In [1]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using Test
import Convex as cvx
import ECOS
using Random
using MathOptInterface
```

```
  Activating project at `c:\CMU\SEM II\OCRL\HW2_S25`
┌ Warning: The active manifest file has dependencies that were resolved with a different julia version (1.10.7). Unexpected behavior may occur.
└ @ nothing C:\CMU\SEM II\OCRL\HW2_S25\Manifest.toml:0
```

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.

Julia Warnings:

1. For a function `foo(x::Vector)` with 1 input argument, it is not neccessary to do `df_dx = FD.jacobian(_x -> foo(_x), x)`. Instead you can just do `df_dx = FD.jacobian(foo, x)`. If you do the first one, it can dramatically slow down your compliation time.
2. Do not define functions inside of other functions like this:

```
function foo(x)
    # main function foo

    function body(x)
        # function inside function (DON'T DO THIS)
        return 2*x
    end

    return body(x)
end
```

This will also slow down your compilation time dramatically.

Q1: Finite-Horizon LQR (55 pts)

For this problem we are going to consider a "double integrator" for our dynamics model. This system has a state $x \in \mathbb{R}^4$, and control $u \in \mathbb{R}^2$, where the state describes the 2D position p and velocity v of an object, and the control is the acceleration a of this object. The state and control are the following:

$$x = [p_1, p_2, v_1, v_2]$$
$$u = [a_1, a_2]$$

And the continuous time dynamics for this system are the following:

$$\dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} u$$

Part A: Discretize the model (5 pts)

Use the matrix exponential (`exp` in Julia) to discretize the continuous time model assuming we have a zero-order hold on the control. See this part of the first recitation if you're unsure of what to do.

In [2]:
```
# double integrator dynamics
function double_integrator_AB(dt)::Tuple{Matrix,Matrix}
    Ac = [0 0 1 0;
          0 0 0 1;
          0 0 0 0;
          0 0 0 0.]
    Bc = [0 0;
```

```
            0 0;
            1 0;
            0 1]
    nx, nu = size(Bc)

    Ad_Bc = [Ac Bc; zeros(nu, nx + nu)]
    exp_Ad_Bc = exp(Ad_Bc * dt)

    A = exp_Ad_Bc[1:nx, 1:nx]
    B = exp_Ad_Bc[1:nx, nx+1:end]

    @assert size(A) == (nx,nx)
    @assert size(B) == (nx,nu)

    return A, B
end
```

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

In [3]:
```
@testset "discrete time dynamics" begin
    dt = 0.1
    A,B = double_integrator_AB(dt)

    x = [1,2,3,4.]
    u = [-1,-3.]

    @test isapprox((A*x + B*u),[1.295, 2.385, 2.9, 3.7];atol = 1e-10)

end
```

```
Test Summary:          | Pass  Total  Time
discrete time dynamics |    1      1  3.9s
Test.DefaultTestSet("discrete time dynamics", Any[], 1, false, false, true, 1.73983835999e9, 1.739838363884e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_W6sZmlsZQ==.jl")
```

Part B: Finite Horizon LQR via Convex Optimization (15 pts)

We are now going to solve the finite horizon LQR problem with convex optimization. As we went over in class, this problem requires $Q \in S_+$(Q is symmetric positive semi-definite) and $R \in S_{++}$ (R is symmetric positive definite). With this, the optimization problem can be stated as the following:

$$\begin{aligned} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \left[\frac{1}{2}x_i^TQx_i + \frac{1}{2}u_i^TRu_i\right] + \frac{1}{2}x_N^TQ_fx_N \\ \text{st} \quad & x_1 = x_{\text{IC}} \\ & x_{i+1} = Ax_i + Bu_i \quad \text{for } i = 1,2,\ldots,N-1 \end{aligned}$$

This problem is a convex optimization problem since the cost function is a convex quadratic and the constraints are all linear equality constraints. We will setup and solve this exact problem using the `Convex.jl` modeling package. (See 2/16 Recitation video for help with this package. Notebook is here.) Your job in the block below is to fill out a function `Xcvx,Ucvx = convex_trajopt(A,B,Q,R,Qf,N,x_ic)`, where you will form and solve the above optimization problem.

In [4]:
```
# utilities for converting to and from vector of vectors <-> matrix
function mat_from_vec(X::Vector{Vector{Float64}})::Matrix
    # convert a vector of vectors to a matrix
    Xm = hcat(X...)
    return Xm
end
function vec_from_mat(Xm::Matrix)::Vector{Vector{Float64}}
    # convert a matrix into a vector of vectors
    X = [Xm[:,i] for i = 1:size(Xm,2)]
    return X
end
"""
X,U = convex_trajopt(A,B,Q,R,Qf,N,x_ic; verbose = false)

This function takes in a dynamics model x_{k+1} = A*x_k + B*u_k
and LQR cost Q,R,Qf, with a horizon size N, and initial condition
x_ic, and returns the optimal X and U's from the above optimization
problem. You should use the `vec_from_mat` function to convert the
solution matrices from cvx into vectors of vectors (vec_from_mat(X.value))
"""
function convex_trajopt(A::Matrix,      # A matrix
                        B::Matrix,      # B matrix
                        Q::Matrix,      # cost weight
                        R::Matrix,      # cost weight
                        Qf::Matrix,     # term cost weight
                        N::Int64,       # horizon size
                        x_ic::Vector;   # initial condition
                        verbose = false
```

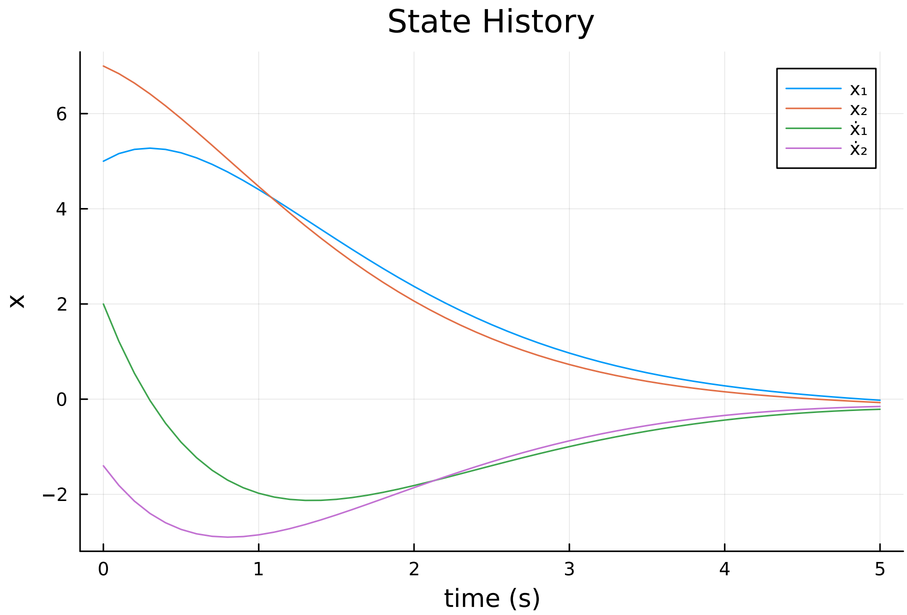
```
                         )::Tuple{Vector{Vector{Float64}},Vector{Vector{Float64}}}

    # check sizes of everything
    nx,nu = size(B)
    @assert size(A) == (nx, nx)
    @assert size(Q) == (nx, nx)
    @assert size(R) == (nu, nu)
    @assert size(Qf) == (nx, nx)
    @assert length(x_ic) == nx

    # TODO:
    # create cvx variables where each column is a time step
    X = cvx.Variable(nx, N)
    U = cvx.Variable(nu, N - 1)

    # create cost
    # hint: you can't do x'*Q*x in Convex.jl, you must do cvx.quadform(x,Q)
    # hint: add all of your cost terms to `cost`
    # hint: x_k = X[:,k], u_k = U[:,k]
    cost = 0
    for k = 1:(N-1)
        cost += cvx.quadform(X[:, k], Q) + cvx.quadform(U[:, k], R)
    end

    # add terminal cost
    cost += cvx.quadform(X[:, N], Qf)

    # initialize cvx problem
    prob = cvx.minimize(cost)

    # TODO: initial condition constraint
    # hint: you can add constraints to our problem like this:
    # prob.constraints = vcat(prob.constraints, (Gz == h))
    prob.constraints = []
    prob.constraints = vcat(prob.constraints, X[:, 1] == x_ic)

    for k = 1:(N-1)
        prob.constraints = vcat(prob.constraints, X[:, k+1] == A * X[:, k] + B * U[:, k])
    end

    # solve problem (silent solver tells us the output)
    cvx.solve!(prob, ECOS.Optimizer; silent = !verbose)

    if prob.status != MathOptInterface.OPTIMAL
        error("Convex.jl problem failed to solve for some reason")
    end

    # convert the solution matrices into vectors of vectors
    X = vec_from_mat(X.value)
    U = vec_from_mat(U.value)

    return X, U
end
```

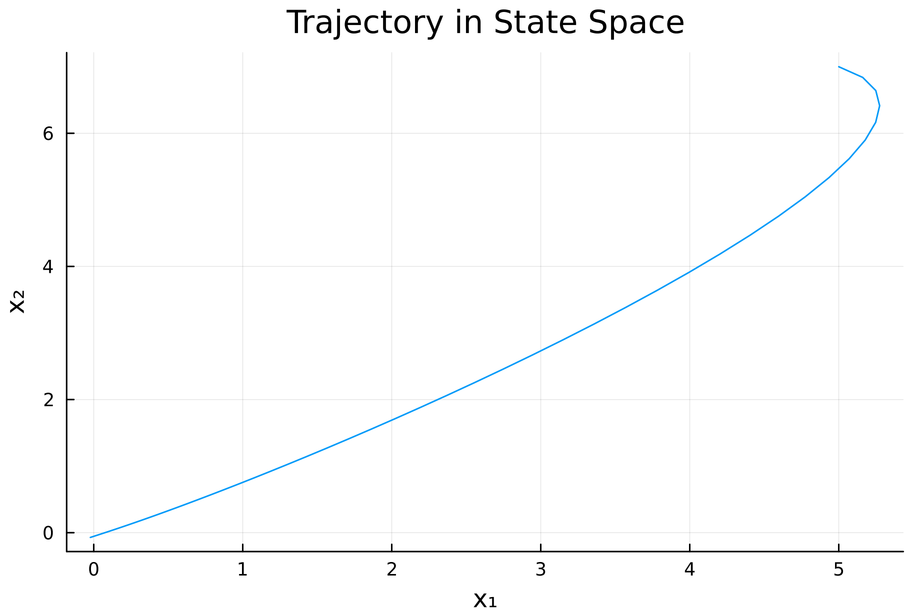
convex_trajopt

Now let's solve this problem for a given initial condition, and simulate it to see how it does:

In [5]:
```
@testset "LQR via Convex.jl" begin

    # problem setup stuff
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 5*Q

    # initial condition
    x_ic = [5,7,2,-1.4]

    # setup and solve our convex optimization problem (verbose = true for submission)
    Xcvx,Ucvx = convex_trajopt(A,B,Q,R,Qf,N,x_ic; verbose = false)

