**2023 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
"DATA ANALYSIS AND MODELING" OR "TESTING AND VALIDATION" TECHNICAL SESSION
AUGUST 15-17, 2023 - NOVI, MICHIGAN**

Transient Torque Response of Tracked Vehicle Suspension Rotary Dampers

David Ostberg[1], Kenneth Redner[2], Samuel Allen[1]

[1]U.S. Army Ground Vehicle Systems Center, Warren, MI
[2]Booz Allen Hamilton, Troy, MI

ABSTRACT

Damper models used in Multi Body Dynamics simulations of tracked vehicles are commonly defined solely by damper curves, that is stabilized damper reaction torque as a function of steady state velocity. In reality, the achievement of the stable reaction torque lags behind damper curve torque upon attainment of a given velocity. As detailed in this paper, the idealization to reduce damper performance to a "damper curve" cannot produce an accurate representation of the two primary terrains military vehicles are designed against: half-round and ride quality courses. By introducing "compliance" and "lash", the damper performance can be accurately represented. With the slight extension of this model to a fully physics based one, future dampers can be designed to expand the operating performance envelope of tracked vehicles.

Citation: D. Ostberg, K. Redner, S. Allen, "Transient Torque Response Modeling of Tracked Vehicle Suspension Rotary Dampers," In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 16-18, 2022.

1. INTRODUCTION

U.S. Army Ground Vehicle Systems Center (GVSC) conducted vehicle experiments in 2022 at Yuma Proving Ground (YPG) and it was found that load correlation between instrumented arms of tracked vehicles and the simple models of springs and dampers that are traditionally used in multi body dynamics (MBD) simulations have incredibly poor correlation for damped stations yet excellent for undamped stations. This difference is due to the traditionally used models being poor predictors of actual reaction torques of dampers.

These same models are also used during the design phase of vehicle development where damper response is tuned to maximize percieved ride quality over terrains, and limit accelerations over discrete events like half-rounds. After vehicles are produced they are then evaluated against these two conditions per TOP-1-1-014 [1].

It is expected that dampers are improperly tuned because of this gap between actual and modeled damper performance. This ultimately limits vehicle mobility by making the ride unnecessarily uncomfortable.

With these motivations GVSC proceeded to conduct bench testing to identify the root cause of the deficiency in the damper modeling practice and develop a damper model that accurately represents damper performance with the minimum possible complexity.

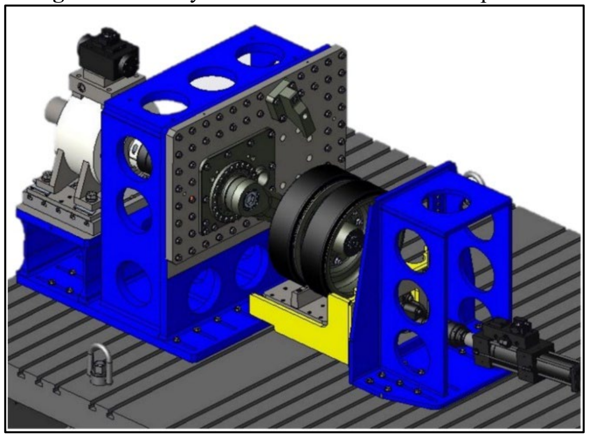
2. BACKGROUND

GVSC began this investigation using a tester specifically developed for tracked vehicle suspensions.

2.1. GVSC Damper Test System

As discussed in a GVSETS paper from 2021 [2], a test system was developed to test complete trailing arm assemblies and has proven valuable to excite failure modes of road arm durability and bearing wear.

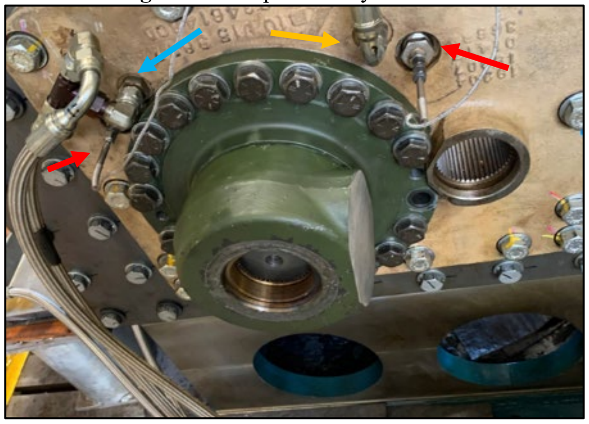
Figure 1. Test System from GVSETS 2021 Paper



This tester was modified to conduct damper evaluation by replacing the full trailing arm assembly with one cut short but retaining all upper spindle (trunnion) interfaces. With this update, the rotary actuator now directly drives the damper in the actual suspension housing.

Additionally, a closed loop heated oil system was installed to control housing temperature by providing approximately 24 Gallons Per Minute (GPM) oil flow from a reservoir with PID controlled heaters capable of maintaining temperatures ranging from unheated (~70F) to 350F. In Figure 2, arrows indicate the oil inlet (blue), oil outlet (orange), and two housing oil monitoring thermocouples (red).

Figure 2. Damper Test System.



Because of the excellent performance envelope of this test setup, the damper reaction torques from actual time/angle series data from the field, over durability and half round courses, as well as traditional damper sinusoids and ramps could be found.

2.2. Damper Model Summaries

As previously stated, the baseline *traditional model* provided a poor prediction of reaction torques. This model consists of a four-part piecewise linear damper curve: high speed compression (jounce), low speed compression, low speed extension (rebound), and high-speed extension.

In Section 4 an *updated model* which improves upon the traditional one by replacing the low speed linear regions with quadratic dominated ones is introduced. In extension the model is quadratic. In compression it is quadratic up until a critical velocity, and beyond this velocity it is linear. Additionally, a term for friction is introduced.

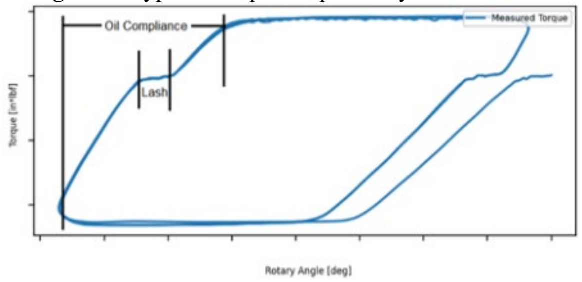
In Section 5 an *updated compliant model* is introduced, which addresses the two critical phenomena missing from the traditional and updated damper curve models: lash and compliance.

"Lash" refers to a finite travel low stiffness region that is observed at low values of torque when the system changes direction.

"Compliance" refers to the non-instantaneous building and decaying of reaction torque as damper velocity changes.

Figure 3 shows a hysteresis plot of angle and damper reaction torque as the damper is cycled through ramp functions. The characteristic “lash” and “compliance” are evident.

Figure 3. Typical Torque Response Hysteresis Plot

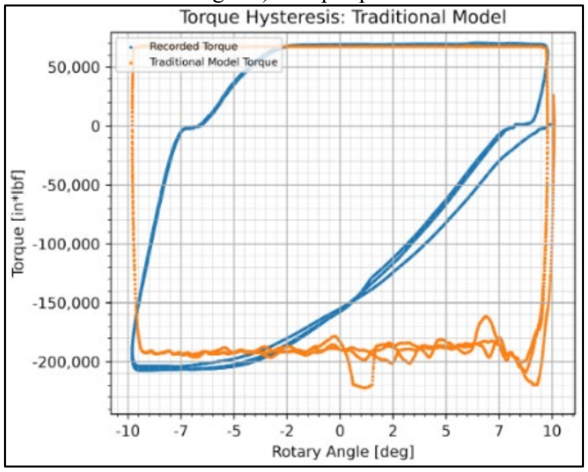


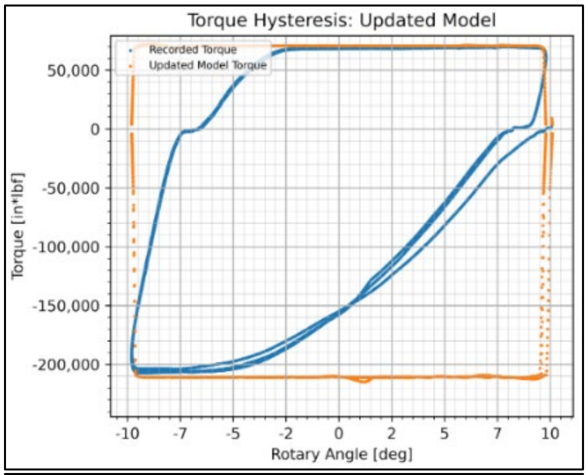
2.3. Damper Model Performance

In this section a series of cross plots of angle and damper reaction torque are shown. Within each figure the laboratory test data is blue and model predictions are orange. The first figure is the *traditional model*, which is followed by the *updated model*, and then the *updated compliant model*.

Figure 4 shows that all three models provide a good estimation of the steady state (constant velocity) torque response for a ramp function at high speeds, however only the updated compliant model captures the lash and compliance characteristic.

Figure 4. Model Comparison with High Velocity (25 deg/sec) Ramp Input





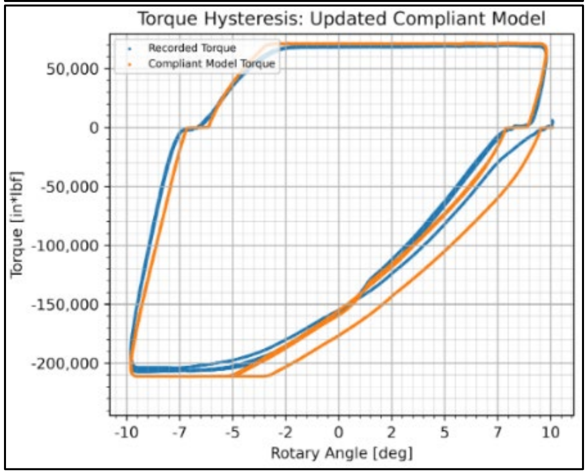
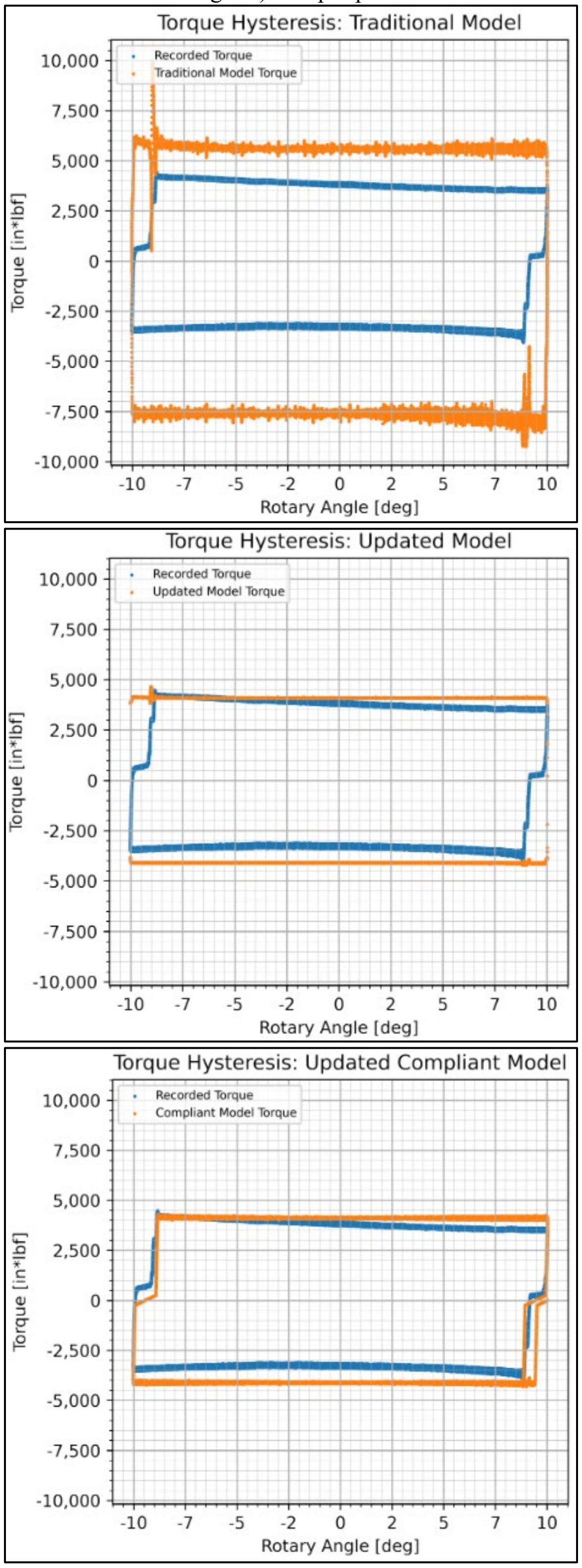


Figure 5 shows for ramp functions at low velocities (1 deg/sec). The lash and compliance characteristics are nearly negligible, hence the strong similarity of the *updated* and *updated compliant models* to the test result. However, the *traditional model* deviates significantly from the test data.

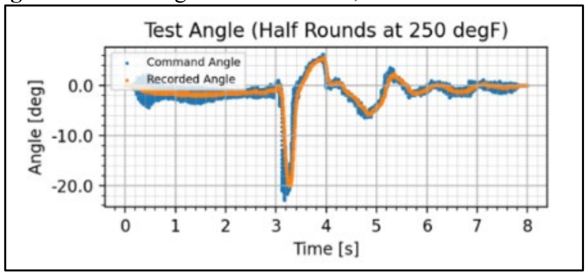
Figure 5. Model comparison with Low Velocity (1 deg/sec) Ramp Input



The angle versus time profile over a practical half-round event shown in Figure 6.

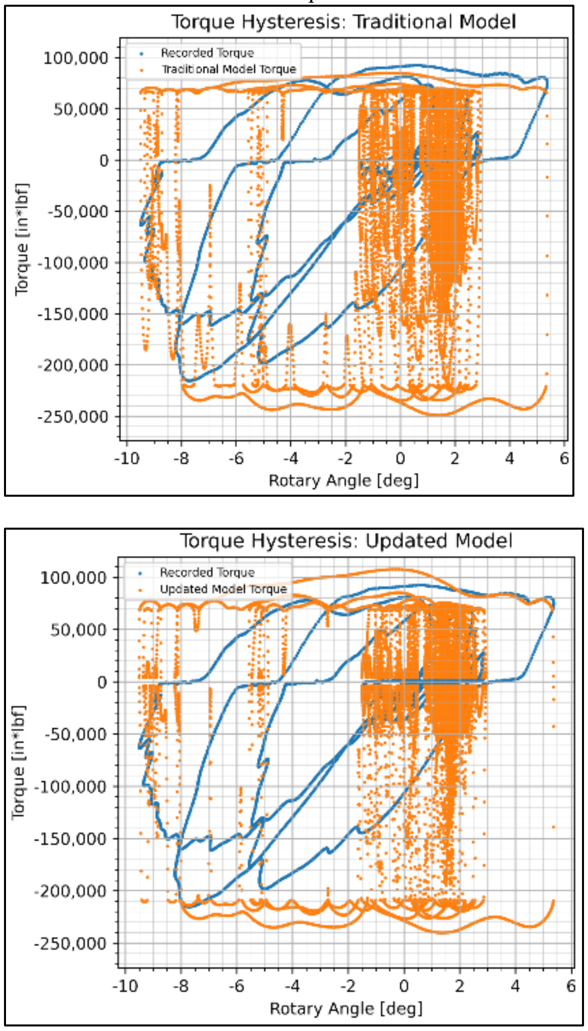
The half-round event provides the most dramatic comparison of these three models.

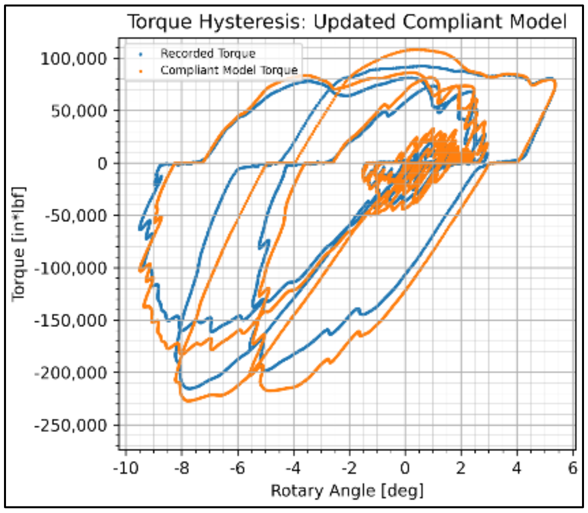
Figure 6. Time-Angle Test Condition, 8" Half Round Test



The model torques over this event are compared to the actual component reaction torque in Figure 7.

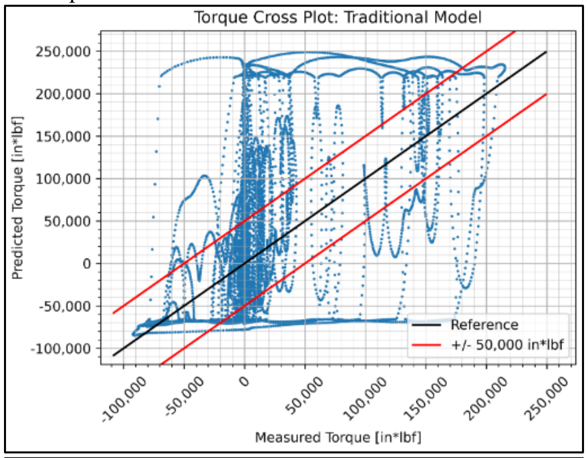
Figure 7. Model Comparison Histograms with Half round Test Input

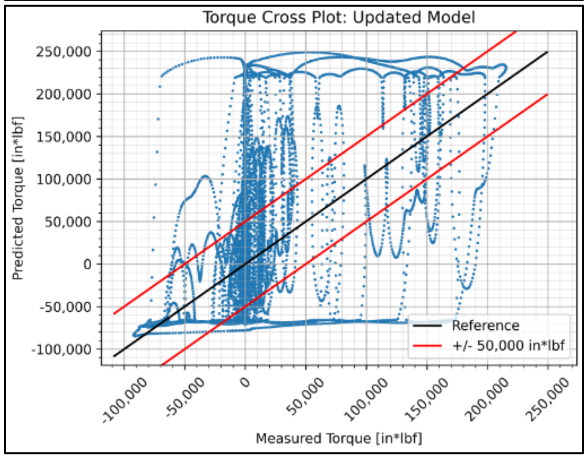


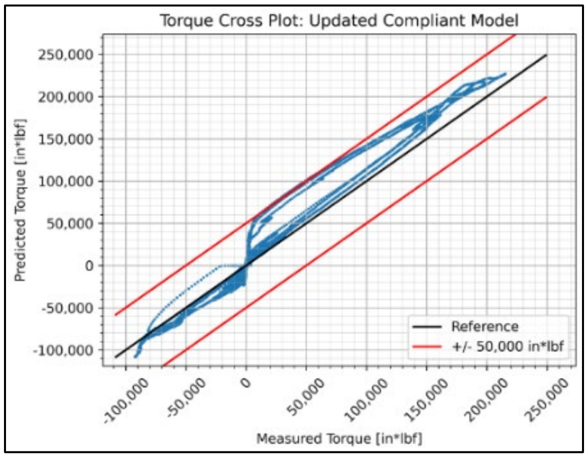


The difference between model prediction and recorded reaction torque is shown in the cross-plots of Figure 8. Included in these Figures are the reference line of zero error, when model prediction and recorded reaction torque exactly match, in black and +/- 50,000 in*lbf in red.

Figure 8. Model Comparison Cross-Plots with Half Round Test Input







The half round example is a practical test which is critical to accurately predict when designing vehicles. The *traditional model* and *updated model* are incredibly poor predictors as shown, with errors exceeding +/- 250,000 in-lbf.

The *updated compliant model* provides a very good prediction but has a slight over prediction of reaction torque magnitudes. This slight over prediction is expected based on the method used to generate the damper curve as clarified in Section 3. Further model tuning could be conducted and would lead to an improved prediction.

3. DAMPER CURVE GENERATION

Damper reaction torque data for generating the updated damper curve was collected on the test bench by running triangle (ramp) wave forms of amplitude 10 degrees at various angular rates.

One single damper was used for the characterization described in this section was well as generation of all datasets discussed in the entirety of this paper.

3.1. Initial Testing

The speeds specified in Table 1 were ran starting from three stabilized thermal states: ambient (~70F), 250°F and 350°F.

Table 1. Bench Test Input Parameters

Ramp Rate [deg/s]
±0.05
±0.1
±0.5
±1
±5
±10
±25
±50
±100
±200

Raw position and torque channels collected from instrumentation were processed before use by forward and backward filtering using a low pass Butterworth filter.

Table 2. Butterworth Filter Parameters

Sample Rate	2000 Hz
Filter Frequency	400 Hz
Filter Order	4

3.2. Smoothing of Velocity

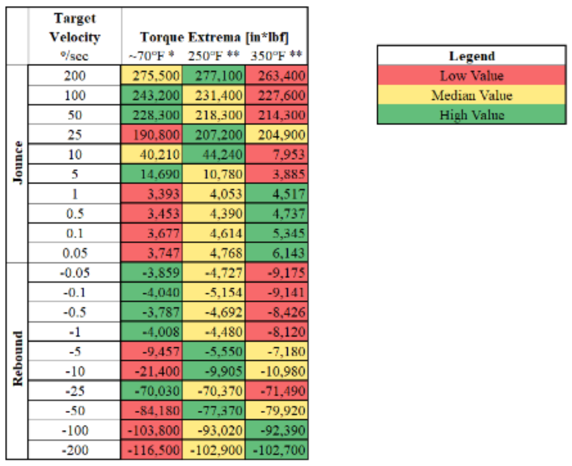
To calculate the reaction force from a damper curve, the angular velocity of the rotor must be known and derived from the raw time angle signals. Because the raw time angle signals from both laboratory and vehicle testing are noisy, they produce non-physical velocities. The method to establish angular velocity from time-angle signals in this paper is to gather subsets of data ranging from 10ms in the past to 10ms in future corresponding to each time step. Then a quadratic polynomial is fit to this vector of angle data and the analytical value of velocity at this time step is calculated from polynomial coefficients. The specific implementation of this algorithm in Python is given in Appendix 1.

