In [1]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
import MeshCat as mc
using JLD2
using Test
using Random
include(joinpath(@__DIR__,"utils/cartpole_animation.jl"))
```

```
  Activating environment at `C:\Users\rdesa\OneDrive\Desktop\OCRL_HW2\HW2_S23
-main\Project.toml`
```

Out[1]:
```
animate_cartpole (generic function with 1 method)
```

Q2: LQR for nonlinear systems (25 pts)

Linearization warmup

Before we apply LQR to nonlinear systems, we are going to treat our linear system as if it's nonlinear. Specifically, we are going to "approximate" our linear system with a first-order Taylor series, and define a new set of $(\Delta x, \Delta u)$ coordinates. Since our dynamics are linear, this approximation is exact, allowing us to check that we set up the problem correctly.

First, assume our discrete time dynamics are the following:

$$x_{k+1} = f(x_k, u_k)$$

And we are going to linearize about a reference trajectory $\bar{x}_{1:N}, \bar{u}_{1:N-1}$. From here, we can define our delta's accordingly:

$$x_k = \bar{x}_k + \Delta x_k$$
$$u_k = \bar{u}_k + \Delta u_k$$

Next, we are going to approximate our discrete time dynamics function with the following first order Taylor series:

$$x_{k+1} \approx f(\bar{x}_k, \bar{u}_k) + \left[\frac{\partial f}{\partial x}\bigg|_{\bar{x}_k,\bar{u}_k}\right](x_k - \bar{x}_k) + \left[\frac{\partial f}{\partial u}\bigg|_{\bar{x}_k,\bar{u}_k}\right](u_k - \bar{u}_k)$$

Which we can substitute in our delta notation to get the following:

$$\bar{x}_{k+1} + \Delta x_{k+1} \approx f(\bar{x}_k, \bar{u}_k) + \left[\frac{\partial f}{\partial x}\bigg|_{\bar{x}_k,\bar{u}_k}\right]\Delta x_k + \left[\frac{\partial f}{\partial u}\bigg|_{\bar{x}_k,\bar{u}_k}\right]\Delta u_k$$

If the trajectory $\bar{x}, \bar{u}$ is dynamically feasible (meaning $\bar{x}_{k+1} = f(\bar{x}_k, \bar{u}_k)$), then we can cancel these equivalent terms on each side of the above equation, resulting in the following:

$$\Delta x_{k+1} \approx \left[\frac{\partial f}{\partial x}\bigg|_{\bar{x}_k,\bar{u}_k}\right]\Delta x_k + \left[\frac{\partial f}{\partial u}\bigg|_{\bar{x}_k,\bar{u}_k}\right]\Delta u_k$$

Cartpole

We are now going to look at two different applications of LQR to the nonlinear cartpole system. Given the following description of the cartpole:

(if this image doesn't show up, check out `cartpole.png`)

with a cart position p and pole angle θ. We are first going to linearize the nonlinear discrete dynamics of this system about the point where $p = 0$, and $\theta = 0$ (no velocities), and use an infinite horizon LQR controller about this linearized state to stabilize the cartpole about this goal state. The dynamics of the cartpole are parametrized by the mass of the cart, the mass of the pole, and the length of the pole. To simulate a "sim to real gap", we are going to design our controllers around an estimated set of problem parameters `params_est` , and simulate our system with a different set of problem parameters `params_real` .

In [2]:

```julia
"""
continuous time dynamics for a cartpole, the state is
x = [p, θ, ṗ, θ̇]
where p is the horizontal position, and θ is the angle
where θ = 0 has the pole hanging down, and θ = 180 is up.

The cartpole is parametrized by a cart mass `mc`, pole
mass `mp`, and pole length `l`. These parameters are loaded
into a `params::NamedTuple`. We are going to design the
controller for a estimated `params_est`, and simulate with
`params_real`.
"""
function dynamics(params::NamedTuple, x::Vector, u)
    # cartpole ODE, parametrized by params.