    # TODO: simulate with the dynamics with control Ucvx, storing the
    # state in Xsim

    # initial condition
    Xsim = [zeros(nx) for i = 1:N]
    Xsim[1] = 1*x_ic
```

```julia
    # TODO dynamics simulation
    for i in 1:(N-1)
        Xsim[i+1] = A * Xsim[i] + B * Ucvx[i][:]
    end


    @test length(Xsim) == N
    @test norm(Xsim[end])>1e-13
    #---------------------plotting----------------------
    Xsim_m = mat_from_vec(Xsim)

    # plot state history
    display(plot(t_vec,Xsim_m',label = ["x₁" "x₂" "ẋ₁" "ẋ₂"],
                 title = "State History",
                 xlabel = "time (s)", ylabel = "x"))

    # plot trajectory in x1 x2 space
    display(plot(Xsim_m[1,:],Xsim_m[2,:],
                 title = "Trajectory in State Space",
                 ylabel = "x₂", xlabel = "x₁", label = ""))
    #---------------------plotting----------------------

    # tests
    @test 1e-14 < maximum(norm.(Xsim .- Xcvx,Inf)) < 1e-3
    @test isapprox(Ucvx[1], [-7.8532442316767, -4.127120137234], atol = 1e-3)
    @test isapprox(Xcvx[end], [-0.02285990, -0.07140241, -0.21259, -0.1540299], atol = 1e-3)
    @test 1e-14 < norm(Xcvx[end] - Xsim[end]) < 1e-3
end
```






```
Test Summary:      | Pass  Total   Time
LQR via Convex.jl |    6      6  19.3s
Test.DefaultTestSet("LQR via Convex.jl", Any[], 6, false, false, true, 1.739838364609e9, 1.739838383914e9, false
, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X14sZmlsZQ==.jl")
```

Bellman's Principle of Optimality

Now we will test Bellman's Principle of optimality. This can be phrased in many different ways, but the main gist is that any section of an optimal trajectory must be optimal. Our original optimization problem was the above problem:

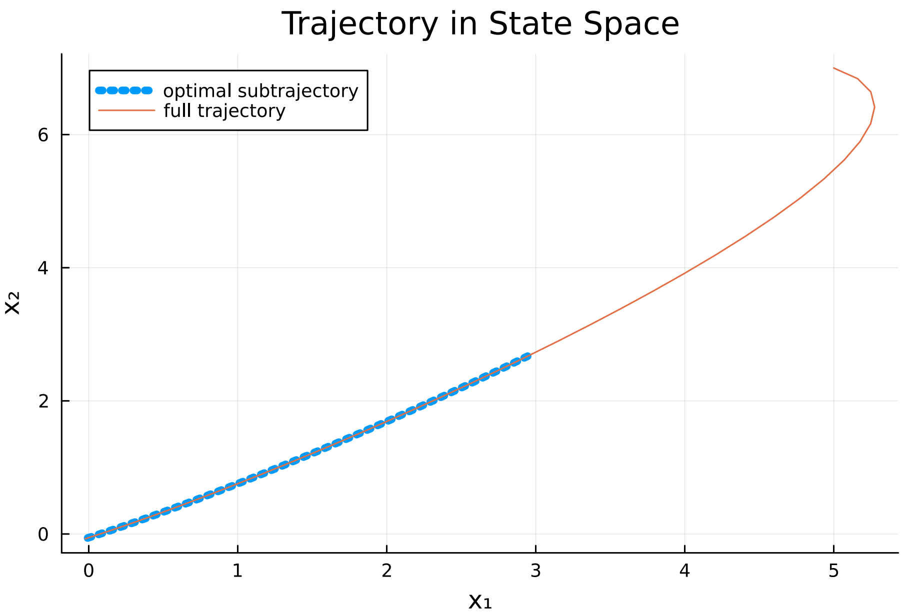
$$\begin{aligned} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N \\ \text{st} \quad & x_1 = x_{\text{IC}} \\ & x_{i+1} = A x_i + B u_i \quad \text{for } i = 1, 2, \ldots, N-1 \end{aligned}$$

which has a solution $x^*_{1:N}, u^*_{1:N-1}$. Now let's look at optimizing over a subsection of this trajectory. That means that instead of solving for $x_{1:N}, u_{1:N-1}$, we are now solving for $x_{L:N}, u_{L:N-1}$ for some new timestep $1 < L < N$. What we are going to do is take the initial condition from x^*_L from our original optimization problem, and setup a new optimization problem that optimizes over $x_{L:N}, u_{L:N-1}$:

$$\begin{aligned} \min_{x_{L:N},u_{L:N-1}} \quad & \sum_{i=L}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N \\ \text{st} \quad & x_L = x^*_L \\ & x_{i+1} = A x_i + B u_i \quad \text{for } i = L, L+1, \ldots, N-1 \end{aligned}$$

In [6]:

```
@testset "Bellman's Principle of Optimality" begin

    # problem setup
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    x0 = [5,7,2,-1.4] # initial condition
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 5*Q

    # solve for X_{1:N}, U_{1:N-1} with convex optimization
    Xcvx1,Ucvx1 = convex_trajopt(A,B,Q,R,Qf,N,x0; verbose = false)

    # now let's solve a subsection of this trajectory
    L = 18
    N_2 = N - L + 1

    # here is our updated initial condition from the first problem
    x0_2 = Xcvx1[L]
    Xcvx2,Ucvx2 = convex_trajopt(A,B,Q,R,Qf,N_2,x0_2; verbose = false)

    # test if these trajectories match for the times they share
    U_error = Ucvx1[L:end] .- Ucvx2
    X_error = Xcvx1[L:end] .- Xcvx2
    @test 1e-14 < maximum(norm.(U_error)) < 1e-3
    @test 1e-14 < maximum(norm.(X_error)) < 1e-3


    # ---------------------------plotting -----------------------------
    X1m = mat_from_vec(Xcvx1)
    X2m = mat_from_vec(Xcvx2)
    plot(X2m[1,:],X2m[2,:], label = "optimal subtrajectory", lw = 5, ls = :dot)
    display(plot!(X1m[1,:],X1m[2,:],
                  title = "Trajectory in State Space",
                  ylabel = "x₂", xlabel = "x₁", label = "full trajectory"))
    # ---------------------------plotting -----------------------------

    @test isapprox(Xcvx1[end], [-0.02285990, -0.07140241, -0.21259, -0.1540299], rtol = 1e-3)
    @test 1e-14 < norm(Xcvx1[end] - Xcvx2[end],Inf) < 1e-3
end
```




```
Test Summary:                       | Pass  Total  Time
Bellman's Principle of Optimality |    4      4  0.5s
Test.DefaultTestSet("Bellman's Principle of Optimality", Any[], 4, false, false, true, 1.739838383932e9, 1.73983
8384422e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X16sZmlsZQ=
=.jl")
```

Part C: Finite-Horizon LQR via Riccati (10 pts)

Now we are going to solve the original finite-horizon LQR problem:

$$\begin{aligned} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N \\ \text{st} \quad & x_1 = x_{\text{IC}} \\ & x_{i+1} = A x_i + B u_i \quad \text{for } i = 1, 2, \ldots, N-1 \end{aligned}$$

with a Riccati recursion instead of convex optimization. We describe our optimal cost-to-go function (aka the Value function) as the following:

$$V_k(x) = \frac{1}{2} x^T P_k x.$$

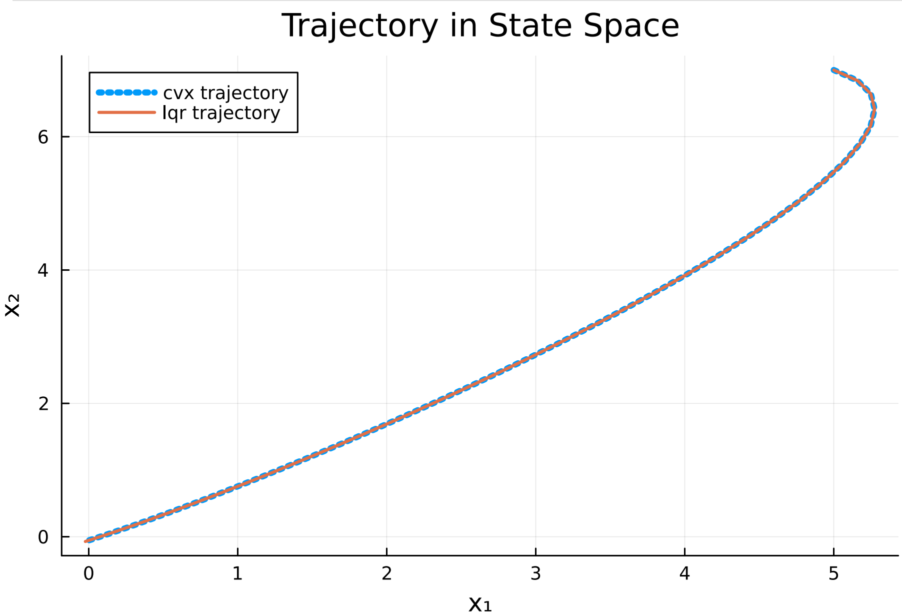
In [7]:

```julia
"""
use the Riccati recursion to calculate the cost to go quadratic matrix P and
optimal control gain K at every time step. Return these as a vector of matrices,
where P_k = P[k], and K_k = K[k]
"""
function fhlqr(A::Matrix, # A matrix
               B::Matrix, # B matrix
               Q::Matrix, # cost weight
               R::Matrix, # cost weight
               Qf::Matrix,# term cost weight
               N::Int64   # horizon size
               )::Tuple{Vector{Matrix{Float64}}, Vector{Matrix{Float64}}} # return two matrices

    # check sizes of everything
    nx,nu = size(B)
    @assert size(A) == (nx, nx)
    @assert size(Q) == (nx, nx)
    @assert size(R) == (nu, nu)
    @assert size(Qf) == (nx, nx)

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

    # initialize S[N] with Qf
    P[N] = deepcopy(Qf)

    # Ricatti
    for k = N-1:-1:1
        S = R + B'*P[k+1]*B
        K[k] = -S\(B'*P[k+1]*A)
        P[k] = Q + A'*P[k+1]*(A+B*K[k])
    end

    return P, K
```

```
end
```

fhlqr

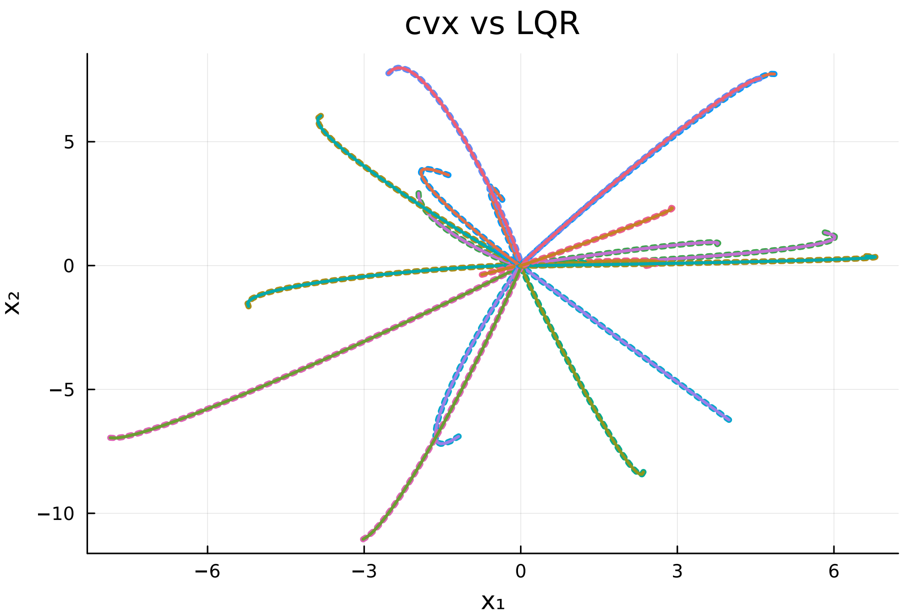
In [8]:
```
@testset "Convex trajopt vs LQR" begin