3.3. Testing Limitations

Test stand limitations began to reduce the achieved angular amplitudes for 50 deg/sec and higher but the angular velocities during the constant velocity portions of the ramps still fell within ±10% of the intended rate.

Since these datasets achieved steady state reaction torques, the achieved minimum and maximum torque at each intended velocity were used to establish the damper curve. These values for the three thermal conditions are given in Table 3 and are shown graphically in Figure 9.

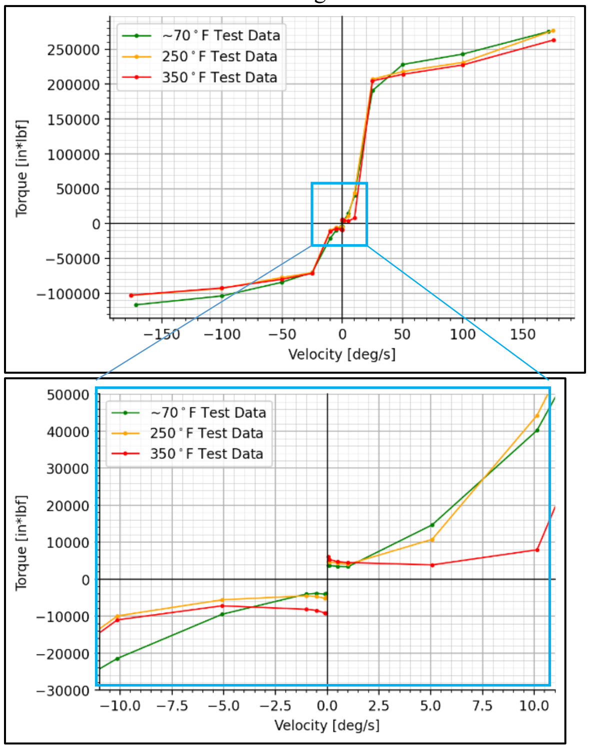
Table 3. Temperature Variation with Ramp Input



	Target Velocity °/sec	Torque Extrema [in*lbf] ~70°F *	250°F **	350°F **
Jounce	200	🟨 275,500	🟩 277,100	🟥 263,400
	100	🟩 243,200	🟨 231,400	🟥 227,600
	50	🟩 228,300	🟨 218,300	🟥 214,300
	25	🟥 190,800	🟩 207,200	🟨 204,900
	10	🟨 40,210	🟩 44,240	🟥 7,953
	5	🟩 14,690	🟨 10,780	🟥 3,885
	1	🟥 3,393	🟨 4,053	🟩 4,517
	0.5	🟥 3,453	🟨 4,390	🟩 4,737
	0.1	🟥 3,677	🟨 4,614	🟩 5,345
	0.05	🟥 3,747	🟨 4,768	🟩 6,143
Rebound	-0.05	🟩 -3,859	🟨 -4,727	🟥 -9,175
	-0.1	🟩 -4,040	🟨 -5,154	🟥 -9,141
	-0.5	🟩 -3,787	🟨 -4,692	🟥 -8,426
	-1	🟩 -4,008	🟨 -4,480	🟥 -8,120
	-5	🟥 -9,457	🟩 -5,550	🟨 -7,180
	-10	🟥 -21,400	🟩 -9,905	🟨 -10,980
	-25	🟩 -70,030	🟨 -70,370	🟥 -71,490
	-50	🟥 -84,180	🟩 -77,370	🟨 -79,920
	-100	🟥 -103,800	🟨 -93,020	🟩 -92,390
	-200	🟥 -116,500	🟨 -102,900	🟩 -102,700

Legend
🟥 Low Value
🟨 Median Value
🟩 High Value

Figure 9. Damper Curve Generated in Initial Bench Testing



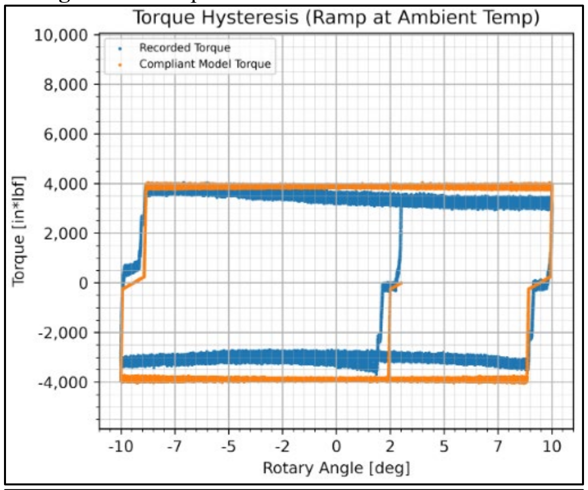
Comparing the results at the three different temperatures, velocity amplitudes 25 deg/sec and higher have very consistent reaction torques, differing at most by 5 %.

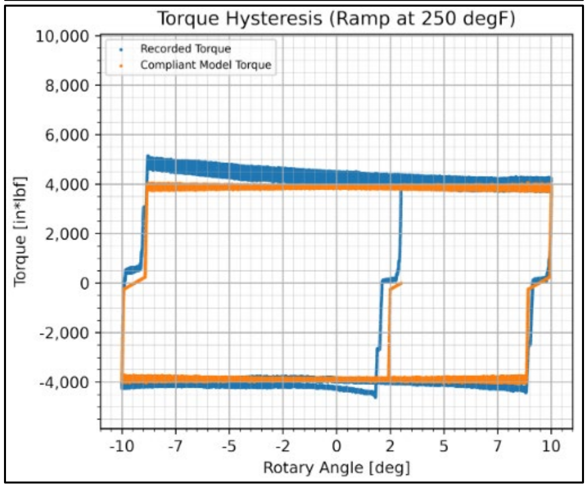
At intermediate velocities, ±5 and ±10 deg /sec, no clear trend exists, and massive differences are observed.

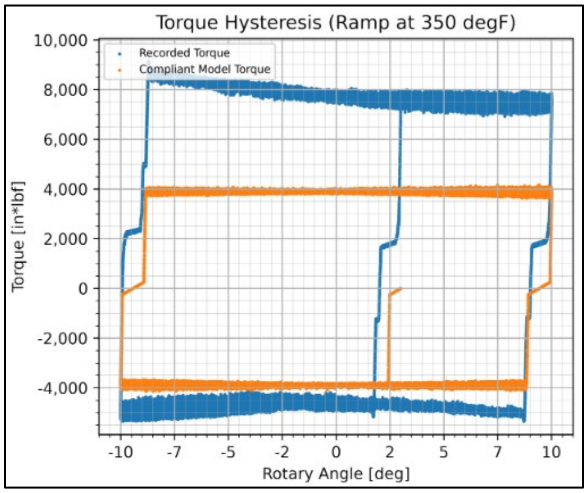
For very small velocities, 1 deg/sec and less, there is a general trend of higher resistance as temperatures grow but the trend is relatively weak.

Figure 10 shows an example of the 0.1 deg/sec condition where ambient and 250°F are similar but 350°F has substantially more resistance, in compression only.

Figure 10. Temperature Variation at Low Velocities







As will later be established, the large velocities correspond to when the pressure relief valves of the damper are active, consistent temperature independent performance is observed. However, the performance at lower velocity amplitudes does not have an obvious temperature dependence.

Temperature as not investigated any further in this study as a factor. Only the data for initial temperature of 250°F was used for development of the damper curve.

3.4. Refinement of Initial Testing

The initial data set provided poor refinement in the low velocity (±25 deg/s) region. Utilizing the same rotary damper and test setup as before, additional testing was conducted to fill in the gaps between existing points.

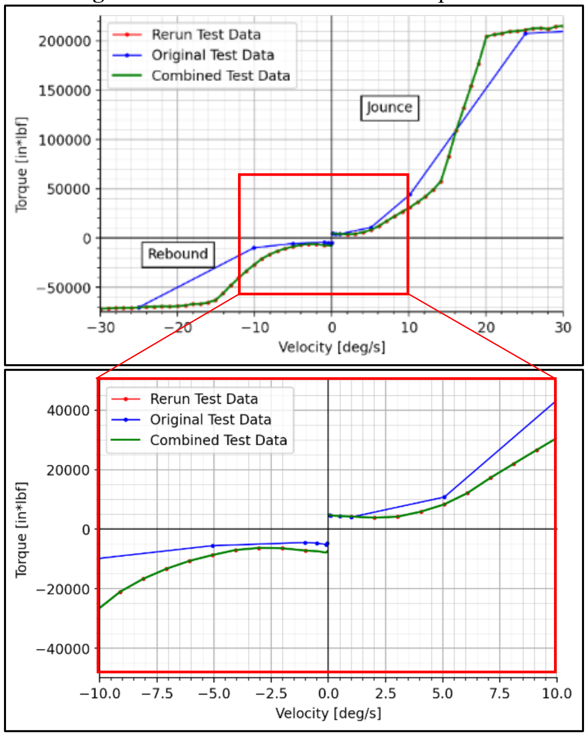
Table 4. Bench Re-Test Input Parameters

Input Function	**Target Velocity [deg/s]**	**Temperature [°F]**
Ramp	Range of [1,50] in 1 deg/s increments	250

This second set of testing for ±1 and ±5 deg/sec was within 22% of the first.

As apparent in Figure 11, significant differences were observed for the ±10 deg/sec speeds. These speeds in the first set of testing had massive jumps in reaction torques as temperatures varied.

Figure 11. Test and Retest Point Comparison

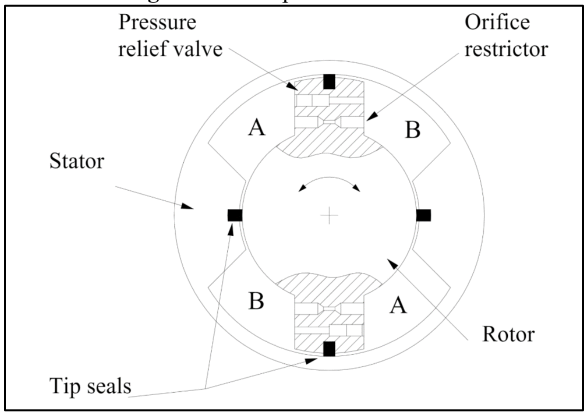


4. DAMPER CONFIGURATION AND MODELING MOTIVATION

4.1. Damper Construction

The general construction of dampers used in tracked vehicles are rotors with two lobes within a stator which creates two sets of oil cavities. This geometry is reflected in the functional diagram from [3] where two oil cavities "A" and "B" are shown as well as the additional features of orifice restrictors, tip seals, and pressure relief valves (PRVs).

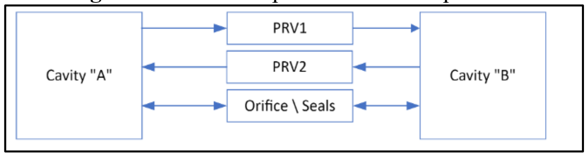
Figure 12. Damper Cross-Section



Note this diagram is a great simplification of the assembly but does contain all necessary elements for this discussion. Some non-essential additional components are seals in the axial direction, bearings to maintain the rotor position relative to the stator, and external oil makeup channels.

In typical damper implementations there is negligible flow past any of the seals within the damper to maintain consistent performance and have good seal durability. Thus, the only flow paths possible between cavities A and B are normally through the PRVs and orifices.

Figure 13. Oil Flow-paths within Damper



There are typically two sets of PRVs, a set for the extension direction and a differently configured set for the compression direction. These valves permit no flow until a threshold pressure is achieved. Once flow commences the restriction is typically linear with flow rate.

At low velocities the PRVs do not permit any flow, thus oil can only flow through the orifices. It has been shown that this flow restriction depends on the particular geometric details of the of the orifices. As with other internal flow problems the restriction would be expected to fall between a linear response due to internal laminar flow (viscous dominated) and due expansion and contraction losses or higher flow rates be quadratic. Some orifices like sharp edged ones the linear region may be imperceivable.

The design of the rotary damper tested in this paper is somewhat anomalous because there are no orifices to control low velocity damping. Instead, low velocity damping is "controlled" by leakage past the seals until sufficient pressure builds to trigger PRV flow.

It would be expected that initially the flow restriction past the seals should be quadratic like a sharp-edged orifice, and this response for low velocities was observed. At higher oil flow rates in the compression direction a characteristic change was observed which may be due to improved engagement of the seal. In the compression direction two distinct regions were observed: initially a quadratic response, then a high slope linear response.

Finally, the bearings and seals of the system would be expected to have a friction like behavior.

In summary, the friction and flow paths restrictions should be sufficient to capture the steady state response.

For transient performance two additional considerations are needed. First since the rotary damper is driven via a set of splines, some lash behavior is expected. Also, the oil and structure of the damper are not rigid, some compliance is expected.

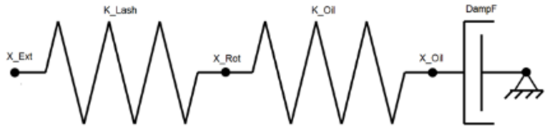
4.2. Steady state versus transient model

In steady state conditions, flow directly relates to angular velocity, however in transient conditions this is generally not the case.

Before deriving the updated damper model two necessary internal variables are introduced to separate the lash, compliance, and damper curve elements.

The variable X_{Ext} is the external "real world" angular position of the rotor. X_{Rot} is introduced to enable the deviation of the external angle from the actual rotor angle due to lash in the system. Lastly X_{Oil} enables incorporation of compliance.

Figure 14. Damper Compliance Model



At the limit of infinitely stiff k_{lash} and k_{oil} this model degenerates to the traditional approach where $X_{Ext} = X_{Oil}$.

4.3. Steady state model

This section deals specifically with creating the steady state model tuned to the data of Section 3 (Damper Curve Generation) which returns a reaction torque for a given angular velocity.

In this model the units are torque (in-lbf), angle (deg), and angular velocity (deg/sec). Upon introduction of effective radii, this model can be alternatively expressed as pressure, volume, and volumetric flow rates. These then have physical meanings that later could be used by engineers to explore damper designs. The primary characteristics that would be varied during design work are PRV cracking pressures and flow restrictions for high speeds as well as orifice coefficient and exponent for low speeds.

In the next section the terms pressure and flow are used interchangeably with torque and angular velocity in order to make the derivation easier to follow.

4.4. Steady state model flow regimes

This section details the mathematical method to replicate the damper curve response described in the previous section.

Following from Figure 13 there are two flow paths, one through the PRVs (v_{PRV}) and one through the orifice (v_{orif}). The total flow (v_{tot}) from cavity "A" to cavity "B" is the sum of these two parallel paths.

$$v_{tot} = v_{orif} + v_{PRV} \text{ [deg/sec]} \qquad (1)$$

The reaction torque of the damper occurs due to a difference in pressures between the two cavities acting over a loaded area of the two rotors. This delta pressure is identical for the two flow paths and its related measure, reaction torque (τ):

$$\tau = \tau_{orif} = \tau_{PRV} \text{ [in-lbf]} \qquad (2)$$

The friction behavior of the damper in this model is lumped into the orifice path restriction based on the observed response. Thus, the restriction of flow through the orifice path includes three terms: an always active quadratic restriction with coefficient A, friction that opposes motion with coefficient B, and for flows which exceed the critical level C an additional linear restriction D.

$$\tau = A\, sgn(v_{orif}) * v_{orif}^2 + B\, sgn(v_{orif}) + D * max(v_{orif} - C, 0) \text{ [in-lbf]} \quad (3)$$

The dimensionality of these parameters are as follows:

$$[A] = [\frac{in * lbf}{\left(\frac{deg}{s}\ of\ orifice\ flow\right)^2}]$$
$$[B] = [in * lbf]$$
$$[C] = [\frac{deg}{s} of\ orifice\ flow]$$
$$[D] = [\frac{in*lbf}{\left(\frac{deg}{s}\ of\ orifice\ flow\right)^2}]$$

Note quadratic, friction, and linear terms all are dissipative, that is they result in torques that oppose flow. Thus, the coefficients A, B, and D are negative. Also note in this model that the convention is taken that compressive flow is positive, so the value of C is positive.

For low flow rates there is no PRV flow ($v_{PRV} = 0$). Beyond certain critical levels there are two active flow paths. Let us find what these critical flow rates are, starting with the compressive flow direction.

As the compression direction flow builds, no flow through the PRV occurs until a critical value of orifice flow is met or exceeded. This critical compression flow rate is denoted v_{comp_crack} and occurs at a torque which is denoted τ_{comp_crack}. For flows beyond the cracking torque threshold, the flow resistance is taken to linearly increase with additional flow through the compression PRV since this matches historical norms:

$$\tau = \tau_{comp\_crack} + F_{comp} * v_{PRV} \quad (4)$$

Because the cracking torque τ_{comp_crack} is a practical real-world design parameter that is easily and commonly controlled but the cracking flow rate v_{comp_crack} is not, τ_{comp_crack} will be retained as a model parameter and v_{comp_crack} eliminated by determining its value from other parameters.

To this end, consider that at the critical compression flow rate $v_{PRV} = 0$, and all flow is through the orifice $v_{tot} = v_{orif} = v_{comp\_crack}$. For this condition, equating the reaction torque of equation 4 with the one of equation 3 reveals a quadratic dependence of v_{comp_crack} on τ_{comp_crack}:

$$\tau_{comp_{crack}} = A\, v_{comp_{crack}}^2 + D\, v_{comp_{crack}} + (B - DC) \quad (5)$$

Here it was used that compression flow is positive ($sgn(v_{orif}) = 1$) and proper values of parameters were assumed which gives the logical consequence that flow exceeds the critical level of C. Only one of the two solutions for v_{comp_crack} in equation 5 provides a flow exceeding C, revealing the critical value of compressive orifice flow:

$$v_{comp\_crack} = \frac{1}{2A} * \left(-D + \sqrt{D^2 - 4A(B - DC - \tau_{comp\_crack})}\right) \quad (6)$$

For larger flows, a torque balance

$$A * v_{orif}^2 + B + D * (v_{orif} - C) = \tau_{comp\_crack} + F_{comp} * (v_{total} - v_{orif}) \quad (7)$$

Gives the relative flow split between orifice and compression PRV:

$$v_{orif\_comp} = \frac{-(D + F_{comp}) + \sqrt{(D + F_{comp})^2 - 4A(B - \tau_{comp\_crack} - DC - F * v\_total}}{2A} \quad (8)$$

With similar reasoning for the extension direction, v_{ext_crack} and v_{orif_ext} is found explicitly from other model parameters.

$$v_{ext\_crack} = \frac{1}{2A}\left(-1 * \sqrt{-4A(B - \tau_{ext_{crack}})}\right) \quad (9)$$

$$v_{orif\_ext} = \frac{F_{ext} - \sqrt{F_{ext}{}^2 - 4A(B + \tau_{ext\_crack} + F_{ext} * v_{total})}}{2A} \quad (10)$$

4.5. Numerical Issue with Friction Term

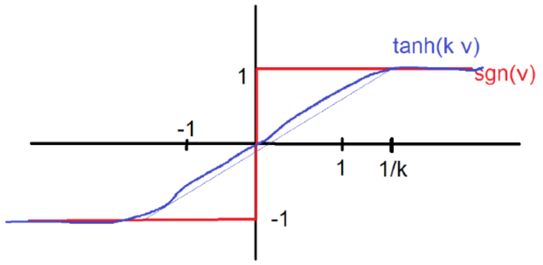
Friction modeling with a strict dependence on the sign of velocity is a classical numerical challenge, the "stick-slip" condition. With the model of the previous section this issue arose leading to large and non-physical instantaneous changes in velocity.

As is traditional this numerical difficulty was overcome by "softening" the response. Specifically, the approach to modify the term $sgn(v_{orif})$ in Equation 3 with a simple "smoothed" equivalent, hyperbolic tangent was done:

$$sgn(v_{orif}) \approx \tanh(v_{orif}/k)$$

The parameter k is responsible for the "softening". As shown in the following figure, the full response of $sgn(v_{orif}) = 1$ is achieved at approximately 1/k which is also approximately the slope of the initial response about zero.

Figure 15. Friction "Softening"



A study was conducted varying the parameter k. In the baseline limit where k tends to infinity severe numerical issues were encountered. The smallest value considered (k = .005 deg/s) resolved these numerical difficulties within the python simulation environment used and represents an absurdly slow condition. The realistic travel of this damper is approximately 60 degrees. With a rate of .005 deg/sec moving through the full travel of the suspension would take in excess of three hours.

This conclusion is not necessarily extensible to an MBD framework so higher values of k (softer response) may be appropriate elsewhere. Since the damper response for small velocities is not relevant from a design perspective it may actually be preferable to move to a even softer response like (k = .5 deg/s) where the full suspension travel would occur in two minutes.

4.6. Invertibility of Damper Curve

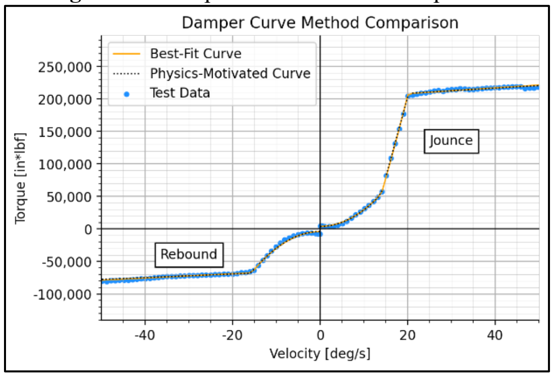
To implement the complaint model introduced in Section 6 a monotonic damper curve is desired such that the inversion is not ill defined. By inclusion of the softened response of Section 4.5 this condition is satisfied.

