In [1]:
```
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"))
include(joinpath(@__DIR__,"utils/basin_of_attraction.jl"))
```

```
  Activating environment at `/home/sman/Work/CMU/Courses/OCRL/OCRL2024/HW/HW2_S24/Project.toml`

plot_basin_of_attraction (generic function with 1 method)
```

Note:

Some of the cells below will have multiple outputs (plots and animations), it can be easier to see everything if you do `Cell -> All Output -> Toggle Scrolling` , so that it simply expands the output area to match the size of the outputs.

Q2: LQR for nonlinear systems (40 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:

$$\begin{aligned} x_k &= \bar{x}_k + \Delta x_k \\ u_k &= \bar{u}_k + \Delta u_k \end{aligned}$$

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
```

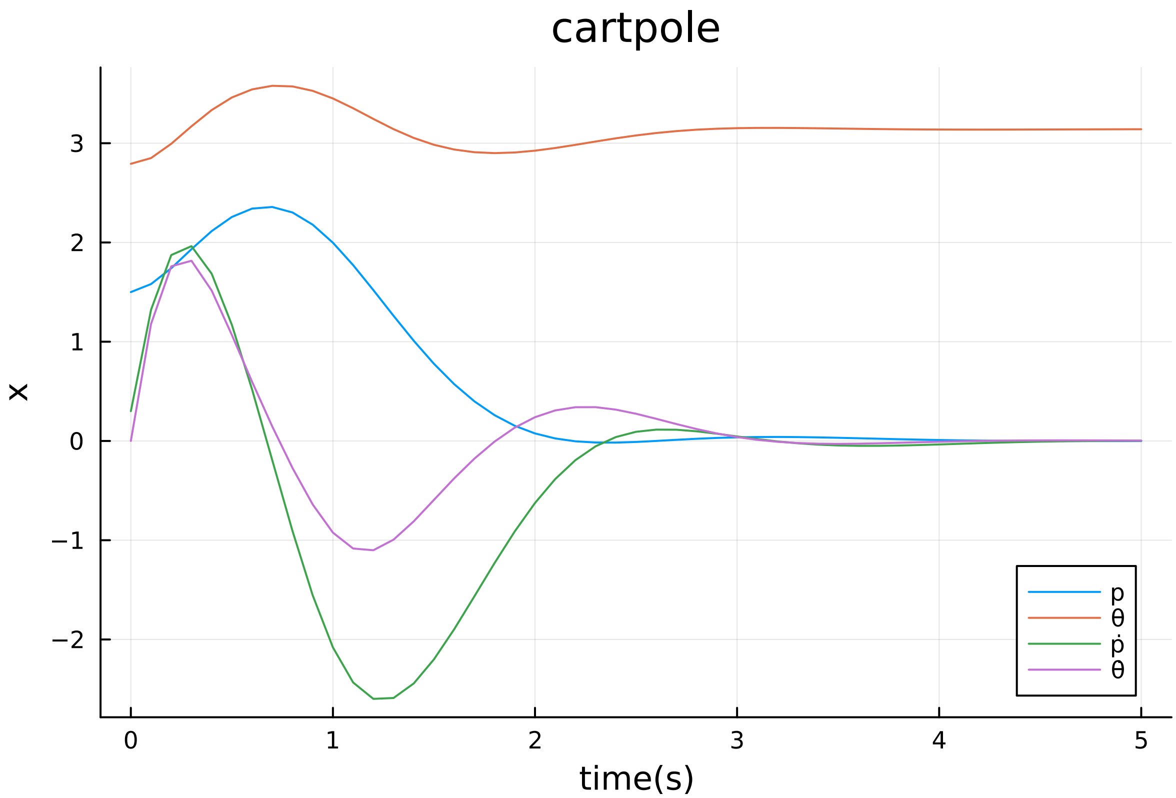
```
rk4 (generic function with 1 method)
```

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]:
```
@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])
    R = 0.1*diagm(ones(nu))

