ECE 486 Final Project:
The Reaction Wheel Pendulum

Fall 2023

Contents

1   Introduction

1.1   The Reaction Wheel Pendulum

Figure 1: The Reaction Wheel Pendulum

The Reaction Wheel Pendulum (RWP), shown in Figure 1, is a simple pendulum with a rotating wheel at the end. The wheel is actuated by a 24-V, permanent magnet DC motor mounted on the pendulum. This motor can produce a torque on the wheel, causing the wheel to spin. According to Newton's *third* law, there is an equal and opposite *reaction* torque on the motor, and hence on the pendulum. This reaction torque can be used to control the motion of the pendulum. We begin by obtaining the equations of motion for the RWP. Next, control of only the reaction wheel's speed is examined. As part of this phase, we investigate counteracting the effect of friction in the motor by "friction compensation". (*Comment*: A similar identification technique was used in Lab 5 to test frictionless motor.) Finally control of the complete RWP is considered.

1.2   Derivation of Mathematical Model

The first step in any control system design problem is to develop a mathematical model of the system to be controlled. Nonlinear models will first be derived using the
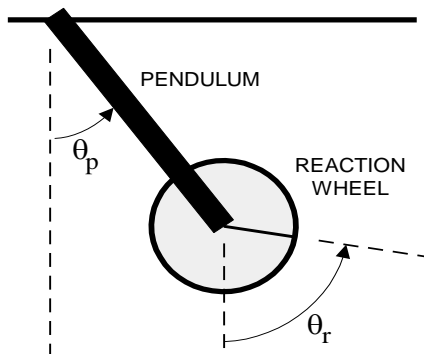
Figure 2: Schematic Diagram

Lagrangian approach. These models will later be linearized, and the linear models will be used to design control strategies.

A schematic diagram of the RWP is shown in Figure 2. We have chosen the angles as in Figure 2 because it is natural to use gravity to line up the pendulum hanging down. The angle θ_p is the angle of the pendulum arm measured *counterclockwise* (*Comment*: also this anti-clockwise direction is defined as *positive* direction.) from the vertical when facing the system, and θ_r is the wheel angle measured likewise.

The RWP is provided with two optical encoders. These encoders are *relative* as opposed to *absolute* encoders and thus measure only the relative angle between their (fixed) stator and (movable) rotor. Their values are initialized to zero at the start of every experiment. (*Comment*: That's why when you run your control later in Chapter 4, 6, and 6, make sure the pendulum is initialized at its resting position with the motor at the bottom.)

One encoder is attached to the fixed mounting bracket with its shaft attached to the pendulum link. It thus provides a measure of the relative angle between the pendulum and the fixed base. The other encoder is attached to the motor fixed at the end of the pendulum. Its shaft is attached to the rotating reaction wheel and thus provides the relative angle between the pendulum and wheel.

If we denote the encoder angles for the pendulum and rotor as φ_p and φ_r, we see that

$$
\begin{aligned}
\theta_p &= \varphi_p \\
\theta_r &= \varphi_p + \varphi_r
\end{aligned}
\qquad . \tag{1}
$$

Later we will discuss such issues as the noise and quantization associated with digital measurement of these angles and the problem of estimating angular velocities from the encoder values.

A convenient way to derive equations of motion for electromechanical systems is the Lagrangian method. The Lagrangian method allows one to deal with scalar energy functions rather than vector forces and accelerations as in the Newtonian method and is, in many cases, simpler.

The RWP has two degrees of freedom. We take as generalized coordinates the angles θ_p of the pendulum and θ_r of the rotor as shown in Figure 2. We also introduce the following variables:

m_p mass of the pendulum and motor housing/stator

m_r mass of the rotor

m combined mass of rotor and pendulum

J_p moment of inertia of the pendulum about its center of mass

J_r moment of inertia of the rotor about its center of mass

ℓ_p distance from pivot to the center of mass of the pendulum

ℓ_r distance from pivot to the center of mass of the rotor

ℓ distance from pivot to the center of mass of pendulum and rotor

k torque constant of the motor

i input current to motor

(*Comment*: Please pay close attention to the three ℓ's above, and you can draw a picture to visualize them if necessary or label them in Figure 2.)

Lagrange's Equations

The Lagrangian method begins by defining a set of *generalized coordinates* $q_1, q_2,...,q_n$, to represent an *n*-degree-of-freedom system. These generalized coordinates are typically position coordinates (distances or angles).

Next, compute the *kinetic energy K*, and the *potential energy V* in terms of these generalized coordinates. Typically, potential energy is only a function of the generalized coordinates, but kinetic energy is a function of the generalized coordinates and their derivatives.

In a multi-body system, the kinetic and potential energies can be computed for each body independently and then added together to form the energies of the complete system. This is an important advantage of the Lagrangian method and works because energy is a scalar-valued function, as opposed to a vector-valued function.

Once the kinetic and potential energies are determined, the *Lagrangian*, $L(q_1,...,q_n,\dot{q}_1,...,\dot{q}_n)$, is then defined as the difference between the kinetic and potential energies. The Lagrangian is therefore a function of the generalized coordinates and their derivatives.

Finally, it can be shown that the equations of motion all have the form

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{q}_k}) - \frac{\partial L}{\partial q_k} = \tau_k \quad k = 1,...,n \tag{2}$$

The variable τ_k represents the generalized force (or torque) in the q_k direction. These equations are called *Lagrange's Equations* and have the remarkable property of remaining invariant with respect to arbitrary changes of coordinates.

We also introduce the quantity

$$J = J_p + m_p \ell_p^2 + m_r \ell_r^2 \tag{3}$$

to represent the moment of inertia with respect to θ_p, and note the relationships

$$m = m_p + m_r \quad , \tag{4}$$

$$m\ell = m_p \ell_p + m_r \ell_r \quad . \tag{5}$$

(*Comment*: J is the total inertia when considering a non rotating rotor attached to the end of the pendulum as an object. You treat rotor plus pendulum as an overall structure.)

> 1-a   Write down the equations for the kinetic energy K and potential energy V of the RWP. (Note that the kinetic energy of the system is the sum of the kinetic energies of each degree of freedom. How many degrees of freedom does the RWP have?) Also point out the generalized coordinates and their derivatives. How many equations of motion will we have? *(Hint: See Appendix A for help with the physics if you're stuck. Comment: If it is not very helpful, check out Wikipedia on Lagrangian Mechanics.)*

★ **Notice:** the RWP includes the motor. So *the motor cannot produce a net torque on the RWP*. Therefore, when it exerts a torque on the rotor it **must exert an equal and opposite torque on the pendulum**. Use the relation $\tau = ki$ for motor torque.

> 1-b   Write Lagrange's equations (see Equation (2)) for this system. Express the equations using three parameters: $\omega_{np}^{2} = \dfrac{mg\ell}{J}$, $\dfrac{k}{J}$, and $\dfrac{k}{J_r}$.
>
> *(Notice that ω_{np} is the frequency of small oscillations of the system around the hanging position. It is* not *the first derivative of position.)*

Derive kinetic energy and potential energy with respect to the generalized coordinates: (*Comment*: For the reasons given above and we defined J in the first place, we are about to derive equations of motion with respect to *generalized coordinates*, hence in the following, subscript "pendulum+rotor" means θ_p motion, "rotor" means θ_r motion.)