4.7. Damper Curve Comparison

The values of A, B, C, D, τ_{comp_crack}, F_{comp}, τ_{ext_crack}, and F_{ext} were determined utilizing the optimization library included [4]. The *minimize* function was used to reduce the average error of the approximation relative to the damper curve points. Constraints were added to ensure a monotonic damper curve was retained.

This procedure resulted in a model which very accurately represented the collected damper curve of section 3.

Figure 16. Damper Curve Method Comparison



5. DAMPER COMPLIANCE MODELING

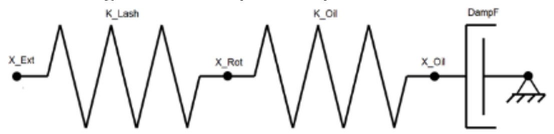
Despite a damper curve which reflects the steady state response of the damper with high accuracy, two critical characteristics are absent.

At low values of torque as the system changes direction a finite travel low stiffness region is observed. This “lash” within the system is independent of velocity. To add this to the model, a spring with a finite travel was added in line to the damper system, replicating the observed response.

Also, a delay in growth and decay of damping reaction torque as observed. This observation is expected to be related to compression and decompression of oil within the damper. This phenomena was modeled with a high-stiffness spring in series with the damper.

Incorporating both observations, a spring-damper system was developed to consider lash and oil compliance within the system. X_{Ext} is the input radial position of the rotor, X_{Rot} is the post-lash position, and X_{Oil} incorporates the oil compliance.

Figure 17. Damper Compliance Model



To calculate the reaction force of the damper, the time series of X_{Oil} must be calculated by integrating a series of functions over the time array:

- Given:
  - X_{Ext} is a position vector over time
  - ‘DampF’ is a function of velocity
  - ‘DampV’ the inverse of ‘DampF’
  - K_{Lash} has a low spring rate
  - K_{Oil} has a high spring rate
- Calculate candidate for X_{Rot} via nodal analysis:

$$x_{RotCan} = \frac{k_{Oil} * x_{Oil} + k_{Lash} * x_{Ext}}{k_{Lash} * k_{Oil}} \quad (11)$$

- X_{Rot} is limited in travel by the lash limit:

$$x_{Rot} = \begin{bmatrix} x_{RotCan} - x_{Ext} < -x_{LashLim} & x_{Ext} - x_{LashLim} \\ -x_{LashLim} < x_{RotCan} - x_{Ext} < x_{LashLim} & x_{RotCan} \\ x_{LashLim} < x_{RotCan} - x_{Ext} & x_{Ext} + x_{LashLim} \end{bmatrix} \quad (12)$$

- The velocity of X_{Oil} is calculated and integrated:

$$x_{Oil} = \int dampV[k_{Oil} * (x_{Oil} x_{Rot})]\, dt \quad (13)$$

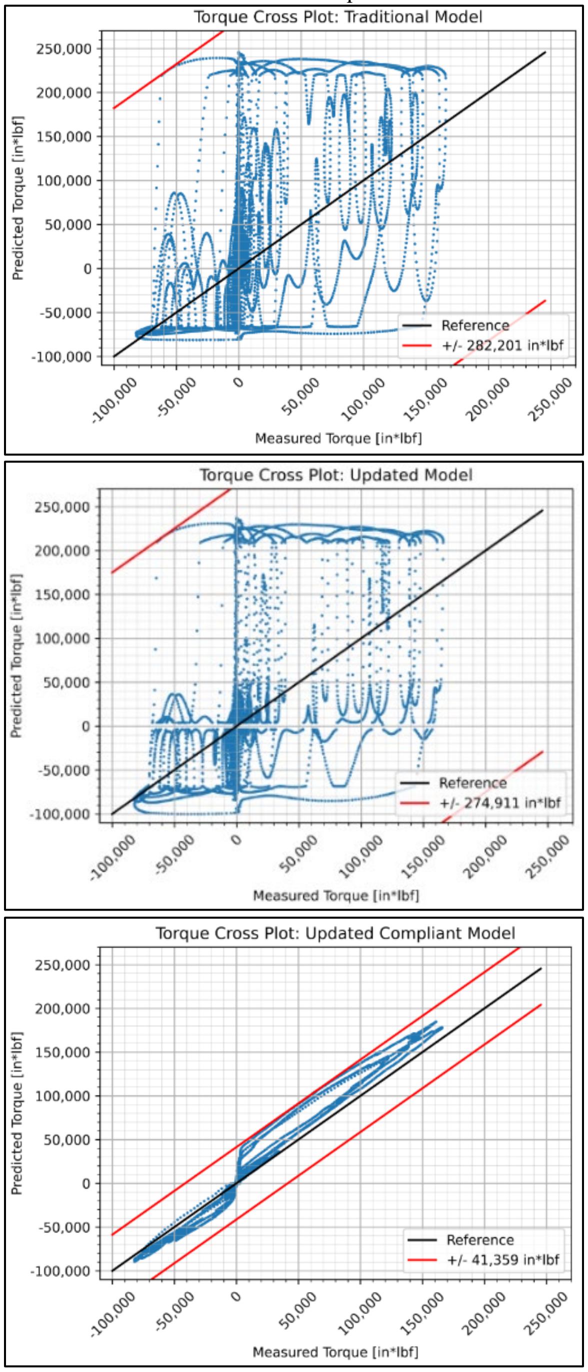
- Using the X_{Oil} previously calculated, X_{Rot} is recalculated and limited (Equations 12,13). The torque on the damper is then calculated:

$$Reaction\,Torque = dampV[k_{Oil} * (x_{Oil} x_{Rot})] \quad (14)$$

5.1. Model Results

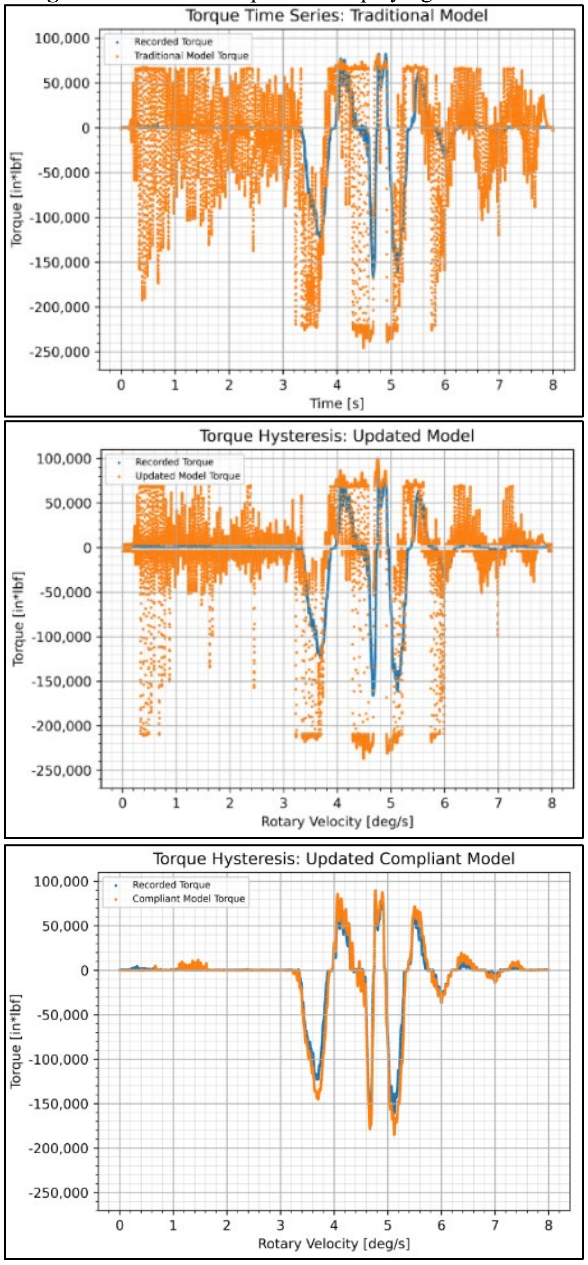
The updated compliant model provides a vast improvement in the modeling of the rotary dampers. By implementing oil and lash compliant factors, the updated compliant model correlation error on half round testing can be reduced to 16% of traditional modeling error: an average error band of ±271,000 in*lbf down to an average of ±45,000 in*lbf. Figure 18 displays an example of the reduction of correlation error on cross-plots for all three modeling methods.

Figure 18. Model Comparison Detailing Correlation to Measured Torque



The time series response shows a vast improvement, as seen in Figure 19.

Figure 19. Model Comparison Displaying Time Series



6. Conclusion

The *updated compliant model* provides a very good prediction but has a slight over prediction of reaction torque magnitudes. By introducing "compliance" and "lash", the damper performance can be accurately represented. With the slight extension of this model to a fully physics based one, future dampers can be designed to expand the operating performance envelope of tracked vehicles.

7. REFERENCES

[1] “Test Operations Procedure (TOP) 1-1-014 Ride Dynamics”, Aberdeen Test Center MD Commander of Development and Test Command, 2005.

[2] S. Allen, D. Ostberg, “Laboratory Testing of Tracked Vehicle Suspensions”, In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 10-12, 2021.

[3] P. Allen, “Models for Dynamic Simulation of Tank Track Components,” PhD Thesis, Def. College of Mgmt. and Tech., Cranfield Univ., 2006.

[4] P. Virtanen *et al.*, “SciPy 1.0: Fundamental algorithms for scientific computing in python,” *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

8. APPENDIXES

Appendix I. Python Code: Derivative Method from Tim Hunter at Wolf Star Tech, Modified by Kenneth Redner for use in Rotary Damper Curve Generation

```
import numpy as np
def tfuDiff(xyData, deriv = 1, smoothPts=20, polyOrder = 3, yTol=0.0, symmetryFlag=True, minPts=1):
    """
    tfuDiff calculates the derivative of the given functions by approximating the function as polynomial over the
    user specified segments. The derivative may be first or second order derivatives.
    Parameters:
    ----------
        tfuDict: Dictionary of xyData pairs to be processed
        tfuOrder: List of keys in the dictionary of xyData pairs to be processed
        deriv: The order of the derivative. Allowable values are 1 or 2
        smoothPts: Number of points to be used in the polynomial fitting
        polyOrder: The order of the polynomial to be fit
    Returns:
    --------
        diffDict, diffOrder - xyData dictionary and keys containing derivatives and the approximated curves

    Usage:
    ------
        diffDict, diffOrder = tfuDiff(tfuDict, tfuOrder, deriv=1, smoothPts=20, polyOrder=3)

    """
    if polyOrder > smoothPts:
        polyOrder = smoothPts
    #end if

    diffDict = {}
    diffOrder = []

    x, y = zip(*xyData)
    x = np.array(x)
    y = np.array(y)
    nPts = len(x)
    yDots = []
    y2Dots = []
    yApproxs = []
    for i in range(nPts):
        iStart = i - smoothPts
        if iStart < 0:
            iStart = 0
        # end if
        iEnd = i + smoothPts
        if iEnd >= nPts:
            iEnd = nPts - 1
        # end if
        if yTol != 0:
            trgtVal = y[i]
            for iTest in range(i, iStart, -1):
                testVal = y[iTest]
                if np.abs(testVal - trgtVal) > yTol:
                    iStart = iTest
                    break
                #end if
            #end iTest
            for iTest in range(i, iEnd):
                testVal = y[iTest]
                if np.abs(testVal - trgtVal) > yTol:
                    iEnd = iTest
                    break
                #end if
            #end iTest
            if symmetryFlag:
                backDiff = i - iStart
                forwardDiff = iEnd - i
                if backDiff != forwardDiff:
                    minDiff = min(backDiff, forwardDiff)
                    iDistStart = i
                    iDistEnd = nPts - i
                    minDist = min(iDistStart, iDistEnd)
                    if minDiff > minDist:
                        minDiff = minDist
                    #end if
                    if minDiff == 0:
                        minDiff = max(backDiff, forwardDiff, minDist)
                    #end if
                    iStart = i - minDiff
                    iEnd = i + minDiff
                    if iStart < 0: iStart = 0
                    if iEnd > nPts: iEnd = nPts
                #end if
            #end if
            if i - iStart < minPts:
                iStart = i - minPts
            #end if
            if iEnd - i < minPts:
                iEnd = i + minPts
            #end if
            if iStart < 0: iStart = 0
            if iEnd > nPts: iEnd = nPts
        #end if

        xSeg = x[iStart:iEnd]
        xSeg = np.array(xSeg)
        ySeg = y[iStart:iEnd]
        ySeg = np.array(ySeg)
        # xx = [xSeg**0, xSeg, xSeg**2, xSeg**3, xSeg**4]  # 5 x 20
        xx = []
        for iOrder in range(polyOrder):
            xx.append(xSeg**iOrder)
        #end iOrder
        xx = np.array(xx)
        xx = xx.T                                    # 20 x 5
        #
        #  x * c = y
        #  x^T * x * c = x^T * y
        #  [x^T * x]^-1 * [x^T * x] * c = [x^T * x]^-1 * x^T * y
        #  c = [x^T * x]^-1 * x^T * y
        #  c = [  x^T     * x]^-1  *     x^T    *     y
        #  c = [[5 x20] [20 x 5]]  *   [5 x 20]  * [20 x 1]
        #  c =      [5 x 5]       *   [5 x 20]  * [20 x 1]
        #  c =                [5 x 20]        * [20 x 1]
        #  c =                               [5 x 1]
        #
        #
        xTemp = np.dot(xx.T, xx)        # 5 x 5
        try:
            xTemp = np.linalg.inv(xTemp)     # 5 x 5
        except:
            # display(xTemp)
            break
        xTemp = np.dot(xTemp, xx.T)      # 5 x 20
        c = np.dot(xTemp, ySeg)         # 5 x 1
        xCur = x[i]
        yApprox = c[0]
        for iOrder in range(1, polyOrder):
            yApprox = yApprox + c[iOrder] * xCur ** iOrder
        # end iOrder
        yApproxs.append(yApprox)
        if deriv == 1:
            # y       = c[0]      +     c[1] * x**1 +      c[2] * x**2 +     c[3] * x**3 + c[4] * x**4
            # dy/dt   = c[1]      + 2 * c[2] * x**1 +  3 * c[3] * x**2 + 4 * c[4] * x**3
            # d2y/dt2 = 2 * c[2]  + 6 * c[3] * x**1 + 12 * c[4] * x**2
            # yApprox = c[0] +   c[1]*xCur +   c[2]*xCur**2 +   c[3]*xCur**3 + c[4]*xCur**4
            # yDot =    c[1] + 2*c[2]*xCur + 3*c[3]*xCur**2 + 4*c[4]*xCur**3
            # y2Dot =   2 * c[2] + 2 * 3 * c[3] * xCur + 3 * 4 * c[4] * xCur**2

            yDot = c[1]
            for iOrder in range(2, polyOrder):
                yDot = yDot + iOrder * c[iOrder] * xCur ** (iOrder-1)
            #end iOrder
            yDots.append(yDot)
        elif deriv == 2:
            if polyOrder == 2:
                print('2nd derivative does not exist for y=C0x^0+C1*x^1')
                return {}, []
            #end if
            # y2Dot = 2*c[2] + 6*c[3]*xCur + 12*c[4]*xCur**2
            y2Dot = 2*c[2]
            for iOrder in range(3, polyOrder):
                y2Dot = (iOrder-1) * iOrder * xCur ** (iOrder - 2)
            #end iOrder
            y2Dots.append(y2Dot)
        # end if
    # end i
    #
    # Store the Approximated curve
    #

    yApproxs = np.array(yApproxs)
    xyData = list(zip(x, yApproxs))
    if deriv == 1:
        yDots = np.array(yDots)
        xyData = list(zip(x, yDots))

    elif deriv == 2:
        y2Dots = np.array(y2Dots)
        xyData = list(zip(x, y2Dots))

# end funcName

    return xyData
```

