the design of robust controllers for uncertain LTI systems. A survey of evolutionary algorithms in control systems engineering, mostly for LTI systems, is provided in [3]. Feedback linearization [4] is a well-established control method for nonlinear systems. In particular, dynamic inversion (DI) control (a type of feedback linearization involving an inner and outer loop) has been widely adopted in aerospace applications as it provides a straight-forward way of designing multi-variable control laws for nonlinear systems which operate over broad, highly nonlinear regimes [5]. Multi-Objective optimization is used to tune DI control laws in [5] using the Multi-Objective Parameter Synthesis [6] software.

In this work we discuss two additional approaches for control system tuning based on Pyomo and Dakota, two open-source software packages developed at Sandia National Labs. Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities. Pyomo requires an "open box" model in order to leverage its capability in determining effective gains for a specific plant and controller. Dakota is written in C++, and unlike Pyomo, can leverage a "closed box" model where Dakota provides parameters as inputs to a model created in MATLAB/Simulink or other modeling packages. Dakota is able to interact with MATLAB/Simulink using its closed box interface. Both methods have advantages and shortcomings which will be highlighted in this work by applying the methods to a nonlinear cart-pole system. We develop two controllers for this well-studied nonlinear system, and then tune each controller with both software packages.

II. BACKGROUND: DAKOTA AND PYOMO

A. Overview of Pyomo

Python Optimization Modeling Objects (Pyomo) was developed by Sandia National Labs and is an open-source software written in Python for devising mathematical models for optimization [7]. Pyomo allows encoding system dynamics, variables, constraints, and objective functions using natural syntax. The Pyomo model is packaged so that it can be automatically differentiated and passed to a user-specified solver for optimization. The extension `pyomo.dae` [8] allows for the incorporation of differential algebraic equations (DAEs) into the Pyomo model. Once the model is formulated, the user can select one of the supported (linear program, nonlinear program, mixed integer linear programming, quadratic program, etc.) solvers to minimize the user-defined objective function. In [9], the authors perform open-loop trajectory generation with Pyomo via nonlinear optimization for a variety of dynamic systems. In that same work, the authors present a method for gain and phase margin-constrained closed-loop control system optimization using Pyomo. Expanding upon this methodology of closed-loop control system optimization with Pyomo is a focus of this paper.

In this work we focus on solving nonlinear control problems via nonlinear optimization [10]. To solve the optimal control problems, the Pyomo model is discretized and constraints are enforced along the time horizon via direct collocation. The horizon time T can be pre-determined by the user or left as an optimization variable. In either case, a

* Equal Contribution.

[1]The authors are with Sandia National Labs, Albuquerque, NM 87123 `{kwilli2, jjwilba, rschlos, dmkozlo, jparish}@sandia.gov`.

simple time transformation $t = T\tau$ is used where $\tau \in [0, 1]$ is normalized time. All subsequent derivative variables are scaled appropriately by the horizon time as, for example, $\frac{dx}{dt} = \frac{1}{T}\frac{dx}{d\tau}$ and $\frac{d^2x}{dt^2} = \frac{1}{T^2}\frac{d^2x}{d\tau^2}$. The extension `pyomo.dae` allows the user to select the desired discretization scheme, e.g, backward Euler finite different or orthogonal collocation, the number of finite elements (nfe), and/or the number of collocation points per finite element (ncp) [8]. In our work `pyomo.dae` is interfaced with IPOPT [11], a nonlinear interior point optimization software which is compiled with the automatic differentiation library "AMPL Solver Library" (ASL). Pyomo handles all of the backend model formatting so the model can be read and differentiated by ASL, which provides first and second order derivatives to the IPOPT solver for use during optimization.

B. Overview of Dakota

The Dakota toolkit is another open-source tool developed by Sandia National Labs that can be leveraged for complex optimization problems in control design. Dakota is written in C++ and offers the ability to apply iterative optimization and sampling techniques to models developed in various programs, including MATLAB [12]. Using a model with Dakota can be completed by either leveraging Dakota's black-box interface or library mode. In this paper, we will focus on using Dakota's black-box interface with models developed in either MATLAB or Simulink, which are ubiquitous in control development problems. Using the black-box interface, a user can easily and quickly define repeatable optimization studies with varying systems or optimization methods. Dakota offers various local and global methods that can be easily interchanged for a given model. Efficacy of the different methods depends on the system and objective function.

In this paper, we use Dakota's black-box interface to solve nonlinear control problems with the various available solution methods. Dakota simplifies the setup of the control problem for optimization because existing models can be used alongside the approach, which reduces the setup time for the method. We will show that Dakota can be easily used alongside MATLAB and Simulink models in order to benefit from existing model development and built-in MATLAB commands. It is not shown in this paper, but MATLAB could easily be replaced with GNU Octave in order to provide a complete open-source solution [13]. Using Dakota provides the ability to apply global optimization methods without costly MATLAB toolboxes. Within this work, the differential equations governing the system of interest are solved using the ordinary differential equation (ODE) solvers in MATLAB/Simulink [14].