    # problem stuff
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    x0 = [5,7,2,-1.4] # initial condition
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 5*Q

    # solve for X_{1:N}, U_{1:N-1} with convex optimization
    Xcvx,Ucvx = convex_trajopt(A,B,Q,R,Qf,N,x0; verbose = false)
    P, K = fhlqr(A,B,Q,R,Qf,N)
    # now let's simulate using Ucvx
    Xsim_cvx = [zeros(nx) for i = 1:N]
    Xsim_cvx[1] = 1*x0
    Xsim_lqr = [zeros(nx) for i = 1:N]
    Xsim_lqr[1] = 1*x0
    for i = 1:N-1
        # simulate cvx control
        Xsim_cvx[i+1] = A*Xsim_cvx[i] + B*Ucvx[i]

        # TODO: use your FHLQR control gains K to calculate u_lqr
        # simulate lqr control
        u_lqr = K[i]*Xsim_lqr[i]
        Xsim_lqr[i+1] = A*Xsim_lqr[i] + B*u_lqr
    end


    @test isapprox(Xsim_lqr[end], [-0.02286201, -0.0714058, -0.21259, -0.154030], rtol = 1e-3)
    @test 1e-13 < norm(Xsim_lqr[end] - Xsim_cvx[end]) < 1e-3
    @test 1e-13 < maximum(norm.(Xsim_lqr - Xsim_cvx)) < 1e-3


    # ------------------------plotting-------------------------
    X1m = mat_from_vec(Xsim_cvx)
    X2m = mat_from_vec(Xsim_lqr)
    # plot trajectory in x1 x2 space
    plot(X1m[1,:],X1m[2,:], label = "cvx trajectory", lw = 4, ls = :dot)
    display(plot!(X2m[1,:],X2m[2,:],
             title = "Trajectory in State Space",
             ylabel = "x₂", xlabel = "x₁", lw = 2, label = "lqr trajectory"))
    # ------------------------plotting-------------------------

end
```




```
Test Summary:          | Pass  Total  Time
Convex trajopt vs LQR |    3      3  0.5s
Test.DefaultTestSet("Convex trajopt vs LQR", Any[], 3, false, false, true, 1.73983838445e9, 1.739838384969e9, fa
lse, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X22sZmlsZQ==.jl")
```

To emphasize that these two methods for solving the optimization problem result in the same solutions, we are now going to sample initial conditions and run both solutions. You will have to fill in your LQR policy again.

In [9]:
```
import Random
Random.seed!(1)
@testset "Convex trajopt vs LQR" begin

    # problem stuff
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 5*Q


    plot()
    for ic_iter = 1:20
        x0 = [5*randn(2); 1*randn(2)]
        # solve for X_{1:N}, U_{1:N-1} with convex optimization
        Xcvx,Ucvx = convex_trajopt(A,B,Q,R,Qf,N,x0; verbose = false)
        P, K = fhlqr(A,B,Q,R,Qf,N)
        Xsim_cvx = [zeros(nx) for i = 1:N]
        Xsim_cvx[1] = 1*x0
        Xsim_lqr = [zeros(nx) for i = 1:N]
        Xsim_lqr[1] = 1*x0
        for i = 1:N-1
            # simulate cvx control
            Xsim_cvx[i+1] = A*Xsim_cvx[i] + B*Ucvx[i]

            # TODO: use your FHLQR control gains K to calculate u_lqr
            # simulate lqr control
            u_lqr = K[i]*Xsim_lqr[i]
            Xsim_lqr[i+1] = A*Xsim_lqr[i] + B*u_lqr
        end

        @test 1e-13 < norm(Xsim_lqr[end] - Xsim_cvx[end]) < 1e-3
        @test 1e-13 < maximum(norm.(Xsim_lqr - Xsim_cvx)) < 1e-3

        # -----------------------plotting-------------------------
        X1m = mat_from_vec(Xsim_cvx)
        X2m = mat_from_vec(Xsim_lqr)
        plot!(X2m[1,:],X2m[2,:], label = "", lw = 4, ls = :dot)
        plot!(X1m[1,:],X1m[2,:], label = "", lw = 2)
    end
    display(plot!(title = "cvx vs LQR", ylabel = "x₂", xlabel = "x₁"))

end
```




```
Test Summary:          | Pass  Total  Time
Convex trajopt vs LQR |   40     40  1.0s
Test.DefaultTestSet("Convex trajopt vs LQR", Any[], 40, false, false, true, 1.739838384986e9, 1.739838386016e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X24sZmlsZQ==.jl")
```

Part D: Why LQR is so great (10 pts)

Now we are going to emphasize two reasons why the feedback policy from LQR is so useful:

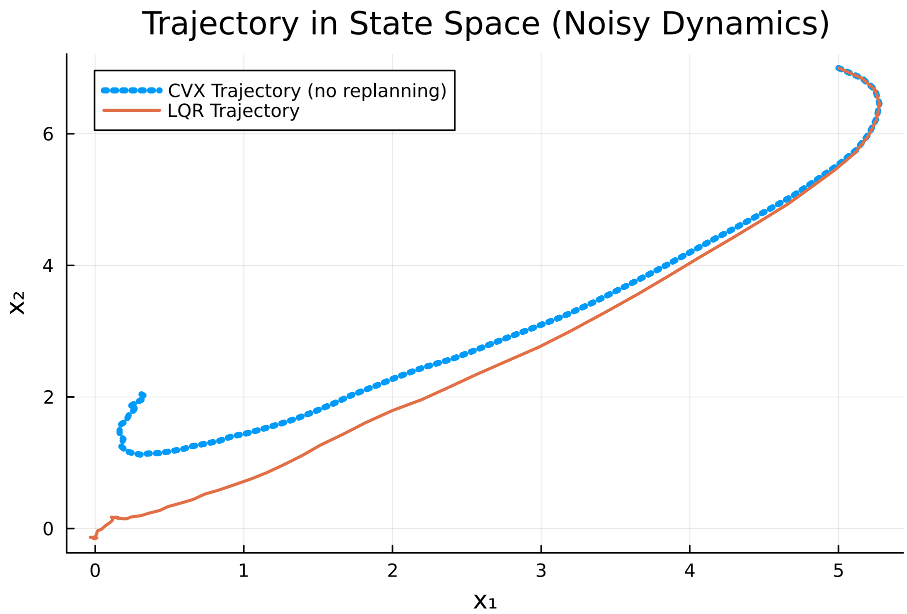
1. It is robust to noise and model uncertainty (the Convex approach would require re-solving of the problem every time the new state differs from the expected state (this is MPC, more on this in Q3)
2. We can drive to any achievable goal state with $u = -K(x - x_{goal})$

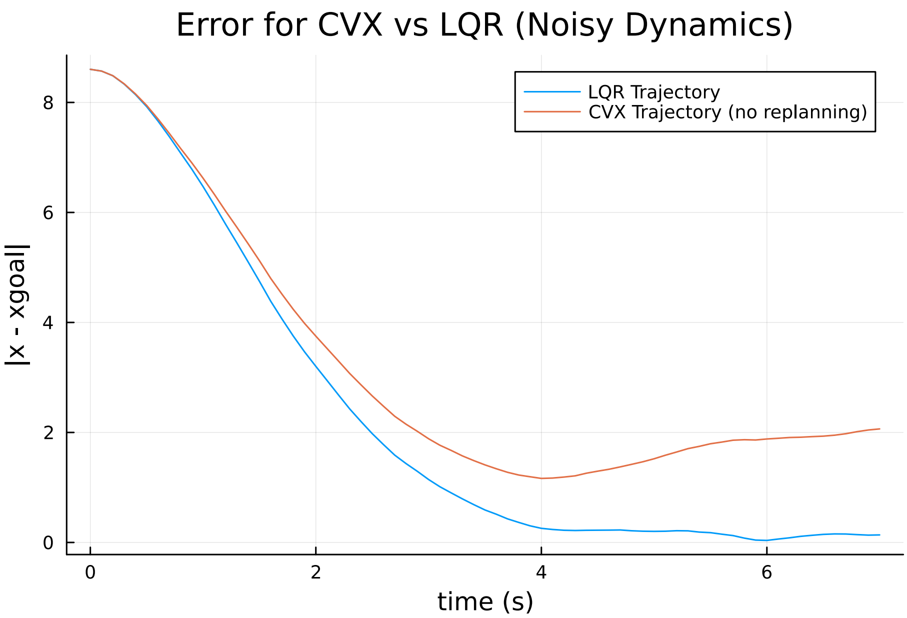
First we are going to look at a simulation with the following white noise:

$$x_{k+1} = Ax_k + Bu_k + \text{noise}$$

Where noise $\sim \mathcal{N}(0, \Sigma)$.

In [10]:
```julia
@testset "Why LQR is great reason 1" begin

    # problem stuff
    dt = 0.1
    tf = 7.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    x0 = [5,7,2,-1.4] # initial condition
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 10*Q

    # solve for X_{1:N}, U_{1:N-1} with convex optimization
    Xcvx,Ucvx = convex_trajopt(A,B,Q,R,Qf,N,x0; verbose = false)
    P, K = fhlqr(A,B,Q,R,Qf,N)
    # now let's simulate using Ucvx
    Xsim_cvx = [zeros(nx) for i = 1:N]
    Xsim_cvx[1] = 1*x0
    Xsim_lqr = [zeros(nx) for i = 1:N]
    Xsim_lqr[1] = 1*x0
    for i = 1:N-1
        # sampled noise to be added after each step
        noise = [.005*randn(2);.1*randn(2)]

        # simulate cvx control
        Xsim_cvx[i+1] = A*Xsim_cvx[i] + B*Ucvx[i] + noise

        # TODO: use your FHLQR control gains K to calculate u_lqr
        # simulate lqr control
        u_lqr = K[i]*Xsim_lqr[i]
        Xsim_lqr[i+1] = A*Xsim_lqr[i] + B*u_lqr + noise
    end

    # make sure our LQR achieved the goal
    @test norm(Xsim_cvx[end]) > norm(Xsim_lqr[end])
    @test norm(Xsim_lqr[end]) < .7
    @test norm(Xsim_cvx[end]) > 2.0