Appendix II. Python Code: Generate Test Points and calculate Best-Fit damper curve coefficients to establish damping function.

```
### Point Generation
from scipy.signal import sosfiltfilt, butter
import matplotlib.pyplot as plt
import numpy as np

def butter_filt(input_signal,sampleRate=2000,filt_freq=400,filt_order=2):
   # sampleRate = 2000   # Hz imported data sample rate, could be calculated instead
   # filt_freq = 400     # Hz for butterworth filter applied to data
   # filt_order = 2      # Applied forward and backward so actually 2x this value
   sos = butter(filt_order, filt_freq, fs = sampleRate, output='sos')
   output_signal = sosfiltfilt(sos,input_signal)
   return(output_signal)

speeds_sorted={
   'Original': ['gas','0p1','0p5','001','005','010','025',
              '050','100','200','300','400','500'],
   'Rerun'   : [str(i).zfill(3) for i in range(1,51)],
   'All'     : ['gas','0p1','0p5']}
speeds_sorted['All'].extend([str(i).zfill(3) for i in range(1,51)])
speeds_sorted['All'].extend(['100','200','300','400','500'])

def make_damp_array(speeds_sorted,run_data,dataset):
   working_df_dict = [f'Ramp_250_{i}' for i in speeds_sorted[dataset]]
   damp_array = []
   damp_array_disp = []
   vel_list = []
   for fileindex in list(working_df_dict):

      data = run_data[dataset][fileindex]
      time_in = np.array(data['Time [s]'])
      ang_in = np.array(data['Rotary Angle [deg]'])
      ang_filt = butter_filt(ang_in)
      torq_in = np.array(data['Rotary Torque [ft-lbf]']) * 12
      target_vel = fileindex[-3:].replace('p','.').replace('gas','.05')

      xyData = list(zip(time_in, ang_filt))
      xyData = tfuDiff(xyData, deriv = 1, smoothPts=20, polyOrder = 3,
                  yTol=0.0, symmetryFlag=True, minPts=1)
      dtime_in, der_vel = zip(*xyData)
      der_vel_abs = [abs(i) for i in der_vel]
      mean_vel = np.mean([i for i in der_vel_abs if i < 1.2*float(target_vel)
                   and i > 0.8*float(target_vel)])

      damp_array.append([mean_vel,-1*min(torq_in),-1*max(torq_in)])
      damp_array_disp.append(['{:.3E}'.format(mean_vel),
                       '{:.3E}'.format(min(-1*torq_in)),
                       '{:.3E}'.format(max(-1*torq_in))])

      vel_list.append(target_vel)
   damp_array = np.transpose(damp_array)
   return damp_array

damp_array = {}
for dataset in list(run_data):
   damp_array[dataset] = make_damp_array(speeds_sorted,run_data,dataset)
del speeds_sorted,run_data,dataset

#adjust points w/ -1<Vel<0 to match curve
damp_array['All'][2][0:3] += damp_array['Rerun'][2][0]-damp_array['Original'][2][3]
import scipy.optimize as spo

def linear_approx(vel_lookup,tq_lookup):
   class opt_curve:
      def __init__(self,vel_lookup,tq_lookup):
         self.vel_lookup = vel_lookup
         self.tq_lookup = tq_lookup

      #Function To minimize
      def f(self, C):
         error = np.zeros(len(self.vel_lookup))
         for (i,vel) in enumerate(self.vel_lookup):
            # Error = Ax + B - y
            error[i] = ((C[0]*vel + C[1] - self.tq_lookup[i])/self.tq_lookup[i])**2
         error_sum = sum(error)*10000                                   #Error Scaling necessary
         return error_sum

   opt_curve1=opt_curve(vel_lookup,tq_lookup)

   #First Guess
   C_start = [1,np.sign(vel_lookup[0])]

   #Constraints
   cons = ({'type': 'ineq', 'fun': lambda C: C[0]}) #B must be non-negative

   #Optimization
   result = spo.minimize(opt_curve1.f,C_start,constraints=cons)

   C_dampV = result.x
   R2_dampV = result.fun
   print('-'*100)
   # print(result.message)
   print('Optimized Linear for Damper Curve:')
   print(f'\t y = ({round(C_dampV[0],4)})x + ({round(C_dampV[1],4)})')
   print(f'\t R^2: {R2_dampV}')

   return(C_dampV)

def quadratic_approx(vel_lookup_neg,tq_lookup_neg,vel_lookup_pos,tq_lookup_pos):
   class opt_curve:
      def __init__(self,vel_lookup_pos,tq_lookup_pos,vel_lookup_neg,tq_lookup_neg):
         self.vel_lookup_pos = vel_lookup_pos
         self.tq_lookup_pos = tq_lookup_pos
         self.vel_lookup_neg = vel_lookup_neg
         self.tq_lookup_neg = tq_lookup_neg

      #Function To minimize
      def f(self, C):
         error = 0
         count = 0
```

```
            for (i,vel) in enumerate(self.vel_lookup_pos):
                # Error = Ax^2 + Bx+ C - y
                error += ((C[0]*pow(vel,2) + C[1]*vel + C[2] - self.tq_lookup_pos[i]))**2
                count += 1
            for (i,vel) in enumerate(self.vel_lookup_neg):
                # Error = Ax^2 + Bx+ C - y
                error += ((-C[0]*pow(vel,2) - C[1]*vel - C[2] - self.tq_lookup_neg[i]))**2
                count += 1
            error_sum = error/count                                    #Error Scaling necessary
            return error_sum

    opt_curve1=opt_curve(vel_lookup_pos,tq_lookup_pos,vel_lookup_neg,tq_lookup_neg)

    #First Guess
    C_start = [1,1,1]

    #Constraints
    # cons = ({'type': 'ineq', 'fun': lambda C: (-C[1])/(2*C[0])}) #-b/2A must be non-negative
    cons = ({'type': 'ineq', 'fun': lambda C: C[0]},
            {'type': 'ineq', 'fun': lambda C: C[2]}) #A,C must be non-negative

    #Optimization
    result = spo.minimize(opt_curve1.f,C_start,constraints=cons)
    # result = spo.minimize(opt_curve1.f,C_start)
    C_dampVpos = [0,0,0]
    C_dampVpos[0] = -1*result.x[0]
    C_dampVpos[2] = -1*result.x[2]

    R2_dampV = result.fun

    print('-'*100)
    print(result.message)
    print('Optimized Quadratics for Damper Curve:')
    print(f'\t y = ({round(C_dampVpos[0],4)})x^2 + ({round(C_dampVpos[1],4)})x + ({round(C_dampVpos[2],4)})')
    print(f'\t R^2: {R2_dampV}')
    return(C_dampVpos)

def Quad_Lin_intercept(C_lin,C_quad):
    class functions:
        def __init__(self,C_lin,C_quad):
            self.C_lin = C_lin
            self.C_quad = C_quad
        def f(self,z):
            x,y = z
            f_lin = self.C_lin[0]*x + self.C_lin[1] - y
            f_quad = self.C_quad[0]*pow(x,2) + self.C_quad[1]*x + self.C_quad[2]-y
            return(f_lin,f_quad)
    functions1 = functions(C_lin,C_quad)

    # #First Guess
    guess = [25*np.sign(C_quad[0]),50000*np.sign(C_quad[0])]
    # #Solve
    intercept = spo.fsolve(functions1.f,guess,)
    print('-'*100)
    print(f'Linear/Quadradtic Intercept: {intercept}')
    return intercept

def Lin_Lin_intercept(C_lin,C_lin2):
    class functions:
        def __init__(self,C_lin,C_lin2):
            self.C_lin = C_lin
            self.C_lin2 = C_lin2
        def f(self,z):
            x,y = z
            f_lin = self.C_lin[0]*x + self.C_lin[1] - y
            f_lin2 = self.C_lin2[0]*x + self.C_lin2[1] - y
            return(f_lin,f_lin2)
    functions1 = functions(C_lin,C_lin2)

    # #First Guess
    guess = [C_lin[0],C_lin[1]]

    # #Solve
    intercept = spo.fsolve(functions1.f,guess)
    print('-'*100)
    print(f'Linear/Linear Intercept: {intercept}')
    return intercept

# Rebound = R, Jounce = J
lookup_vel = {
    'R-Valve':           [-1*i for i in damp_array['All'][0][17:55]],      #damp_array['All']
    # 'R-Leakage, Seal':    [-1*i for i in damp_array['All'][0][17:17]],
    'R-Leakage, No Seal': [-1*i for i in damp_array['All'][0][:17]],
    'J-Leakage, No Seal': damp_array['All'][0][:17],
    'J-Leakage, Seal':    damp_array['All'][0][17:23],
    'J-Valve':           damp_array['All'][0][23:55]}
lookup_tq = {
    'R-Valve':           damp_array['All'][2][17:55],
    # 'R-Leakage, Seal':    damp_array['All'][2][17:17],
    'R-Leakage, No Seal': damp_array['All'][2][:17],
    'J-Leakage, No Seal': damp_array['All'][1][:17],
    'J-Leakage, Seal':    damp_array['All'][1][17:23],
    'J-Valve':           damp_array['All'][1][23:55]}

lookup_coeff = {
    'R-Valve':           linear_approx(lookup_vel['R-Valve'],
                                       lookup_tq['R-Valve']),
    # 'R-Leakage, Seal':    linear_approx(lookup_vel['R-Leakage, Seal'],
    #                                      lookup_tq['R-Leakage, Seal']),
    'R-Leakage, No Seal': quadratic_approx(lookup_vel['R-Leakage, No Seal'],
                                         lookup_tq['R-Leakage, No Seal'],
                                         lookup_vel['J-Leakage, No Seal'],
                                         lookup_tq['J-Leakage, No Seal']),
    'NegV Fix': [],
    'PosV Fix': [],
    'J-Leakage, No Seal': [],
    'J-Leakage, Seal':    linear_approx(lookup_vel['J-Leakage, Seal'],
                                       lookup_tq['J-Leakage, Seal']),
    'J-Valve':           linear_approx(lookup_vel['J-Valve'],
                                       lookup_tq['J-Valve'])}
lookup_coeff['J-Leakage, No Seal'] = [-1*i for i in lookup_coeff['R-Leakage, No Seal']]
```

```
fixpoint = 0.005
lookup_intercept = {
    'R-Valve_R-Leakage, No Seal':           Quad_Lin_intercept(lookup_coeff['R-Valve'],lookup_coeff['R-Leakage, No Seal']),
    'R-Leakage, No Seal_NegV Fix':[-fixpoint,lookup_coeff['R-Leakage, No Seal'][0]*pow(-fixpoint,2) + lookup_coeff['R-Leakage, No Seal'][1]*(-fixpoint) + lookup_coeff['R-Leakage, No Seal'][2]],
    'PosV Fix_J-Leakage, No Seal':[fixpoint,lookup_coeff['J-Leakage, No Seal'][0]*pow(fixpoint,2) + lookup_coeff['J-Leakage, No Seal'][1]*(fixpoint) + lookup_coeff['J-Leakage, No Seal'][2]],
    'J-Leakage, No Seal_J-Leakage, Seal':   Quad_Lin_intercept(lookup_coeff['J-Leakage, Seal'],lookup_coeff['J-Leakage, No Seal']),
    'J-Leakage, Seal_J-Valve':             Lin_Lin_intercept(lookup_coeff['J-Leakage, Seal'],lookup_coeff['J-Valve'])}
del fixpoint
lookup_coeff['NegV Fix'] = lookup_intercept['R-Leakage, No Seal_NegV Fix'][1]/lookup_intercept['R-Leakage, No Seal_NegV Fix'][0]
lookup_coeff['PosV Fix'] = lookup_intercept['PosV Fix_J-Leakage, No Seal'][1]/lookup_intercept['PosV Fix_J-Leakage, No Seal'][0]

# print(lookup_coeff['NegV Fix'])
# print(lookup_coeff['PosV Fix'])

# declare damping function "dampF"
def dampF(vel_in):
    f_out = np.piecewise(vel_in,
                        [(vel_in<=lookup_intercept['R-Valve_R-Leakage, No Seal'][0]),
                        ((lookup_intercept['R-Valve_R-Leakage, No Seal'][0] < vel_in)&(vel_in < lookup_intercept['R-Leakage, No Seal_NegV Fix'][0])),
                        (lookup_intercept['R-Leakage, No Seal_NegV Fix'][0]<=vel_in) & (vel_in<=0),
                        (0<vel_in) & (vel_in<=lookup_intercept['PosV Fix_J-Leakage, No Seal'][0]),
                        ((lookup_intercept['PosV Fix_J-Leakage, No Seal'][0] < vel_in) & (vel_in < lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][0])),
                        ((lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][0] < vel_in) & (vel_in < lookup_intercept['J-Leakage, Seal_J-Valve'][0])),
                        (lookup_intercept['J-Leakage, Seal_J-Valve'][0] <= vel_in)],

                        [lambda vel: lookup_coeff['R-Valve'][0]*vel + lookup_coeff['R-Valve'][1],
                         lambda vel: lookup_coeff['R-Leakage, No Seal'][0]*pow(vel,2) + lookup_coeff['R-Leakage, No Seal'][1]*vel + lookup_coeff['R-Leakage, No Seal'][2],
                         lambda vel: lookup_coeff['NegV Fix']*vel,
                         lambda vel: lookup_coeff['PosV Fix']*vel,
                         lambda vel: lookup_coeff['J-Leakage, No Seal'][0]*pow(vel,2) + lookup_coeff['J-Leakage, No Seal'][1]*vel + lookup_coeff['J-Leakage, No Seal'][2],
                         lambda vel: lookup_coeff['J-Leakage, Seal'][0]*vel + lookup_coeff['J-Leakage, Seal'][1],
                         lambda vel: lookup_coeff['J-Valve'][0]*vel + lookup_coeff['J-Valve'][1]])
    return(f_out)
```

Appendix III. Python Code: Physics Motivated Model and Optimization Function

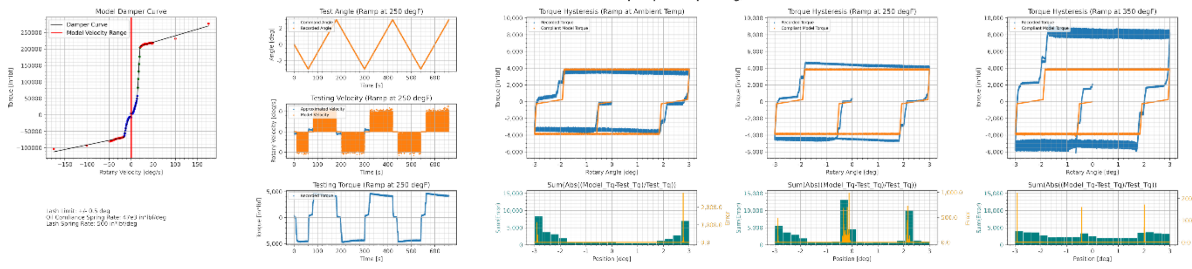
```python
def reacF_physics(vel_in,
    A = 257,
    B = 3999,
    C = 14,
    D = 16391,
    J = 208318,
    F = 448,
    K = -67386,
    H = 346
    ):
    from math import sqrt
    leakage = {
        'Quad' : A, #Quadratic Coefficient, A
        'Fric' : B, #Friction Coefficient, B
        'Thrsh' : C, #Higher Leakage resistance threshould, C
        'Slope' : D #Higher Leakage resistance slope, D
    }
    valve = {
        'C Tq' : J, #Cracking Tq, J
        'C Vel Rt' : [], #Cracking Vel Rate, E
        'C Fl Cr': F, #Flow Characteristic, F
        'E Tq' : K, #Cracking Tq, K
        'E Vel Rt' : [], #Cracking vel rate, G
        'E Fl Cr': H, #Flow Characteristic, H
        'Area': 13.84, #Loaded Area
        'C of P': 4.24 #Center of Pressure (radius)
    }

    valve['C Vel Rt'] = (-leakage['Slope'] + sqrt(leakage['Slope']**2-4*
        leakage['Quad']*(leakage['Fric']-valve['C Tq']-leakage['Slope']*
        leakage['Thrsh'])))/(2*leakage['Quad'])
    valve['E Vel Rt'] = (-1*sqrt(-1*(valve['E Tq']+leakage['Fric'])/(leakage['Quad'])))
    valve['C Pressure'] = valve['C Tq']/valve['C of P']/valve['Area']
    valve['E Pressure'] = valve['E Tq']/valve['C of P']/valve['Area']
    curve = []
    vol_leakage = []
    reacF = []
    valve_flow = []

    for [i,vel] in enumerate(vel_in):
        if vel < valve['E Vel Rt']:
            curve.append('hi_ext')
        elif vel < leakage['Thrsh']:
            curve.append('lo_sp')
        elif vel < valve['C Vel Rt']:
            curve.append('med_comp')
        else:
            curve.append('hi_comp')

        if curve[i]=='lo_sp' or curve[i]=='med_comp':
            vol_leakage.append(vel)
        elif curve[i]=='hi_comp':
            vol_leakage.append((-(leakage['Slope']+valve['C Fl Cr'])+sqrt(((
                leakage['Slope']+valve['C Fl Cr'])**2)-4*leakage['Quad']*(
                leakage['Fric']-valve['C Tq']-leakage['Slope']*leakage['Thrsh']-
                valve['C Fl Cr']*vel)))/(2*leakage['Quad']))
        elif curve[i]=='hi_ext':
            # (H_-SQRT(H_^2-4*A_*(B_+K_+H_*V4)))/(2*A_)
            vol_leakage.append((valve['E Fl Cr']-sqrt((valve['E Fl Cr']**2)-4*
                leakage['Quad']*(leakage['Fric']+valve['E Tq']+valve['E Fl Cr']*
                vel)))/(2*leakage['Quad']))
        valve_flow.append(vel-vol_leakage[i])
        reacF.append(leakage['Quad']*np.sign(vol_leakage[i])*(vol_leakage[i]**2) +
                    leakage['Fric']*np.sign(vol_leakage[i]) + leakage['Slope']*
                    max(vol_leakage[i]-leakage['Thrsh'],0))
    func_outputs = {
        'Vol Leakage': vol_leakage,
        'Vol Valve': valve_flow,
        'Tq Reac': reacF,
        'Curve': curve
    }
    return func_outputs
def P_model_opt(vel_lookup,tq_lookup):
    class opt_func:
        def __init__(self,vel_lookup,tq_lookup):
            self.vel_lookup = vel_lookup
            self.tq_lookup = tq_lookup

        def f(self,params):
            error = 0
            count = 0
            for (i,vel) in enumerate(self.vel_lookup):
                error += abs(reacF_physics([vel],
                    A=params[0],
                    B=params[1],
                    # C=params[2],
                    D=params[3],
                    J=params[4],
                    F=params[5],
                    K=params[6],
                    H=params[7],
                    )['Tq Reac']-self.tq_lookup[i])
                count += 1
            error_sum = error/count
            return error_sum
    opt_func1 = opt_func(vel_lookup,tq_lookup)

    param_start =[255,3800,14,16000,205000,400,-66000,400]
    # param_start =[255,3800,14,16000]

    cons = {
            'type':'ineq','fun': lambda params: params[3]**2-4*params[0]*(
                params[1]-params[4]-params[3]*params[2]),
            'type':'ineq','fun':lambda params:-1*(params[4]+params[1])/(params[0]),
            'type':'ineq','fun':lambda params: params[0],
            'type':'ineq','fun':lambda params: params[1],
            'type':'ineq','fun':lambda params: params[2],
            'type':'ineq','fun':lambda params: params[3],
            'type':'ineq','fun':lambda params: params[4],
            'type':'ineq','fun':lambda params: params[5],
            'type':'ineq','fun':lambda params: -1*params[6],
            'type':'ineq','fun':lambda params: params[7],
            }
    result = spo.minimize(opt_func1.f,param_start,constraints=cons)
    display(f'A = {result.x[0]}')
    display(f'B = {result.x[1]}')
    display(f'C = {result.x[2]}')
    display(f'D = {result.x[3]}')
    display(f'E = {result.x[4]}')
    display(f'F = {result.x[5]}')
    display(f'G = {result.x[6]}')
    display(f'H = {result.x[7]}')
    display(result.message)
    display(f'R^2: {result.fun}')

vel_lookup = [-i for i in damp_array['All'][0][:-3]]
vel_lookup.extend([i for i in damp_array['All'][0][:-3]])
tq_lookup = [i for i in damp_array['All'][2][:-3]]
tq_lookup.extend([i for i in damp_array['All'][1][:-3]])
P_model_opt(vel_lookup,tq_lookup)
```

Appendix IV. Python Code: Damper Updated Compliant Model

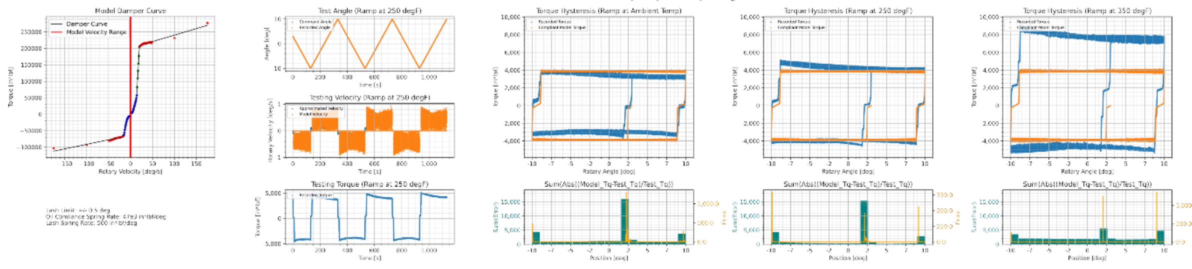
```
def get_reacF_from_ang(time,ang,inv_vel=False,inv_tq=False):
    """This function imports necessary libraries, as well as runs the ODE to find the calculated torque.
    You may need to invert `reactionF` depending on the orientation of
    the sensor."""
    import numpy  as np
    from re import X
    from scipy.integrate import odeint
    from scipy.interpolate import interp1d
    class calc_dampf:
        def __init__(self,time_filt,ang_filt,inv_vel,inv_tq):
            self.x_lashLim = 0.5
            self.k_oil = 47e3
            self.k_lash = 500

            self.time_filt = time_filt
            self.ang_filt = ang_filt

            self.lookup_coeff = {
                'R-Valve': [275.782,-63748.0],
                'R-Leakage, No Seal':[-255.259,0,-3837.05],
                'NegV Fix': [767412],
                'PosV Fix': [767412],
                'J-Leakage, No Seal': [255.259,0,3837.05],
                'J-Leakage, Seal': [24529.1,-287731],
                'J-Valve': [395.477,200814]
            }
            self.lookup_intercept = {
                'R-Valve_R-Leakage, No Seal': [-15.8699,-68124.6],
                'R-Leakage, No Seal_NegV Fix':[-0.005,-3837.06],
                'PosV Fix_J-Leakage, No Seal':[0.005,3837.06],
                'J-Leakage, No Seal_J-Leakage, Seal':[13.8961,53127.9],
                'J-Leakage, Seal_J-Valve':[20.2433,208820]
            }

            if inv_vel: self.inv_vel = 1
            else: self.inv_vel = -1

            if inv_tq: self.inv_tq = -1
            else: self.inv_tq = 1

        def vel_k_damper(self,x_oil, time_val):
            """
            INPUTS k12, k01, x2, x0, time_val
            CALCULATED x1, x.2
            RETURN x.2

            Node x1 is point between finite travel spring and external angle x0
                It is connected to external angle by a low spring rate finite travel
                k12 * (x2 - x1) = k01 * (x1 - x0)   (Force blance at node x1)
                We know x0, x2 comes from ODE find x1 directly
                x1_canidate = (k12 * x2 + k01* x0) / (k01+k12)
                Clamp x1 by limiting diff from x0

            Solve ODE with spring BTW x2 and x1
                x2 is point between damper and 50k spring
                x.2 = dampV[k12 * (x2 - x1)]  (Force blance between spring and damper)
            """
            x_ext = self.ang_filt[np.searchsorted(self.time_filt, time_val)-1]        # Get cmd angle
            x_rot_candidate = (self.k_oil * x_oil + self.k_lash* x_ext) / (self.k_lash+self.k_oil)
            xdiff = x_rot_candidate - x_ext
            if -self.x_lashLim < xdiff < self.x_lashLim:
                x_rot = x_rot_candidate
            elif self.x_lashLim < xdiff:
                x_rot = x_ext + self.x_lashLim
            else:
                x_rot = x_ext - self.x_lashLim
            force = self.k_oil*(x_oil-x_rot)
            vel_new = self.inv_vel*self.dampV(force*self.inv_tq)
            return (vel_new)
        def calc_res(self):
            x_ext = self.ang_filt
            res = odeint(self.vel_k_damper, x_ext[0], self.time_filt)
            x_oil = res[:,0]
            x_rot_candidate = (self.k_oil * x_oil + self.k_lash* x_ext) / (self.k_lash+self.k_oil)
            force = 0*x_oil
            x_rot = 0*x_rot_candidate
            for (i,x1_val) in enumerate(x_rot_candidate):
                xdiff = x_rot_candidate[i] - x_ext[i]
                if -self.x_lashLim <= xdiff <= self.x_lashLim:
                    x_rot[i] = x_rot_candidate[i]
                elif self.x_lashLim < xdiff:
                    x_rot[i] = x_ext[i] + self.x_lashLim
                elif self.x_lashLim > xdiff:
                    x_rot[i] = x_ext[i] - self.x_lashLim

                force[i] = self.k_oil*(x_oil[i]-x_rot[i])
            x_rot_vel = self.inv_vel*self.dampV(force*self.inv_tq)
            fric_force = self.dampF(x_rot_vel*self.inv_vel)*self.inv_tq - self.k_oil*(x_oil-x_rot)
            reactionF = self.dampF(x_rot_vel*self.inv_vel)*self.inv_tq - fric_force
            return (reactionF, x_rot_vel, x_ext, x_rot, x_oil)
        def dampF(self, vel_in):
            f_out = np.piecewise(vel_in,
                [(vel_in<=self.lookup_intercept['R-Valve_R-Leakage, No Seal'][0]),
                ((self.lookup_intercept['R-Valve_R-Leakage, No Seal'][0] < vel_in)&(vel_in < self.lookup_intercept['R-Leakage, No Seal_NegV Fix'][0])),
                (self.lookup_intercept['R-Leakage, No Seal_NegV Fix'][0]<=vel_in) & (vel_in<=0),
                (0<vel_in) & (vel_in<=self.lookup_intercept['PosV Fix_J-Leakage, No Seal'][0]),
                ((self.lookup_intercept['PosV Fix_J-Leakage, No Seal'][0] < vel_in) & (vel_in < self.lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][0])),
                ((self.lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][0] < vel_in) & (vel_in < self.lookup_intercept['J-Leakage, Seal_J-Valve'][0])),
                (self.lookup_intercept['J-Leakage, Seal_J-Valve'][0] <= vel_in)],

                [lambda vel: self.lookup_coeff['R-Valve'][0]*vel + self.lookup_coeff['R-Valve'][1],
                 lambda vel: self.lookup_coeff['R-Leakage, No Seal'][0]*pow(vel,2) + self.lookup_coeff['R-Leakage, No Seal'][1]*vel + self.lookup_coeff['R-Leakage, No Seal'][2],
                 lambda vel: self.lookup_coeff['NegV Fix']*vel,
                 lambda vel: self.lookup_coeff['PosV Fix']*vel,
                 lambda vel: self.lookup_coeff['J-Leakage, No Seal'][0]*pow(vel,2) + self.lookup_coeff['J-Leakage, No Seal'][1]*vel + self.lookup_coeff['J-Leakage, No Seal'][2],
                 lambda vel: self.lookup_coeff['J-Leakage, Seal'][0]*vel + self.lookup_coeff['J-Leakage, Seal'][1],
                 lambda vel: self.lookup_coeff['J-Valve'][0]*vel + self.lookup_coeff['J-Valve'][1]])
            return(f_out)
        def dampV(self, f_in):
            vel_out = np.piecewise(f_in,
                [(f_in<=self.lookup_intercept['R-Valve_R-Leakage, No Seal'][1]),
                 ((self.lookup_intercept['R-Valve_R-Leakage, No Seal'][1] < f_in)&(f_in < self.lookup_intercept['R-Leakage, No Seal_NegV Fix'][1])),
                 (self.lookup_intercept['R-Leakage, No Seal_NegV Fix'][1]<=f_in) & (f_in<=0),
                 (0<f_in) & (f_in<=self.lookup_intercept['PosV Fix_J-Leakage, No Seal'][1]),
                 ((self.lookup_intercept['PosV Fix_J-Leakage, No Seal'][1] < f_in) & (f_in < self.lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][1])),
                 ((self.lookup_intercept['J-Leakage, No Seal_J-Leakage, Seal'][1] < f_in) & (f_in < self.lookup_intercept['J-Leakage, Seal_J-Valve'][1])),
                 (self.lookup_intercept['J-Leakage, Seal_J-Valve'][1] <= f_in)],
                [lambda torq: (torq - self.lookup_coeff['R-Valve'][1])/self.lookup_coeff['R-Valve'][0],
                lambda torq: (pow(4*self.lookup_coeff['R-Leakage, No Seal'][0]*torq-4*self.lookup_coeff['R-Leakage, No Seal'][0]*self.lookup_coeff['R-Leakage, No Seal'][2]+pow(self.lookup_coeff['R-Leakage, No
Seal'][1],2),1/2)+self.lookup_coeff['R-Leakage, No Seal'][1])/(2*self.lookup_coeff['R-Leakage, No Seal'][0]),
                lambda torq: torq/self.lookup_coeff['NegV Fix'],
                lambda torq: torq/self.lookup_coeff['PosV Fix'],
                lambda torq: (pow(4*self.lookup_coeff['J-Leakage, No Seal'][0]*torq-4*self.lookup_coeff['J-Leakage, No Seal'][0]*self.lookup_coeff['J-Leakage, No Seal'][2]+pow(self.lookup_coeff['J-Leakage, No
Seal'][1],2),1/2)+self.lookup_coeff['J-Leakage, No Seal'][1])/(2*self.lookup_coeff['J-Leakage, No Seal'][0]),
                lambda torq: (torq - self.lookup_coeff['J-Leakage, Seal'][1])/self.lookup_coeff['J-Leakage, Seal'][0],
                lambda torq: (torq - self.lookup_coeff['J-Valve'][1])/self.lookup_coeff['J-Valve'][0]])
            return(vel_out)
    get_results = calc_dampf(time,ang,inv_vel,inv_tq)
    return (get_results.calc_res())
```