    # cartpole physical parameters
    mc, mp, l = params.mc, params.mp, params.l
    g = 9.81

    q = x[1:2]
    qd = x[3:4]

    s = sin(q[2])
    c = cos(q[2])

    H = [mc+mp mp*l*c; mp*l*c mp*l^2]
    C = [0 -mp*qd[2]*l*s; 0 0]
    G = [0, mp*g*l*s]
    B = [1, 0]

    qdd = -H\(C*qd + G - B*u[1])
    return [qd;qdd]

end

function rk4(params::NamedTuple, x::Vector,u,dt::Float64)
    # vanilla RK4
    k1 = dt*dynamics(params, x, u)
    k2 = dt*dynamics(params, x + k1/2, u)
    k3 = dt*dynamics(params, x + k2/2, u)
    k4 = dt*dynamics(params, x + k3, u)
    x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
end
```

Out[2]: rk4 (generic function with 1 method)

Part A: Infinite Horizon LQR about an equilibrium (10 pts)

Here we are going to solve for the infinite horizon LQR gain, and use it to stabilize the cartpole about the unstable equilibrium.

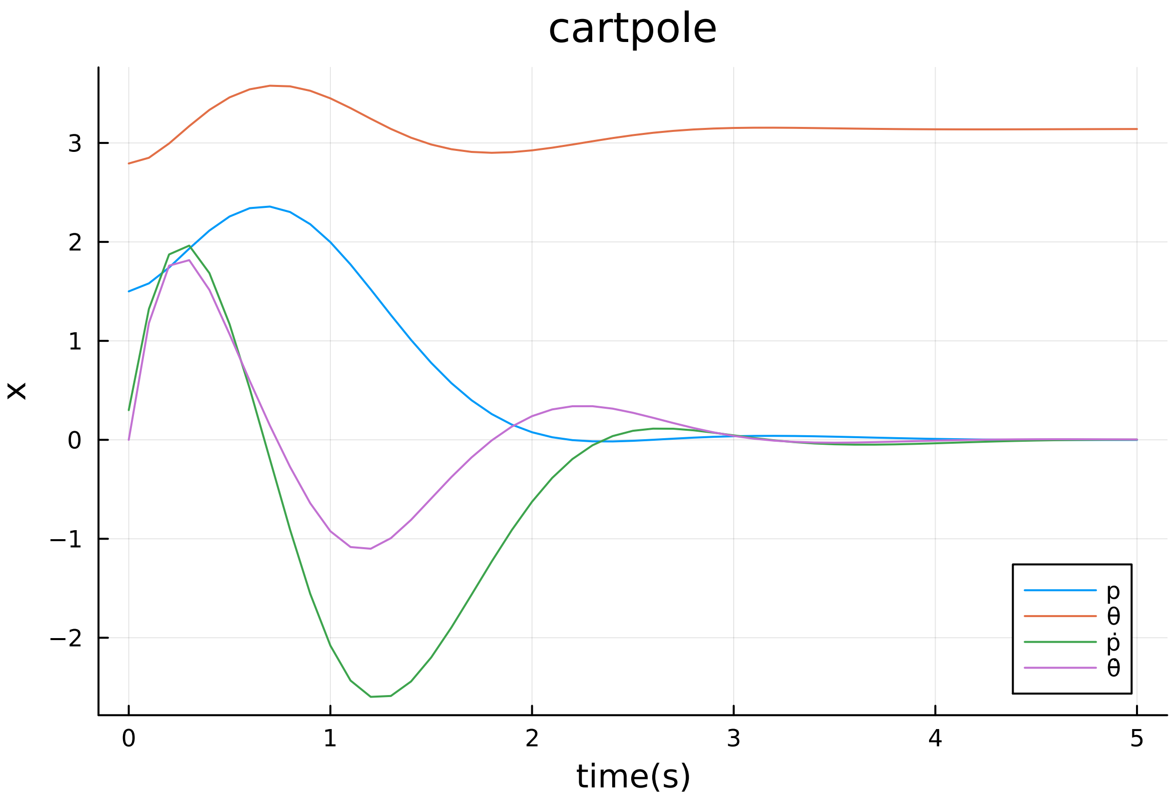
In [3]:
```
function IH_ricatti(t_vec,Qf,A,B,Q,R)
    for k = 1:(length(t_vec)-1)
            P = deepcopy(Qf)
            for ricatti_iter = 1:1000
                K_k = (R + B'*P*B)\(B'*P*A)
                Pnew = Q + A'*P*(A-B*(K_k))