KE_{pendulum+rotor} =

PE_{pendulum+rotor} =

KE_{rotor}   =

PE_{rotor}   =

L_{pendulum+rotor}   =

L_{rotor}   =

Lagrange Equation_{pendulum+rotor}=

Lagrange Equation_{rotor}   =

Your final representation should be:

$$\begin{cases} \ddot{\theta}_p + \omega_{np}{}^2 \sin\theta_p = -\dfrac{k}{J}i \\ \ddot{\theta}_r \qquad\qquad = \dfrac{k}{J_r}i \end{cases}. \tag{6}$$

So far we have ignored friction. The mass on the pendulum is large enough that the friction on the pendulum link can be ignored. However, there is a significant amount of friction on the rotor link (mostly due to motor friction). Fortunately, the rotor is attached directly to the motor, making friction easy to model. The motor current i is generated by a pulse width modulation system, which is controlled from the computer. Due to current feedback, the current is proportional to the control command u from the computer. The control variable used in the computer is scaled so that 10 units correspond to maximum current. Therefore we can write

$$ki = k_u u, \quad |u| \le 10. \tag{7}$$

We assume the friction is a function of the rotor speed $F(\omega_r)$. Initially, we will model friction in command units (units of 'u'). Applying Equation (7),

$$\begin{cases} \ddot{\theta}_p + \omega_p{}^2 \sin\theta_p = -\dfrac{k_u}{J}(u + F(\dot{\theta}_r)) \\ \ddot{\theta}_r \qquad\qquad = \dfrac{k_u}{J_r}(u + F(\dot{\theta}_r)) \end{cases}. \tag{8}$$

Finally, to clear up the clutter, we can also introduce variables $a = \omega_{np}{}^2 = \dfrac{mg\ell}{J}$, $b_p = \dfrac{k_u}{J}$, and using (8), $b_r = \dfrac{k_u}{J_r}$ becomes:

$$\begin{cases} \ddot{\theta}_p + a\sin\theta_p = -b_p(u + F(\dot{\theta}_r)) \\ \ddot{\theta}_r \qquad\quad = b_r(u + F(\dot{\theta}_r)) \end{cases}. \tag{9}$$

This is a satisfactory representation of the RWP. Before we begin its control, however, let us take a detour and consider speed control of a DC motor. This will allow us to model and play with friction.

2   Friction Identification Using the Reaction Wheel

In this section we will identify friction, using only the reaction wheel. Recall that we made the assumption that the pendulum link has negligible friction. Isolate the motor by attaching the pendulum arm to the mounting bracket using the provided hex spacers. (*Comment*: Ask your TA to fix the pendulum arm.) We will design velocity controllers to identify friction. Since velocity information is not directly available to us, we will explore alternate methods. Keep in mind that in this part we are trying to model the real world, not perform actual control systems analysis. Therefore the emphasis will be on contrasting calculated or simulated behavior with actual behavior, rather than on designing controllers to meet certain performance specifications.

Safety note: When doing experiments in this part, remember a couple of things. Even though the motor controller has a safety mechanism to prevent the motor from spinning extremely fast, the combination of spinning fast and running for a long time will heat up or even burn out the motor. Be prepared to shut off the controller either when the system runs long enough for you to collect data or when it becomes unstable, either through Windows Target or the switch on the amplifier board. **Consider 200 rad/s as fast**.

In past labs we have analyzed the dynamics of a DC motor, using the armature voltage as the input. Here however, we select the armature current i as input. Then the motor becomes merely a current-torque transducer (see Figure 3), meaning electrical
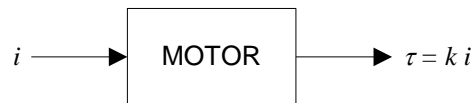

$$i \longrightarrow \boxed{\text{MOTOR}} \longrightarrow \tau = k\,i$$

Figure 3: DC Motor Model

energy will be converted to mechanical energy.

The torque τ is applied to the reaction wheel (rotor) having moment of inertia J_r and speed (relative to the motor housing) of ω_r. There is also friction, due mainly to the motor brushes and represented as a torque τ_F. So the motion of the wheel is given by

$$J_r \dot{\omega}_r = ki - \tau_F \tag{10}$$

using the motor (rotor) speed $\omega_r = \dot{\theta}_r$ as the output. Putting it in terms we are familiar with, we get the following: (*Comment*: The τ_F above is "positive" while the $F(\dot{\theta}_r)$ below is "negative".)
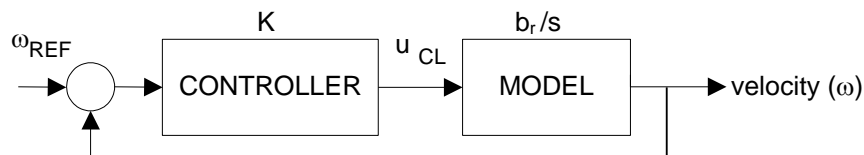

Figure 4: General Block Diagram of Velocity Controller

$$\dot{\omega}_r = \ddot{\theta}_r = b_r(u + F(\dot{\theta}_r))$$ (11)

> 2-a   Design a proportional controller with a rise time of 0.2s and no steady state error. Use an input step of 100 rad/s. Assume b_r = 198 (rad/s). Simulate your controller using Simulink. (See Figure 4)

2.1   Velocity Estimation

Consider Equation (11) but ignore friction for now. Of course, this is an ideal model, so a few real-world issues must be dealt with.

Recall from Section 1.2, page 5, that the control input is limited to 10. That is simple enough to simulate in Simulink. *(Hint: Check out the "Saturation" block.)*

Another issue is the determination of angle from the encoder output. Think of the encoder as the Wheel of Fortune wheel; counting ticks tells you that it's turning. (There's also a provision for determining direction of spin; this is analogous to the "ticker" sounding different in either direction.) The ticks add as the angle changes. There are two issues here:

First, how does the software know where "zero" is? By convention, zero is the encoder angle when you "Start" the run.

Second, how do you determine angle from ticks? Since the motor encoder has 4000 ticks/revolution, multiply by $2\pi/4000$ to scale to radians. (The pendulum encoder has 5000 ticks/revolution.) One other detail: the reaction wheel encoder and motor use opposite sign conventions in this setup. In other words, when a positive current is applied to the motor, it spins in a direction that the encoder calls negative. **Therefore, place an inverter before the input to the motor**.

The encoder measures position. How can the velocity $\omega = d\theta/dt$ be obtained? This is done either by using a transfer function that approximates a derivative or by using a discrete version of the same. We will implement both and compare them.