Appendix V. Test Dashboard: Ramp Input with Temperature Variation

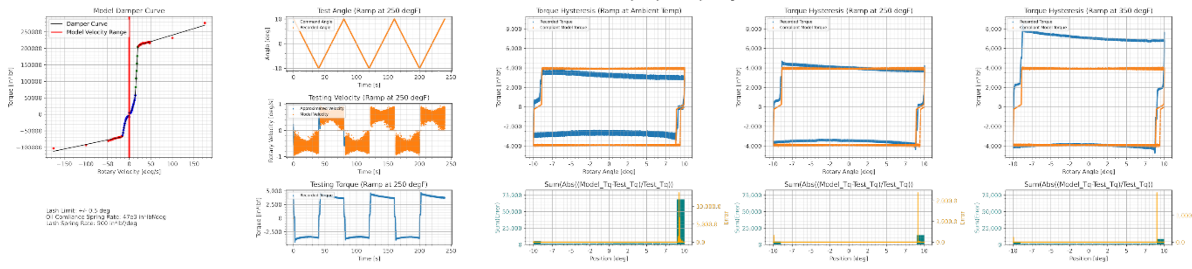
Model and Test Data for "Ramp" Input at 0p05 deg/s



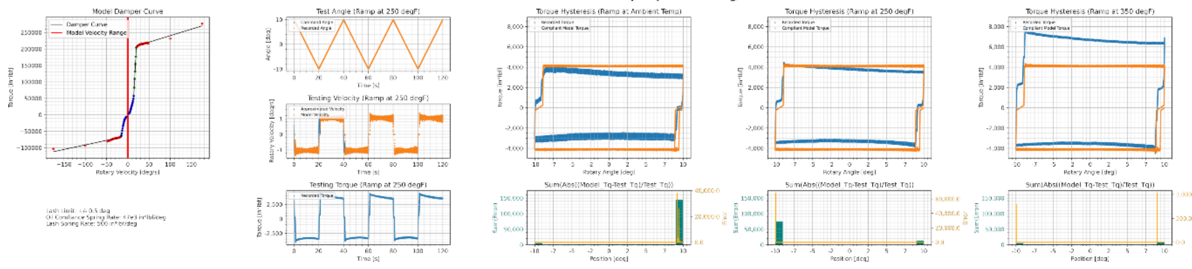
Model and Test Data for "Ramp" Input at 0p1 deg/s



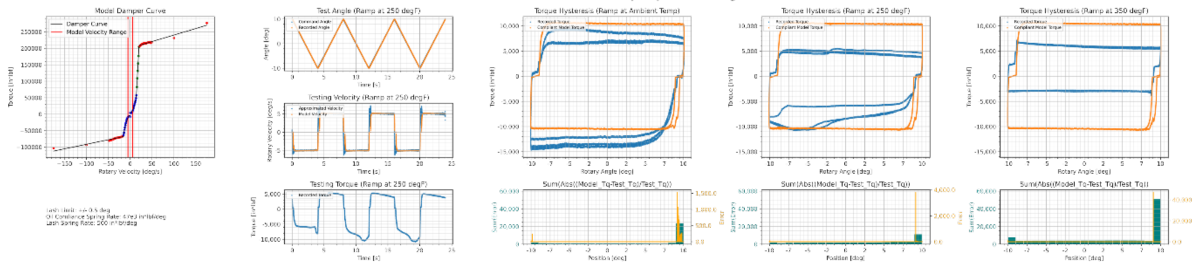
Model and Test Data for "Ramp" Input at 0p5 deg/s



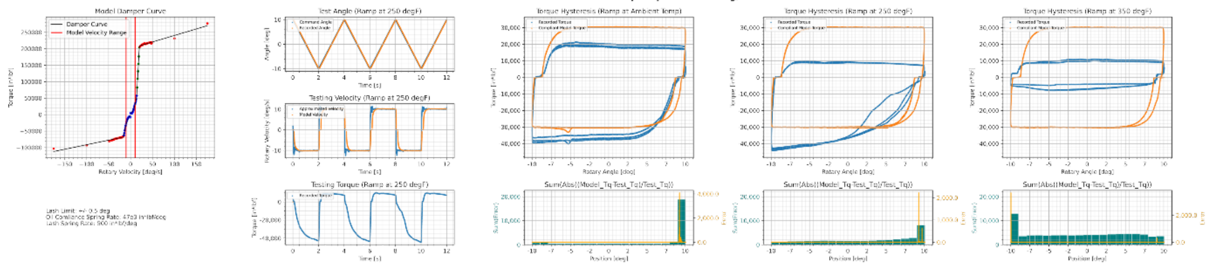
Model and Test Data for "Ramp" Input at 001 deg/s



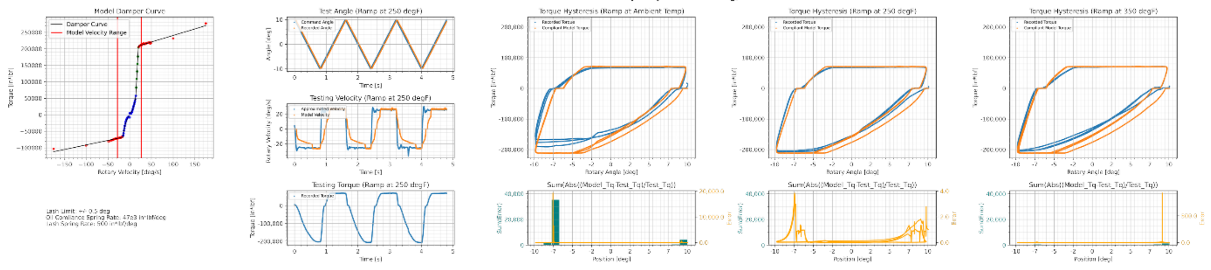
Model and Test Data for "Ramp" Input at 005 deg/s



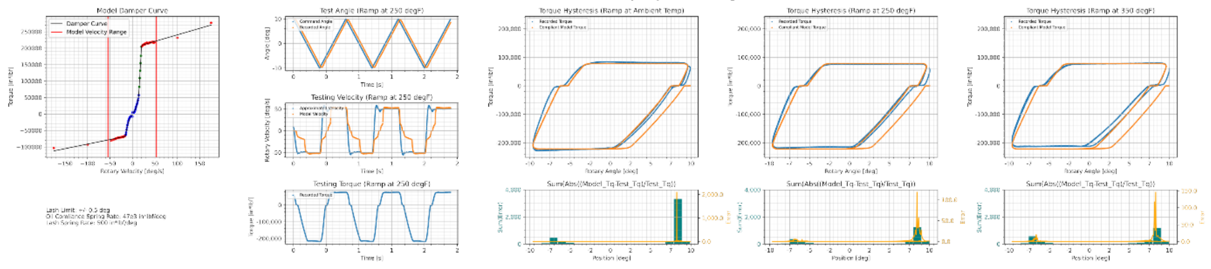
Model and Test Data for "Ramp" Input at 010 deg/s



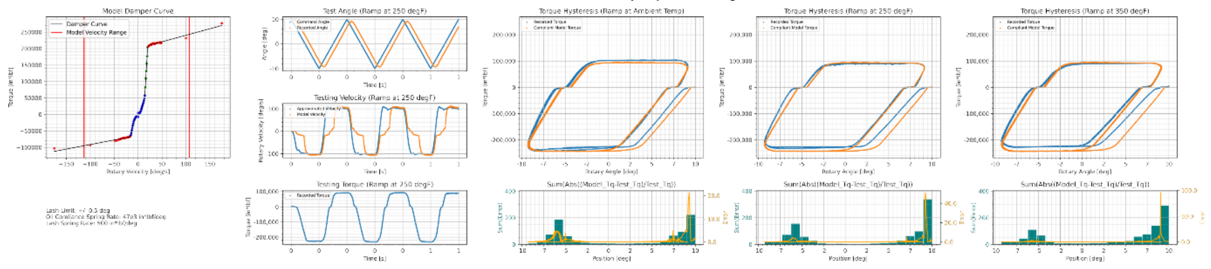
Model and Test Data for "Ramp" Input at 025 deg/s



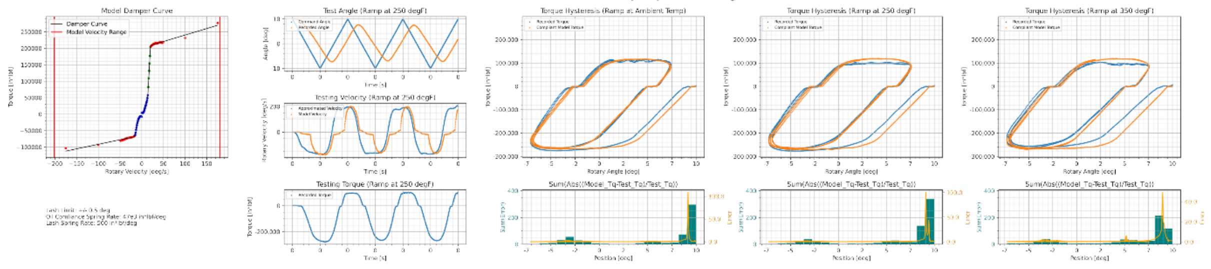
Model and Test Data for "Ramp" Input at 050 deg/s



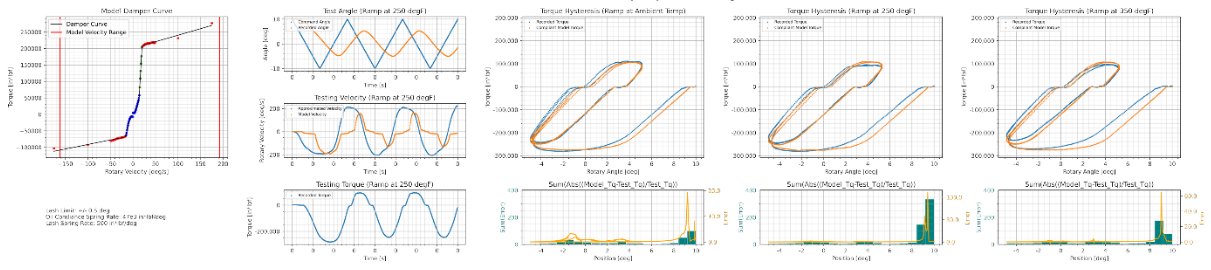
Model and Test Data for "Ramp" Input at 100 deg/s

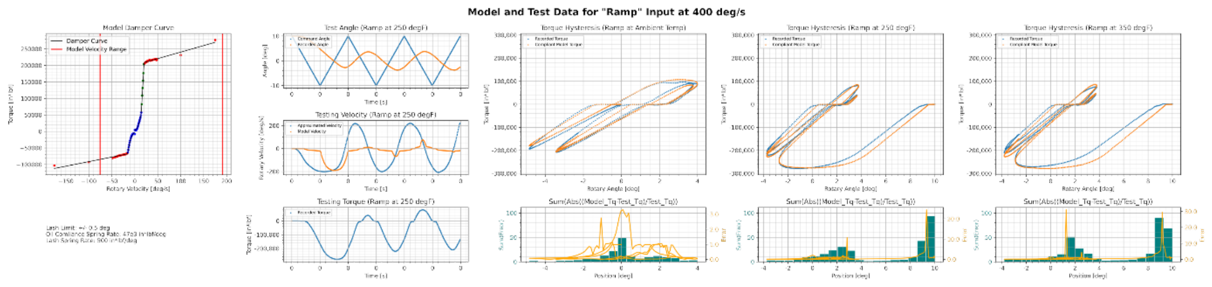


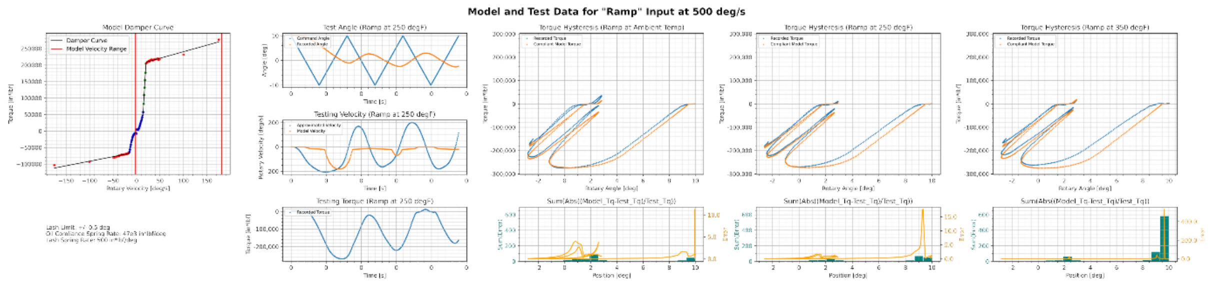
Model and Test Data for "Ramp" Input at 200 deg/s



Model and Test Data for "Ramp" Input at 300 deg/s

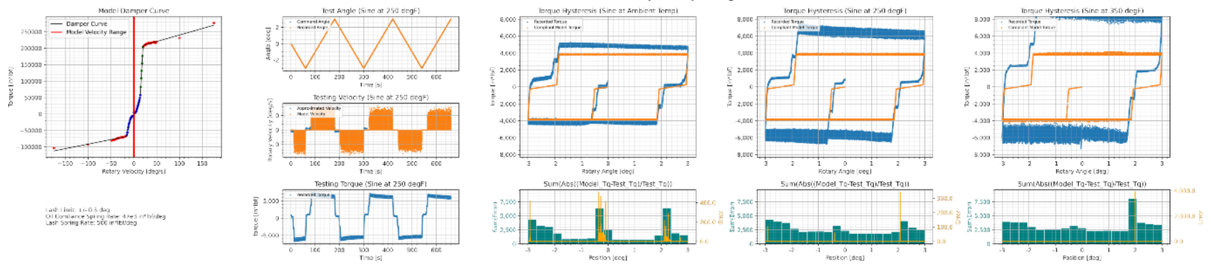



Model and Test Data for "Ramp" Input at 400 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Test Angle (Ramp at 250 degF)
Testing Velocity (Ramp at 250 degF)
Testing Torque (Ramp at 250 degF)
Torque Hysteresis (Ramp at Ambient Temp)
Torque Hysteresis (Ramp at 250 degF)
Torque Hysteresis (Ramp at 350 degF)
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 900 in*lbf/deg

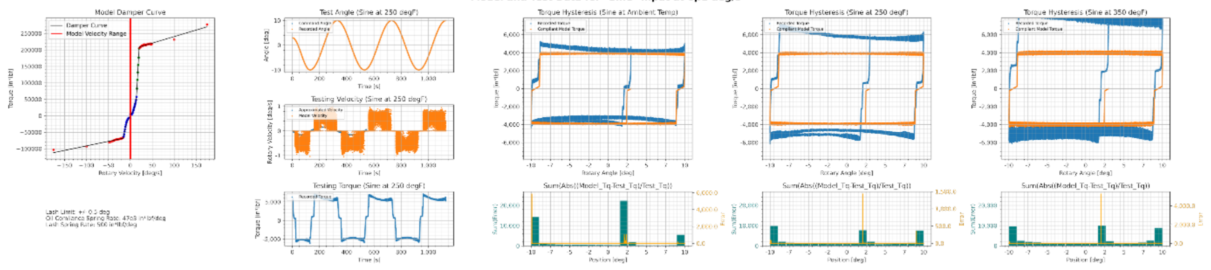

Model and Test Data for "Ramp" Input at 500 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Test Angle (Ramp at 250 degF)
Testing Velocity (Ramp at 250 degF)
Testing Torque (Ramp at 250 degF)
Torque Hysteresis (Ramp at Ambient Temp)
Torque Hysteresis (Ramp at 250 degF)
Torque Hysteresis (Ramp at 350 degF)
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 900 in*lbf/deg

Appendix VI. Test Dashboard: Sinusoidal Input with Temperature Variation

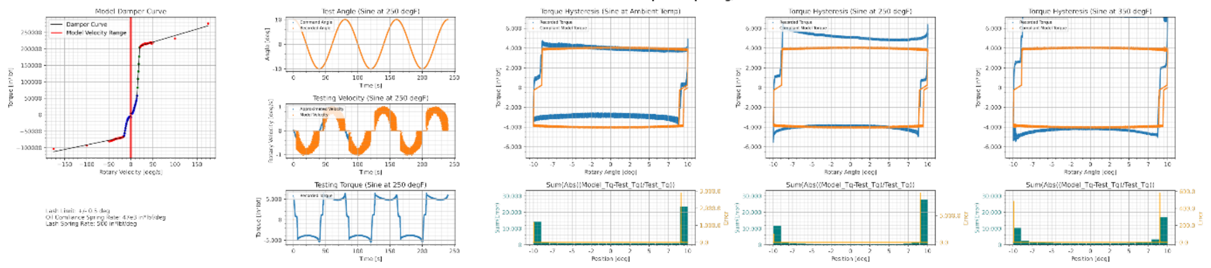
Model and Test Data for "Sine" Input at 0p05 deg/s



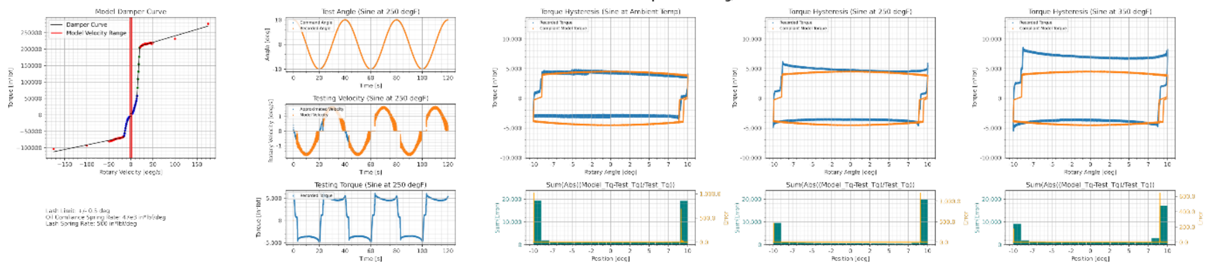
Model and Test Data for "Sine" Input at 0p1 deg/s



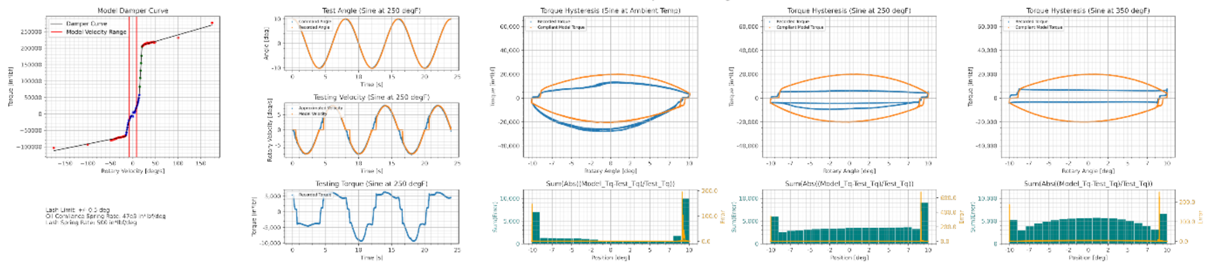
Model and Test Data for "Sine" Input at 0p5 deg/s