    Kinf = zeros(1,4)
    P = deepcopy(Q)
    tol = 1e-5

    x_linearize = xgoal
    u_linearize = 0

    Ac = FD.jacobian(_x -> dynamics(params_est, _x, u_linearize), x_linearize)
    Bc = FD.derivative(_u -> dynamics(params_est, x_linearize, _u), u_lineariz
e)

    Dc = zeros(nx+nu,nx+nu)
    Dc[1:nx, 1:nx+nu] = [Ac Bc]
    Dd = exp(Dc*dt)
    A, B = (Dd[1:nx, 1:nx], Dd[1:nx, (nx+1):(nx+nu)])

    for ricatti_iter = 1:1000
        K = (R + B' * P * B) \ (B' * P * A)
        P_kp1 = Q + A'* P * (A - B * K)

        if norm(P - P_kp1) < tol
            Kinf = K
            break
```

```
        end

        P = P_kp1
    end

    # TODO: simulate this controlled system with rk4(params_real, ...)
    for ii = 2:N
        X[ii] = rk4(params_real, X[ii-1], -Kinf*(X[ii-1]-xgoal), dt)
    end

    # ---------------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
```




```
┌ Info: Listening on: 127.0.0.1:8700, thread id: 1
└ @ HTTP.Servers /root/.julia/packages/HTTP/1EWL3/src/Servers.jl:369
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
│ http://127.0.0.1:8700
└ @ MeshCat /root/.julia/packages/MeshCat/I6NTX/src/visualizer.jl:63
```

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

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

Part B: Infinite horizon LQR basin of attraction (5 pts)

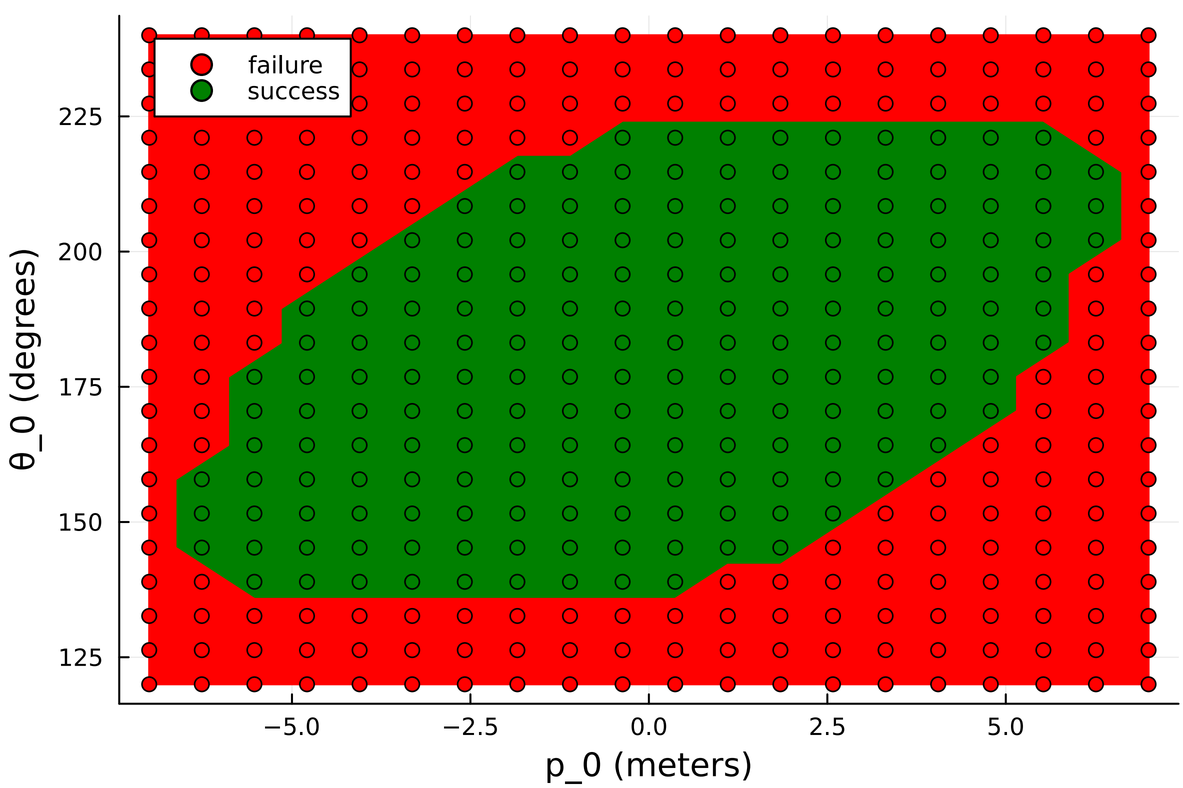
In part A we built a controller for the cartpole that was based on a linearized version of the system dynamics. This linearization took place at the `(xgoal, ugoal)`, so we should only really expect this model to be accurate if we are close to this linearization point (think small angle approximation). As we get further from the goal state, our linearized model is less and less accurate, making the performance of our controller suffer. At a certain point, the controller is unable to stabilize the cartpole due to this model mismatch.

To demonstrate this, you are now being asked to take the same controller you used above, and try it for a range of initial conditions. For each of these simulations, you will determine if the controller was able to stabilize the cartpole. From here, you will plot the successes and failures on a plot and visualize a "basin of attraction", that is, a region of the state space where we expect our controller to stabilize the system.

In [4]:
```
function create_initial_conditions()
    # create a span of initial configurations
    M=20
    ps = LinRange(-7, 7, M)
    thetas = LinRange(deg2rad(180-60), deg2rad(180+60), M)