A simple discrete version can be found by using Euler's method (FPE pp.167, or FPE 3rd Ed pp. 138). It states that

$$\frac{df}{dt} = \lim_{\Delta t \to 0} \frac{f(t) - f(t - \Delta t)}{\Delta t}$$ (12)

The discrete derivative approximation is implemented by using the "Unit Delay" block in Simulink. In order to keep life simple, when doing calculations we will still consider our velocity estimate to be ideal. Applying Equation (12) to our system,

$$\omega(t) \cong \frac{\theta_r(t) - \theta_r(t - \Delta t)}{\Delta t}$$ (13)

Trick: *To make sure you don't have to change Δt each time you change the simulation step size (it must be fixed-step) in Simulink, use "str2num(get_param(bdroot, 'FixedStep'))" in place of Δt. This causes Simulink to look up the value you've entered in the Fixed Step Size field in Configuration Parameters.*

The continuous derivative approximation can be understood by looking at the frequency response of the derivative function *s*. We want to keep the response similar at

low frequencies, but refrain from amplifying the high-frequency noise. This is accomplished by placing a pole at a sufficiently high[1] frequency, giving the transfer function

$$\frac{s}{\tau s + 1}, \tag{14}$$

where $\omega = 1/\tau$ is the pole location.

For your continuous derivative, use Equation (14) with $\tau = 1/50$. Now that we have a velocity estimate, we can solve the following problem:

2-b   Implement your controller designed above in Windows Target (see Windows Target Notes in Appendix B).

    ① Use a "Manual Switch" to choose between your continuous and discrete velocity estimates. Pick the better derivative.

    ② Compare the simulated response in 2-a with the Windows Target response in here 2-b. What is the source of this discrepancy?

The answer to question ② leads us to the ultimate goal of this section: friction identification and modeling.

2.2   PI Control for Friction Identification

As you know, we can counteract a constant disturbance by adding an integrator.

2-c   Choose a PI controller to regulate the system to 100 rad/s (similar to Question 2-a). Simulate it, and also implement it using Windows Target. Compare results, especially steady-state velocities and steady-state control efforts. Explain why the steady-state control effort differs.

This nonzero control effort can be used to our advantage. We can use it to characterize friction. It may seem odd to be doing system identification while applying a controller; this is called *closed-loop system identification*, and is a relatively new and exciting area of study. In this case, closed-loop system identification allows us to work with a stable system and use straightforward procedures (c.f. open-loop friction identification in Lab 4).

Let's see how this works. Consider Equation (11) at steady-state, with friction included.

---

[1] *Sufficiently high:* application-specific, often selected iteratively by simulation or test runs.

$$\dot{\omega}_r = \ddot{\theta}_r = b_r(u + F(\dot{\theta}_r)) = 0 \tag{15}$$

We see that for non-zero friction, the control effort will be non-zero as well. In fact, the value of friction for any velocity is merely the steady-state control effort for a setpoint of that velocity. In other words,

$$u = -F(\dot{\theta}_r) + u_{cl} \tag{16}$$

where u_{cl} is the closed-loop controller.

2-d   Run the motor at various speeds (i.e. vary the setpoint) and record the steady-state control effort for each speed. Do this for both positive (counter-clockwise) and negative (clockwise) velocities. Fit this to two lines *(hint: see MATLAB command "polyfit")*, and note the static and dynamic frictions **in both directions** (they will probably differ). Write down the expression for $F(\dot{\theta}_r)$.


Now that we have characterized friction, we can explore a way of negating its effects on our system.

2.3   Friction Compensation for Velocity Control

Friction compensation is a popular topic. Friction affects all systems, and can add to or modify system dynamics, bring in noise, decrease resolution, and introduce offsets. We have implemented one method of dealing with offsets introduced by Coulomb friction – integral control. However, the dynamics of the system are still affected by friction. By considering friction as a linear function (a velocity gain and offset for each direction), we see another way of dealing with it. Since we now know the value of friction (in control units; see Equation (7) and the associated discussion on page 5) for any speed, we can let Simulink "adjust" for friction by counteracting its value as a function of speed (see Figure 5). (*Tip:* Keep in mind that a "Switch" block can be used to implement a conditional function.)

2-e   Implement friction compensation in Windows Target.[2] A proportional control should now regulate the system to (or very close to) the desired velocity. Is integral control still needed, or is proportional control sufficient? Observe the effects of PI control. Now, **remove your controller** and simply implement friction compensation. What do you expect will happen? Reason out what you expect to see, then manually start the motor spinning in either direction and see what happens. Try adjusting your dynamic friction gains and see how the behavior changes.

Friction compensation can do wonders for velocity control. Now that friction is well understood and accurately modeled, we can return to the overall Reaction Wheel Pendulum. First, we will do some System Identification and Model Verification, then finally delve into control.
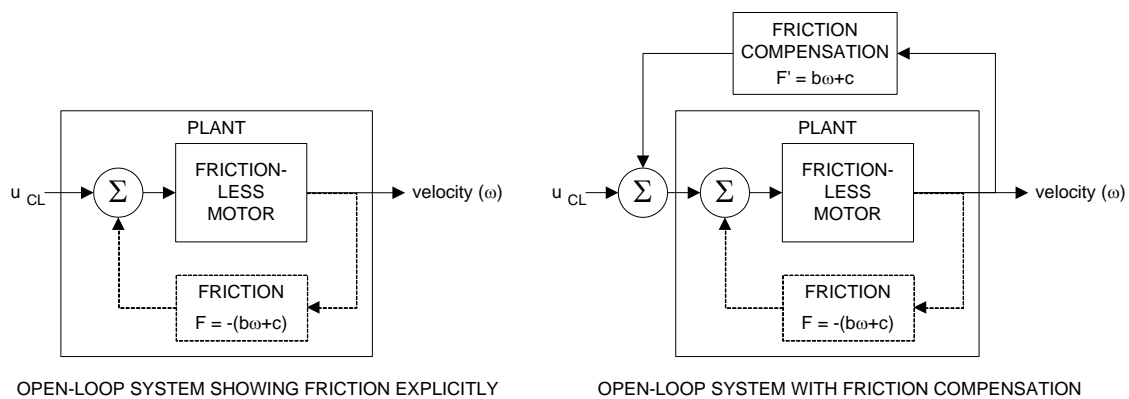
Figure 5: Functionality of Friction Compensation

---

[2] Typing **fricblocks** at the MATLAB prompt opens a block that calculates asymmetrical friction.

3   System Identification

We can now determine the parameters of the Reaction Wheel Pendulum (RWP). Here we set up the RWP in the standard configuration. The parameters can be determined from physical construction data and by direct experiments on the system. It is useful to combine both methods to find all the parameters, and to make cross-checks. It also verifies that our mathematical model is reasonable.

By measuring the dimensions of the components, weighing them, and computing moments of inertia using simplified formulas we find:

m_p	$= 0.2164$ kg
m_r	$= 0.0850$ kg
J_p	$= 2.233{\cdot}10^{-4}$ kgm^2
J_r	$= 2.495{\cdot}10^{-5}$ kgm^2
ℓ_p	$= 0.1173$ m
ℓ_r	$= 0.1270$ m

For you to find:
J	$=$	kgm^2
m	$=$	kg
ℓ	$=$	m
ω_{np}	$=$	rad/s
$\omega_{np}{}'$	$=$	rad/s
measured frequency of oscillation:		
$\omega_{np}{}'_{meas} =$		rad/s

3-a   Use the relations and definitions given in Section 1.2 to get values for J, m, ℓ, and ω_{np}. Also find ω_{np}'. (Defined by Equation (17) below)

In order to verify the natural frequency, we can do a free swing test (below).

3.1   Checking the Harmonic Frequency

3-b   Set the motor input u to zero. Initialize the pendulum to 90° and let it swing freely. When the wheel stays stuck to the pendulum, i.e. the encoder reading φ_r is constant, determine the frequency of oscillation ($\omega_{np}{}'_{meas}$).

This measured frequency is different from ω_p because the rotor is contributing to the moment of inertia. The quantity you measured is actually

$$\omega_{np}{}' = \sqrt{\frac{mg\ell}{J + J_r}} \tag{17}$$

Compare the experimental value with the theoretical value, computed from the parameters (all known).

Notice also that the decay in the swing amplitude is slow. On the other hand, if the rotor is excited with the maximum current, and then the current is removed, it takes only a

few seconds to come to complete rest. In both cases, friction is the only deceleration force (for the pendulum, consider conservation of energy and for the rotor, apply Newton's first law of motion). This helps to validate our assumption that the friction in the pendulum link is negligible, but friction in the motor is not.

3.2 Determination of Torque Constant (k_u)

We must consider one implementation detail: the signs on the inputs and outputs.

> 3-c   Verify the signs:
>
> ① With NO input to the motor, check that the sign conventions on rotor and pendulum angle are consistent with Figure 2. If not, add –1 signs where necessary.
>
> ② Apply a POSITIVE input to the motor to check that the rotor velocity and INITIAL pendulum velocity are consistent with Equation (9), ignoring friction. If not, add –1 gains where necessary.

If any –1 signs are necessary, these are purely an implementation detail, and should *not* be considered as part of your controller. They will be absolutely critical however, and if you forget them, they can result in total instability of a controller that should be stabilizing.

With that taken care of, there are two parameters that we still don't know. These are b_p and b_r. By examining Equation (18), we see a way to find b_p and b_r.

$$\begin{cases} \ddot{\theta}_p + a\sin\theta_p = -b_p(u + F(\dot{\theta}_r)) \\ \ddot{\theta}_r = b_r(u + F(\dot{\theta}_r)) \end{cases} \tag{18}$$

Our sensors directly measure θ_p, θ_r. We developed an approximation for $\dot{\theta}_p$ and $\dot{\theta}_r$ in section 2.1. If we estimate $\ddot{\theta}_p$ and $\ddot{\theta}_r$ by differentiating again and use our friction model from section 2.3 to replace $F(\dot{\theta}_r)$, the only unknowns in the top equation of (18) are b_p and b_r. Solving for them is trivial, assuming we can find the second derivative. Alas, Figure 6 shows that the first derivative is noisy, and the second derivative is worthless. A better approach would find a polynomial fit for θ_p, θ_r, and differentiate this fit to get clean values for $\dot{\theta}_p$, $\dot{\theta}_r$, $\ddot{\theta}_p$ and $\ddot{\theta}_r$. These clean values can then be put into Equation (18) to solve for the torque constants. In practice, a cubic fit of the RWP response to a step input from rest gives good results.
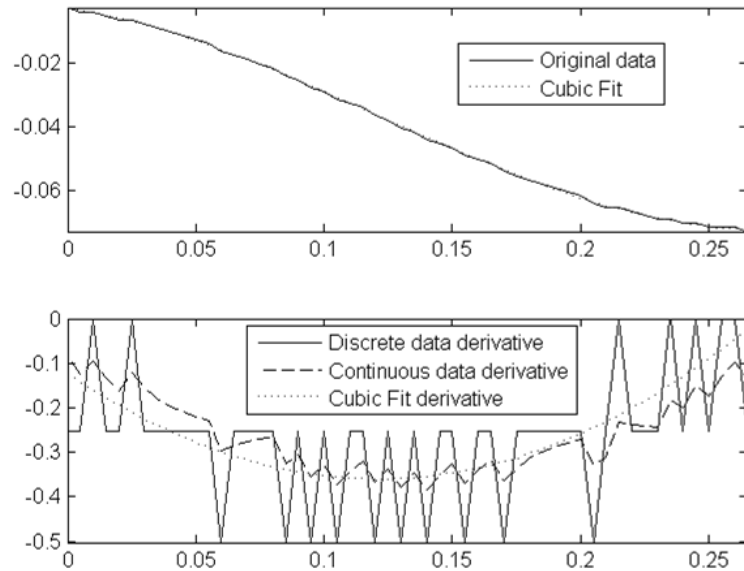
Figure 6: Derivative approximations add significant noise

3-d   Use the method described above to find b_p and b_r. A partial m-file is provided for you on the lab website. Use a step of magnitude 5 for u.

Another way to determine b_p and b_r (and thus k_u), uses equation (7). The i is determined for the maximum input u of 10. Properties of the motor and controller tell us the value of i_{max} and k, giving

$b_p = 1.08$

$b_r = 198$

Your results should approximately agree (within 30%).

4 Stabilizing the Inverted Reaction Wheel Pendulum

4.1 Linearization and Controllability

The Reaction Wheel Pendulum (RWP) has equations of motion, ignoring friction, given by

$$\begin{cases} \ddot{\theta}_p + a\sin\theta_p = -b_p u \\ \ddot{\theta}_r \qquad\qquad = \;\; b_r u \end{cases} \tag{19}$$

4-a   Linearize this system about the equilibrium position of $\theta_p = \pi$. Write the state-space model for this system in the blanks provided in Equation (20) and check for controllability from the single input *u*. *Note:* Your new state variables are delta-angles, where $\delta\theta_p = \theta_p - \pi$ and $\delta\theta_r = \theta_r$.

$$\dot{\mathbf{x}} \quad = \qquad \mathbf{A} \qquad \mathbf{x} \; + \; \mathbf{B}\,u$$

$$\begin{bmatrix} \\ \\ \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} \delta\theta_p \\ \dot{\theta}_p \\ \delta\theta_r \\ \dot{\theta}_r \end{bmatrix} + \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix} u \tag{20}$$

The system should be controllable; otherwise we would need to add another actuator to be able to complete the project.

4.2 Inverted Stabilization Using Two-State Feedback

In the interest of keeping our controller as simple as possible, and because we don't care about the position of the rotor, let us first design a PD controller to regulate the pendulum angle, completely ignoring the rotor angle and velocity. Here we are only considering the first equation of Equation (19) – so we actually have a 2nd order system.

> 4-b   Design a state-feedback controller for the RWP using the MATLAB command *place*. Constraints for θ_p: keep $\omega_{np} > \omega'_{np}$ (ω_{np} as found in Section 3) – do you know why it must be greater? Make $\zeta < {}^1/\sqrt{2}$, and keep the K values less than 300. *(Hint: meet the last constraint by trial and error.)* Simulate your system using the nonlinear "Reaction Wheel Block Diagram Model" and "Reaction Wheel Animation" blocks – see Appendix B (don't bother estimating velocity; just use the exact states). Simulate ① IC deviation ($\delta\theta_p$ or $\delta\dot{\theta}_p$ nonzero), ② a pulse (simulating a tap) disturbance input to the pendulum arm (τ_p) with duty cycle of 5% and period of 4 seconds, and ③ a constant disturbance input to the pendulum arm. Is the response satisfactory (i.e. stable and fairly fast)? Now look at rotor velocity. Do you see any problems?

Table 1: Robustness Comparisons

	Two-State Feedback (4.2)		Three-State Feedback (4.3)		Observer (5.1)	
① Max IC deviations	$\delta\theta_p$	$\delta\dot{\theta}_p$				
② Max pulse						
③ Max disturbance						

Note: Remember that the controller uses delta states, whereas the nonlinear model outputs absolute states. You will need to remove the offset(s) accordingly. (*Comment*: In order to fill out the first row in the above table, you need to test IC deviations for both $\delta\theta_p$ and $\delta\dot{\theta}_p$. But you only test one at a time, i.e., for example, when you test IC deviation, add small deviation to $\delta\theta_p$ but keep zero deviation for $\delta\dot{\theta}_p$. Then you test $\delta\dot{\theta}_p$ similarly.)

As you can see, the rotor velocity stays constant at steady-state without any disturbances. However, with a constant disturbance, however small, the motor undergoes a constant acceleration to counteract it, which causes the velocity to increase without bound. Since we have a bound on velocity (there is *always* a bound on velocity!), this is not practical for implementation. Therefore we must feedback the rotor velocity information.

This may raise a question: In simulation, if a constant u can cause velocity to become arbitrarily large, why can't that happen in our system? Because as velocity increases, friction increases, and (u-F) decreases until friction effectively "cancels out" u!

4.3   Inverted Stabilization Using Three-State Feedback

Consider the eigenvalues of the 4-state state-space model you found in Question 4-a. The zero eigenvalues represent the rotor position and velocity. Our goal is to pull the velocity eigenvalue into the LHP, but leave the position eigenvalue alone. This will **stabilize** the rotor velocity, while still **ignoring** its position.

4-c   Using the same constraints as 4-b, design a state feedback controller with θ_p, $\dot{\theta}_p$ and $\dot{\theta}_r$ feedback. Place the $\dot{\theta}_r$ eigenvalue between the other two LHP poles. *(Hint:* MATLAB *"place" or "acker" may work – just keep the fourth pole at zero. This makes the fact that we're ignoring θ_r evident.)* Simulate conditions ①, ②, and ③ from 4-b again. Record in Table 1 the maximum IC deviation that the system can correct, as well as the maximum tap. Use Windows Target to implement this controller on the actual RWP.

If the RWP is too sensitive to be positioned and doesn't stay up very long, check the rotor velocity. If the RWP is spinning up to a high velocity before falling, then your $\dot{\theta}_r$ feedback gain may be too small. Adjust the gains until you get a satisfactory response, then show it to your TA. When the RWP is successfully stabilized, you should see limit cycle behavior[3] .

4-d   Include friction compensation (which you designed in Question 2-e) and observe the change in behavior. Demonstrate it to your TA.

We will next explore the observer's approach.

---

[3] Persistent (but not necessarily precisely) repeating behavior that does not die out is called *Limit Cycle Behavior*.

5   Observer Design

We will now design an observer for the Reaction Wheel Pendulum (RWP) to replace the full state feedback controller we designed previously. The observer will estimate both velocities of the system. And since we're designing a full-order observer, it will also "estimate" both positions.

Look back at your full-state feedback design; you pulled all of the open-loop poles except the θ_r pole into the left half plane. When we design an observer, however, we must place *all* of the poles in the left half plane; our criteria being, as before: "significantly farther" than the desired closed-loop poles. See Figure 7 for an illustration of this (not to scale, and relative pole locations may vary by design).
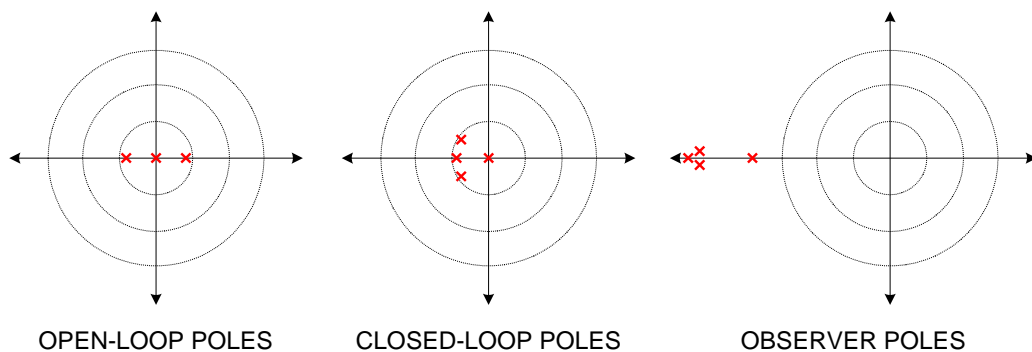

Figure 7: Illustration of Pole Locations

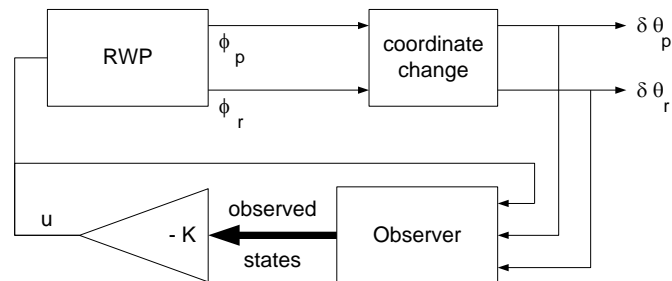## 5.1   Observing Four States Together

Figure 8: Block Diagram of System with Observer

Figure 8 shows the structure of our closed-loop system with observer-based control. In Simulink, the observer block can be modeled as a State-Space block. However, in order to do so, we need to define the **A**, **B**, **C**, and **D** matrices (where the state equation is defined as $\{\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{B}u, \mathbf{y} = \mathbf{Cx} + \mathbf{D}u\}$).

The standard differential equation for an observer is

$$\dot{\hat{\mathbf{x}}} = \mathbf{A}\hat{\mathbf{x}} + \mathbf{B}u + \mathbf{L}(\mathbf{y} - \hat{\mathbf{y}})$$

$$\hat{\mathbf{y}} = \mathbf{C}\hat{\mathbf{x}} + \mathbf{D}u$$

$$(21)$$

Keeping in mind that we want $\hat{\mathbf{x}}$ as the states, $\hat{\mathbf{x}}$ as the outputs, and both delta-angles (collectively called $\mathbf{y}$, where $\mathbf{y} = [\delta\theta_p , \delta\theta_r]^{\mathrm{T}}$) as well as u as the input, Equation (21) can be manipulated to give

$$\begin{cases} \dot{\hat{\mathbf{x}}} = (\mathbf{A} - \mathbf{LC})\hat{\mathbf{x}} + [\mathbf{B}\ \mathbf{L}]\begin{bmatrix} u \\ \mathbf{y} \end{bmatrix} \\ \qquad\quad \mathbf{z} = \mathbf{I_4}\hat{\mathbf{x}} \end{cases}$$

$$(22)$$

where $\mathbf{z}$ represents the output of the observer. **You should be comfortable going from Equation (21) to Equation (22)**. (*Hint:* when doing matrix algebra, always check dimensions!)

In particular, what is $\mathbf{C}$? From Equation (21), we see that $\mathbf{y}$ and $\hat{\mathbf{y}}$ must have the same dimension, and, in order for their difference to be meaningful, must represent the same physical phenomena (e.g., subtracting a velocity from an angle is meaningless).

since $\mathbf{y} = \mathbf{Cx},\qquad \mathbf{C} = \begin{bmatrix} & & \\ & & \end{bmatrix}$ *(fill in your **C** matrix)*

Also, why $\mathbf{I_4}$? ($\mathbf{I_4}$ represents the 4×4 identity matrix) Because we want to output all of our states **individually**. If we used a scalar z and defined $z = \hat{x}_1 + \hat{x}_2 + \hat{x}_3 + \hat{x}_4$, then instead of having more information from the observer, we would actually have less!

5-a   Using the MATLAB *place* command, place the observer poles significantly farther than your closed-loop poles (as designed in Section 4.3). Five to ten times faster is a good distance. Keep these poles near or on the real axis. Also note that the 'place' command cannot solve for repeated roots. Check the full system's eigenvalues using MATLAB *eig* to make sure they are stable.

5-b   Simulate your observer design in Simulink. Test and record the same things you tested in Question 4-b. How does this controller compare to the three-state feedback controller? Now vary the nonlinear model's parameters slightly. Does the controller still work?

The extremely high sensitivity of this controller to variations in the plant may surprise you, but in the next section we will explore the RWP model in more depth in order to understand why this is the case, and find a way to modify your controller in order to make it less sensitive to plant variations.

5.2   Decoupling and Redesigning the Observer

Recall that the state-space model for our system has the form

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ a & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ -b_p \\ 0 \\ b_r \end{bmatrix} u \tag{23}$$

We see that $\mathbf{A}$ has a specific form – the top right four values and the bottom left four values are zero. There is a special term for that form – *block diagonal*. Let us take a small digression and explore the implications of $\mathbf{A}$ being in block diagonal form. If we rewrite the system as

$$\begin{bmatrix} \dot{\mathbf{x}}_{1,2} \\ \dot{\mathbf{x}}_{3,4} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{N} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{1,2} \\ \mathbf{x}_{3,4} \end{bmatrix} + \begin{bmatrix} \mathbf{P} \\ \mathbf{Q} \end{bmatrix} u \tag{24}$$

where the "elements" of these vectors and matrices are now vectors and matrices themselves, this representation of $\mathbf{A}$ makes $\mathbf{A}$ look like a diagonal matrix. Separating the two vector equations, we get

$$\begin{cases} \dot{\mathbf{x}}_{1,2} = \mathbf{M}\mathbf{x}_{1,2} + \mathbf{P}u \\ \dot{\mathbf{x}}_{3,4} = \mathbf{N}\mathbf{x}_{3,4} + \mathbf{Q}u \end{cases} \tag{25}$$

And then writing each vector equation as a system,

$$\begin{cases} \dot{\mathbf{x}}_{1,2} = \begin{bmatrix} 0 & 1 \\ a & 0 \end{bmatrix} \mathbf{x}_{1,2} + \begin{bmatrix} 0 \\ -b_p \end{bmatrix} u, & \mathbf{C}_{1,2} = \begin{bmatrix} \quad \end{bmatrix} \\ \dot{\mathbf{x}}_{3,4} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x}_{3,4} + \begin{bmatrix} 0 \\ b_r \end{bmatrix} u, & \mathbf{C}_{3,4} = \begin{bmatrix} \quad \end{bmatrix} \end{cases} \tag{26}$$

This is quite significant. It says that the dynamics of these subsystems are *decoupled*, or independent of each other. This is a direct result of $\mathbf{A}$ being in block diagonal form. (Note that this does *not* mean that you can control the dynamics of both arbitrarily – the same input u applies to both.) Another implication is that the eigenvalues of $\mathbf{A}$ are the union of the eigenvalues of $\mathbf{M}$ and the eigenvalues of $\mathbf{N}$.

This last fact can be used to our advantage. We wish to make the observer response converge very quickly, and therefore want to set the eigenvalues of $(\mathbf{A}\text{-}\mathbf{LC})$ to be fast. Remember that we're now designing the internal dynamics of the observer; the input u is not applied here. Here is the breakdown of $(\mathbf{A}\text{-}\mathbf{LC})$:

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} l_{11} & l_{12} \\ l_{21} & l_{22} \\ l_{31} & l_{32} \\ l_{41} & l_{42} \end{bmatrix}, \quad (\mathbf{A}-\mathbf{LC}) = \begin{bmatrix} -l_{11} & 1 & -l_{12} & 0 \\ a-l_{21} & 0 & -l_{22} & 0 \\ -l_{31} & 0 & -l_{32} & 1 \\ -l_{41} & 0 & -l_{42} & 0 \end{bmatrix} \tag{27}$$

So the first and third columns can be modified. Must the values that are not on the block diagonal be nonzero? For example, look at l_{12}. It determines the effect of $\hat{x}_3$ on $\dot{\hat{x}}_1$. But since the dynamics of the first two states are decoupled from the last two, that term is unnecessary. In other words, we can arbitrarily determine the dynamics of the observer using only l_{11}, l_{21}, l_{32}, and l_{42}! In fact, the other terms only serve to make the observer more sensitive to model errors and noise.

If the dynamics are independent, then why did MATLAB give nonzero off-diagonal terms? Answer: the algorithms MATLAB uses do not check for independence.

5-c   Split up the system as shown in Equation (26), and use MATLAB to find the **L** matrix. Compare it with the **L** matrix found in Question 5-a. Now take the **L** matrix found in Question 5-a and zero out the terms off the block diagonal. Compare again. Is it important to split up and redesign, or do you find it sufficient just to zero out the terms off the block diagonal? *(Hint: Look at their effects on the observer eigenvalues.)* Repeat Question 5-b with the new **L** matrix. Any improvement?

Now we have a good design, and are ready to put it to work on the actual RWP.

5-d   Implement your observer design with Windows Target on the RWP. How does this controller compare to three-state feedback control?

Sure enough, this design does not yield a controller as robust as the full-state feedback controller. However, this demonstrates the tradeoff between performance and design time in real-world engineering. Although the full state feedback design (including derivative approximations, etc.) may have been "easier" for you, in general the observer is easier to design because the design process is methodical.

6   Up and Down Stabilizing Control

We will now explore the topic of *switching control*. We will first discuss this topic, and then consider the example of the RWP, which we are quite familiar with. We can then design a switching controller for the RWP without much trouble.

Most systems that we control are nonlinear. We simply choose an operating condition and linearize about that condition. However, what if we want to control this system over a broader range of operating conditions? For example, airplanes are extremely nonlinear systems. Fighter jets are even more so, due to the enormous range of airspeeds and maneuverability requirements. A nonlinear controller would be extremely complicated and may even become unstable near the extremes, due to modeling errors. The approach used is to switch between many different linear controllers based on the states. Each controller uses a different model for the system, and applies a different type of controller, but all share the broad goal of keeping the jet in the sky[4]. The difficult aspect of switching control is handling the switching transients: When switching from one model and controller to a completely different one, how do you guarantee that the system won't go unstable?

To explore this further, let us consider again the RWP. Unlike a fighter jet, a pendulum has only two equilibrium points: up and down. We have designed a controller to balance up. If we make a switching control to balance down when the pendulum swings past the upward stabilizable region, will we be able to guarantee stability? Keeping in mind that the downward equilibrium is a stable equilibrium, it is not rocket science to determine that after the switching transients, the down controller will be able to stabilize the pendulum.
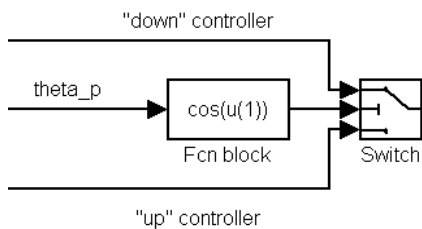
Figure 9: Implementing Switching Control

6-a   Design a switching controller that will stabilize the RWP in either the up or down position based on the pendulum angle θ_p. To make your life easier, use three-state feedback controllers as in Section 4.3. *(Hint: use the "Switch" block controlled by a function of θ_p—see figure above.)* First simulate it, then implement it using Windows Target.

---

[4] Controlling a nonlinear plant by switching between a family of linear controllers, each tuned for certain operating conditions, is called *gain scheduling*

If you don't see the effects of the controller in the down position, swing the pendulum freely and see how long it takes for the swinging to stop. There should be a significant decrease in that time with your new controller.

7   Swing-Up Control

After all that talk of avoiding nonlinear control by using switching control, let us now look at nonlinear control itself. It can be a very useful tool, especially when it is used along with other control algorithms. We can use switching control to switch between a nonlinear controller and a linear controller. This may sound complicated, but actually "switching control" is nothing more than an algorithm that switches between multiple controllers. We can use a nonlinear controller to get the system into the region that is stabilizable with linear control, and then switch over to the linear controller. That is the approach we will use to swing-up the pendulum and then stabilize it at the top position.

The concept of nonlinear control may sound daunting, but look at the problem in this way: how can we pump energy into this system properly, and how can we get the system to recognize that it is in the "region" of stabilizability? A clue lies inside that question: energy. We can measure kinetic and potential energy, we want a certain setting of kinetic and potential energy, and we can apply kinetic energy.

Let us look at this from a naïve point of view. Assume that we want to tap the pendulum really hard at the bottom, but just hard enough to get it to swing up and come to rest (briefly, of course) at the top. **(Do not try this! The RWP is fragile.)** You can imagine tapping it harder or softer based on how high up it swings. Is there a way to figure out just how hard you need to tap it? Yes! The energy at the bottom is purely kinetic, and you want the energy at the top to be purely potential. Therefore, you can compute how intensely you must tap.
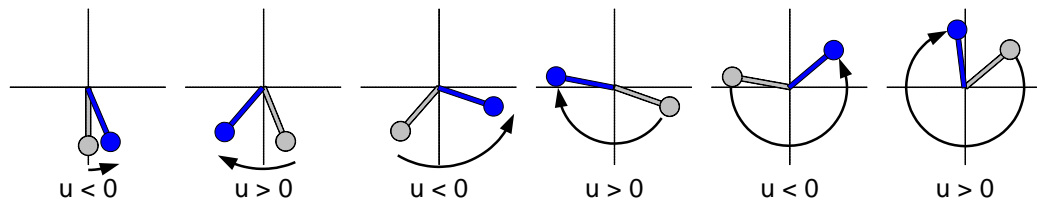
Figure 10: Illustration of Nonlinear Swingup Control.

Light grey is the starting position, black is the final position after applying u.

On the actual RWP, you have the added advantage that the motor is mounted on the pendulum, so you do not need to tap. Rather, you can apply a long-term force (see Figure 10). But there is a disadvantage to this: the force is limited by the maximum velocity of the rotor (as we have seen before). It turns out that the motor cannot provide the necessary energy input in a single swing. Therefore, the motor must dump some energy during one swing, then dump energy in the other direction during swingback, and so on until the RWP has the correct amount of energy. Figure 11 shows a plot of the intersection pendulum energy and the ideal energy when the pendulum is balanced in an inverted position

There is, of course, the added complication of friction. To account for this, we can simply dump in more energy than required without friction and hope it works. This process takes much trial and error.

This is only one of many methods of doing swing-up control. There are many implementation details involved in making a swing-up controller. Have fun!

Appendix A: Useful Physics Theory

Conversions (units of MKS)

- Length (m), Mass (kg), and Time (s) are basic units.
- Force is mass • acceleration, and has units of newtons ($N = kg•m/s^2$).
- Energy is force applied over a certain distance, and has units of joules ($J = N•m = kg•m^2/s^2$).
- Power is an impulse of energy, with units of watts ($W = J/s = N•m/s$).
- Inertia is the change in force required to make a unit change in acceleration. It has units of change in force per change in acceleration, or $N/(m/s^2)$.
- Moment of inertia is the analog of inertia for rotational objects. It is the change in torque required to make a unit change in angular acceleration. It has units of change in torque per change in angular acceleration, or $Nm/(rad/s^2) = kg•m^2$.

Note: (rad) is considered unitless

Energy Equations

Potential energy …

… for a mass	$= mgh$
… for a spring	$= ½\,kx^2$

Kinetic energy …

… for a mass	$= ½\,mv^2$
… for a moment of inertia	$= ½\,J\omega^2$
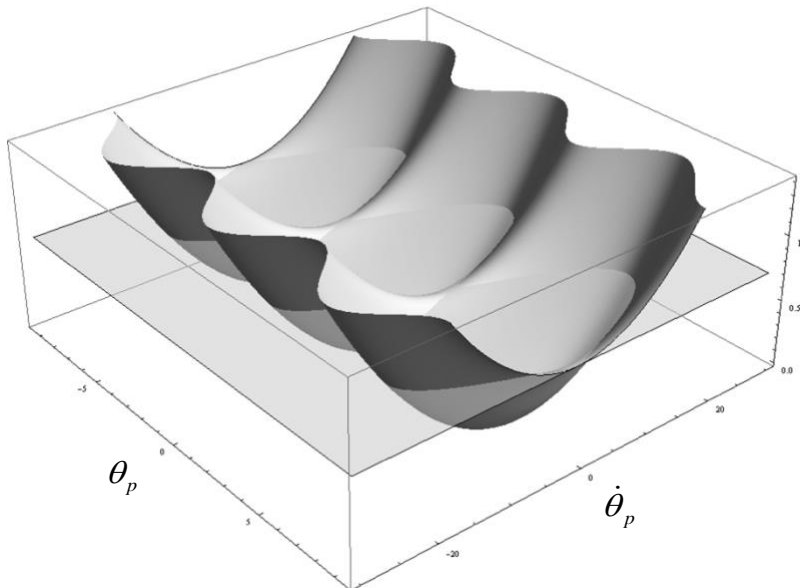
Figure 11: Pendulum Energy (surface) intersected by ideal energy (plane)

Appendix B: Implementation Notes

This appendix contains many details that will be critical during simulation and actual control. Keep this nearby, and refer to it often.

Simulink Notes

- *To start Simulink:* First open MATLAB. Then either type `simulink` in the command window, or click on the Simulink button (enlarged below) on the toolbar.

- *Setting up Simulink parameters:* When running a Simulink simulation, you need to set up a few parameters in order to keep conformity with Windows Target. From the **Simulation** menu, select **Configuration Parameters**, then in the **Solver Options** box, set Type to "Fixed-step" and "ode1: Euler".

- *Changing simulation Start/Stop time:* You can also change Start Time and Stop Time in the Simulation Parameters box.

- *Nonlinear RWP model:* There is a nonlinear RWP model available for simulation. To find it, type `pend_blks` in the MATLAB command window. You will need the Reaction Wheel Block Diagram Model (nonlinear model) and the Reaction Wheel Animation (animation block). Points to remember:

  - For the Reaction Wheel Block Diagram Model, "Tau1" corresponds to the pendulum arm. (That arm is not actuated, but that's where the disturbances are applied.)

  - "Tau2" corresponds to the rotor. (That's where the control effort should be applied.)

  - Don't forget to include a Saturation block (to saturate the control effort at ± 10) before "Tau2".

  - The nonlinear model outputs encoder states (φ_p, φ_r), but your controller uses delta states ($\delta\theta_p$ instead of θ_p).

  - You can change the initial conditions of the Reaction Wheel Block Diagram Model by double-clicking on the block. You should only have to change the first initial condition (pendulum position, where π signifies upwards).

  - To slow down animation speed, go to **Simulation** » **Configuration Parameters**, and decrease the "Max step size". (This will force Simulink to do more calculations, thus slowing down the simulation.)
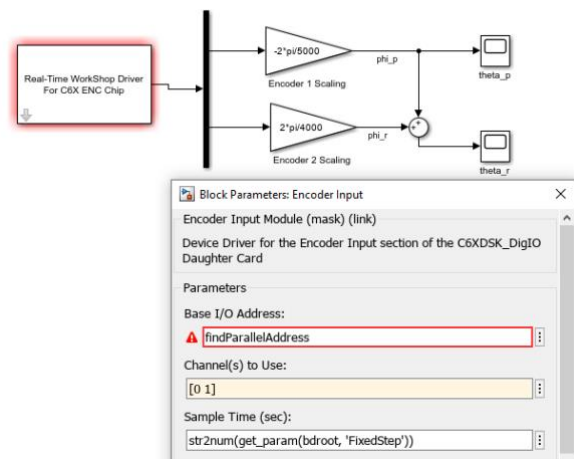
Windows Target Notes

- *To make a Windows Target model:* At the MATLAB prompt, type `rtwintgt_starter`. Delete the PCIINT32_DAQ block (we will use different I/O blocks for the RWP) and save the model in your c:/ directory with a filename 22 characters or less. Make sure to change MATLAB's current to your directory where you just saved the file. Then go to the **Simulation** menu, select **Configuration Parameters**, and make sure the **Fixed-step size** is 0.005. This sets the sample rate of your controller.

- *To interface to the RWP:* Blocks are available for you which make interfacing with the RWP transparent. To access them, type `c6xlib` at the MATLAB command window. The "Encoder Input" block outputs encoder ticks, which must be converted into angles. The "PWM Output" block takes an input of command units (in control units – see Equation (7) and the associated discussion on page 5), and automatically saturates at ± 10.

- *Encoder Input block details:* The system has two encoder values. We will only use one Encoder block for both data channels. The encoder block can output either channel 0 or 1, or both. To access both, use the *Demux* block at the Encoder block output. The upper output of the Demux is channel 0 and the bottom channel is channel 1. See Table 2.

Table 2: Encoder settings

Motor encoder is Encoder 2 (channel 1)
Pendulum encoder is Encoder 1 (channel 0)

Also see **Error! Reference source not found.** for more clarification. (If you simply enter 0 in the "Channel(s) to Use:" box, only channel 0 of the data will be output.) Notice that here, only the pendulum encoder output needs to be negated, whereas in Section 2, the motor encoder needed to be negated. This has to do with the orientation of the encoders on the RWP and our designation of positive angles. Generally, encoders designate positive as the clockwise direction when looking towards the encoder along the shaft.

Figure 12: Reading Encoders

- *To run a Windows Target model:*
  - First step is to compile (or build) your real-time controller. From the **Tools->Real-Time Workshop** menu, select **Build Model**. Or you can click anywhere in the Simulink window and type "Ctrl-B" and this will also build your real-time controller. Before going to the next step, make sure your real-time controller has been completely built by watching the MATLAB command window for the message "Sucessful Completion of the Real-Time Workshop Build"

  - To setup your scope blocks for plotting and data storage double click on the Simulink scope blocks to open them. Right click in the plot area to set the data's Y range. Click the scope window's "Parameters" button to set the scopes time range. In the "Data History" tab make sure "Limit data points" is **unchecked** and "Save date to workspace" is **checked**. Assign a "Variable name" and change the format of the data to type "Array".

  - (*Comment*: This note is outdated, probably valid for very old Matlab. In order to run Windows Target with newer Matlab and Simulink, you need to follow the notes in your lab book. See notes in Lab 5.) There are two steps to start your real-time controller. First you must "Connect to Target" or in other words load your real-time controller to the Windows Target run engine. Select the **Simulation->Connect to Target** menu item or click anywhere in your simulink model and type "Ctrl-T" or click the "Connect to Target" icon just to the right of the "Start" icon. After connecting to the target click the "Start" icon to start your real-time controller.

  - To stop your real-time controller click the "Stop" icon
  .

Note: If you make simple gain changes or aesthetic changes, you need not re-"Build" the code. However, if you change anything else, such as connections between blocks, block parameters, simulation parameters, etc, you will need to rebuild. If you don't, you will either get an error message, or see that the response didn't change since your last revision.