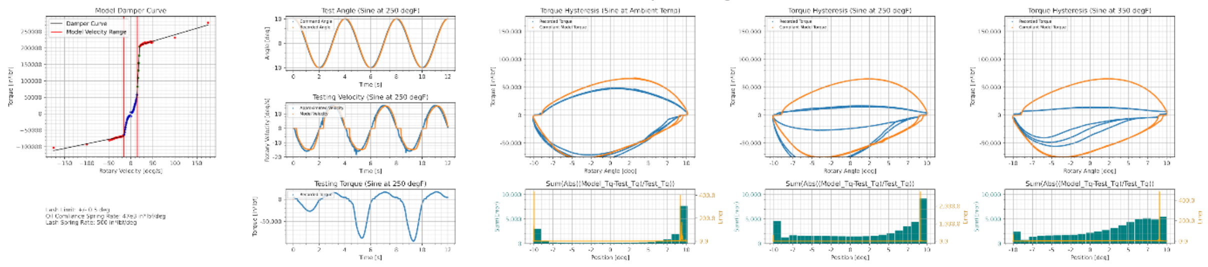
Model and Test Data for "Sine" Input at 001 deg/s



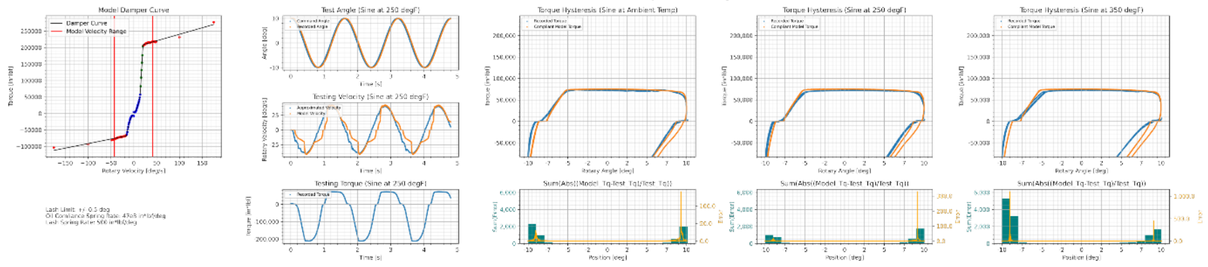
Model and Test Data for "Sine" Input at 005 deg/s



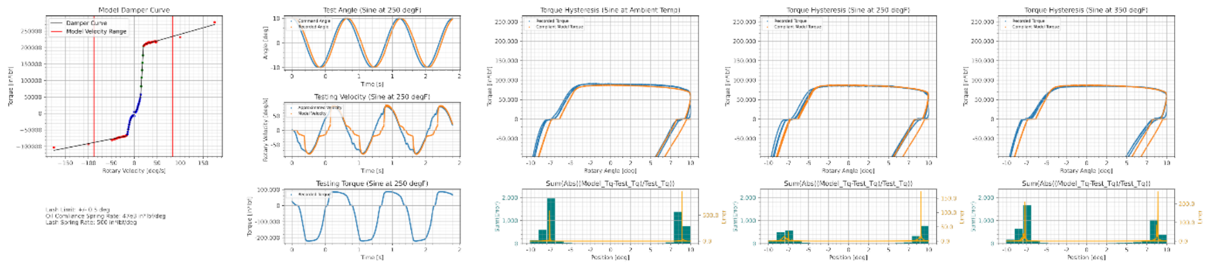
Model and Test Data for "Sine" Input at 010 deg/s



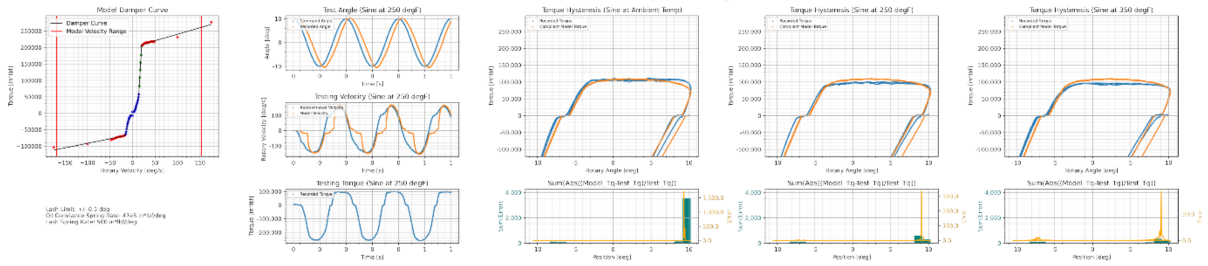
Model and Test Data for "Sine" Input at 025 deg/s



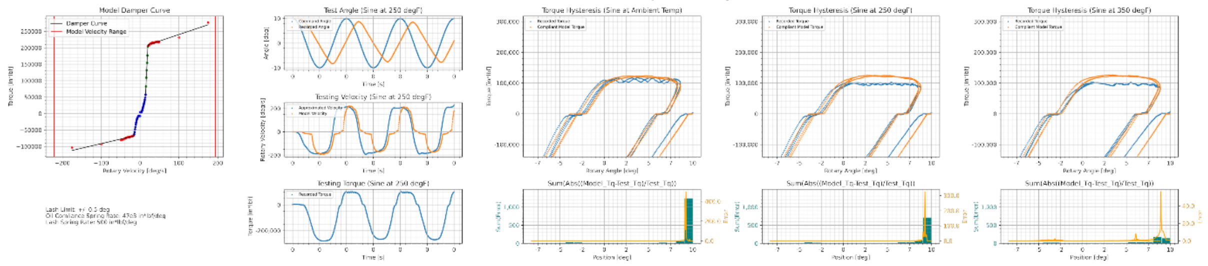
Model and Test Data for "Sine" Input at 050 deg/s



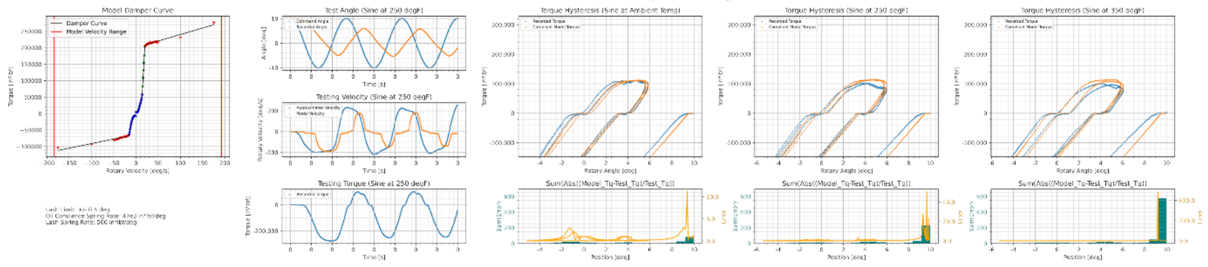
Model and Test Data for "Sine" Input at 100 deg/s

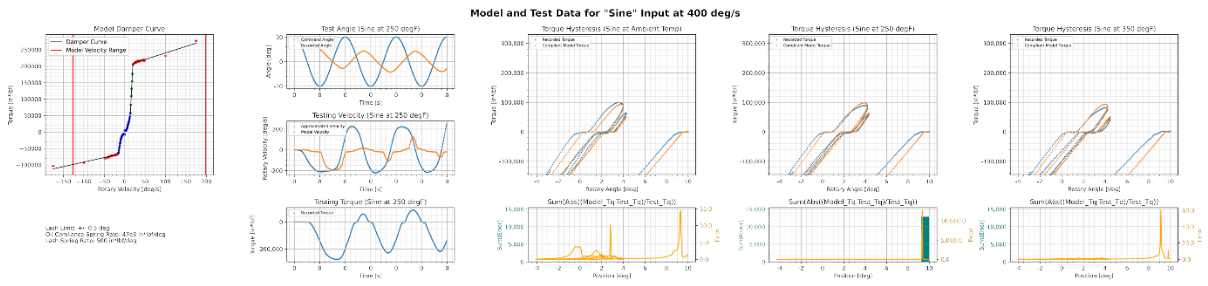


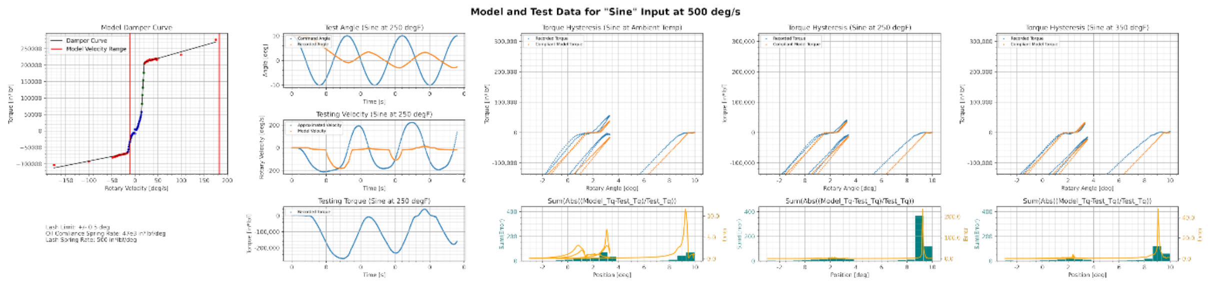
Model and Test Data for "Sine" Input at 200 deg/s



Model and Test Data for "Sine" Input at 300 deg/s

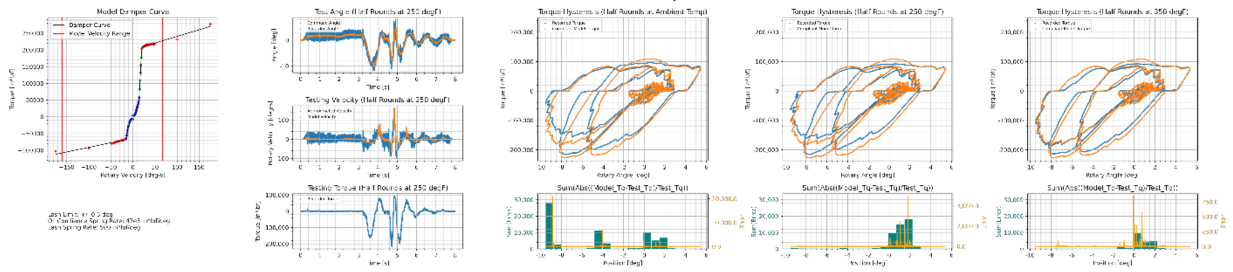



Model and Test Data for "Sine" Input at 400 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Sine at 250 degF)
Command Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Sine at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Sine at 250 degF)
Recorded Torque
Torque [in*lbf]
Torque Hysteresis (Sine at Ambient Temp)
Torque Hysteresis (Sine at 250 degF)
Torque Hysteresis (Sine at 350 degF)
Recorded Torque
Compliant Model Torque
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]

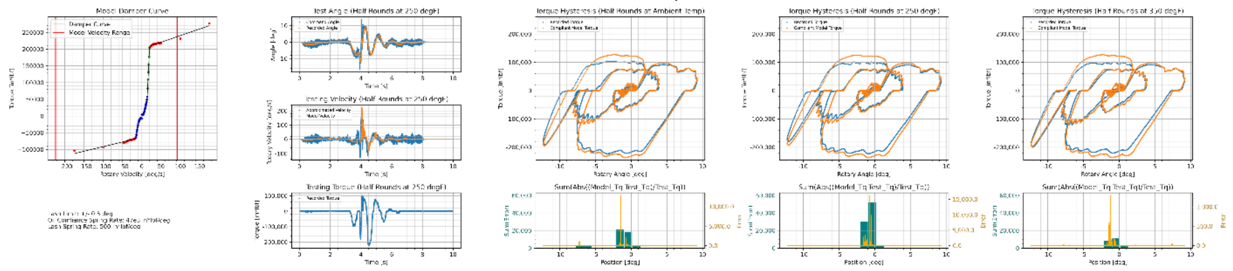

Model and Test Data for "Sine" Input at 500 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Sine at 250 degF)
Command Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Sine at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Sine at 250 degF)
Recorded Torque
Torque [in*lbf]
Torque Hysteresis (Sine at Ambient Temp)
Torque Hysteresis (Sine at 250 degF)
Torque Hysteresis (Sine at 350 degF)
Recorded Torque
Compliant Model Torque
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]

Appendix VII. Test Dashboard: Half Round Input with Temperature Variation

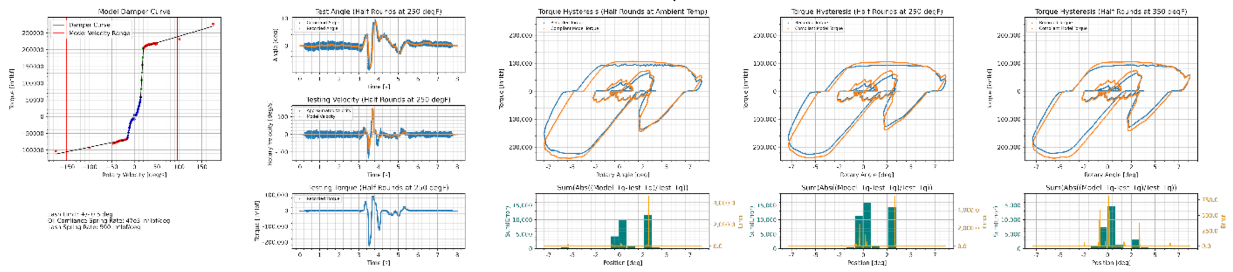
Model and Test Data for "Half Rounds" Input and Test 158 RT7



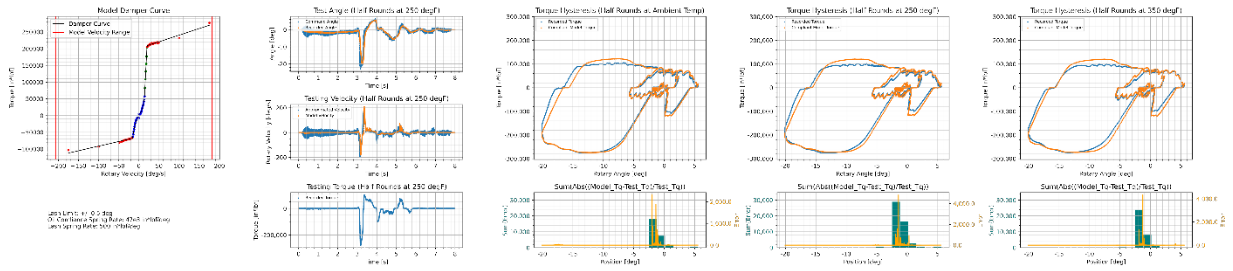
Model and Test Data for "Half Rounds" Input and Test 161 LT7



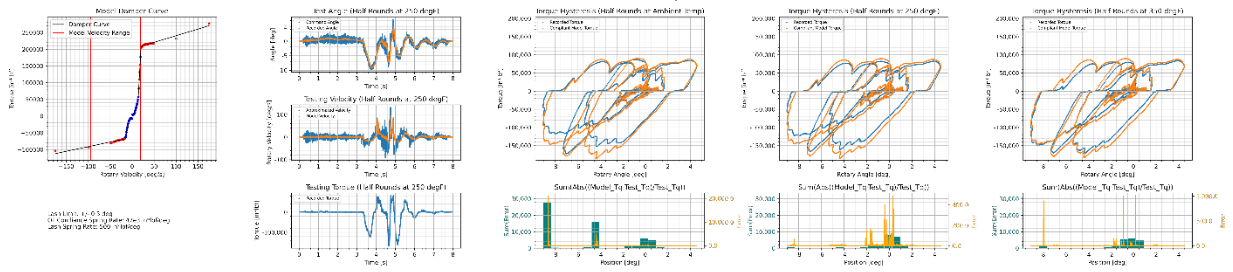
Model and Test Data for "Half Rounds" Input and Test 158 RT2

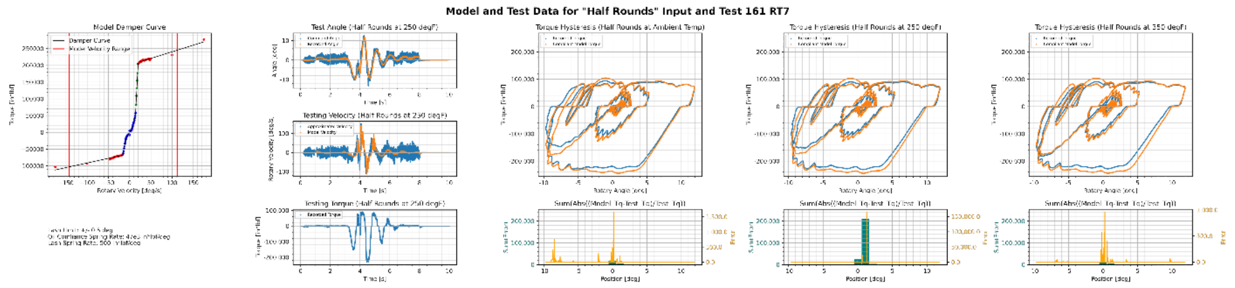


Model and Test Data for "Half Rounds" Input and Test 158 LT1

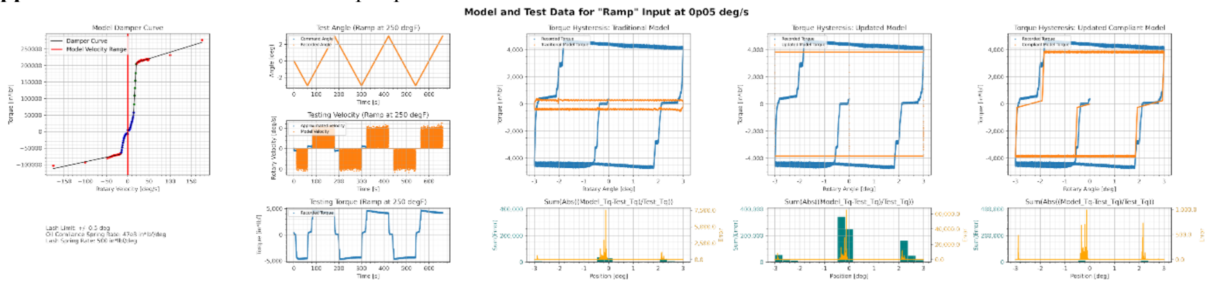


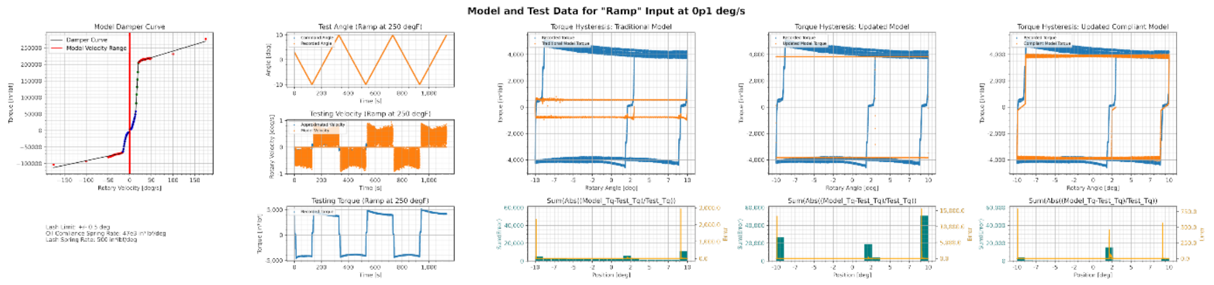
Model and Test Data for "Half Rounds" Input and Test 158 LT7

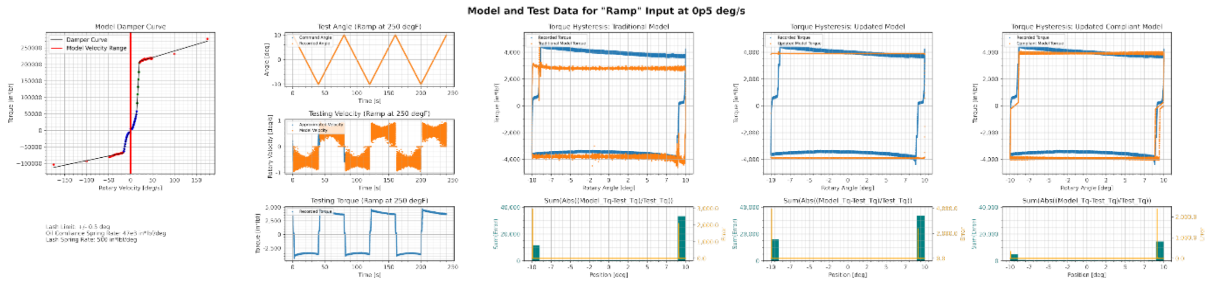


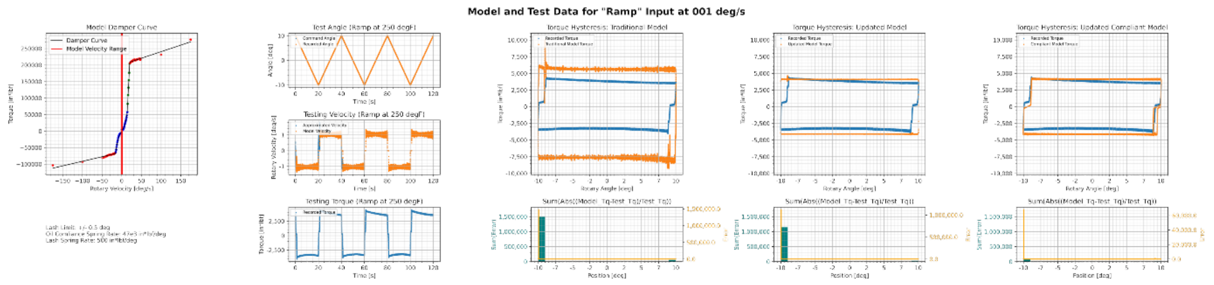

Model and Test Data for "Half Rounds" Input and Test 161 RT7
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Compliance Spring Rate: 4763 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Half Rounds at 250 degF)
Angle [deg]
Time [s]
Testing Velocity (Half Rounds at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Half Rounds at 250 degF)
Recorded Torque
Torque Hysteresis (Half Rounds at Ambient Temp)
Torque Hysteresis (Half Rounds at 250 degF)
Torque Hysteresis (Half Rounds at 350 degF)
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Sum Error
Error
Position [deg]