    initial_conditions = []

    for p in ps
        for theta in thetas
            push!(initial_conditions, [p, theta, 0, 0.0])
        end
    end

    return initial_conditions, ps, thetas
end

function check_simulation_convergence(params_real, initial_condition, Kinf, xg
oal, N, dt)
    """
    args
        params_real: named tuple with model dynamics parametesr
        initial_condition: X0, length 4 vector
        Kinf: IHLQR feedback gain
        xgoal: desired state, length 4 vector
        N: number of simulation steps
        dt: time between steps

    return
        is_controlled: bool
    """

    nx = 4
    xgoal = [0, pi, 0, 0]
    X = [zeros(nx) for i = 1:N]
    x0 = 1 * initial_condition
    X[1] = x0

    is_controlled = false

    # TODO: simulate the closed-loop (controlled) cartpole starting at the ini
tial condition

    # for some of the unstable initial conditions, the integrator will "blow u
p", in order to
    # catch these errors, you can stop the sim and return is_controlled = fals
e if norm(x) > 100

    # you should consider the simulation to have been successfuly controlled i
f the
    # L2 norm of |xfinal - xgoal| < 0.1. (norm(xfinal-xgoal) < 0.1 in Julia)

    for ii = 2:N
        X[ii] = rk4(params_real, X[ii-1], -Kinf*(X[ii-1]-xgoal), dt)
        if norm(X[ii]) > 100
            return is_controlled
```

```
        end
    end

    is_controlled = (norm(X[end]-xgoal)) < 0.1

    return is_controlled
end

let

    nx = 4
    nu = 1
    xgoal = [0, pi, 0, 0]
    ugoal = [0]
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)


    # 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
    # this is the same controller as part B

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

    Kinf = zeros(1,4)
    P = deepcopy(Q)
    tol = 1e-5

    x_linearize = xgoal
    u_linearize = 0

    Ac = FD.jacobian(_x -> dynamics(params_est, _x, u_linearize), x_linearize)
    Bc = FD.derivative(_u -> dynamics(params_est, x_linearize, _u), u_lineariz
e)

    Dc = zeros(nx+nu,nx+nu)
    Dc[1:nx, 1:nx+nu] = [Ac Bc]
    Dd = exp(Dc*dt)
    A, B = (Dd[1:nx, 1:nx], Dd[1:nx, (nx+1):(nx+nu)])

    for ricatti_iter = 1:1000
        K = (R + B' * P * B) \ (B' * P * A)
        P_kp1 = Q + A'* P * (A - B * K)

        if norm(P - P_kp1) < tol
            Kinf = K
            break
        end
```

```
        P = P_kp1
    end

    # create the set of initial conditions we want to test for convergence
    initial_conditions, ps, thetas = create_initial_conditions()

    convergence_list = []

    for initial_condition in initial_conditions

        convergence = check_simulation_convergence(params_real,
                                                    initial_condition,
                                                    Kinf, xgoal, N, dt)

        push!(convergence_list, convergence)
    end

    plot_basin_of_attraction(initial_conditions, convergence_list, ps, rad2de
g.(thetas))