                if norm(Pnew - P) <= 1e-5
                    K = (R + B'*Pnew*B)\(B'*Pnew*A)
                    return K
                    break
                end
                P = Pnew
            end
            Kinf = K
            #Kinf = zeros(1,4)
        end
end
```

Out[3]:
```
IH_ricatti (generic function with 1 method)
```

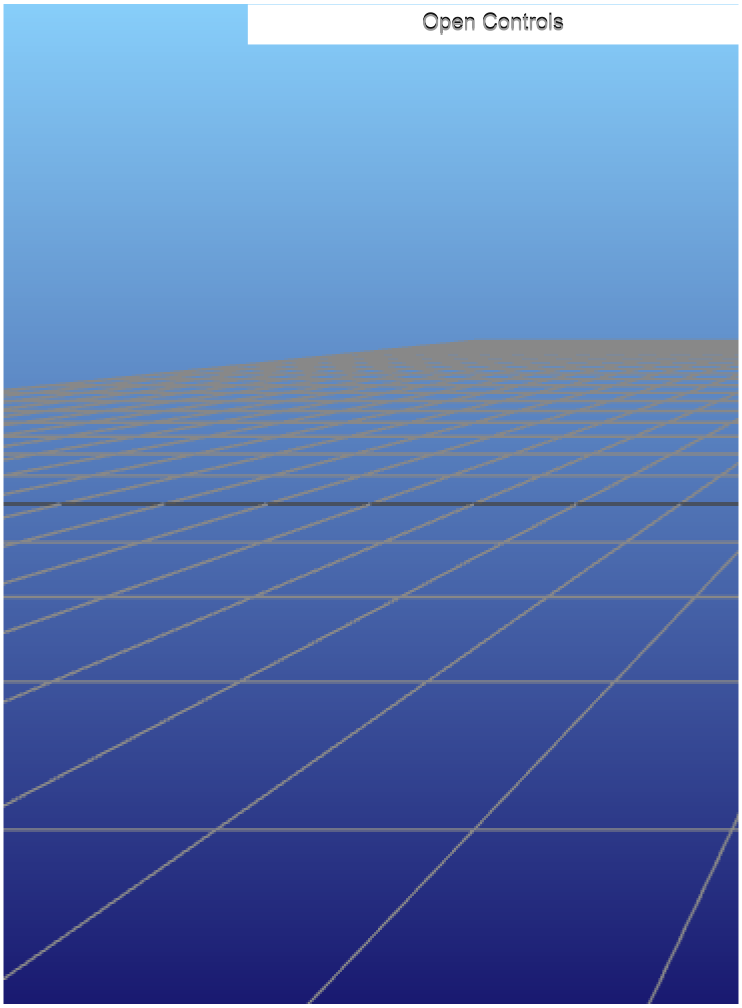
In [4]:
```
@testset "LQR about eq" begin

    # states and control sizes
    nx = 4
    nu = 1

    # desired x and g (linearize about these)
    xgoal = [0, pi, 0, 0]
    ugoal = [0]

    # initial condition (slightly off of our linearization point)
    x0 = [0, pi, 0, 0] + [1.5, deg2rad(-20), .3, 0]

    # simulation size
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    X = [zeros(nx) for i = 1:N]
    X[1] = x0

    # estimated parameters (design our controller with these)
    params_est = (mc = 1.0, mp = 0.2, l = 0.5)

    # real paremeters (simulate our system with these)
    params_real = (mc = 1.2, mp = 0.16, l = 0.55)

    # TODO: solve for the infinite horizon LQR gain Kinf

    # cost terms
    Q = diagm([1,1,.05,.1])
    Qf = 1*Q
    R = 0.1*diagm(ones(nu))

    ######################################################################

    #Determine A and B

    A = FD.jacobian(Dx -> rk4(params_est,Dx,ugoal,dt),xgoal)
    B = FD.jacobian(Du -> rk4(params_est,xgoal,Du,dt),ugoal)

    Kinf = IH_ricatti(t_vec,Qf,A,B,Q,R)

    U = [zeros(nu) for i = 1:N-1]

    #######################################################################

    # TODO: simulate this controlled system with rk4(params_real, ...)
    for k = 1:(length(t_vec)-1)
        U[k] = -Kinf*(X[k]-xgoal)
        X[k+1] = rk4(params_real, X[k], (U[k]-ugoal), dt)
    end
    print("\n Simulation complete \n")
    ###########################################################################

    # ---------------tests and plots/animations---------------
```

```
    @test X[1] == x0
    @test norm(X[end])>0
    @test norm(X[end] - xgoal) < 0.1

    Xm = hcat(X...)
    display(plot(t_vec,Xm',title = "cartpole",
                 xlabel = "time(s)", ylabel = "x",
                 label = ["p" "θ" "ṗ" "θ̇"]))

    # animation stuff
    display(animate_cartpole(X, dt))
    # ---------------tests and plots/animations---------------
end
```