III. The Cart-Pole Nonlinear System

In this work we focus on the cart-pole system described in [15], shown in Fig. 1(a). The horizontal position of the cart with mass m_c is given by x, and the angular position of the pendulum with mass m_p is given by θ, where $\theta = \pi$ is the vertical orientation. The length of the pole is given by ℓ.

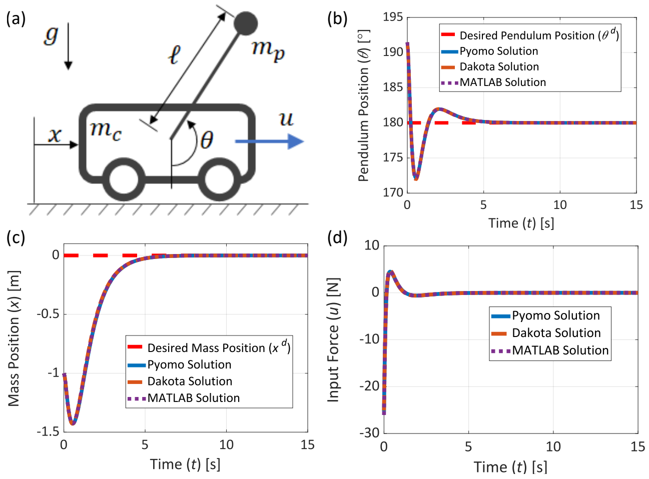

Fig. 1: (a) Cart-pole system. (b), (c), and (d) compare the resulting θ, x. and u trajectories using the Dakota-, Pyomo-, and MATLAB-generated LQR controllers.

A single linear force input u is exerted on the cart. For this system the accelerations can be solved for directly [15],

$$\ddot{x} = \frac{1}{m_c + m_p \sin^2\theta}\left[u + m_p \sin\theta(\ell\dot{\theta}^2 + g\cos\theta)\right] \tag{1a}$$

$$\ddot{\theta} = \frac{1}{\ell(m_c + m_p\sin^2\theta)}\Big[-u\cos\theta - m_p\ell\dot{\theta}^2\cos\theta\sin\theta - (m_c + m_p)g\sin\theta\Big] \tag{1b}$$

The state of the system is defined as $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]^T$ with control vector $\mathbf{u} = u$. The task in this work is to stabilize the unstable equilibrium $\mathbf{x}^{eq} = [0, 0, \pi, 0]^T$, $\mathbf{u}^{eq} = 0$. Note that substituting $\mathbf{x}^{eq}$ and $\mathbf{u}^{eq}$ into (1) produces $\ddot{x} = 0, \ddot{\theta} = 0$. We focus our attention on two approaches for closed-loop controller synthesis. The first approach produces a linear full state feedback controller based on linear quadratic regulator (LQR) optimization. The resulting linear control law is valid locally around $\mathbf{x}^{eq}$. The second approach produces a globally valid nonlinear feedback control based on a partial feedback linearization. These two approaches are discussed in the following sections.

IV. LQR Optimization with Pyomo and Dakota

LQR synthesis is formulated as the following optimization problem subject to linear dynamics [16]

$$\begin{aligned} \text{minimize} \quad & J = \int_0^T \mathbf{x}(t)^T Q\mathbf{x}(t) + \mathbf{u}(t)^T R\mathbf{u}(t)dt \\ \text{subject to} \quad & \dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t) \end{aligned} \tag{2}$$

For a finite horizon the solution to (2) is a time-varying state-feedback control of the form $\mathbf{u}(t) = -K(t)\mathbf{x}(t)$. The solution to the time-varying gain feedback matrix $K(t)$ is given by the well-known matrix differential Riccati equation, which can be solved numerically backward in time along the horizon from a known boundary condition. The matrix $K(t)$ reaches steady state due to the limiting behavior of the matrix differential Riccati equation, and when the horizon is infinite ($T \to \infty$), the solution of (2) produces a fixed gain state-feedback control $\mathbf{u}(t) = -K\mathbf{x}(t)$.

In this work we take an alternative approach to LQR synthesis, solving a modified version of (2). We numerically minimize the LQR objective over a finite horizon subject to the full nonlinear dynamics.

$$\begin{aligned} &\text{minimize} && J = \int_0^T \tilde{\mathbf{x}}(t)^T Q \tilde{\mathbf{x}}(t) + \tilde{\mathbf{u}}(t)^T R \tilde{\mathbf{u}}(t) dt \\ &\text{subject to} && \text{Nonlinear dynamics given by (1)} \\ & && \tilde{\mathbf{u}}(t) = -K\tilde{\mathbf{x}}(t) \end{aligned} \tag{3}$$

where $\mathbf{x}^{eq}$ and $\mathbf{u}^{eq}$ are defined above, and $\tilde{\mathbf{x}}(t) = \mathbf{x}(t) - \mathbf{x}^{eq}(t), \tilde{\mathbf{u}}(t) = \mathbf{u}(t) - \mathbf{u}^{eq}(t)$. We have assumed a linear form of the feedback control, and the gain matrix K is directly optimized by the software. Using the LQR controller the goal is to drive $\tilde{\mathbf{x}}$ to zero, so we consider the initial error $\tilde{\mathbf{x}}_0 =$ [-1 m 0 m/s 0.2 rad 0 rad/s]^T where the pole's orientation is slightly offset from vertical.