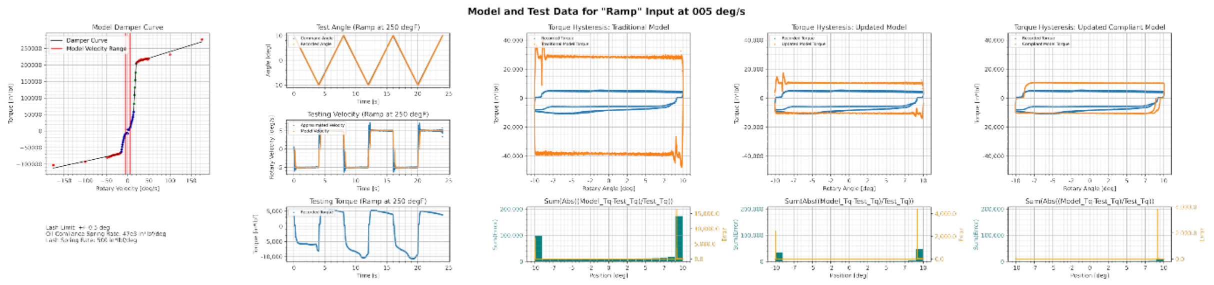
Appendix VIII. Test Dashboard: Ramp Input with Model Variation



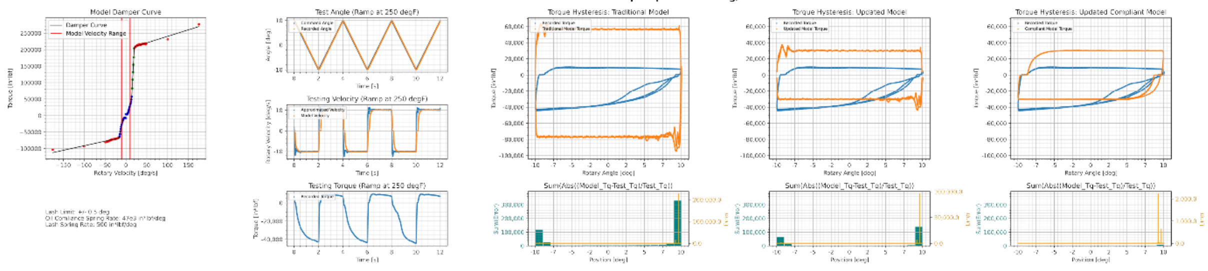




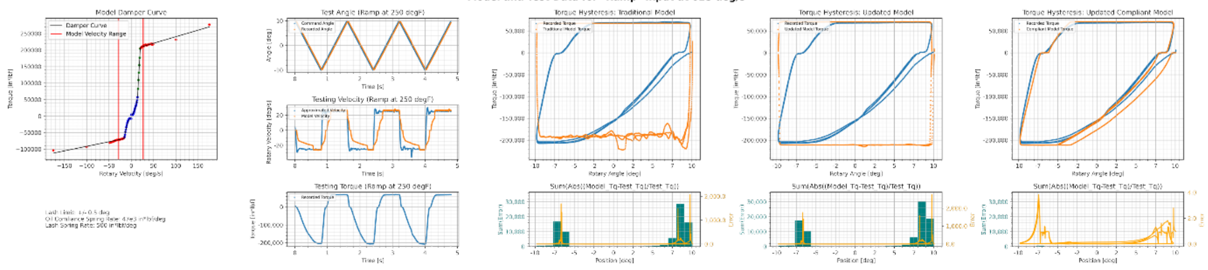




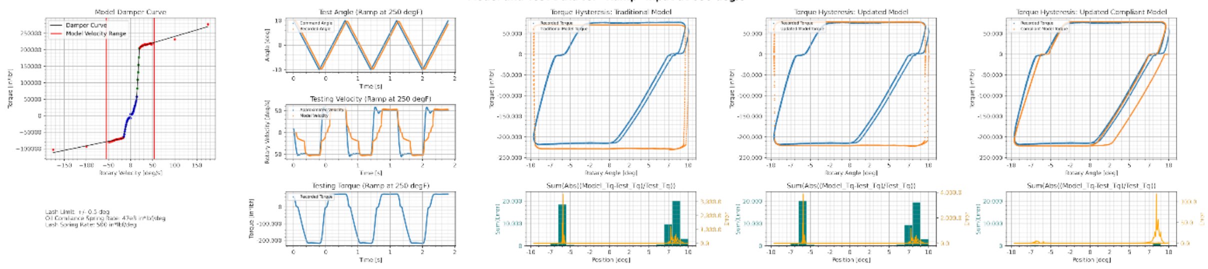
Model and Test Data for "Ramp" Input at 010 deg/s



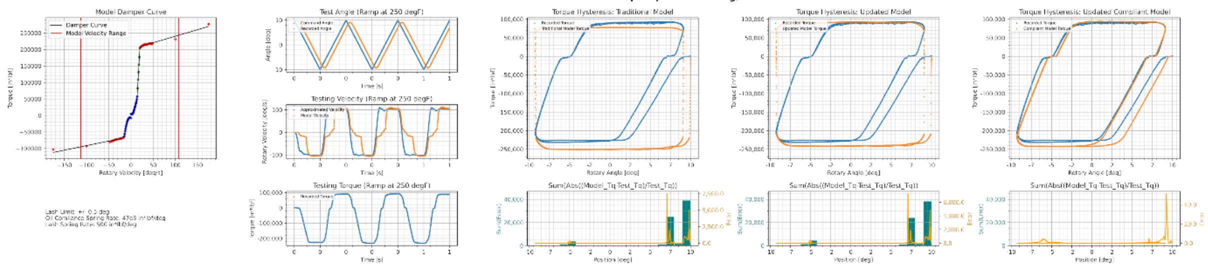
Model and Test Data for "Ramp" Input at 025 deg/s



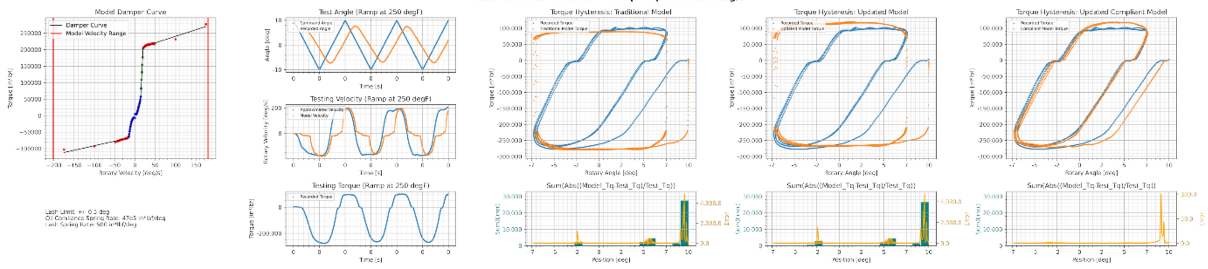
Model and Test Data for "Ramp" Input at 050 deg/s



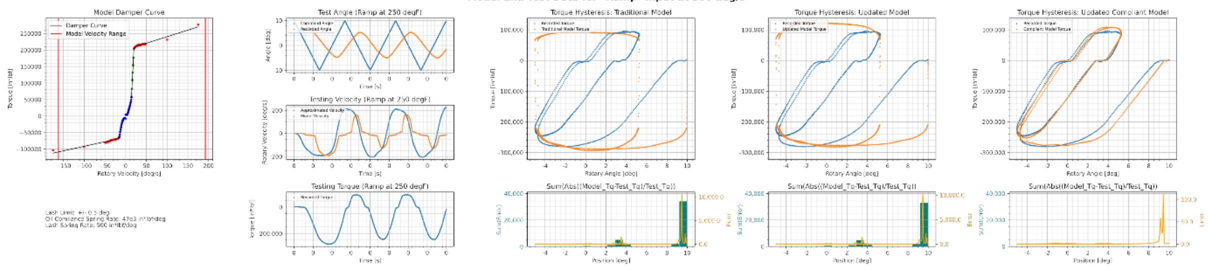
Model and Test Data for "Ramp" Input at 100 deg/s

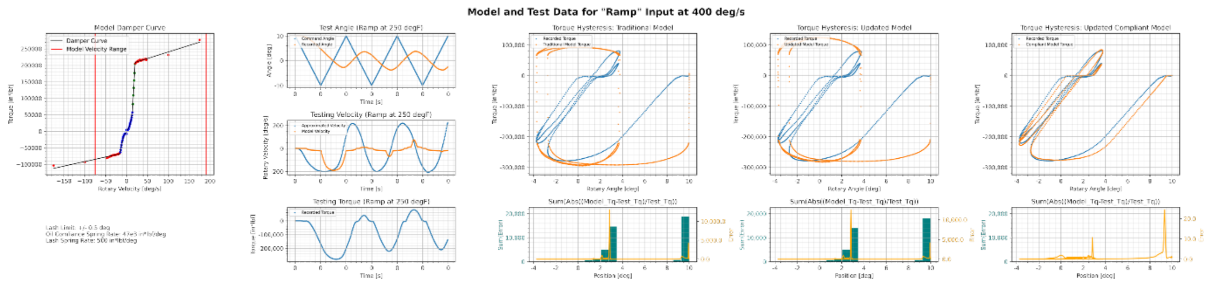


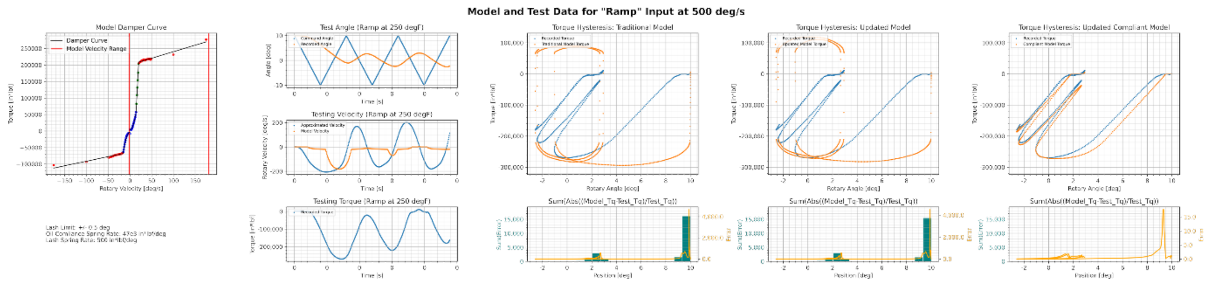
Model and Test Data for "Ramp" Input at 200 deg/s



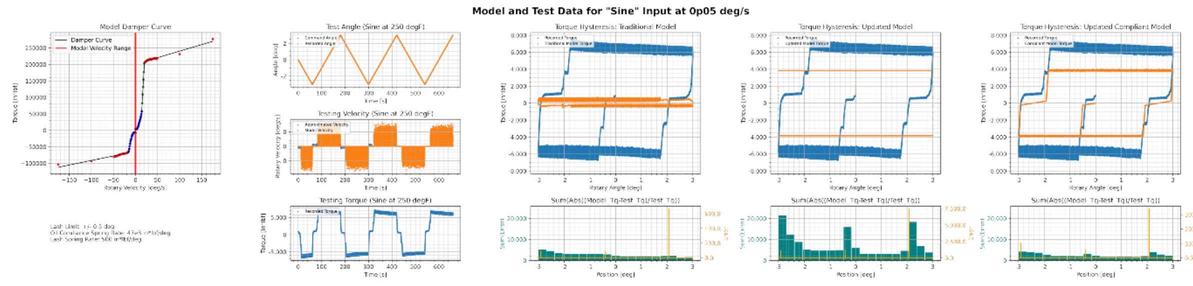
Model and Test Data for "Ramp" Input at 300 deg/s

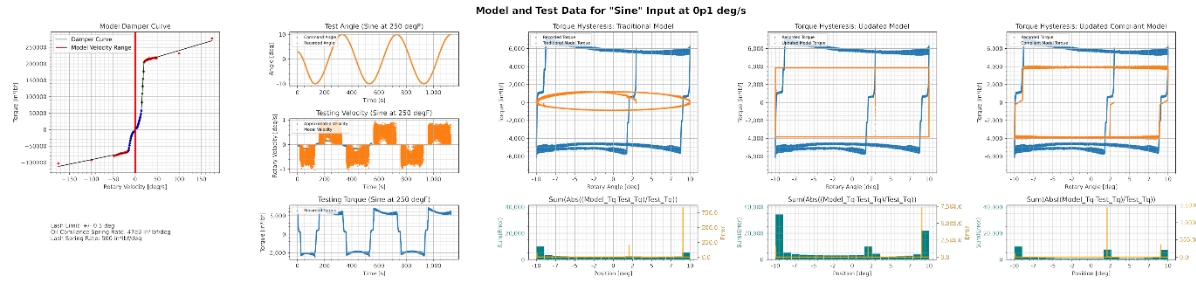


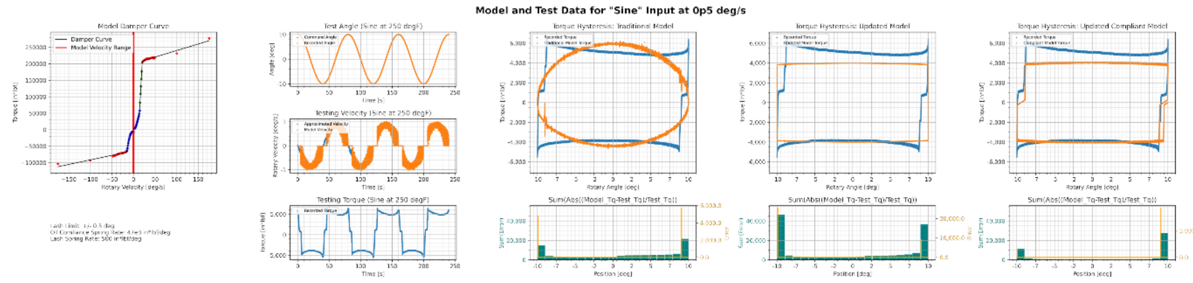

Model and Test Data for "Ramp" Input at 400 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Ramp at 250 degF)
Command Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Ramp at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Ramp at 250 degF)
Recorded Torque
Torque [in*lbf]
Torque Hysteresis: Traditional Model
Recorded Torque
Traditional Model Torque
Rotary Angle [deg]
Torque Hysteresis: Updated Model
Updated Model Torque
Torque Hysteresis: Updated Compliant Model
Compliant Model Torque
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]

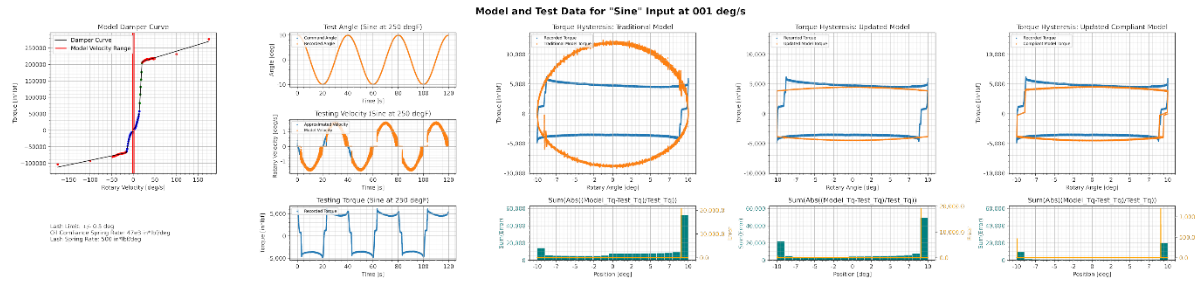

Model and Test Data for "Ramp" Input at 500 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Comliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Ramp at 250 degF)
Command Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Ramp at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Ramp at 250 degF)
Recorded Torque
Torque [in*lbf]
Torque Hysteresis: Traditional Model
Recorded Torque
Traditional Model Torque
Rotary Angle [deg]
Torque Hysteresis: Updated Model
Updated Model Torque
Torque Hysteresis: Updated Compliant Model
Compliant Model Torque
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]

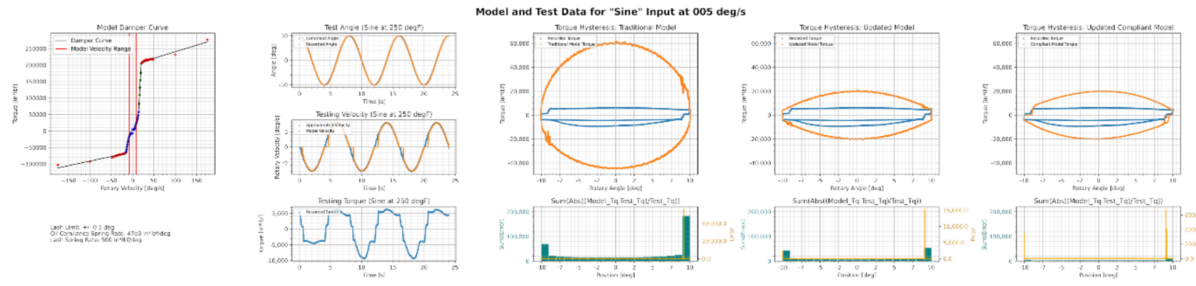
Appendix IX. Test Dashboard: Sinusoidal Input with Model Variation



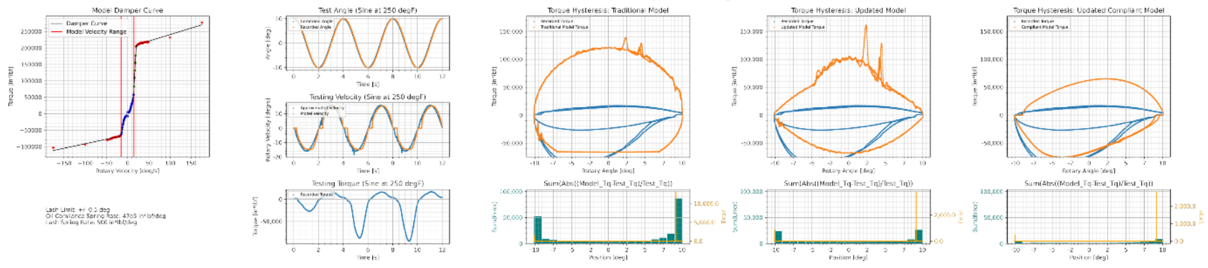




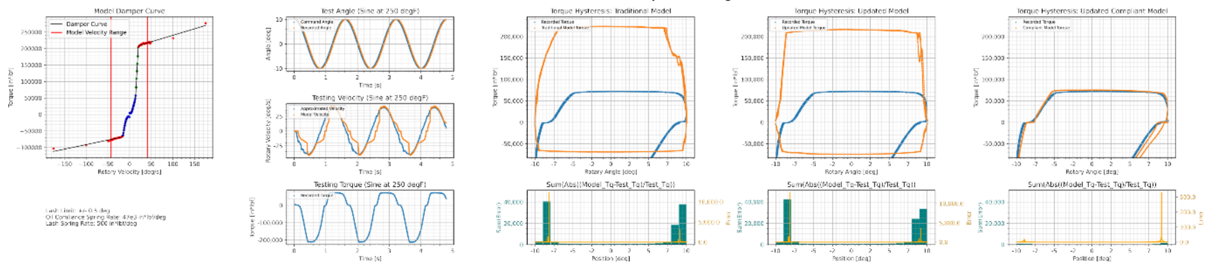




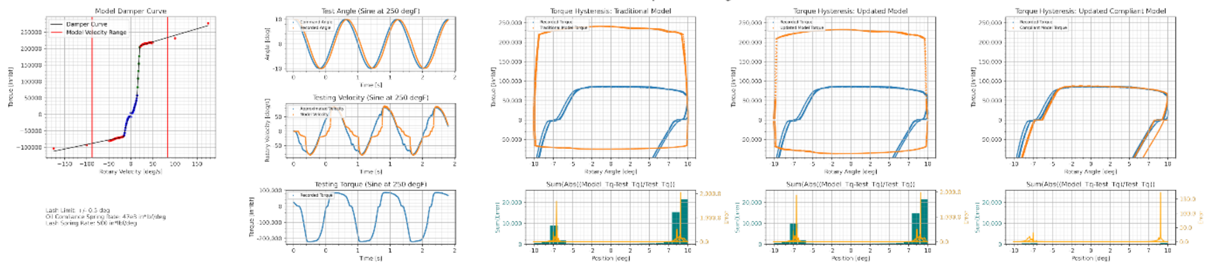
Model and Test Data for "Sine" Input at 010 deg/s



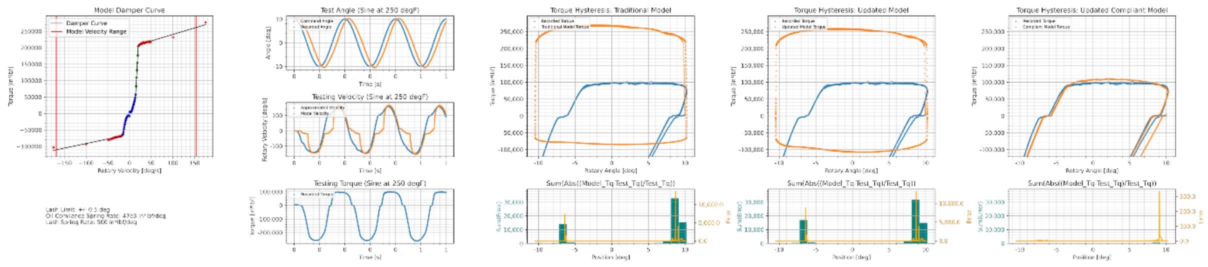
Model and Test Data for "Sine" Input at 025 deg/s