    # ------------tests------------------
    @test sum(convergence_list) < 190
    @test sum(convergence_list) > 180
    @test length(convergence_list) == 400
    @test length(initial_conditions) == 400

end
```




```
Test Passed
```

Part C: Infinite horizon LQR cost tuning (5 pts)

We are now going to tune the LQR cost to satisfy our following performance requirement:

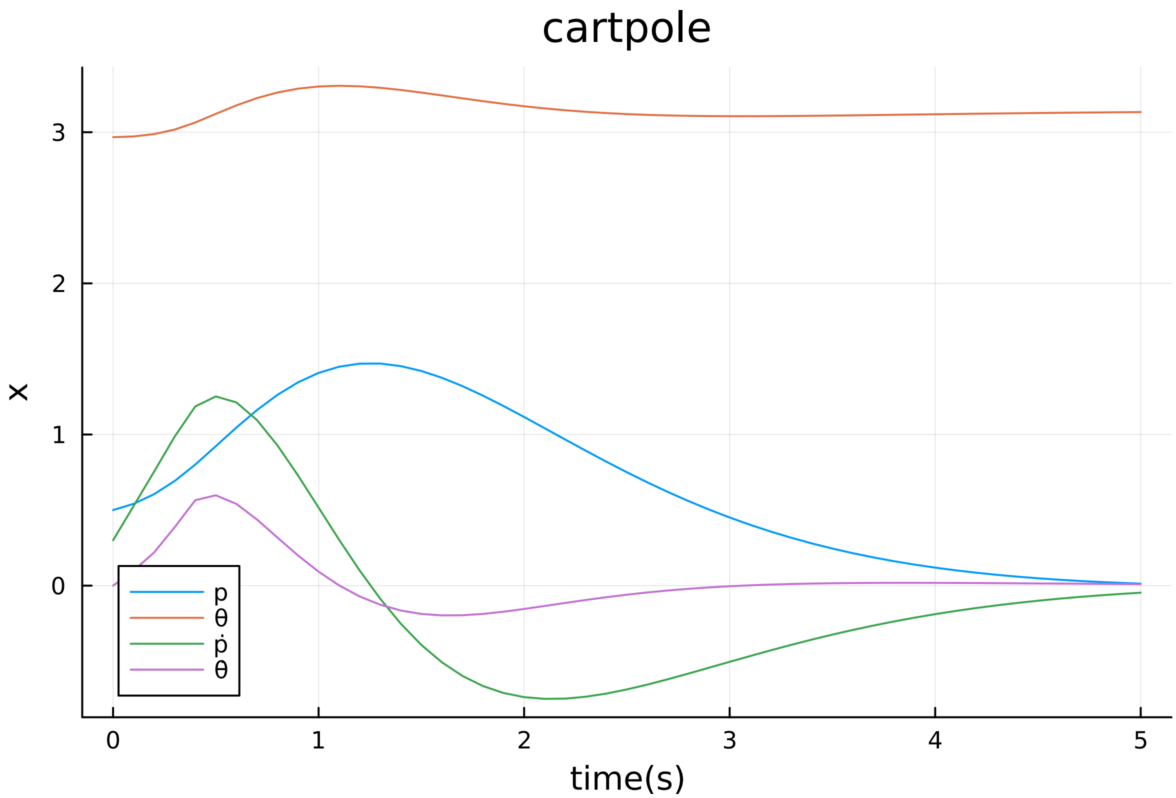
$\|x(5.0) - x_{\text{goal}}\|_2 =$ `norm(X[N] - xgoal) < 0.1`

which says that the L2 norm of the state at 5 seconds (last timestep in our simulation) should be less than 0.1. We are also going to have to deal with the following actuator limits: $-3 \leq u \leq 3$. You won't be able to directly reason about this actuator limit in our LQR controller, but we can tune our cost function to avoid saturating the actuators (reaching the actuator limits) for too long. Here are our suggestions for tuning successfully:

1. First, adjust the values in Q and R to find a controller that stabilizes the cartpole. The key here is tuning our cost to keep the control away from the actuator limits for too long.
2. Now that you can stabilize the system, the next step is to tune the values in Q and R accomplish our performance goal of `norm(X[N] - xgoal) < 0.1`. Think about the individual values in Q, and which states we really want to penalize. The positions (p, θ) should be penalized differently than the velocities $(\dot{p}, \dot{\theta})$.

In [5]:
```
@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] + [0.5, deg2rad(-10), .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([10,10,1,10])
    R = 10*diagm(ones(nu))

    Kinf = zeros(1,4)

    P = deepcopy(Q)
    tol = 1e-5

    x_linearize = xgoal
    u_linearize = 0

    Ac = FD.jacobian(_x -> dynamics(params_est, _x, u_linearize), x_linearize)
    Bc = FD.derivative(_u -> dynamics(params_est, x_linearize, _u), u_lineariz
e)

    Dc = zeros(nx+nu,nx+nu)
    Dc[1:nx, 1:nx+nu] = [Ac Bc]
    Dd = exp(Dc*dt)
    A, B = (Dd[1:nx, 1:nx], Dd[1:nx, (nx+1):(nx+nu)])

    for ricatti_iter = 1:1000
        K = (R + B' * P * B) \ (B' * P * A)
        P_kp1 = Q + A'* P * (A - B * K)

        if norm(P - P_kp1) < tol
            Kinf = K
```

```
            break
        end

        P = P_kp1
    end

    # vector of length 1 vectors for our control
    U = [zeros(1) for i = 1:N-1]

    # TODO: simulate this controlled system with rk4(params_real, ...)
    # TODO: make sure you clamp the control input with clamp.(U[i], -3.0, 3.0)

    for ii = 2:N
        U[ii-1] = clamp.(-Kinf*(X[ii-1]-xgoal),-3,3)
        X[ii] = rk4(params_real, X[ii-1],U[ii-1], dt)
    end

    @show norm(X[end] - xgoal)

    # ---------------tests and plots/animations---------------
    @test X[1] == x0  # initial condition is used
    @test norm(X[end])>0 # end is nonzero
    @test norm(X[end] - xgoal) < 0.1 # within 0.1 of the goal
    @test norm(vcat(U...), Inf) <= 3.0 # actuator limits are respected

    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
```




```
norm(X[end] - xgoal) = 0.050373809497486495

┌ Info: Listening on: 127.0.0.1:8701, thread id: 1
└ @ HTTP.Servers /root/.julia/packages/HTTP/1EWL3/src/Servers.jl:369
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
│ http://127.0.0.1:8701
└ @ MeshCat /root/.julia/packages/MeshCat/I6NTX/src/visualizer.jl:63
```

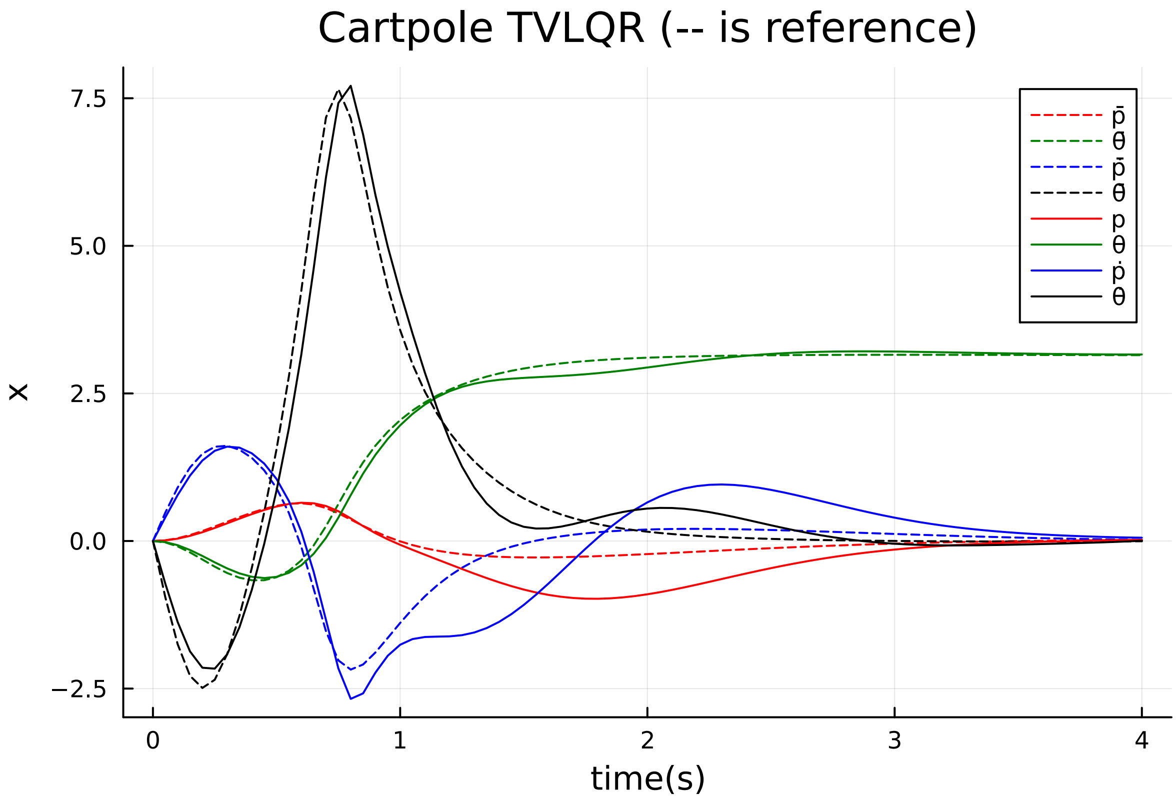
```
Test Summary: | Pass  Total
LQR about eq  |    4      4

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

Part D: 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 [55]:
```julia
@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 = Qf
    Ks = [zeros(nx) for i = 1:N-1]
    Dc = zeros(nx+nu,nx+nu)
    u = 0

    for ii = N:-1:2
        Ac = FD.jacobian(_x -> dynamics(params_est, _x, Ubar[ii-1]), Xbar[ii])
        Bc = FD.jacobian(_u -> dynamics(params_est, Xbar[ii], _u), Ubar[ii-1])
        Dc[1:nx, 1:nx+nu] = [Ac Bc]
        Dd = exp(Dc*dt)
        A, B = (Dd[1:nx, 1:nx], Dd[1:nx, (nx+1):(nx+nu)])

        K = (R + B'*P*B) \ B'*P*A
        Ks[ii-1] = vec(K)
        P = Q + K'*R*K + (A-B*K)'*P*(A-B*K)
    end
```

```julia
    # TODO: simulate this controlled system with rk4(params_real, ...)
    for ii = 2:N
        U = Ubar[ii-1]-[Ks[ii-1]⋅(X[ii-1] - Xbar[ii-1])]
        X[ii] = rk4(params_real, X[ii-1],U, 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" "θ̄̇"],lc = [:red :green :blue :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: Listening on: 127.0.0.1:8709, thread id: 1
└ @ HTTP.Servers /root/.julia/packages/HTTP/1EWL3/src/Servers.jl:369
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
│ http://127.0.0.1:8709
└ @ MeshCat /root/.julia/packages/MeshCat/I6NTX/src/visualizer.jl:63
```

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

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

Part E (5 pts): One sentence short answer

1. Will the LQR controller from part A be stable no matter where the cartpole starts?

No, it's only linearized about the goal region, so will not be stable when too far from the goal

1. In order to build an infinite-horizon LQR controller for a nonlinear system, do we always need a state to linearize about?

Yes, LQR is an optimal controller for linear systems

1. If we are worried about our LQR controller saturating our actuator limits, how should we change the cost?

Increase values in the cost matrix R with respect to values in the Q matrix