A. Optimization in Pyomo

In Listing 1 below we provide pseudocode illustrating the creation of a fixed-horizon Pyomo model for the LQR optimization problem.

Listing 1: Creating Pyomo model

```
import pyomo.environ as pe
import pyomo.dae as pdae
#Define the Pyomo model
m = pe.ConcreteModel()
#Define normalized time horizon set
m.tau = pdae.ContinuousSet(bounds=(0,1))
#Define the actual time horizon (fixed-length)
m.T = pe.Param(initialize=(15))
#Pyomo phase variables indexed along horizon
m.q1 = pe.Var(m.tau)     # x deviation from x_eq
m.q2 = pe.Var(m.tau)     # dx/dtau
m.q3 = pe.Var(m.tau)     # theta deviation from eq
m.q4 = pe.Var(m.tau)     # dtheta/dtau
#Pyomo derivative vars with respect to tau
m.dq1dtau = pdae.DerivativeVar(m.q1, wrt=m.tau)
m.dq2dtau = pdae.DerivativeVar(m.q2, wrt=m.tau)
m.dq3dtau = pdae.DerivativeVar(m.q3, wrt=m.tau)
m.dq4dtau = pdae.DerivativeVar(m.q4, wrt=m.tau)
#Pyomo control variable indexed along horizon
m.u = pe.Var(m.tau) # u deviation
# Objective function, J
m.J = pe.Var(m.tau)
m.dJdtau = pdae.DerivativeVar(m.J, wrt=m.tau)
```

After the time horizon-indexed variables have been defined we can define the system dynamics constraints along the horizon. We first discretize the horizon and then populate the dynamics constraint along the horizon. We also define the quadratic objective function dynamics. The position dynamics are outlined in the example Listing below,

Listing 2: Defining dynamic constraints in Pyomo

```
#Discretize the horizon
discretizer = \
  pe.TransformationFactory('dae.collocation')
discretizer.apply_to(m, wrt=m.tau, nfe=10, ncp=5, \
  scheme='LAGRANGE-RADAU')
#Dynamics constraints
m.dxdt_Constraint = pe.Constraint(m.tau)
m.dJdt_Constraint = pe.Constraint(m.tau)
#Populate the constraints along the horizon
for tau in m.tau:
    # Define absolute coordinates
    x = m.q1[tau] + x_eq
    u = m.u[tau] + u_eq
    # Dynamics
    m.dxdt_Constraint[tau] =\
      m.dq1dtau[tau] == q2[tau]
  # LQR control law
  m.control_Constraint[tau] =\
    m.u[tau] == (RHS of LQR control law (3))
  # Objective
   dJdt = (integrand of (2))
   m.dJdtau_constraint[tau] = \
     m.dJdtau[tau] == m.T * dJdt
```

We then define the system boundary conditions and the objective function to be minimized by the software,

Listing 3: Defining boundary conditions in Pyomo

```
#Define boundary conditions
def BCs(m):
   yield m.q1[0] == -1
   yield m.q2[0] == 0
   yield m.q3[0] == 0.2
   yield m.q4[0] == 0
   yield m.J[0] == 0
#Pyomo constraint list
m.BC_Constraint = pe.ConstraintList(rule=BCs)
#Pyomo objective function
m.obj = pe.Objective(expr=m.J[1], sense=pe.Minimize)
```

The final step is to specify the solver (in our case, IPOPT) and solve the model. We do not illustrate these steps here, and leave the details to [7], [8]. The bounds and initial values of K are shown in Listing 4:

Listing 4: Initializing LQR K gains in Pyomo.

```
m.K_x = pe.Var(initialize=0, bounds=(-200, 200))
m.K_xd = pe.Var(initialize=0, bounds=(-200,200))
m.K_theta = pe.Var(initialize=0, bounds=(-200,200))
m.K_thetad = pe.Var(initialize=0, bounds=(-200,200))
```

The dynamic constraints are imposed in Pyomo by converting (1) to four first order differential equations. Before imposing these constraints, we shift values from relative error state ($\tilde{\mathbf{x}}(t)$) to absolute state ($\mathbf{x}(t)$), and impose constraints on the absolute state. Pyomo is then used to encode (3). We select the Q and R values to be $Q = 50 \times I_{4\times4}$ and $R = 1$, where $I_{4\times4}$ is the 4×4 identity matrix.

We solve this problem with $T = 15$ s, using orthogonal collocation with Gauss-Radau roots, 35 nfe, and 5 ncp. All Pyomo-based optimization is completed on a Windows 10 laptop PC with 32.0 GB of RAM and a Intel Core i7-8850U CPU. The time to build and solve the Pyomo model is 1.64 s, and Pyomo selects the gains as $K = [-6.82, -12.45, 92.32, 28.68]$. The results are shown in Fig. 1(b)-(d). The most interesting item to note is that the Pyomo formulation does not require the user to explicitly linearize the dynamics, which is a necessary step for nonlinear systems when using the Riccati equation to solve for K.

B. Optimization in Dakota

In order to run a Dakota analysis, a set of files to define the optimization process and simulation of the cart-pole system is required. In this paper, MATLAB and Simulink are used to complete the simulation of the cart-pole system. The following files are required to setup the Dakota black-box interface for control design:

- ***.in**: The input file (*.in) specifies the solver type for the Dakota optimization process as well as the ranges and initial values for the design variables. For the black-box interface, a *.vbs/*.sh script is used to open and interface with MATLAB/Simulink. Temporary results and parameter files are specified in the *.in in order to update the parameters in MATLAB/Simulink and send the current value of the objective function back to Dakota to inform the next iteration's parameter choices.
- ***.sh / *.vbs**: The *.sh or*.vbs file opens MATLAB in either Linux (*.sh) or Windows (*.vbs) in a working directory. Using Windows and a *.vbs allows for MATLAB to be called as a COM Automation Server, which can speed up simulations when calling Dakota in serial on Windows compared to Linux.
- ***_Wrapper.m**: A MATLAB file is used to specify the current parameter choices made by Dakota in the MATLAB/Simulink model of interest, the cart-pole system here, and save the current values of the objective functions as a temporary text output file that can be read by Dakota to continue to subsequent iterations.
- ***.m**: An additional MATLAB function is used to contain the dynamic model and control implementation of interest, which can subsequently call a Simulink model (*.slx). This function is called within the *_Wrapper.m file using the current parameter choices made by Dakota. Using Dakota and the black-box interface allows for the various built-in functionalities of MATLAB/Simulnk to be used within this process.

The design variables for the Dakota process were defined as the four values that define the K vector. The ranges for the values of the K vector were defined in the input file for the process as continuous design variables with specified ranges, as shown in Listing 5:

Listing 5: Initializing LQR K gains in Dakota.

```
variables ,
   continuous_design = 4
   cdv_initial_point  0       0       0       0
   cdv_lower_bounds  -100    -100    -100    -100
   cdv_upper_bounds   100     100     100     100
   cdv_descriptor    'k1'    'k3'    'k2'    'k4'
```

where 'k1', 'k2', 'k3', and 'k4' is K_x, $K_{\dot{x}}$, K_θ, $K_{\dot{\theta}}$, respectively. As was stated in the previous section for Pyomo, the design variables are not warm-started and are all initialized at 0 with $\mathbf{x}_0 = [\text{-1 m} \quad \text{0 m/s} \quad \pi + 0.2 \text{ rad} \quad \text{0 rad/s}]^T$. The system of four first order differential equations are solved using `ode45` within MATLAB, which uses an an explicit Runge-Kutta (4,5) (RK45) formula [14]. The Q and R values are again selected to be $Q = 50 \times I_{4\times4}$ and $R = 1$. For the Dakota simulations, we define $T = 10$ s. The Dakota simulations are completed on a Windows 10 laptop PC with 16.0 GB of RAM and a Intel Core i7-8650U CPU.

The optimization problem in Dakota is solved using the COLINY EA solver, which is an evolutionary algorithm provided in the Sandia Colin Optimization Library (SCOLIB) collection of non-gradient-based optimizers which support the Common Optimization Library Interface (COLIN) [12]. The maximum function evaluations were defined as 8000 with a population size of 75 in the minimization of (3). The overall time to run the optimization process with the Dakota black-box interface is 31.3 min, and Dakota selects the gains as $K = [-6.85, -12.42, 91.13, 28.10]$.

The response of the system and input force defined with the Dakota gains is provided in Fig. 1(b)-(d). The Dakota-defined LQR controller is able to settle the pendulum within 2% of the desired value of π rad in 4.68 s. As with Pyomo, it was also possible to acquire K without linearization using Dakota.