```
Simulation complete
```




```
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8700
```




```
Test Summary: | Pass  Total
LQR about eq  |    3      3
```

Out[4]: `Test.DefaultTestSet("LQR about eq", Any[], 3, false, false)`

In [ ]:

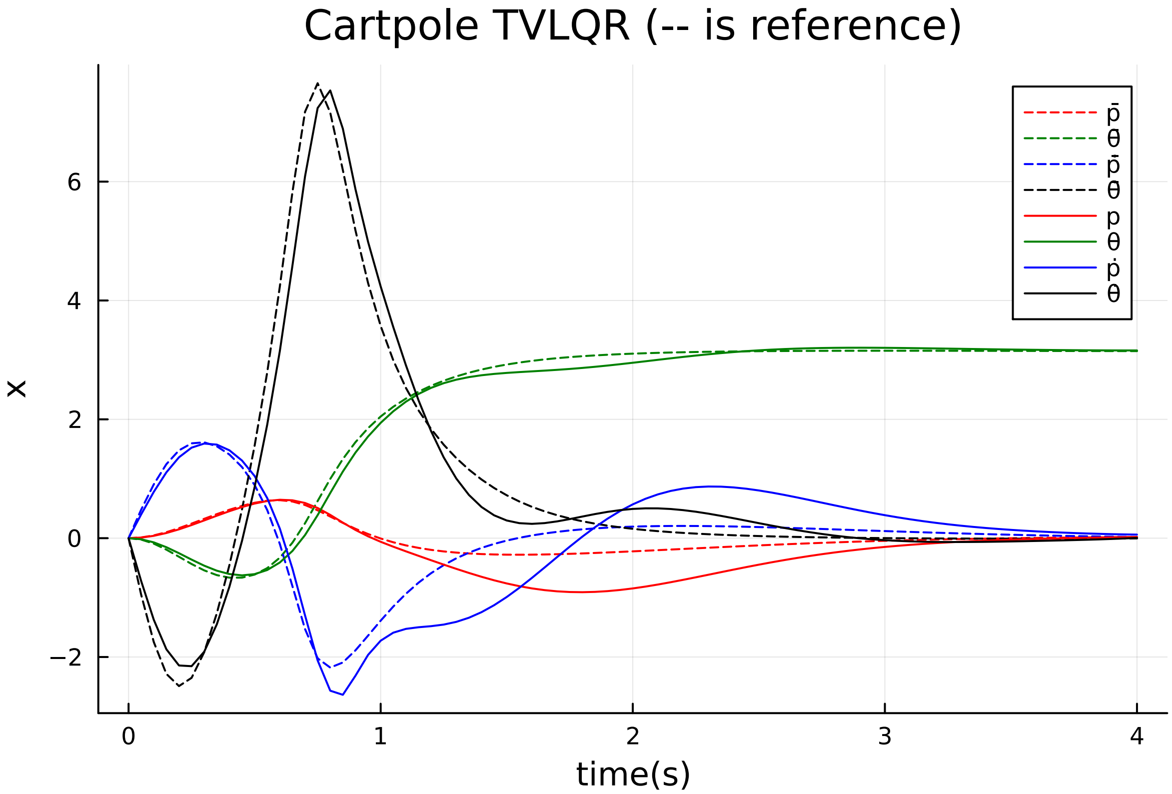
```
```

Part B: TVLQR for trajectory tracking (15 pts)

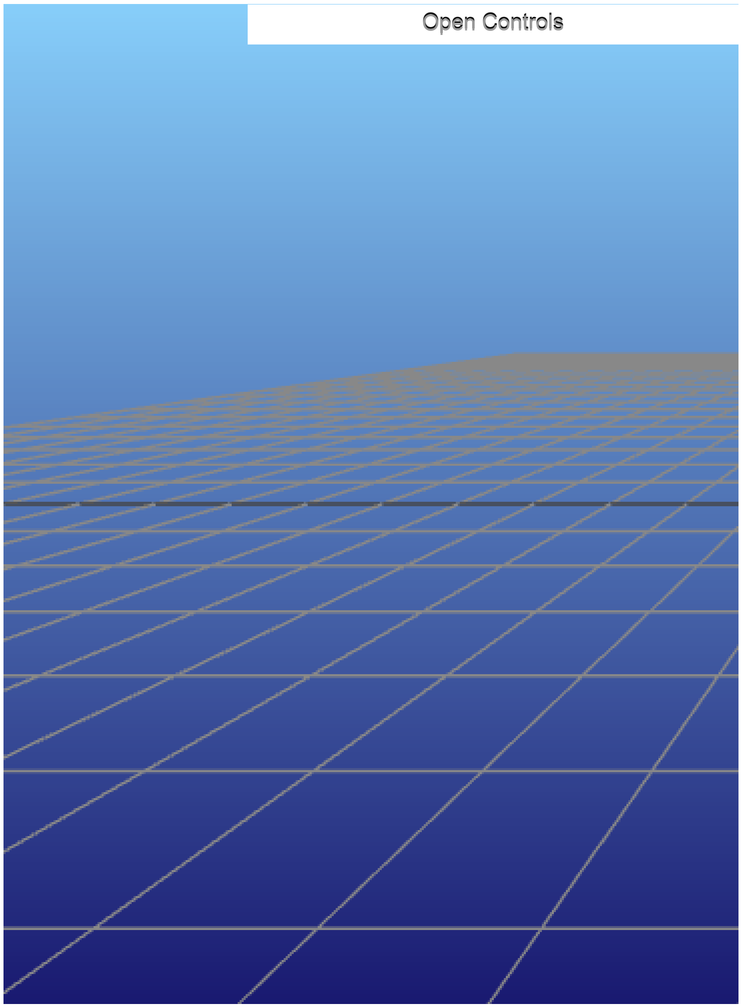
Here we are given a swingup trajectory that works for `params_est` , but will fail to work with `params_real` . To account for this sim to real gap, we are going to track this trajectory with a TVLQR controller.

In [5]:
```
@testset "track swingup" begin