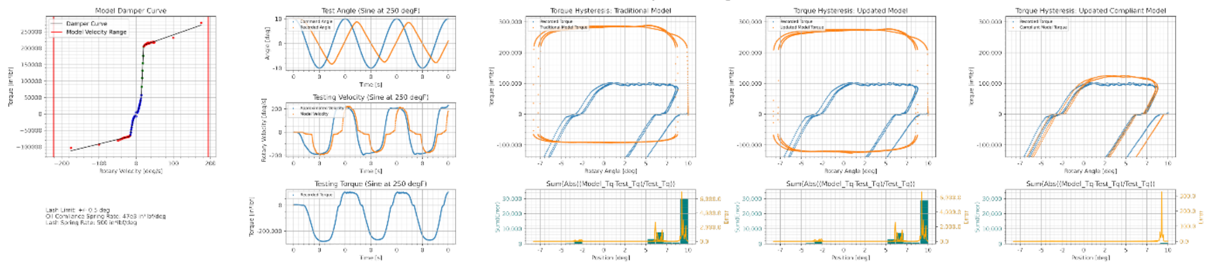
Model and Test Data for "Sine" Input at 050 deg/s



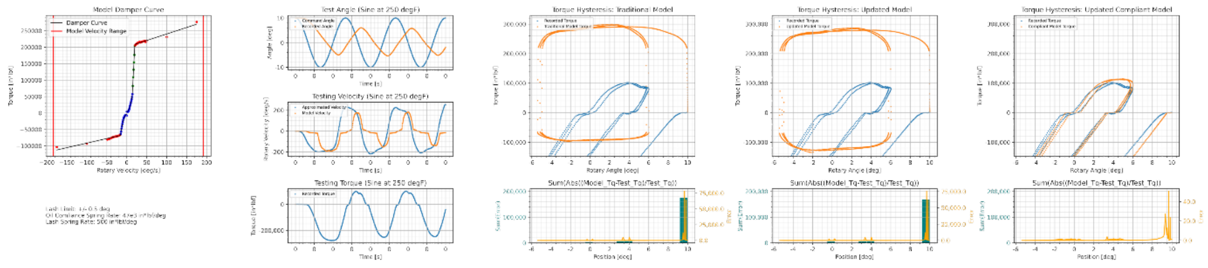
Model and Test Data for "Sine" Input at 100 deg/s

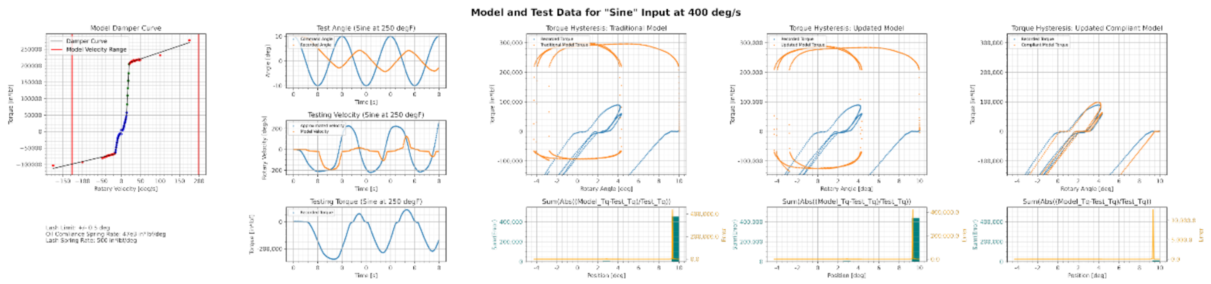


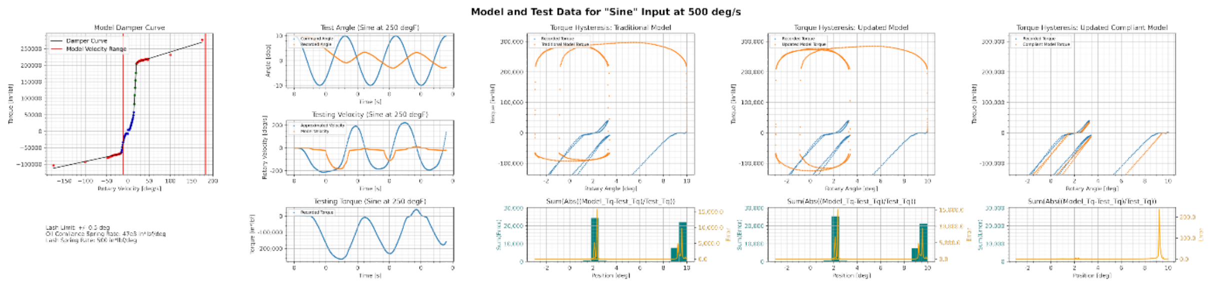
Model and Test Data for "Sine" Input at 200 deg/s



Model and Test Data for "Sine" Input at 300 deg/s

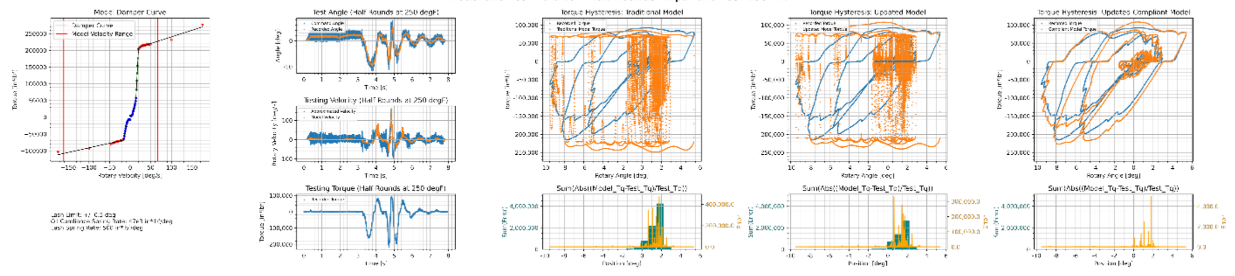



Model and Test Data for "Sine" Input at 400 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Compliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Sine at 250 degF)
Commanded Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Sine at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Sine at 250 degF)
Recorded Torque
Torque Hysteresis: Traditional Model
Recorded Torque
Traditional Model Torque
Torque Hysteresis: Updated Model
Updated Model Torque
Torque Hysteresis: Updated Compliant Model
Compliant Model Torque
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]
Error

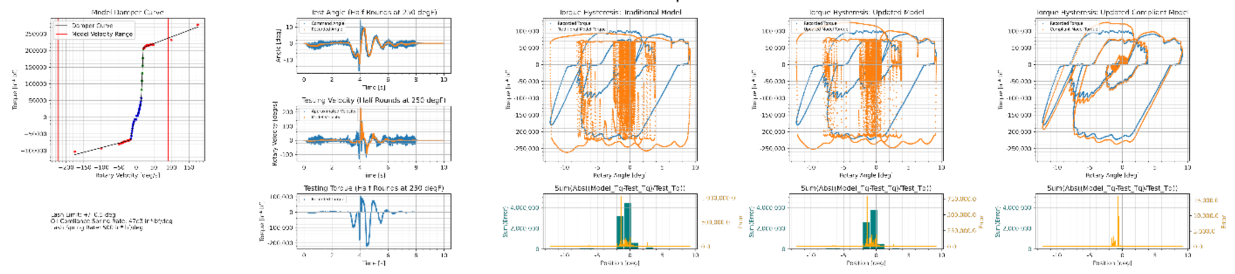

Model and Test Data for "Sine" Input at 500 deg/s
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Compliance Spring Rate: 47e3 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Sine at 250 degF)
Commanded Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Sine at 250 degF)
Approximated Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Sine at 250 degF)
Recorded Torque
Torque Hysteresis: Traditional Model
Traditional Model Torque
Torque Hysteresis: Updated Model
Updated Model Torque
Torque Hysteresis: Updated Compliant Model
Compliant Model Torque
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]
Error

Appendix X. Test Dashboard: Half Round Input with Model Variation

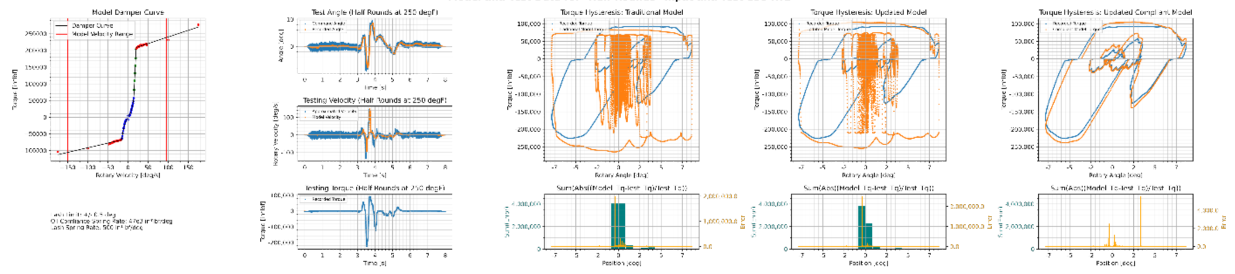
Model and Test Data for "Half Rounds" Input and Test 158 RT7



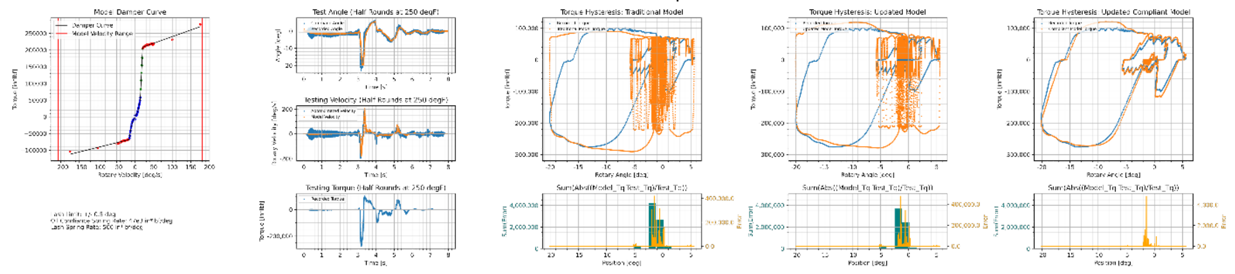
Model and Test Data for "Half Rounds" Input and Test 161 LT7



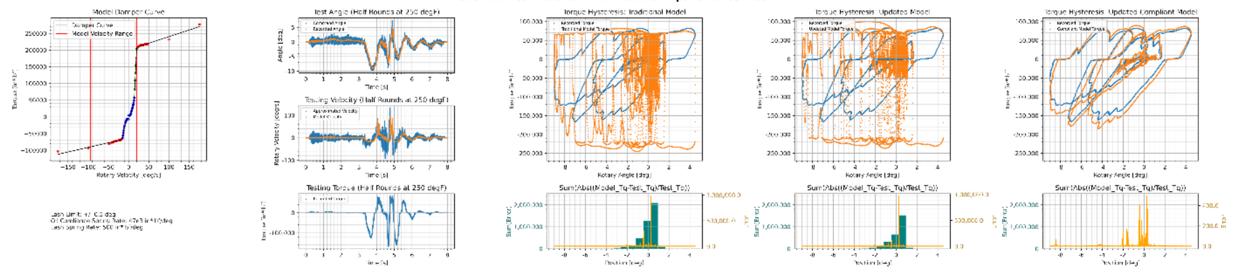
Model and Test Data for "Half Rounds" Input and Test 158 RT2

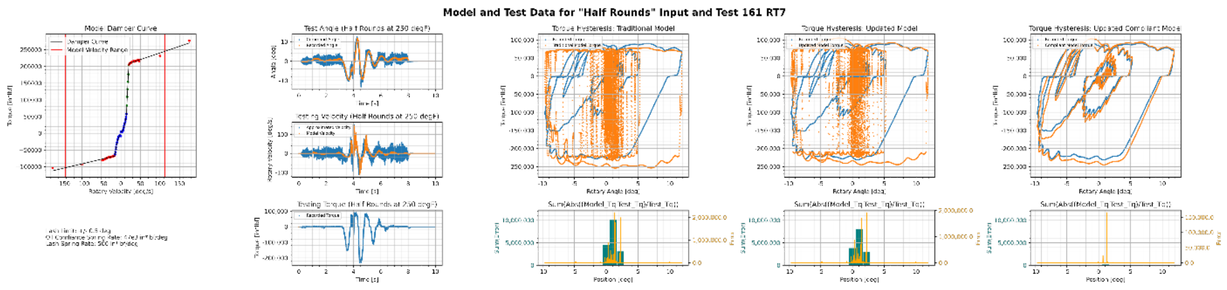


Model and Test Data for "Half Rounds" Input and Test 158 LT1

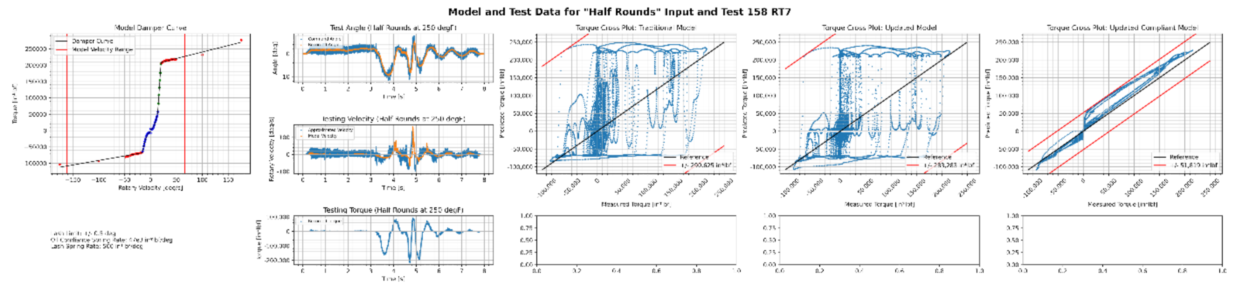


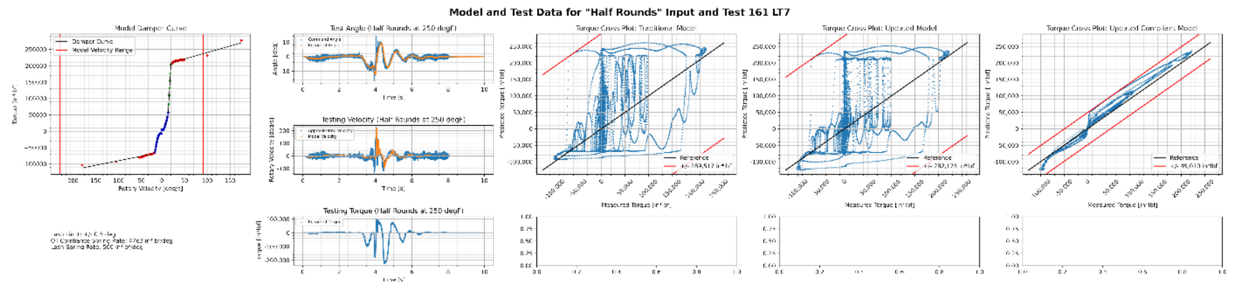
Model and Test Data for "Half Rounds" Input and Test 158 LT7

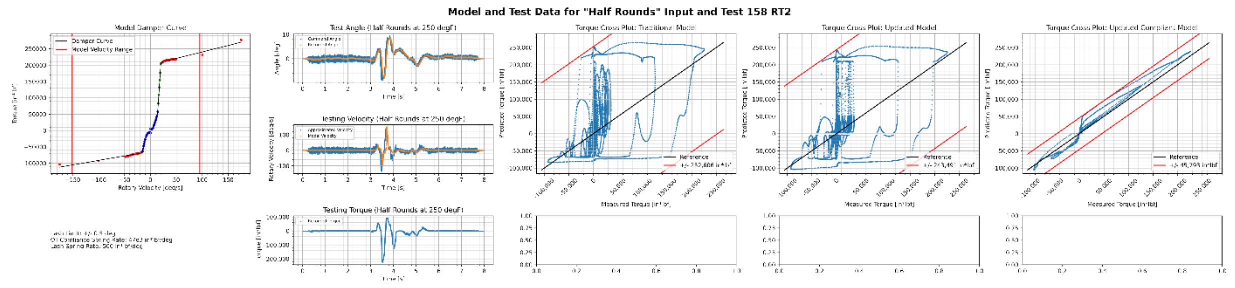


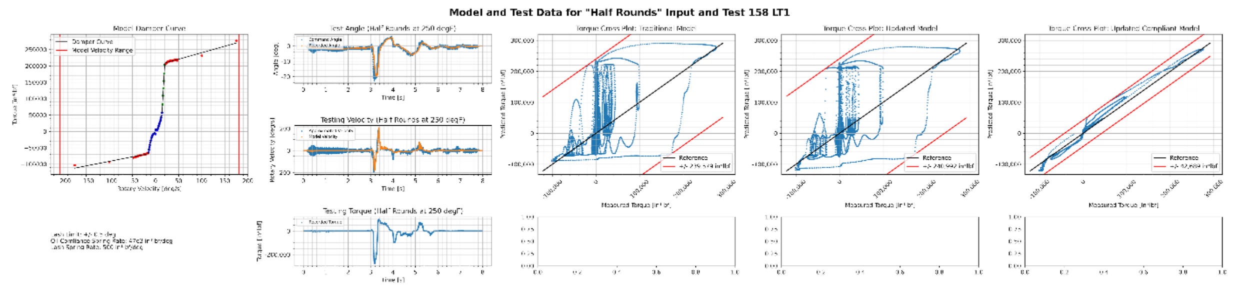

Model and Test Data for "Half Rounds" Input and Test 161 RT7
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash Limit: +/- 0.5 deg
Oil Compliance Spring Rate: 4763 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Half Rounds at 250 degF)
Commanded Angle
Recorded Angle
Angle [deg]
Time [s]
Testing Velocity (Half Rounds at 250 degF)
Approximate Velocity
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Half Rounds at 250 degF)
Recorded Torque
Torque Hysteresis: Traditional Model
Torque Hysteresis: Updated Model
Torque Hysteresis: Updated Compliant Model
Rotary Angle [deg]
Sum(Abs((Model_Tq-Test_Tq)/Test_Tq))
Position [deg]

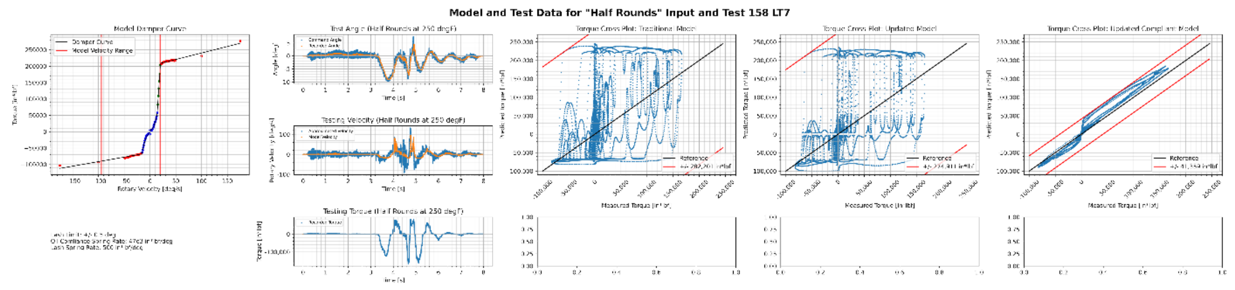
Appendix XI. Test Dashboard: Half Round Input with Model Variation Cross-Plots

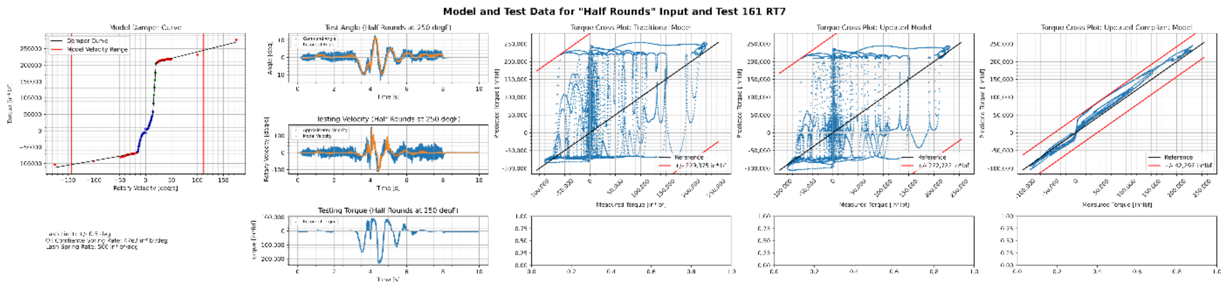











Model and Test Data for "Half Rounds" Input and Test 161 RT7
Model Damper Curve
Damper Curve
Model Velocity Range
Torque [in*lbf]
Rotary Velocity [deg/s]
Lash limit: +/- 0.5 deg
Oil Compliance Spring Rate: 4760 in*lbf/deg
Lash Spring Rate: 500 in*lbf/deg
Test Angle (Half Rounds at 250 degF)
Command Angle
Angle [deg]
Time [s]
Testing Velocity (Half Rounds at 250 degF)
Model Velocity
Rotary Velocity [deg/s]
Testing Torque (Half Rounds at 250 degF)
Torque [in*lbf]
Torque Cross Plot: Traditional Model
Torque Cross Plot: Updated Model
Torque Cross Plot: Updated Compliant Model
Predicted Torque [in*lbf]
Measured Torque [in*lbf]
Reference