C. Validation

The gains obtained with Dakota and Pyomo can be compared with the results obtained using the `lqr` function within MATLAB, which requires the system to be linearized. Table I and Fig. 1(b)-(d) compare the resulting LQR controller definitions from Pyomo, Dakota, and MATLAB. Final cost function values, defined using (3), for the Pyomo and Dakota gains were 0.23% and 0.32% lower than the final cost with the MATLAB gains obtained with `lqr`, respectively. The largest difference in the gains obtained with the optimization approaches was 4.67% for the value of $K_{\dot{\theta}}$ obtained with Dakota. Responses of the mass and pole of the system with Dakota and Pyomo are indistinguishable from what is observed with the MATLAB gains, which can be observed in Fig. 1(b) and 1(c). Maximum position error for the pole is 4.3% for the MATLAB and Pyomo gains and slightly higher at 4.5% for the Dakota solution. The required input force trajectories obtained with the Pyomo and Dakota gains, shown in Fig. 1(d), closely follow the MATLAB solution. Maximum forces observed with the Pyomo and Dakota gains are 2.76% and 4.02% less than the value obtained with the MATLAB gains.

TABLE I: LQR controller gains obtained with Pyomo, Dakota, and MATLAB.

Method	K_x	$K_{\dot{x}}$	K_θ	$K_{\dot{\theta}}$	J
Pyomo	-6.82	-12.45	92.32	28.68	60.92
Dakota	-6.85	-12.42	91.13	28.10	60.86
MATLAB	-7.07	-12.98	94.94	29.48	61.06

V. Partial Feedback Linearization Optimization with Pyomo and DAKOTA

In the previous section we synthesized a linear full-state feedback controller. This was accomplished by directly optimizing a finite horizon objective with Pyomo and Dakota. The resulting control law is valid locally near the equilibrium $\mathbf{x}^{eq} = [0, 0, \pi, 0]^T$, $\mathbf{u}^{eq} = 0$. We now synthesize a nonlinear controller based on partial feedback linearization [15], which produces a globally valid control law. The gains of the controller will be optimized through Pyomo and Dakota. The goal once again is to stabilize the unstable equilibrium, which occurs when the pendulum is vertically oriented. We proceed by solving (1b) for u such that $\ddot{\theta} = v$ where

$$v \triangleq k_d(\dot{\theta}^d - \dot{\theta}) + k_p(\theta^d - \theta) \qquad (4)$$

Solving for u gives

$$u = -\frac{1}{\cos\theta}\Big[v\ell(m_c + m_p \sin^2\theta) + m_p \ell \dot{\theta}^2 \cos\theta \sin\theta + (m_c + m_p) g \sin\theta\Big] \tag{5}$$

This nonlinear control law is valid globally for $\theta \neq \{\pi, -\pi\}$. Assuming a perfect system model, substituting this control into (1b) produces the closed loop system

$$\theta(s) = \underbrace{\frac{k_p + k_d s}{s^2 + k_d s + k_p}}_{T(s)} \theta^d(s) \tag{6}$$

Here $T(s)$ is the closed loop transfer function which is stable for all values of k_p, k_d such that $\mathrm{Re}\{s^2 + k_d s + k_p\} < 0$.

In addition to this perfect model assumption, we also consider a time delay of τ second on the control signal in which case $u(t-\tau)$ is applied to (1b) at time t. We note this is not the same as introducing a transport delay in the forward path transfer function of (6), since the time-delayed control signal produces imperfect cancellation of the system dynamics. As a result, the form of the closed loop system (6) is lost in the time-delayed control case.

We adjust gains k_d, k_p of (4) to solve the following optimization problem

$$\begin{aligned} &\text{minimize} \quad J = \int_0^T (\theta^d(t) - \theta(t))^2 dt + w_1 k_p + W_{ST} t_s \\ &\text{subject to} \quad (1), (4), (5) \end{aligned} \tag{7}$$

Here w_1 is a regularization term weight, limiting the preferred size of k_p (and indirectly, the size of k_d). W_{ST} is a penalty on the settling time t_s, where a larger W_{ST} encourages a shorter settling time.

A. Optimization with Pyomo

We now perform optimization in Pyomo for the two cases discussed above: control input without and with time delay. In both cases the initial condition is $\mathbf{x}_0 = [0 \text{ m} \quad 0 \text{ m/s} \quad 1.92 \text{ rad} \quad 0 \text{ rad/s}]^T$. We solve this problem with a horizon of $T = 5$ seconds using backward Euler collocation with 500 nfe. We set $w_1 = 0.02$ and $W_{ST} = 0$ in (7). The weight W_{ST} is set to zero since modeling the settling time in Pyomo is non-trivial, and is not attempted in this work. We note that in order to include settling time in the objective during Pyomo optimization, the settling time would need to be written in terms of Pyomo variables (states and control inputs), without any logical rule type expressions (e.g., if/else statements) since Pyomo does not support such expressions.

1) No Time Delay: We first consider the case without any time delay on the control signal. The time to build and solve the Pyomo model is 1.65 seconds. The gains found through optimization are $k_p = 11.17, k_d = 3.24$. Fig. 2 shows the Pyomo solution compared to the numerical simulation of the closed loop system using Simulink [17] under a RK45 numerical differential equation solver scheme.

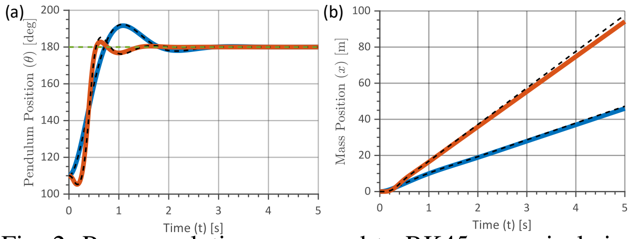

Fig. 2: Pyomo solution compared to RK45 numerical simulation. (a) and (b) compare the resulting θ and x trajectories, respectively. Blue curves are with no time delay on the control signal, red curves are with a time delay. Dashed black curves are the corresponding RK45 numerical simulation.

2) Time Delay: We next consider the case with a 0.1 second time delay on the control signal. We use Pyomo to produce an adjusted set of gains that are robust to the delay. To model the delay we use Pyomo's previous time step feature, which allows the user to directly implement the value of a quantity obtained from previous time steps. Since we are using a 5 second horizon with 500 evenly distributed discretization points, each point corresponds to a 0.01 second time step. To model a 0.1 second delay we therefore require 10 delay steps as indicated below in Listing 6. We note that for the first 0.1 seconds the control input to the system is zero.

Listing 6: Implementing time delay in control signal.

```
# Define control input
m.u = pe.Var(m.tau)
# Define the number of delay steps
tau_idx_delay = 10
# time step index for time tau
tau_idx = list(m.tau.data()).index(tau) + 1
if tau_idx > tau_idx_delay:
  tau_prev = m.tau.prev(tau, step=tau_idx_delay)
  u = m.u[tau_prev]
else:
  u = 0
```

To promote solution stability, we use the previous solution (without time delay) as a warm-start. The time to build and solve the Pyomo model is 3.87 seconds (including time for the previous build and solve). The gains found through optimization are $k_p = 13.92, k_d = 6.12$. We again compare the Pyomo solution with the numerical simulation of the closed loop system using Simulink shown in Fig. 2.

3) Cross Validation: We now numerically simulate the nominal and time-delayed systems in closed loop under each set of gains (nominal and adjusted). The simulations are again performed in Simulink under RK45. In these simulations the pendulum is initially oriented straight down. The control input is limited to ± 100 N to prevent infinite control effort when $\theta = \pm\pi/2$.

Figure 3(b) shows the robustness of the adjusted gains in the time-delayed system. As expected, the adjusted gains provide good performance under both the nominal and time-delayed systems. We also show the sensitivity of the time-delayed system under the nominal gains in Fig. 3(a), noting that the nominal gains perform poorly in the presence of time delay.

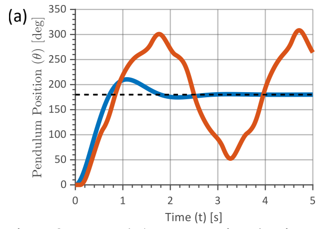

Fig. 3: RK45 numerical simulation for cross validation: (a) nominal gains on nominal system (blue) and time-delayed system (red); (b) adjusted gains on nominal system (blue) and time-delayed system (red).

B. Optimization with Dakota

Dakota was also leveraged to choose the best gains for the partial feedback linearization input defined in (4) and (5). Optimization is initially performed with $w_1 = W_{ST} = 0$ in (7). Later, W_{ST} will be made nonzero to reduce settling time oscillations. We note that no weighting is required on k_p (w_1 can be zero) when W_{ST} is nonzero. As a result, $w_1 = 0$ in the Dakota approach. Several different solutions methods were used to determine k_d and k_p for the partial feedback linearization controller. The resulting gains are provided in Table II and the resulting pendulum position trajectory is provided in Fig. 4. The solution methods used alongside the Dakota black-box interface are:

- **COLINY EA**: Evolutionary algorithm provided in Sandia Colin Optimization Library (SCOLIB) non-gradient-based methods that are part of the Common Optimization Library Interface (COLIN)
- **SOGA**: Single-objective genetic algorithm
- **COLINY PS**: Derivative free pattern search provided in SCOLIB collection
- **CONMIN FRCG (Gradient)**: Gradient-based optimization approach

where [12] provides additional details for each method.

TABLE II: Partial feedback linearization controller gains obtained with various Dakota solution methods and corresponding computation times ($t_{Computation}$).