    # optimized trajectory we are going to try and track
    DATA = load(joinpath(@__DIR__,"swingup.jld2"))
    Xbar = DATA["X"]
    Ubar = DATA["U"]

    # states and controls
    nx = 4
    nu = 1

    # problem size
    dt = 0.05
    tf = 4.0
    t_vec = 0:dt:tf
    N = length(t_vec)

    # states (initial condition of zeros)
    X = [zeros(nx) for i = 1:N]
    X[1] = [0, 0, 0, 0.0]

    # make sure we have the same initial condition
    @assert norm(X[1] - Xbar[1]) < 1e-12

    # real and estimated params
    params_est = (mc = 1.0, mp = 0.2, l = 0.5)
    params_real = (mc = 1.2, mp = 0.16, l = 0.55)

    # TODO: design a time-varying LQR controller to track this trajectory
    # use params_est for your control design, and params_real for the simulati
on

    # cost terms
    Q = diagm([1,1,.05,.1])
    Qf = 10*Q
    R = 0.05*diagm(ones(nu))

    # TODO: solve for tvlqr gains K

    P = [zeros(nx,nx) for i = 1:N]
    K = [zeros(nu,nx) for i = 1:N-1]

    #Determine A and B
    P[N] = deepcopy(Qf)

    P_k = P[N]

    U = [zeros(nu) for i = 1:N-1]

    for k = (N-1):-1:1
        #Generate A and B
        A = FD.jacobian(Dx -> rk4(params_est,Dx,Ubar[k],dt),Xbar[k])
        B = FD.jacobian(Du -> rk4(params_est,Xbar[k],Du,dt),Ubar[k])
        #Generate K
        K[k] = (R + B'*P_k*B)\(B'*P_k*A)
        K_k = K[k]
```

```
        #Generate P
        P[k] = Q + A'*P_k*(A-B*(K_k))
        P_k = P[k]
    end
        #Get U
    for k = 1:(length(t_vec)-1)
        # TODO: simulate this controlled system with rk4(params_real, ...)
        U[k] = -K[k]*(X[k]-Xbar[k])
        X[k+1] = rk4(params_real, X[k], (U[k]+Ubar[k]), dt)
    end

    # ---------------tests and plots/animations---------------
    xn = X[N]
    @test norm(xn)>0
    @test 1e-6<norm(xn - Xbar[end])<.2
    @test abs(abs(rad2deg(xn[2])) - 180) < 5 # within 5 degrees

    Xm = hcat(X...)
    Xbarm = hcat(Xbar...)
    plot(t_vec,Xbarm',ls=:dash, label = ["p̄" "θ̄" "ṗ̄" "θ̇̄"],lc = [:red :green :b
lue :black])
    display(plot!(t_vec,Xm',title = "Cartpole TVLQR (-- is reference)",
                 xlabel = "time(s)", ylabel = "x",
                 label = ["p" "θ" "ṗ" "θ̇"],lc = [:red :green :blue :black]))

    # animation stuff
    display(animate_cartpole(X, dt))
    # ---------------tests and plots/animations---------------


end
```




```
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8702
```




```
Test Summary: | Pass  Total
track swingup |    3      3
```

Out[5]: `Test.DefaultTestSet("track swingup", Any[], 3, false, false)`

In [ ]: