This document provides an overview of the **FullStateFeedback** class and its subclasses. It models a system by integrating a nonlinear plant, a Kalman filter, and full-state feedback, allowing you to design and evaluate system behavior under disturbances.

Like an abstract class in C#, **FullStateFeedback** class is not meant to be instantiated directly. It includes essential methods for setting full-state feedback gains and running simulations. All subclasses, such as the **InvertedPendulumOnWheel** class, share a common set of methods, functioning like abstract methods inherited from **FullStateFeedback** class.

I will first introduce the methods in the **FullStateFeedback** class so that you can simulate the system I created. Then, I will explain the methods in its subclasses, allowing you to develop your own custom systems. Finally, I will discuss the initialization of the subclasses I have written. The derivation of the system is documented in "controlSystemAndModels.pdf".

FullStateFeedback.CtrbObsv

FullStateFeedback.CtrbObsv(self)

Print the rank of the controllability matrix and the rank of the observability matrix.

Parameters:

None

Returns:

None

FullStateFeedback.LQR

FullStateFeedback.LQR(self, QW, RW)

Calculate the full-state feedback gain **Kr** using the LQR method and assign this value to the instance's **Kr**. The LQR method is sourced from the control module and is implemented using control.lqr(self.A,self.B,QW,RW). It will also print the resulting eigenvalue of (A-B@Kr).

Parameters:

QW: array_like or float

State and input weight matrices. If provided as an array, it must be a diagonal matrix with a shape of (x, x), where x is the number of states.

RW: array_like or float

State and input weight matrices. If provided as an array, it must be a diagonal matrix with a shape of (x, x), where x is the number of inputs.

Returns:

None

FullStateFeedback.PolePlacement

FullStateFeedback.PolePlacement (self, poles)

Calculate the full-state feedback gain **Kr** using the pole placement method and assign this value to the instance's **Kr**. The pole placement method is sourced from the control module and is implemented using control.place(self.A,self.B,poles).

Parameters:

poles: array_like

Desired eigenvalue locations. It must be a one-dimensional array with a length equal to the number of states.

Returns:

None

FullStateFeedback.SetKr

FullStateFeedback.SetKr(self, Kr)

Assign the Kr value directly. It will also print the resulting eigenvalue of (A-B@Kr).

Parameters:

Kr: array_like

Desired Kr value. It must have a shape of (1, x), where x represents the number of states.

Returns:

None

FullStateFeedback.SimulationSimple

FullStateFeedback.SimulationSimple (self, interval=0.001, duration=4, initialState=None)

Simulate $x_{k+1} = x_k + (A - B\ Kr)x_k \times \Delta t$ and export the state at each moment throughout the duration. x_{k+1} is the state at time k+1. x_k is the state at time k. Δt is the interval

Parameters:

interval: float

The interval between each discrete step. The default is 0.001 second.

duration: float

The duration of the simulation. The default is 4 seconds.

initialState: array_like

The initial state of simulation. If None, then the initial state will be all zeros. It must have a shape of (x, 1), where x represents the number of states.

Returns:

timeVector: array_like

The array of times at each moment, with a shape of (round(duration/interval),).

outputState: array_like

The array of states at each moment, with a shape of (round(duration/interval), x), where x represents the number of states.

FullStateFeedback.Simulation

FullStateFeedback.Simulation(self, duration=4, outputInterval=1/25, feedbackInterval=0.002, phyiscInterval=0.0005, initialState=None, QN=None, RN=None, fullState=False, estimateBias=True, setPoint=0, acceleration=5, delay=0.000, event=[], animation=False, windowH=500, windowW=500, imageScale=1000)

Simulate the system if the motor is a stepper motor. You can introduce events during the simulation and display the animation to visualize the system's behavior.

Parameters:

duration: float

The duration of the simulation. The default is 4 seconds.

outputInterval: float

The time interval between each frame. The default is 0.001 second.

feedbackInterval: float

The time interval between each time micro-computer calculating the feedback. The default is 0.002 second.

physicInterval: float

The time interval between each update of the nonlinear system. The default is 0.0005 seconds.

initialState:array_like

The initial state of simulation. If None, then the initial state will be all zeros. It must have a shape of (x, 1), where x represents the number of states.

QN: array_like or float

State and input weight matrices. If provided as an array, it must be a diagonal matrix with a shape of **(x, x)**, where **x** is the number of states. If it’s a float number, the value will be broadcast to all the state. The default is a diagonal matrix. If the state is not a bias, the value is 0.00001; otherwise, it is 0.000001. Note that the given value represents the variance of the total disturbance accumulated over one second. The matrix used in the Kalman filter is this matrix multiplied by the feedback interval.

QN_noise: array_like or float

Covariance of disturbance noise. If provided as an array, it must be a diagonal matrix with a shape of **(x, x)**, where **x** is the number of states. If it’s a float number, the value will be broadcast to all the state. The default is the same as QN.

RN: array_like or float

State and input weight matrices. If provided as an array, it must be a diagonal matrix with a shape of **(x, x)**, where **x** is the number of measurements. If it's a float number, the value will be broadcast to all the output. The default is a diagonal matrix and the value is 0.00001.

RN_noise: array_like or float

Covariance of noise detection. If provided as an array, it must be a diagonal matrix with a shape of **(x, x)**, where **x** is the number of states. If it's a float number, the value will be broadcast to all the state. The default is the same as RN.

fullState: bool

If True, it bypasses the Kalman filter and uses the full state directly.

estimateBias: bool

If False, the Kalman filter will not estimate bias state.

setpoint: float

The setpoint for one of the states, which varies depending on the specific subclass. The default is 0.

acceleration: float

The acceleration of stepper motor. The default is 5 1/s^2.

delay: float

The delay between computing the input and its effect taking place. The default is 0 seconds.

event: list

Each element in the list must be a tuple in the format **(time, "key", value)**:

- **time**: A float representing the event timing.
- **key**: A string indicating the type of event. The available options are **"setpoint"** and **"disturbance"**.

- If **"setpoint"**, **value** must be a float, which sets the setpoint to the given value.
- If **"disturbance"**, **value** must be a 1D array with a shape of **(x,1)**, where **x** is the number of states. The **value** will then be added to the current state.

animation: bool

If True, display the animation.

windowH: float

The pixel height of animation window. The default is 240 pixels.

windowW: float

The pixel width of animation window. The default is 427 pixels.

imageScale: float

The pixel length of one meter in the animation. The default is 1000 pixels.

playbackSpeed: float

Playback speed of animation. The default is 1.

exportFileName: string

The file name of exported animation. The only supported file extension is gif.

Returns:

timeVector: array_like

The array of times at each moment, with a shape of (round(duration/outputInterval),).

outputState: array_like

The array of states at each moment, with a shape of

(round(duration/outputInterval), x), where x represents the number of states.

outputStateHat: array_like

The array of estimated states at each moment, with a shape of (round(duration/outputInterval), x), where x represents the number of states.

outputTorque: array_like

The array of average torque at each moment, with a shape of (round(duration/outputInterval)).

Notes:

Here's a deeper look into how I designed the simulation:

Each time the microcomputer measures the output, the data is processed through a Kalman filter. The estimated state is then multiplied by the LQR gain to determine the input. However, since a stepper motor only supports **set speed** or **set position** functions, we cannot directly apply torque as an input.

To address this, I use the nonlinear system to calculate the target motor speed for the next measurement cycle. The microcomputer then updates the motor speed to match this target speed. The simulation assumes that when the stepper motor is set to a target speed, it accelerates at a linear rate. Based on observations, if the acceleration is fast enough for the motor to reach the target speed, the system behaves as if the desired torque were being applied.

The following are the methods in the subclasses.

Subclass.State

Subclass.State(self)

Print the state, input and output of the system.

Parameters:

None

Returns:

None

Subclass.Ud

Subclass.Ud(self, setpoint)

The required value u_d ensures that when the input is $u = -Kx + u_d \times setpoint$, a specific state approaches the setPoint.

Parameters:

setpoint: float

Desired setpoint value.

Returns:

ud: float

the u_d value

Subclass.CurrentSpeed

Subclass.CurrentSpeed(self, x)

Print the current motor speed.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the number of states.

Returns:

speed: float

the current motor speed

Subclass.AccFromU

Subclass.State(self, x, u)

The acceleration on the motor if input u is applied to system.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the number of states.

u: float

The input to the system.

Returns:

acc: float

The acceleration on the motor.

Subclass.TorqueFromAcc

Subclass.TorqueFromAcc(self, x, acc)

The torque applied to the system if the motor accelerates at the rate of acc.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the number of states.

acc: float

Acceleration of motor.

Returns:

torque: float

The torque applied on the system by motor.

Subclass.dXdt

Subclass.dXdt(self, x, u)

The time derivative of the state.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the number of states.

u: float

The input to the system.

Returns:

acc: array_like

The time derivative of the state.

Subclass.IntegralVelocity

Subclass.IntegralVelocity(self, x)

The state whose integral is useful for displaying the animation.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the number of states.

Returns:

state: float

a single whose integral is useful for displaying the animation

Subclass.UpdateAnimation

Subclass.UpdateAnimation(self, x,delta,WINDOW_NAME,windowH,windowW,imageScale)

Display the animation.

Parameters:

x: array_like

Current state. It must have a shape of (1, x), where x represents the

number of states.

delta: float

The integral of the state of **Subclass.IntegralVelocity** method.

WINDOW_NAME: string

The name of the displaying window.

windowH: float

The pixel height of animation window. The default is 500 pixels.

windowW: float

The pixel width of animation window. The default is 500 pixels.

imageScale: float

The pixel length of one meter in the animation. The default is 1000 pixels.

Returns:

Image: array_like

The updated image of the animation.

The following are the initiation in each subclass. Also, all the units are SI units.

InvertedPendulumOnWheel(self, m,m_w,R,L,I,I_w,Br,Bm)

State: $\theta, \dot{\phi}, \dot{\theta}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta

Input: torque in the $\phi - \theta$ direction

output: θ

set point state: $\dot{\phi}$

m: the mass of body

m_w: the mass of the wheel

R: the radius of the wheel

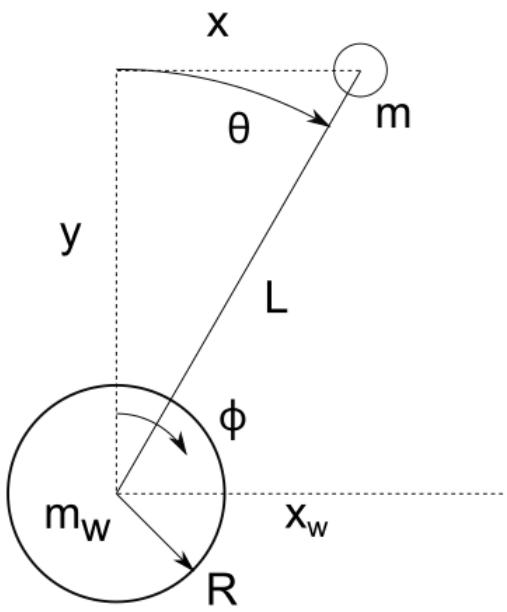
L: the length of the body

I: the moment of inertia of the body

I_w: the moment of inertia of the wheel

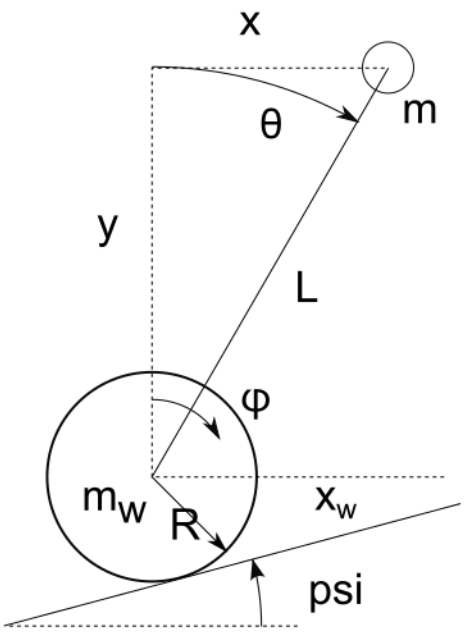
Br: the friction coefficient in the $-\dot{\phi}$ direction

Br: the friction coefficient in the $\dot{\theta} - \dot{\phi}$ direction



InvertedPendulumOnWheelOnSlope(self, m,m_w,R,L,I,I_w,Br,Bm, psi)

State: $\theta, \dot{\phi}, \dot{\theta}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta



Input: torque in the $\phi - \theta$ direction

output: θ

set point state: $\dot{\phi}$

m: the mass of body

m_w: the mass of the wheel

R: the radius of the wheel

L: the length of the body

I: the moment of inertia of the body

I_w: the moment of inertia of the wheel

Br: the friction coefficient in the $-\dot{\phi}$ direction

Br: the friction coefficient in the $\dot{\theta} - \dot{\phi}$ direction

Psi: angle of the slope

ReactionWheelOnInvertedPendulum(self, m,m_w,L,L_w,I,I_w,Ba,Bm)

State: $\theta, \dot{\phi}, \dot{\theta}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta

Input: torque in the $\phi - \theta$ direction

output: $\theta, \dot{\phi} - \dot{\theta}$

set point state: $\dot{\phi}$

m: the mass of body

m_w: the mass of the wheel

R: the radius of the wheel

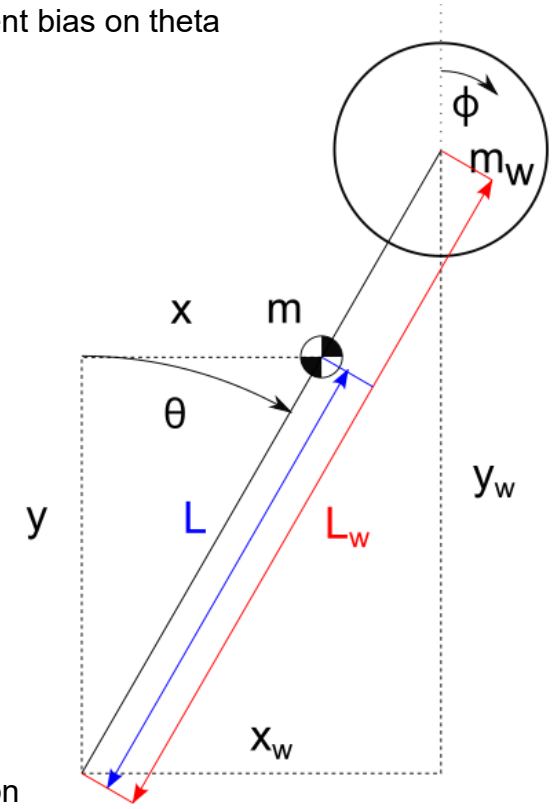
L: the length of the body

I: the moment of inertia of the body

I_w: the moment of inertia of the wheel

Ba: the friction coefficient in the $-\dot{\phi}$ direction

Bm: the friction coefficient in the $\dot{\theta} - \dot{\phi}$ direction



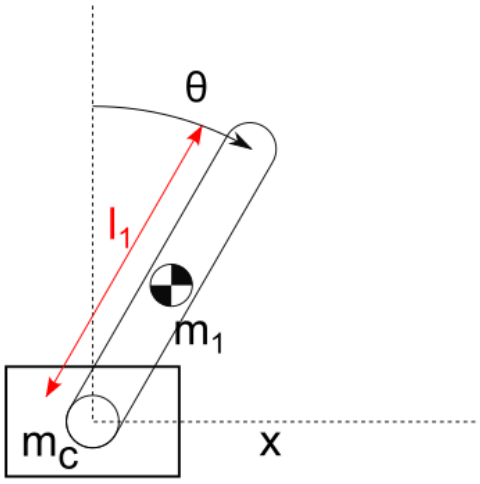
InvertedPendulumOnCart(self,mc,m1,l1,I1,b1,R)

State: $\theta, \dot{x}, \dot{\theta}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta

Input: torque = R * force in the x direction

output: $\theta, \dot{x}$

set point state: $\dot{x}$



mc: the mass of the cart

m1: the mass of the pendulum 1

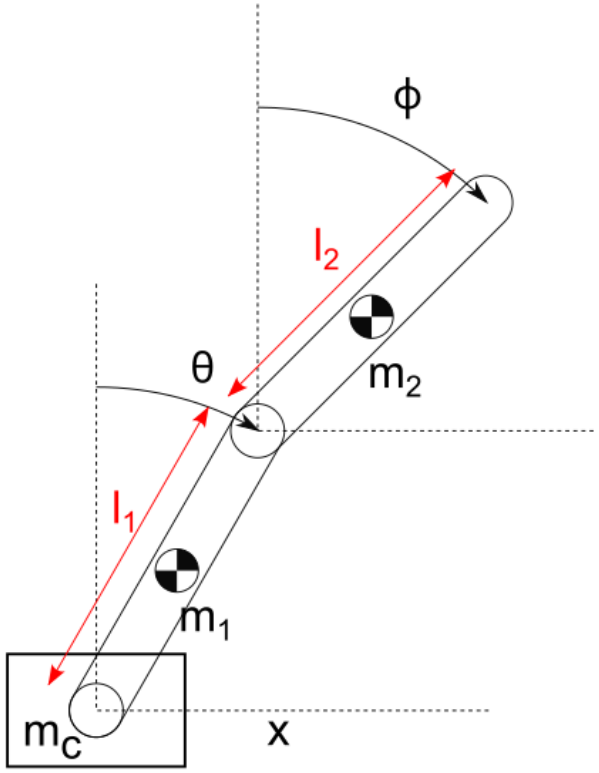
l1: the length of the pendulum 1

I1: the moment of inertia of the pendulum 1

b1: the friction coefficient in the $\dot{\theta}$ direction

R: if the motor generates a torque, it is transformed to a $force = \frac{torque}{R}$ in x direction

DoubleInvertedPendulumOnCart(self,mc,m1,l1,I1,b1,m2,l2,I2,b2,R)



State: $\theta, \phi, \dot{x}, \dot{\theta}, \dot{\phi}$

Input: torque = R * force in the x direction

output: $\theta, \phi - \theta, \dot{x}$

set point state: $\dot{x}$

mc: the mass of the cart

m1: the mass of the pendulum 1

l1: the length of the pendulum 1

I1: the moment of inertia of the pendulum 1

b1: the friction coefficient in the $\dot{\theta}$ direction

m2: the mass of the pendulum 2

l2: the length of the pendulum 2

I2: the moment of inertia of the pendulum 2

b2: the friction coefficient in the $\dot{\phi} - \dot{\theta}$ direction

R: if the motor generates a torque, it is transformed to a $force = \frac{torque}{R}$ in x direction

BallOnBeam(self,m,r,d,l,b,ratio)

State: $\theta, R, \dot{R}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta

Input: $\dot{\theta}$

output: θ, R

set point state: R

m: the mass of the ball

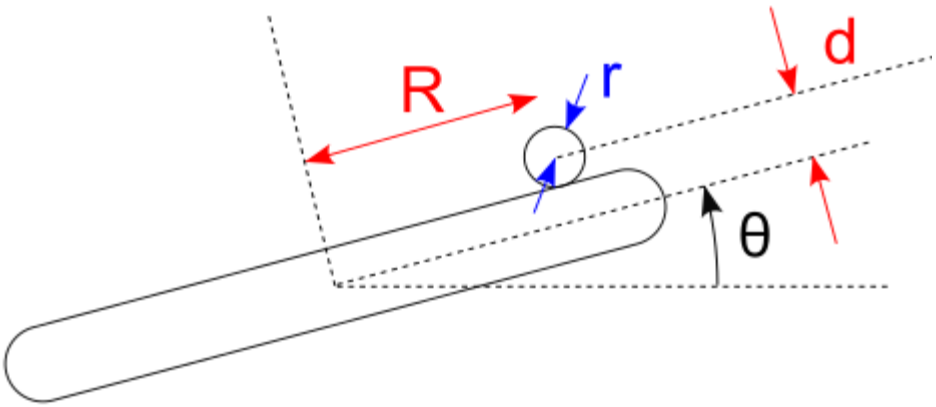
r: the radius of the ball

d: the vertical distance of the ball from the beam

l: moment of inertia of the beam

b: the friction coefficient in the R direction

ratio: a ratio in the control matrix. Please refer to the "controlSystemAndModels.pdf" document.



ControlMomentGyroscope(self,r2,r1,h,m,mb,r,rb,Bw,psidot)

State: $\theta, \phi, \dot{\theta}, \theta_{Bias}$. θ_{Bias} is the measurement bias on theta

Input: $\dot{\phi}$

output: θ, ϕ

set point state: ϕ

r2: the outer radius of disk

r1: the inner radius of disk

h: the thickness of disk

m: the mass of disk

mb: the mass of body

r: the height of the center of mass of disk

rb: the height of the center of mass of body

Bw: the friction coefficient in $\dot{\theta}$ direction

psidot: the anguler velocity of disk