Solution Method	k_p	k_d	$J_{Reduction}$	$t_{Computation}$
COLINY EA	125.00	11.00	91.03%	6.84 min
SOGA	123.66	10.20	90.95%	7.32 min
COLINY PS	125.00	11.18	91.03%	0.89 min
GRADIENT	125.00	11.18	91.03%	0.38 min

All of the solution methods provide more than 90% reduction in the cost function defined in (7) when gains k_d and k_p are set to 1. For the COLINY EA and SOGA solution methods, the maximum number of function evaluations allowed was set to 1000 with a population size of 125. Initial conditions for the system in theses analyses matched those used in Pyomo. Gains k_d and k_p were initialized with an arbitrary value of 1 and kept within a range from 0.1 to 125. Dakota does not require these gain values to be warm-started. With these parameters, the COLINY PS and GRADIENT solution methods were able to converge significantly faster at 0.89 min and 0.38 min, respectively, compared to the times of 6.84 min and 7.32 min for COLINY EA and SOGA, respectively. All of the solution methods resulted in similar responses for the pendulum of the system, which can be observed in Fig. 4. The resulting settling times (t_s), maximum force (u_{Max}), and overshoot values for the pendulum component of the system are within 5% of each other. The cart-pole system in these cases was modeled using Simulink to allow for simple implementation of the time-delay on the input, $u(t - \tau)$, and additional validation of the gains generated with Pyomo. The position trajectory of the pendulum and input signal for the optimized gains with varying input delays can be observed in Fig. 5(a)-(b).

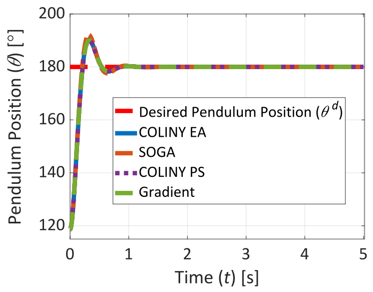

Fig. 4: Impact of the solution method on the resulting θ trajectory using the Dakota-generated partial feedback linearization controller.

C. Comparison and Validation

Table III compares the gains obtained with Pyomo accounting for a 0.1 s input delay to the gains obtained with Dakota with and without accounting for the settling time in the cost function. Pyomo provides a solution that reduces the cost function by 91.0% relative to the initial value with the gains defined as 1. The reduction observed with Pyomo is slightly less than the reduction of 92.71% observed with Dakota without accounting for the settling time, but the Dakota gains result in a 44.3% increase in the settling time.

TABLE III: Partial feedback linearization controller gains obtained with Pyomo and Dakota with $\tau = 0.10$ s.

Method	k_p	k_d	$J_{Reduction}$	t_s	u_{Max}
Pyomo	13.92	6.12	90.99%	1.31 s	97.9 N
Dakota ($W_{ST} = 0$)	27.92	9.08	92.71%	1.90 s	156.6 N
Dakota ($W_{ST} = 1$)	11.22	5.85	89.91%	0.76 s	87.9 N

VI. Conclusions

Pyomo and Dakota are both powerful tools for control designers to optimize the gains of a controller given various objective functions and constraints. The primary trade-off between Pyomo and Dakota is setup time versus optimization time. Using Dakota leads to shorter setup time since one can leverage existing plant models, but optimization time can be significant due to the coupling of the algorithm and simulation model. Setup of the Pyomo model can take significant time, but the optimization tends to be much faster.

In this tutorial we have addressed a single-input multi-output system. Implementing Pyomo and Dakota on multi-input systems is a straight-forward extension of the methods described here. An example of using Pyomo on a multi-input aerospace optimization problem is shown in [9].

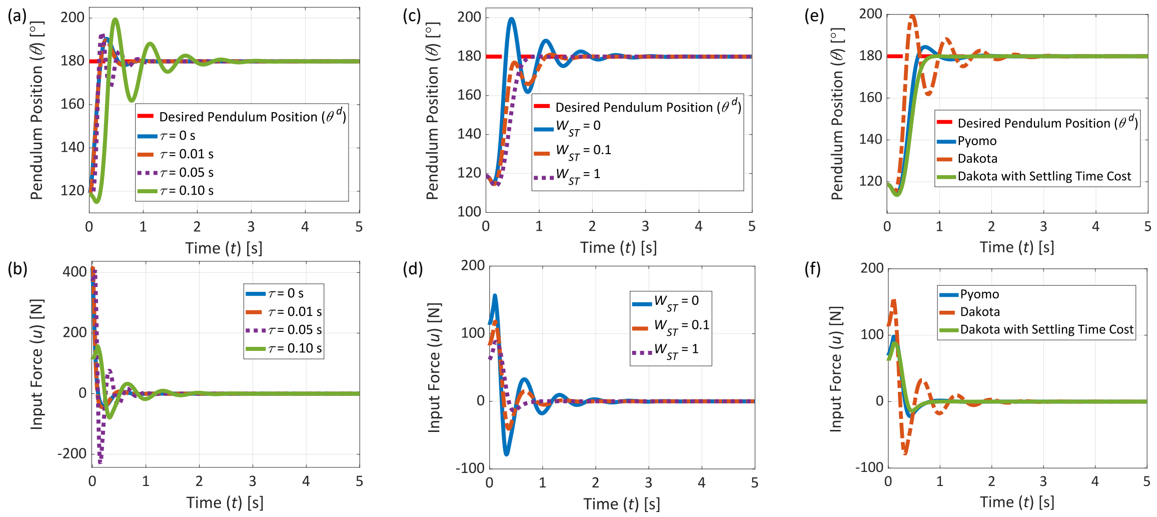

Fig. 5: (a) and (b) show the resulting θ and input trajectories using the Dakota-generated partial feedback linearization controller under various input time delays. (c) and (d) show the trajectories using the Dakota-generated partial feedback linearization controller with settling time cost. (e) and (f) compares the trajectories resulting from Pyomo and Dakota.

In addition to the methods discussed here, novel approaches based on Bayesian Optimization have been developed for automated tuning of nonlinear control systems. For example, Bayesian Optimization is applied to LQR synthesis for a robotic inverted pole balancing problem in [18]. The associated Q and R matrices are parameterized and then tuned via Bayesian Optimization. The (expensive to evaluate) latent objective function is represented as a Gaussian Process where the evaluation points are determined via Entropy Search [19]. More recently, the Python package BoTorch [20] was developed as a framework for Bayesian Optimization.

Acknowledgement

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] C. H. Houpis and S. N. Sheldon, *Linear Control System Analysis and Design with MATLAB®*. CRC Press, 2013.

[2] G. Balas, R. Chiang, A. Packard, and M. Safonov, "Robust control toolbox 3," *The Mathworks, Inc., Natick, MA*, 2005.

[3] P. J. Fleming and R. C. Purshouse, "Evolutionary algorithms in control systems engineering: a survey," *Control engineering practice*, vol. 10, no. 11, pp. 1223–1241, 2002.

[4] J.-J. E. Slotine, W. Li, *et al.*, *Applied nonlinear control*. Prentice hall Englewood Cliffs, NJ, 1991, vol. 199, no. 1.

[5] G. Looye and H.-D. Joos, "Design of robust dynamic inversion control laws using multi-objective optimization," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2001, p. 4285.

[6] H.-D. Joos, "Multi-objective parameter synthesis (mops)," in *Robust Flight Control*. Springer, 1997, pp. 199–217.

[7] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, and J. D. Siirola, *Pyomo-optimization modeling in python*. Springer, 2017, vol. 67.

[8] B. Nicholson, J. D. Siirola, J.-P. Watson, V. M. Zavala, and L. T. Biegler, "pyomo. dae: a modeling and automatic discretization framework for optimization with differential and algebraic equations," *Mathematical Programming Computation*, vol. 10, no. 2, pp. 187–223, 2018.

[9] R. Schlossman, K. Williams, D. Kozlowski, and J. J. Parish, "Open-source, object-oriented, multi-phase pseudospectral optimization using pyomo," in *AIAA Scitech 2021 Forum*, 2021, p. 1951.

[10] M. Kelly, "An introduction to trajectory optimization: How to do your own direct collocation," *SIAM Review*, vol. 59, no. 4, pp. 849–904, 2017.

[11] L. T. Biegler and V. M. Zavala, "Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization," *Computers & Chemical Engineering*, vol. 33, no. 3, pp. 575–582, 2009.

[12] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, M. S. Ebeida, J. P. Eddy, M. S. Eldred, R. W. Hooper, P. D. Hough, K. T. Hu, J. D. Jakeman, M. Khalil, K. A. Maupin, J. A. Monschke, E. M. Ridgway, A. A. Rushdi, D. T. Seidl, J. A. Stephens, L. P. Swiler, and J. G. Winokur, "Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 6.13 user's manual," *Sandia Technical Report*, no. SAND2020-12495, 2020.

[13] J. W. Eaton, D. Bateman, H. Søren, and R. Wehbring, "Gnu octave version 6.2.0 manual: a high-level interactive language for numerical computations," 2021.

[14] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," *SIAM Journal on Scientific Computing*, vol. 18, pp. 1–22, 1997.

[15] R. Tedrake, "Underactuated robotics: Algorithms for walking, running, swimming, flying, and manipulation (course notes for mit 6.832). downloaded on 4-30-2021 from http://underactuated.mit.edu."

[16] D. E. Kirk, *Optimal control theory: an introduction*. Courier Corporation, 2004.

[17] S. Documentation, "Simulation and model-based design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[18] A. Marco, P. Hennig, J. Bohg, S. Schaal, and S. Trimpe, "Automatic lqr tuning based on gaussian process optimization: Early experimental results," in *Second Machine Learning in Planning and Control of Robot Motion Workshop at International Conference on Intelligent Robots and Systems*, 2015.

[19] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization." *Journal of Machine Learning Research*, vol. 13, no. 6, 2012.

[20] M. Balandat, B. Karrer, D. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "Botorch: A framework for efficient monte-carlo bayesian optimization," *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.