    # ------------------------plotting-------------------------
    X1m = mat_from_vec(Xsim_cvx)
    X2m = mat_from_vec(Xsim_lqr)
    # plot trajectory in x1 x2 space
    plot(X1m[1,:],X1m[2,:], label = "CVX Trajectory (no replanning)", lw = 4, ls = :dot)
    display(plot!(X2m[1,:],X2m[2,:],
                 title = "Trajectory in State Space (Noisy Dynamics)",
                 ylabel = "x₂", xlabel = "x₁", lw = 2, label = "LQR Trajectory"))
    ecvx = [norm(x[1:2]) for x in Xsim_cvx]
    elqr = [norm(x[1:2]) for x in Xsim_lqr]
    plot(t_vec, elqr, label = "LQR Trajectory",ylabel = "|x - xgoal|",
         xlabel = "time (s)", title = "Error for CVX vs LQR (Noisy Dynamics)")
    display(plot!(t_vec, ecvx, label = "CVX Trajectory (no replanning)"))
    # ------------------------plotting-------------------------

end
```

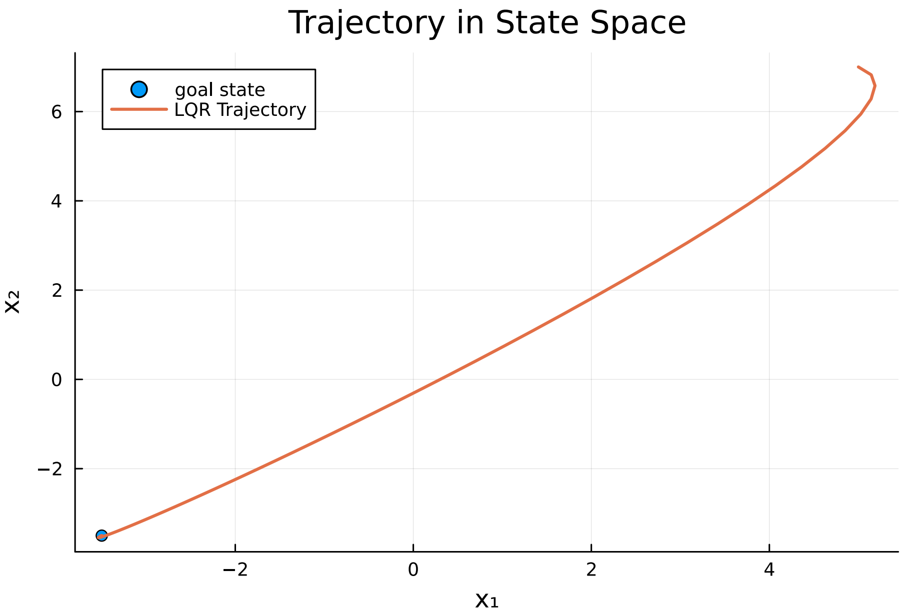





```
Test Summary:              | Pass  Total  Time
Why LQR is great reason 1 |    3      3  0.6s
Test.DefaultTestSet("Why LQR is great reason 1", Any[], 3, false, false, true, 1.73983838604e9, 1.739838386681e9
, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X26sZmlsZQ==.jl")
```

In [11]:
```
@testset "Why LQR is great reason 2" begin

    # problem stuff
    dt = 0.1
    tf = 20.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)
    x0 = [5,7,2,-1.4] # initial condition
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 10*Q

    P, K = fhlqr(A,B,Q,R,Qf,N)

    # TODO: specify any goal state with 0 velocity within a 5m radius of 0
    xgoal = [-3.5,-3.5,0,0]
    @test norm(xgoal[1:2])< 5
    @test norm(xgoal[3:4])<1e-13 # ensure 0 velocity

    Xsim_lqr = [zeros(nx) for i = 1:N]
    Xsim_lqr[1] = 1*x0

    for i = 1:N-1
        # TODO: use your FHLQR control gains K to calculate u_lqr
        # simulate lqr control
        u_lqr = K[i]*(Xsim_lqr[i]-xgoal)
        Xsim_lqr[i+1] = A*Xsim_lqr[i] + B*u_lqr
    end
```

```
    @test norm(Xsim_lqr[end][1:2] - xgoal[1:2]) < .1

    # ------------------------plotting-------------------------
    Xm = mat_from_vec(Xsim_lqr)
    plot(xgoal[1:1],xgoal[2:2],seriestype = :scatter, label = "goal state")
    display(plot!(Xm[1,:],Xm[2,:],
                 title = "Trajectory in State Space",
                 ylabel = "x₂", xlabel = "x₁", lw = 2, label = "LQR Trajectory"))


end
```




```
Test Summary:              | Pass  Total  Time
Why LQR is great reason 2 |    3      3  0.4s
Test.DefaultTestSet("Why LQR is great reason 2", Any[], 3, false, false, true, 1.739838386696e9, 1.739838387109e
9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X30sZmlsZQ==.jl")
```

Part E: Infinite-horizon LQR (10 pts)

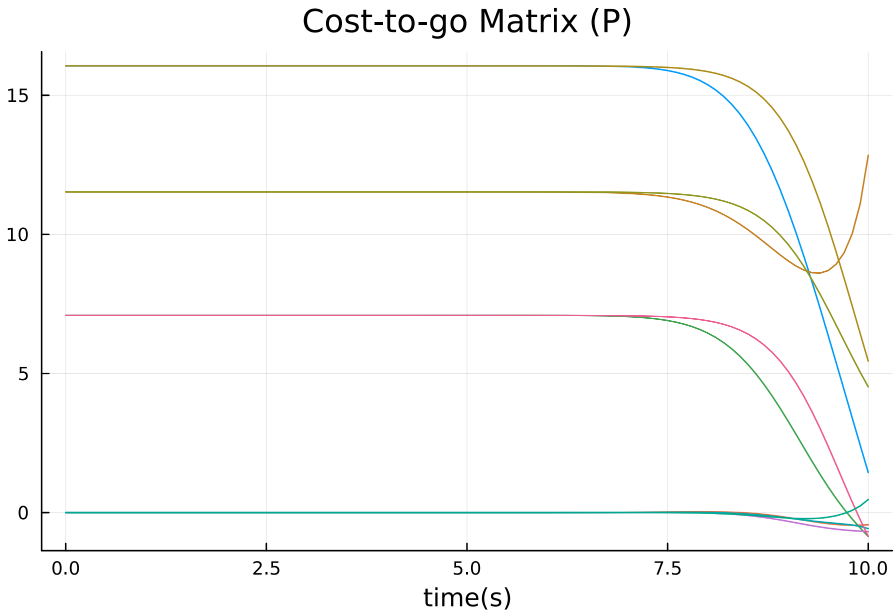
Up until this point, we have looked at finite-horizon LQR which only considers a finite number of timesteps in our trajectory. When this problem is solved with a Riccati recursion, there is a new feedback gain matrix K_k for each timestep. As the length of the trajectory increases, the first feedback gain matrix K_1 will begin to converge on what we call the "infinite-horizon LQR gain". This is the value that K_1 converges to as $N \to \infty$.

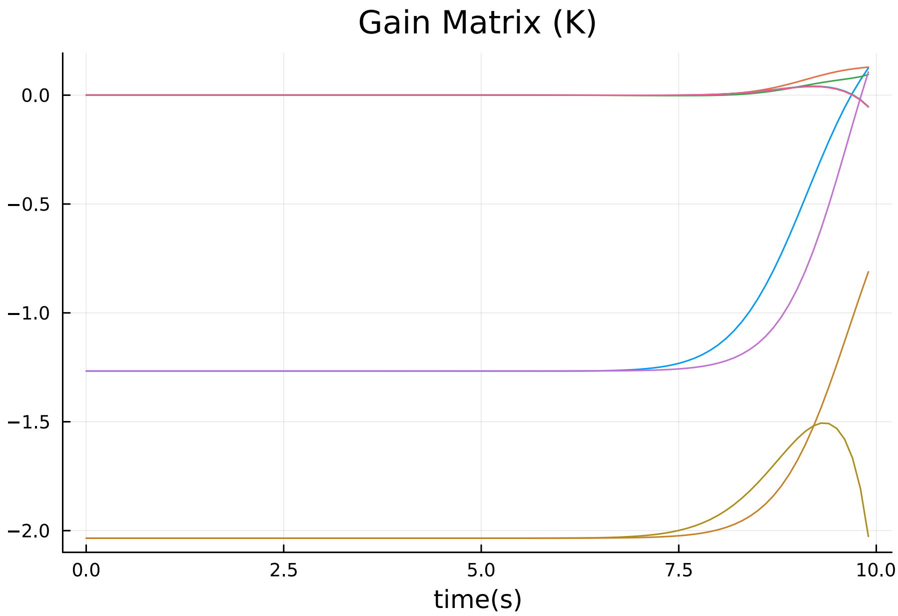
Below, we will plot the values of P and K throughout the horizon and observe this convergence.

In [12]:

```
# half vectorization of a matrix
function vech(A)
    return A[tril(trues(size(A)))]
end
@testset "P and K time analysis" begin

    # problem stuff
    dt = 0.1
    tf = 10.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)

    # cost terms
    Q = diagm(ones(nx))
    R = .5*diagm(ones(nu))
    Qf = randn(nx,nx); Qf = Qf'*Qf + I;

    P, K = fhlqr(A,B,Q,R,Qf,N)

    Pm = hcat(vech.(P)...)
    Km = hcat(vec.(K)...)

    # make sure these things converged
    @test 1e-13 < norm(P[1] - P[2]) < 1e-3
    @test 1e-13 < norm(K[1] - K[2]) < 1e-3

    display(plot(t_vec, Pm', label = "",title = "Cost-to-go Matrix (P)", xlabel = "time(s)"))
    display(plot(t_vec[1:end-1], Km', label = "",title = "Gain Matrix (K)", xlabel = "time(s)"))
```

```
end
```






```
Test Summary:          | Pass  Total  Time
P and K time analysis |    2      2  0.7s
Test.DefaultTestSet("P and K time analysis", Any[], 2, false, false, true, 1.739838387123e9, 1.739838387816e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X32sZmlsZQ==.jl")
```

Complete this infinite horizon LQR function where you do a Riccati recursion until the cost to go matrix P converges:

||

And return the steady state P and K.

In [16]:

```julia
"""
P,K = ihlqr(A,B,Q,R)

TODO: complete this infinite horizon LQR function where
you do the Riccati recursion until the cost to go matrix
P converges to a steady value |P_k - P_{k+1}| ≤ tol
"""
function ihlqr(A::Matrix,       # vector of A matrices
               B::Matrix,       # vector of B matrices
               Q::Matrix,       # cost matrix Q
               R::Matrix;       # cost matrix R
               max_iter = 1000, # max iterations for Riccati
               tol = 1e-5       # convergence tolerance
               )::Tuple{Matrix, Matrix} # return two matrices

    # get size of x and u from B
    nx, nu = size(B)

    # initialize S with Q
    P = deepcopy(Q)
```

```julia
    K = zeros(nu, nx)

    # Riccati
    for riccati_iter = 1:max_iter
        S = (R + B'*P*B)
        K = S\(B'*P*A)
        P_new = Q + A'*P*(A-B*K)
        if norm(P_new - P) <= tol
            return P_new, K
        end
        P = P_new
    end
    error("ihlqr did not converge")
end
@testset "ihlqr test" begin
    # problem stuff
    dt = 0.1
    A,B = double_integrator_AB(dt)
    nx,nu = size(B)

    # we're just going to modify the system a little bit
    # so the following graphs are still interesting

    Q = diagm(ones(nx))
    R = .5*diagm(ones(nu))
    P, K = ihlqr(A,B,Q,R)

    # check this P is in fact a solution to the Riccati equation
    @test typeof(P) == Matrix{Float64}
    @test typeof(K) == Matrix{Float64}
    @test 1e-13 < norm(Q + K'*R*K + (A - B*K)'P*(A - B*K) - P) < 1e-3

end
```

```
Test Summary: | Pass  Total  Time
ihlqr test    |    3      3  0.0s
Test.DefaultTestSet("ihlqr test", Any[], 3, false, false, true, 1.73983894737e9, 1.739838947385e9, false, "c:\\C
MU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X34sZmlsZQ==.jl")
```

In [14]:

```
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 project at `c:\CMU\SEM II\OCRL\HW2_S25`
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:

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

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:

No description has been provided for this image

(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 [15]:
```
"""
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

# true nonlinear dynamics of the system
# if I want to simulate, this is what I do
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)
    return x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
end
```

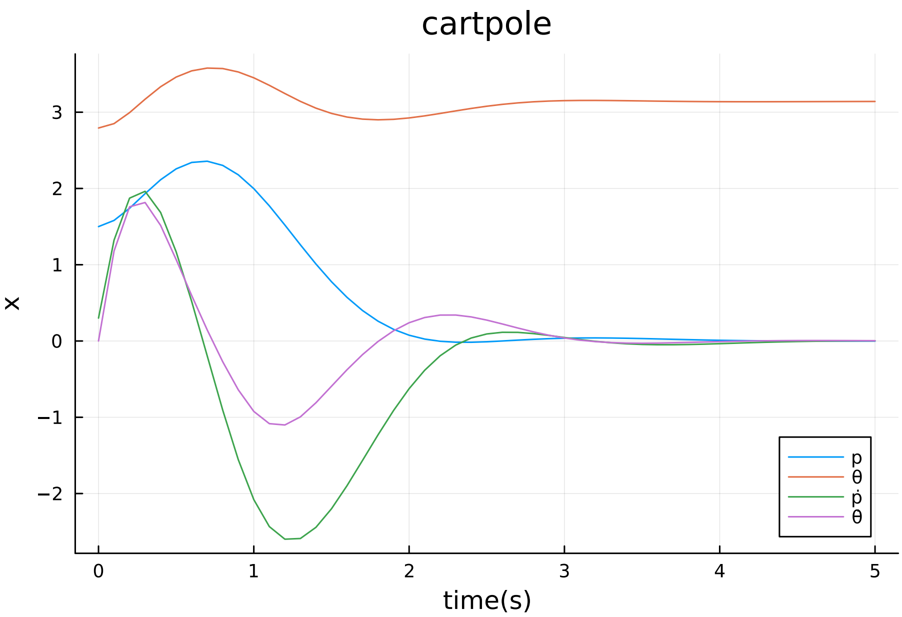
```
rk4 (generic function with 1 method)
```

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

    # states and control sizes
```

```julia
    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))

    A = FD.jacobian(x -> dynamics(params_est, x, ugoal), xgoal)
    B = FD.jacobian(u -> dynamics(params_est, xgoal, u), ugoal)

    Ad_Bc = [A B; zeros(nu, nx + nu)]
    exp_Ad_Bc = exp(Ad_Bc * dt)

    A = exp_Ad_Bc[1:nx, 1:nx]
    B = exp_Ad_Bc[1:nx, nx+1:end]

    Kinf = zeros(1,4)

    max_iter = 10000
    tol = 1e-8
    P = deepcopy(Q)

    for riccati_iter = 1:max_iter
        S = R + B'*P*B
        Kinf = S\(B'*P*A)
        P_new = Q + A'*P*(A-B*Kinf)
        if norm(P_new - P) <= tol
            break
        end
        P = P_new
    end

    # TODO: simulate this controlled system with rk4(params_real, ...)
    for i in 1:N-1
        u = -Kinf*(X[i]-xgoal) + ugoal
        X[i+1] = rk4(params_real, X[i], u, 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: 127.0.0.1:8706, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8706
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

```
Test Summary: | Pass  Total  Time
LQR about eq  |    3      3  8.7s
Test.DefaultTestSet("LQR about eq", Any[], 3, false, false, true, 1.740008332326e9, 1.740008340983e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X10sZmlsZQ==.jl")
```

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

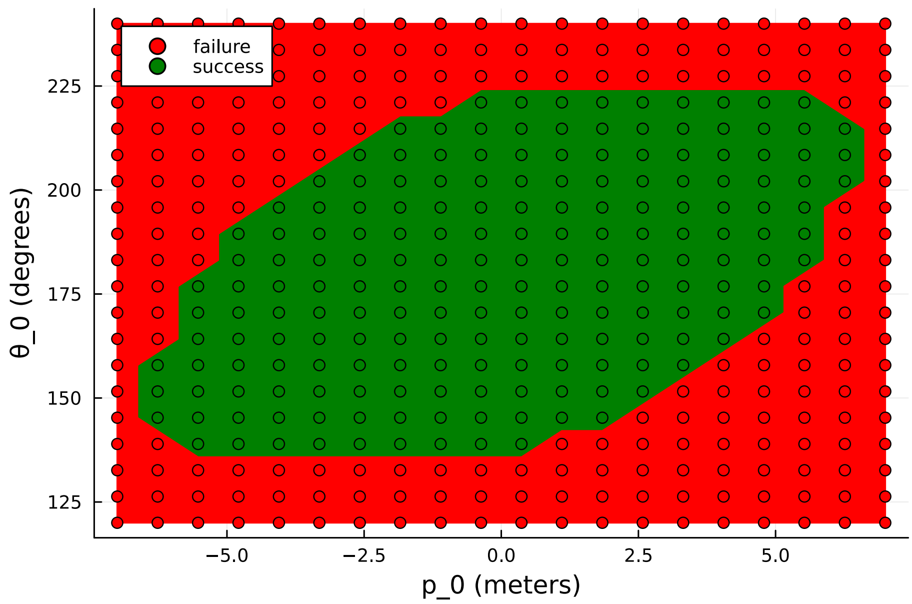
In part A we built a controller for the cartpole that was based on a linearized version of the system dynamics. This linearization took place at the `(xgoal, ugoal)`, so we should only really expect this model to be accurate if we are close to this linearization point (think small angle approximation). 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 [24]:
```
function create_initial_conditions()
    # create a span of initial configurations
    M=20
```

```julia
    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, xgoal, 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
    """

    x0= 1 * initial_condition

    is_controlled = false

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

    x = x0

    for i in 1:N-1
        u = -Kinf*(x-xgoal)
        x_new = rk4(params_real, x, u, dt)
        if(norm(x_new) > 100)
            return false
        end
        x = x_new
    end

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

    # you should consider the simulation to have been successfuly controlled if the
    # L2 norm of |xfinal - xgoal| < 0.1. (norm(xfinal-xgoal) < 0.1 in Julia)
    if norm(x - xgoal) < 0.1
        is_controlled = true
    end


    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
    A = FD.jacobian(x -> dynamics(params_est, x, ugoal), xgoal)
    B = FD.jacobian(u -> dynamics(params_est, xgoal, u), ugoal)

    Ad_Bc = [A B; zeros(nu, nx + nu)]
    exp_Ad_Bc = exp(Ad_Bc * dt)
```

```
    A = exp_Ad_Bc[1:nx, 1:nx]
    B = exp_Ad_Bc[1:nx, nx+1:end]

    Kinf = zeros(1,4)

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

    max_iter = 1000
    tol = 1e-8
    P = deepcopy(Q)

    for riccati_iter = 1:max_iter
        S = R + B'*P*B
        Kinf = S\(B'*P*A)
        P_new = Q + A'*P*(A-B*Kinf)
        if norm(P_new - P) <= tol
            break
        end
        P = P_new
    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, rad2deg.(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:

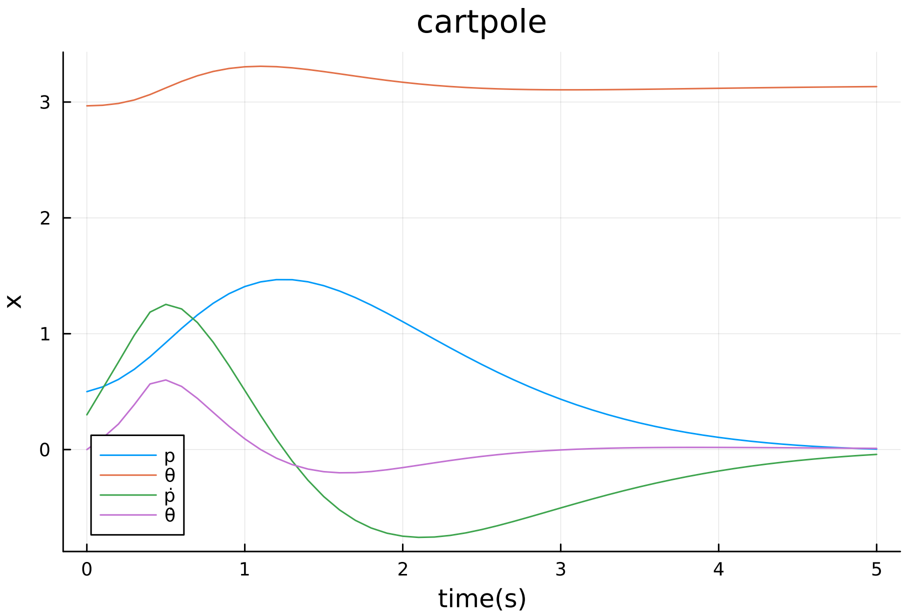
$\|x(5.0) - x_{\text{goal}}\|_2 =$ `norm(X[N] - xgoal) < 0.1`

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 [27]:
```
@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
    A = FD.jacobian(x -> dynamics(params_est, x, ugoal), xgoal)
    B = FD.jacobian(u -> dynamics(params_est, xgoal, u), ugoal)

    Ad_Bc = [A B; zeros(nu, nx + nu)]
    exp_Ad_Bc = exp(Ad_Bc * dt)

    A = exp_Ad_Bc[1:nx, 1:nx]
    B = exp_Ad_Bc[1:nx, nx+1:end]

    # cost terms
    Q = diagm([1,1,.01,.01])
    R = 1*diagm(ones(nu))

    Kinf = zeros(1,4)

    max_iter = 1000
    tol = 1e-8
    P = deepcopy(Q)

    for riccati_iter = 1:max_iter
        S = R + B'*P*B
        Kinf = S\(B'*P*A)
        P_new = Q + A'*P*(A-B*Kinf)
        if norm(P_new - P) <= tol
            break
        end
        P = P_new
    end


    # vector of length 1 vectors for our control
    U = [[0.0] 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 i in 1:N-1
        U[i] = -Kinf*(X[i]-xgoal) + ugoal
        U[i] .= clamp.(U[i], -3.0, 3.0)
        X[i+1] = rk4(params_real, X[i], U[i], dt)
    end
```

```julia
    # ---------------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
```




```
┌ Info: Listening on: 127.0.0.1:8707, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8707
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

```
Test Summary: | Pass  Total  Time
LQR about eq  |    4      4  5.1s
Test.DefaultTestSet("LQR about eq", Any[], 4, false, false, true, 1.740008866499e9, 1.740008871633e9, false, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X14sZmlsZQ==.jl")
```

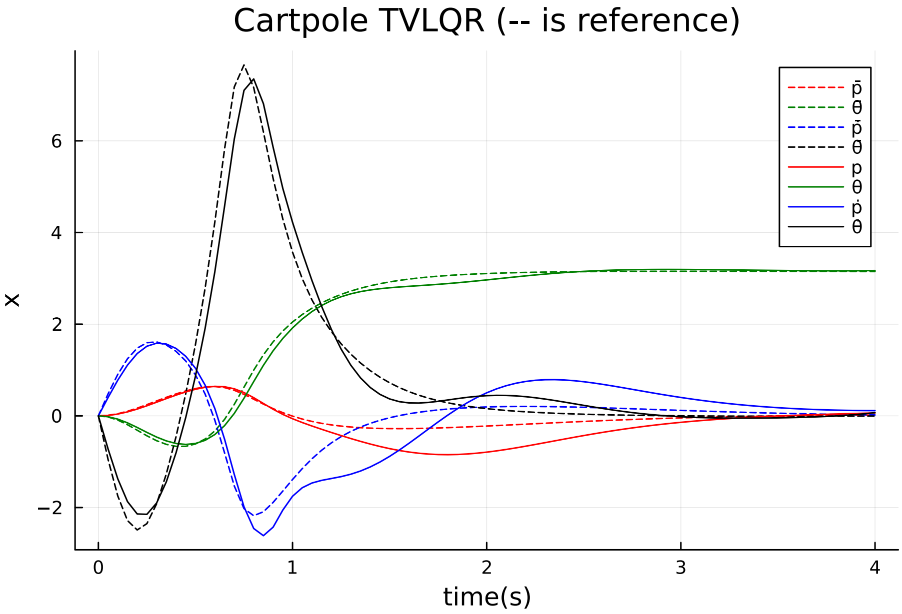
Part D: TVLQR for trajectory tracking (15 pts)

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 [30]:

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

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

    U = [[0.0] for i = 1:N-1]

    # TODO: solve for tvlqr gains K
    for i in N-1:-1:1
        A = FD.jacobian(x -> dynamics(params_est, x, Ubar[i]), Xbar[i])
        B = FD.jacobian(u -> dynamics(params_est, Xbar[i], u), Ubar[i])

        Ad_Bc = [A B; zeros(nu, nx + nu)]
        exp_Ad_Bc = exp(Ad_Bc * dt)

        A = exp_Ad_Bc[1:nx, 1:nx]
        B = exp_Ad_Bc[1:nx, nx+1:end]

        S = R + B'*P[i+1]*B
        K[i] = S\(B'*P[i+1]*A)
        P[i] = Q + A'*P[i+1]*(A-B*K[i])
    end



    # TODO: simulate this controlled system with rk4(params_real, ...)
    for i in 1:N-1
        U[i] = -K[i]*(X[i]-Xbar[i]) + Ubar[i]
        X[i+1] = rk4(params_real, X[i], U[i], 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 = ["p" "θ" "ṗ" "θ̇"],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: 127.0.0.1:8708, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8708
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

```
Test Summary: | Pass  Total  Time
track swingup |    3      3  5.4s
Test.DefaultTestSet("track swingup", Any[], 3, false, false, true, 1.740009367918e9, 1.740009373356e9, false, "c
:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X16sZmlsZQ==.jl")
```

Processing math: 100%

In [1]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
import MeshCat as mc
using Test
using Random
import Convex as cvx
import ECOS      # the solver we use in this hw
# import Hypatia # other solvers you can try
# import COSMO   # other solvers you can try
using ProgressMeter
include(joinpath(@__DIR__,"utils/rendezvous.jl"))
```

```
  Activating project at `c:\CMU\SEM II\OCRL\HW2_S25`
┌ Warning: The active manifest file has dependencies that were resolved with a different julia version (1.10.7). Unexpected behavior may occur.
└ @ nothing C:\CMU\SEM II\OCRL\HW2_S25\Manifest.toml:0
thruster_model (generic function with 1 method)
```

Notes:

1. 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.
2. Things in space move very slowly (by design), because of this, you may want to speed up the animations when you're viewing them. You can do this in MeshCat by doing `Open Controls -> Animations -> Time Scale`, to modify the time scale. You can also play/pause/scrub from this menu as well.
3. You can move around your view in MeshCat by `clicking + dragging`, and you can pan with `right click + dragging`, and zoom with the scroll wheel on your mouse (or trackpad specific alternatives).

In [2]:
```julia
# utilities for converting to and from vector of vectors <-> matrix
function mat_from_vec(X::Vector{Vector{Float64}})::Matrix
    # convert a vector of vectors to a matrix
    Xm = hcat(X...)
    return Xm
end
function vec_from_mat(Xm::Matrix)::Vector{Vector{Float64}}
    # convert a matrix into a vector of vectors
    X = [Xm[:,i] for i = 1:size(Xm,2)]
    return X
end
```

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

Is LQR the answer for everything?

Unfortunately, no. LQR is great for problems with true quadratic costs and linear dynamics, but this is a very small subset of convex trajectory optimization problems. While a quadratic cost is common in control, there are other available convex cost functions that may better motivate the desired behavior of the system. These costs can be things like an L1 norm on the control inputs ($\|u\|_1$), or an L2 goal error ($\|x - x_{goal}\|_2$). Also, control problems often have constraints like path constraints, control bounds, or terminal constraints, that can't be handled with LQR. With the addition of these constraints, the trajectory optimization problem is stil convex and easy to solve, but we can no longer just get an optimal gain K and apply a feedback policy in these situations.

The solution to this is Model Predictive Control (MPC). In MPC, we are setting up and solving a convex trajectory optimization at every time step, optimizing over some horizon or window into the future, and executing the first control in the solution. To see how this works, we are going to try this for a classic space control problem: the rendezvous.

Q3: Optimal Rendezvous and Docking (55 pts)

In this example, we are going to use convex optimization to control the SpaceX Dragon 1 spacecraft as it docks with the International Space Station (ISS). The dynamics of the Dragon vehicle can be modeled with Clohessy-Wiltshire equations, which is a linear dynamics model in continuous time. The state and control of this system are the following:

$$x = [r_x, r_y, r_z, v_x, v_y, v_z]^T,$$
$$u = [t_x, t_y, t_z]^T,$$

where r is a relative position of the Dragon spacecraft with respect to the ISS, v is the relative velocity, and t is the thrust on the spacecraft. The continuous time dynamics of the vehicle are the following:

$$\dot{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 3n^2 & 0 & 0 & 0 & 2n & 0 \\ 0 & 0 & 0 & -2n & 0 & 0 \\ 0 & 0 & -n^2 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} u,$$

where $n = \sqrt{\mu/a^3}$, with μ being the standard gravitational parameter, and a being the semi-major axis of the orbit of the ISS.

We are going to use three different techniques for solving this control problem, the first is LQR, the second is convex trajectory optimization, and the third is convex MPC where we will be able to account for unmodeled dynamics in our system (the "sim to real" gap).

Part A: Discretize the dynamics (5 pts)

Use the matrix exponential to convert the linear ODE into a linear discrete time model (hint: the matrix exponential is just `exp()` in Julia when called on a matrix.

In [7]:

```
function create_dynamics(dt::Real)::Tuple{Matrix,Matrix}
    mu = 3.986004418e14 # standard gravitational parameter
    a = 6971100.0       # semi-major axis of ISS
    n = sqrt(mu/a^3)    # mean motion

    # continuous time dynamics ẋ = Ax + Bu
    A = [0     0  0    1   0   0;
         0     0  0    0   1   0;
         0     0  0    0   0   1;
         3*n^2 0  0    0   2*n 0;
         0     0  0   -2*n 0   0;
         0     0 -n^2  0   0   0]

    B = Matrix([zeros(3,3);0.1*I(3)])

    # TODO: convert to discrete time X_{k+1} = Ad*x_k + Bd*u_k
    nx = 6;
    nu = 3;

    Ad_Bc = [A B; zeros(nu, nx + nu)]
    exp_Ad_Bc = exp(Ad_Bc * dt)

    Ad = exp_Ad_Bc[1:nx, 1:nx]
    Bd = exp_Ad_Bc[1:nx, nx+1:end]

    return Ad, Bd
end
```

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

In [8]:

```
@testset "discrete dynamics" begin
    A,B = create_dynamics(1.0)

    x = [1,3,-.3,.2,.4,-.5]
    u = [-.1,.5,.3]

    # test these matrices
    @test isapprox(A*x + B*u, [1.195453, 3.424786, -0.78499972, 0.190925, 0.4495759, -0.4699993], atol = 1e-3)
    @test isapprox(det(A), 1, atol = 1e-8)
    @test isapprox(norm(B,Inf), 0.0999999803, atol = 1e-5)

end
```

```
Test Summary:     | Pass  Total  Time
discrete dynamics |    3      3  3.1s
Test.DefaultTestSet("discrete dynamics", Any[], 3, false, false, true, 1.740009718095e9, 1.740009721151e9, false
, "c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X10sZmlsZQ==.jl")
```

Part B: LQR (10 pts)

Now we will take a given reference trajectory `X_ref` and track it with finite-horizon LQR. Remember that finite-horizon LQR is solving this problem: min where our policy is u_i = -K_i(x_i - x_{ref, i}). Use your code from the previous problem with your `fhlqr` function to generate your gain matrices.

One twist we will throw into this is control constraints `u_min` and `u_max`. You should use the function `clamp.(u, u_min, u_max)` to clamp the values of your `u` to be within this range.

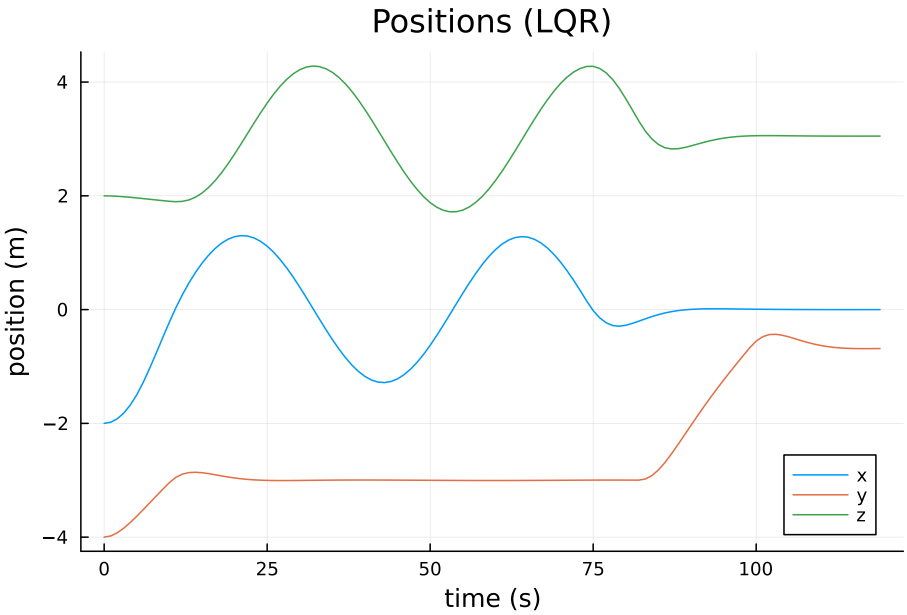
If implemented correctly, you should see the Dragon spacecraft dock with the ISS successfuly, but only after it crashes through the ISS a little bit.

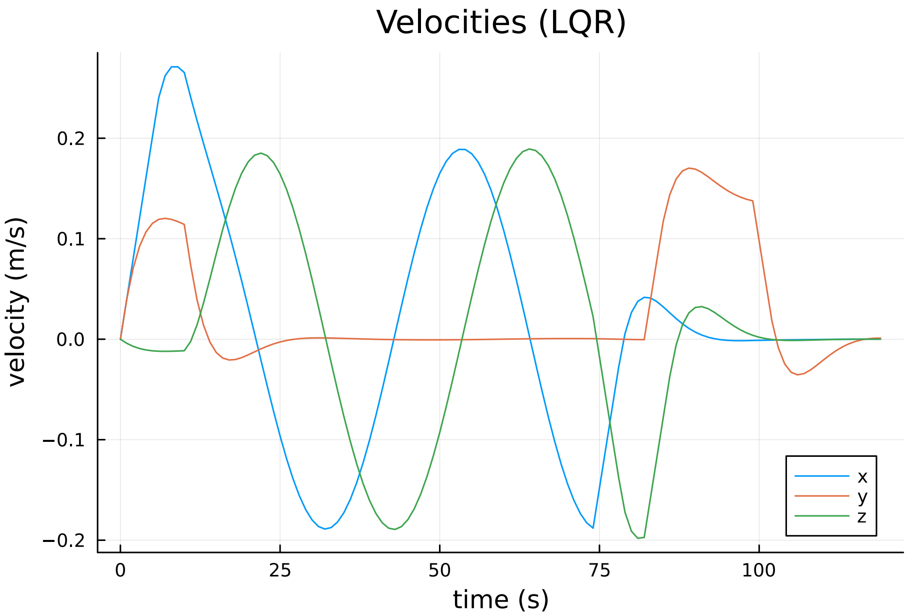
In [9]:
```
function fhlqr(A::Matrix, # A matrix
    B::Matrix, # B matrix
    Q::Matrix, # cost weight
    R::Matrix, # cost weight
    Qf::Matrix,# term cost weight
    N::Int64   # horizon size
    )::Tuple{Vector{Matrix{Float64}}, Vector{Matrix{Float64}}} # return two matrices

    # check sizes of everything
    nx,nu = size(B)
    @assert size(A) == (nx, nx)
    @assert size(Q) == (nx, nx)
    @assert size(R) == (nu, nu)
    @assert size(Qf) == (nx, nx)

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

    # initialize S[N] with Qf
    P[N] = deepcopy(Qf)

    # Ricatti
    for k = N-1:-1:1
    S = R + B'*P[k+1]*B
    K[k] = -S\(B'*P[k+1]*A)
    P[k] = Q + A'*P[k+1]*(A+B*K[k])
    end

    return P, K
end
```

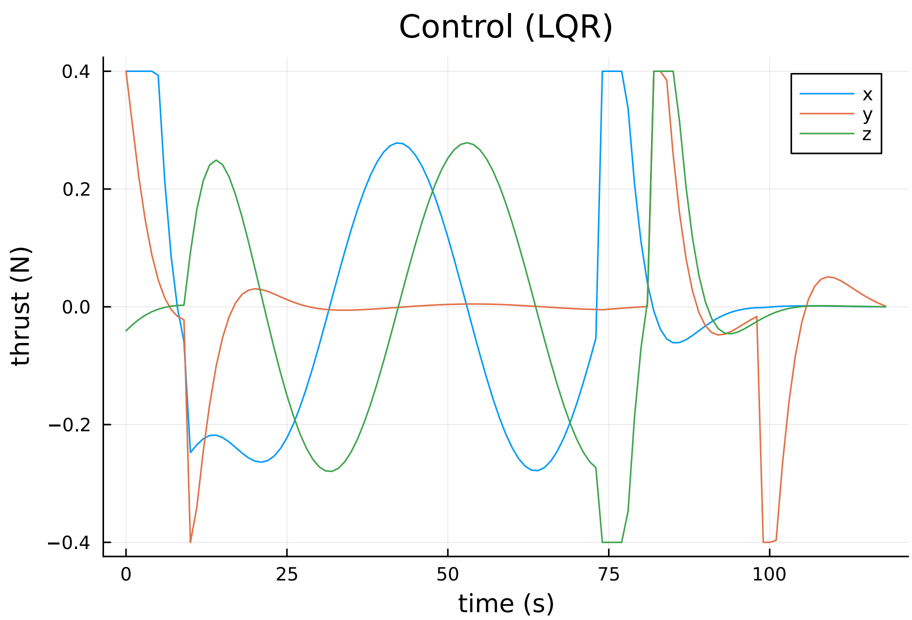
```
fhlqr (generic function with 1 method)
```

In [15]:
```
@testset "LQR rendezvous" begin

    # create our discrete time model
    dt = 1.0
    A,B = create_dynamics(dt)

    # get our sizes for state and control
    nx,nu = size(B)

    # initial and goal states
    x0 = [-2;-4;2;0;0;.0]
    xg = [0,-.68,3.05,0,0,0]

    # bounds on U
    u_max = 0.4*ones(3)
    u_min = -u_max

    # problem size and reference trajectory
    N = 120
    t_vec = 0:dt:((N-1)*dt)
    X_ref = desired_trajectory_long(x0,xg,200,dt)[1:N]

    # TODO: FHLQR
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 10*Q
    # TODO get K's from fhlqr
    P, K = fhlqr(A, B, Q, R, Qf, N)
    # P = [P for i = 1:N]
    # K = [K for i = 1:N-1]

    # simulation
    X_sim = [zeros(nx) for i = 1:N]
    U_sim = [zeros(nu) for i = 1:N-1]
    X_sim[1] = x0
    for i = 1:(N-1)
        # TODO: put LQR control law here
        # make sure to clamp
        U_sim[i] = K[i]*(X_sim[i] - X_ref[i])
        U_sim[i] .= clamp.(U_sim[i], u_min, u_max)
        # simulate 1 step
        X_sim[i+1] = A*X_sim[i] + B*U_sim[i]
    end

    # -------------plotting/animation---------------------------
    Xm = mat_from_vec(X_sim)
```

```julia
    Um = mat_from_vec(U_sim)
    display(plot(t_vec,Xm[1:3,:]',title = "Positions (LQR)",
                 xlabel = "time (s)", ylabel = "position (m)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec,Xm[4:6,:]',title = "Velocities (LQR)",
            xlabel = "time (s)", ylabel = "velocity (m/s)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec[1:end-1],Um',title = "Control (LQR)",
            xlabel = "time (s)", ylabel = "thrust (N)",
                 label = ["x" "y" "z"]))

    # feel free to toggle `show_reference`
    display(animate_rendezvous(X_sim, X_ref, dt;show_reference = false))
    # -------------plotting/animation--------------------------


    # testing
    xs=[x[1] for x in X_sim]
    ys=[x[2] for x in X_sim]
    zs=[x[3] for x in X_sim]
    @test norm(X_sim[end] - xg) < .01 # goal
    @test (xg[2] + .1) < maximum(ys) < 0 # we should have hit the ISS
    @test maximum(zs) >= 4 # check to see if you did the circle
    @test minimum(zs) <= 2 # check to see if you did the circle
    @test maximum(xs) >= 1 # check to see if you did the circle
    @test maximum(norm.(U_sim,Inf)) <= 0.4 # control constraints satisfied

end
```








```
┌ Info: Listening on: 127.0.0.1:8711, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8711
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

```
Test Summary:  | Pass  Total  Time
LQR rendezvous |    6      6  0.9s
Test.DefaultTestSet("LQR rendezvous", Any[], 6, false, false, true, 1.74001027413e9, 1.740010275013e9, false, "c
:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X12sZmlsZQ==.jl")
```

Part C: Convex Trajectory Optimization (15 pts)

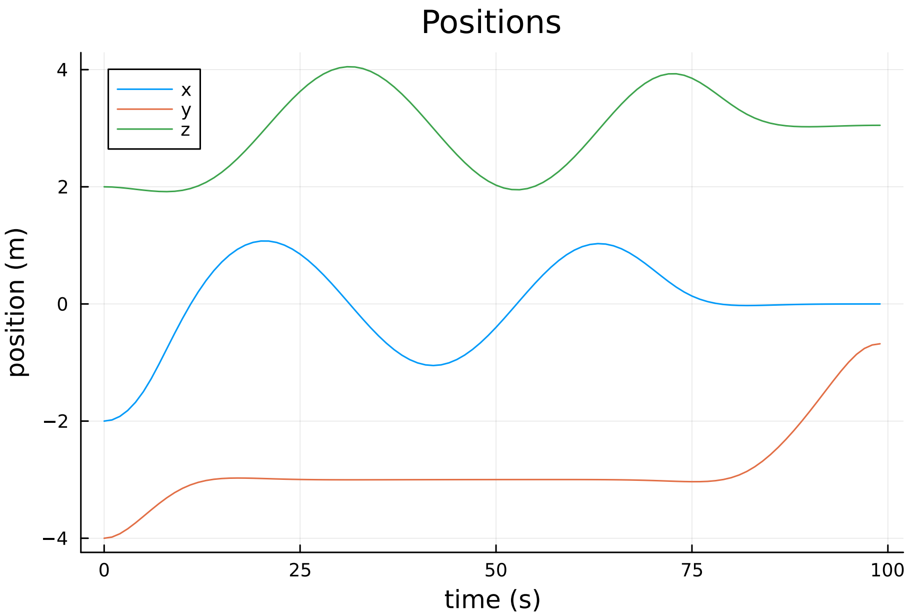
Now we are going to assume that we have a perfect model (assume there is no sim to real gap), and that we have a perfect state estimate. With this, we are going to solve our control problem as a convex trajectory optimization problem.

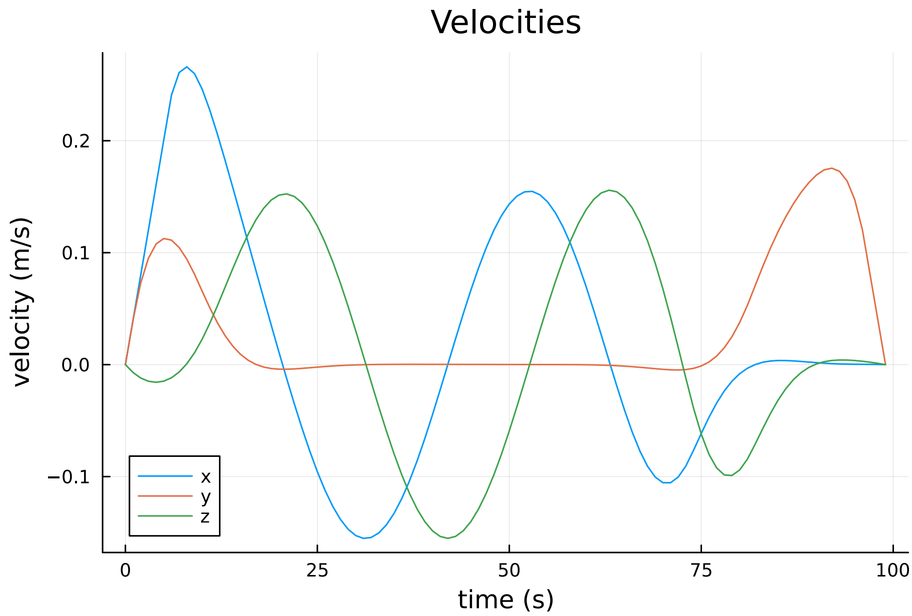
\begin{align} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \bigg[\frac{1}{2} (x_i - x_{ref, i})^TQ(x_i - x_{ref, i}) + \frac{1}{2} u_i^TRu_i \bigg] \\ \text{st} \quad & x_1 = x_{\text{IC}} \\ & x_{i+1} = A x_i + Bu_i \quad \text{for } i = 1,2,\ldots,N-1 \\ & u_{min} \leq u_i \leq u_{max} \quad \text{for } i = 1,2,\ldots,N-1 \\ & x_i[2] \leq x_{goal} [2]\quad \text{for } i = 1,2,\ldots,N \\ & x_N = x_{goal} \end{align}

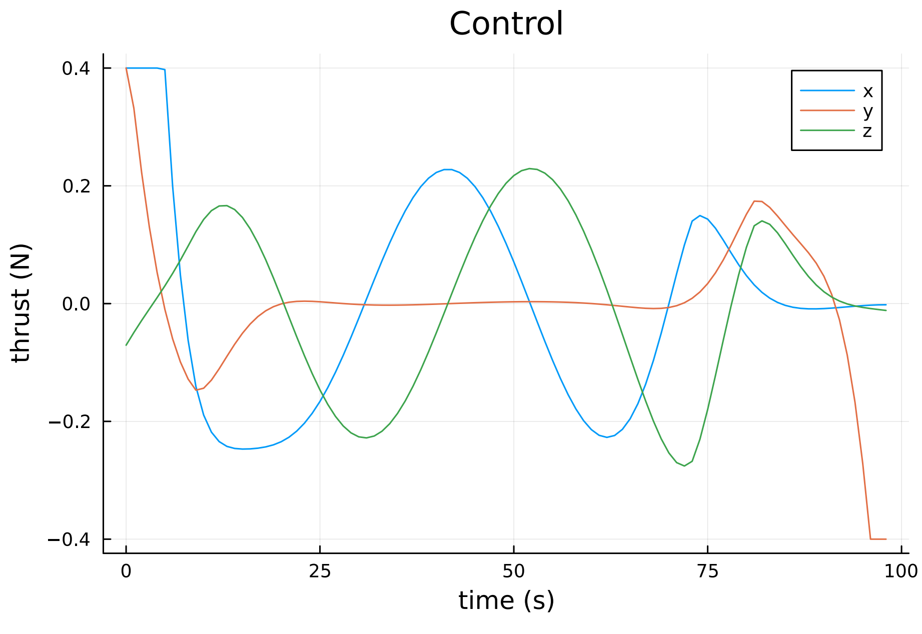
Where we have an LQR cost, an initial condition constraint (x_1 = x_{\text{IC}}), linear dynamics constraints (x_{i+1} = A x_i + Bu_i), bound constraints on the control (\leq u_i \leq u_{max}), an ISS collision constraint (x_i[2] \leq x_{goal} [2]), and a terminal constraint (x_N = x_{goal}). This problem is convex and we will setup and solve this with `Convex.jl`.

In [ ]:
```julia
"""
Xcvx,Ucvx = convex_trajopt(A,B,X_ref,x0,xg,u_min,u_max,N)

setup and solve the above optimization problem, returning
the solutions X and U, after first converting them to
vectors of vectors with vec_from_mat(X.value)
"""
function convex_trajopt(A::Matrix, # discrete dynamics A
                        B::Matrix, # discrete dynamics B
                        X_ref::Vector{Vector{Float64}}, # reference trajectory
                        x0::Vector, # initial condition
                        xg::Vector, # goal state
                        u_min::Vector, # lower bound on u
                        u_max::Vector, # upper bound on u
                        N::Int64, # length of trajectory
                        )::Tuple{Vector{Vector{Float64}}, Vector{Vector{Float64}}} # return Xcvx,Ucvx

    # get our sizes for state and control
    nx,nu = size(B)

    @assert size(A) == (nx, nx)
    @assert length(x0) == nx
    @assert length(xg) == nx

    # LQR cost
    Q = diagm(ones(nx))
    R = diagm(ones(nu))

    # variables we are solving for
    X = cvx.Variable(nx,N)
    U = cvx.Variable(nu,N-1)

    # TODO: implement cost
    obj = sum(0.5 * cvx.quadform(X[:, i] - X_ref[i], Q) for i in 1:N) +
      sum(0.5 * cvx.quadform(U[:, i], R) for i in 1:N-1)

    # create problem with objective
    prob = cvx.minimize(obj)

    # TODO: add constraints with prob.constraints = vcat(prob.constraints, ...)
    constraints = []
    push!(constraints, X[:, 1] == x0)
    for i in 1:N-1
        push!(constraints, X[:, i+1] == A*X[:, i] + B*U[:, i])
        push!(constraints, X[2, i] <= xg[2])
        push!(constraints, U[:, i] >= u_min)
        push!(constraints, U[:, i] <= u_max)
    end
    push!(constraints, X[:, N] == xg)

    prob.constraints = constraints

    cvx.solve!(prob, ECOS.Optimizer; silent = true)

    X = X.value
    U = U.value

    Xcvx = vec_from_mat(X)
    Ucvx = vec_from_mat(U)

    return Xcvx, Ucvx
end

@testset "convex trajopt" begin

    # create our discrete time model
    dt = 1.0
    A,B = create_dynamics(dt)

    # get our sizes for state and control
    nx,nu = size(B)

    # initial and goal states
    x0 = [-2;-4;2;0;0;.0]
    xg = [0,-.68,3.05,0,0,0]

    # bounds on U
    u_max = 0.4*ones(3)
    u_min = -u_max
```

```julia
    # problem size and reference trajectory
    N = 100
    t_vec = 0:dt:((N-1)*dt)
    X_ref = desired_trajectory(x0,xg,N,dt)

    # solve convex trajectory optimization problem
    X_cvx, U_cvx = convex_trajopt(A,B,X_ref, x0,xg,u_min,u_max,N)

    X_sim = [zeros(nx) for i = 1:N]
    X_sim[1] = x0
    for i = 1:N-1
        X_sim[i+1] = A*X_sim[i] + B*U_cvx[i]
    end

    # -------------plotting/animation--------------------------
    Xm = mat_from_vec(X_sim)
    Um = mat_from_vec(U_cvx)
    display(plot(t_vec,Xm[1:3,:]',title = "Positions",
                 xlabel = "time (s)", ylabel = "position (m)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec,Xm[4:6,:]',title = "Velocities",
            xlabel = "time (s)", ylabel = "velocity (m/s)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec[1:end-1],Um',title = "Control",
            xlabel = "time (s)", ylabel = "thrust (N)",
                 label = ["x" "y" "z"]))


    display(animate_rendezvous(X_sim, X_ref, dt;show_reference = false))
    # -------------plotting/animation--------------------------


    @test maximum(norm.( X_sim .- X_cvx, Inf)) < 1e-3
    @test norm(X_sim[end] - xg) < 1e-3 # goal
    xs=[x[1] for x in X_sim]
    ys=[x[2] for x in X_sim]
    zs=[x[3] for x in X_sim]
    @test maximum(ys) <= (xg[2] + 1e-3)
    @test maximum(zs) >= 4 # check to see if you did the circle
    @test minimum(zs) <= 2 # check to see if you did the circle
    @test maximum(xs) >= 1 # check to see if you did the circle
    @test maximum(norm.(U_cvx,Inf)) <= 0.4 + 1e-3 # control constraints satisfied

end
```








```
┌ Info: Listening on: 127.0.0.1:8715, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8715
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

```
Test Summary:  | Pass  Total  Time
convex trajopt |    7      7  1.9s
Test.DefaultTestSet("convex trajopt", Any[], 7, false, false, true, 1.740013198067e9, 1.740013199931e9, false, "
c:\\CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X14sZmlsZQ==.jl")
```

Part D: Convex MPC (20 pts)

In part C, we solved for the optimal rendezvous trajectory using convex optimization, and verified it by simulating it in an open loop fashion (no feedback). This was made possible because we assumed that our linear dynamics were exact, and that we had a perfect estimate of our state. In reality, there are many issues that would prevent this open loop policy from being successful; Here are a few:

- imperfect state estimation
- unmodeled dynamics
- misalignments
- actuator uncertainties

Together, these factors result in a "sim to real" gap between our simulated model, and the real model. Because there will always be a sim to real gap, we can't just execute open loop policies and expect them to be successful. What we can do, however, is use Model Predictive Control (MPC) that combines some of the ideas of feedback control with convex trajectory optimization.

A convex MPC controller will set up and solve a convex optimization problem at each time step that incorporates the current state estimate as an initial condition. For a trajectory tracking problem like this rendezvous, we want to track x_{ref}, but instead of optimizing over the whole trajectory, we will only consider a sliding window of size N_{mpc} (also called a horizon). If $N_{mpc} = 20$, this means our convex MPC controller is reasoning about the next 20 steps in the trajectory. This optimization problem at every timestep will start by taking the relevant reference trajectory at the current window from the current step i, to the end of the window $i + N_{mpc} - 1$. This slice of the reference trajectory that applies to the current MPC window will be called $\tilde{x}_{ref} = x_{ref}[i,(i + N_{mpc} - 1)]$.

$$\begin{align} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \bigg[\frac{1}{2} (x_i - \tilde{x}_{ref, i})^TQ({x}_i - \tilde{x}_{ref, i}) + \frac{1}{2} u_i^TRu_i \bigg] + \frac{1}{2}(x_N- \tilde{x}_{ref, N})^TQ ({x}_N- \tilde{x}_{ref, N})\\ \text{st} \quad & x_1 = x_{\text{IC}} \\ & x_{i+1} = A x_i + Bu_i \quad \text{for } i = 1,2,\ldots,N-1 \\ & u_{min} \leq u_i \leq u_{max} \quad \text{for } i = 1,2,\ldots,N-1 \\ & x_i[2] \leq x_{goal} [2]\quad \text{for } i = 1,2,\ldots,N \end{align}$$

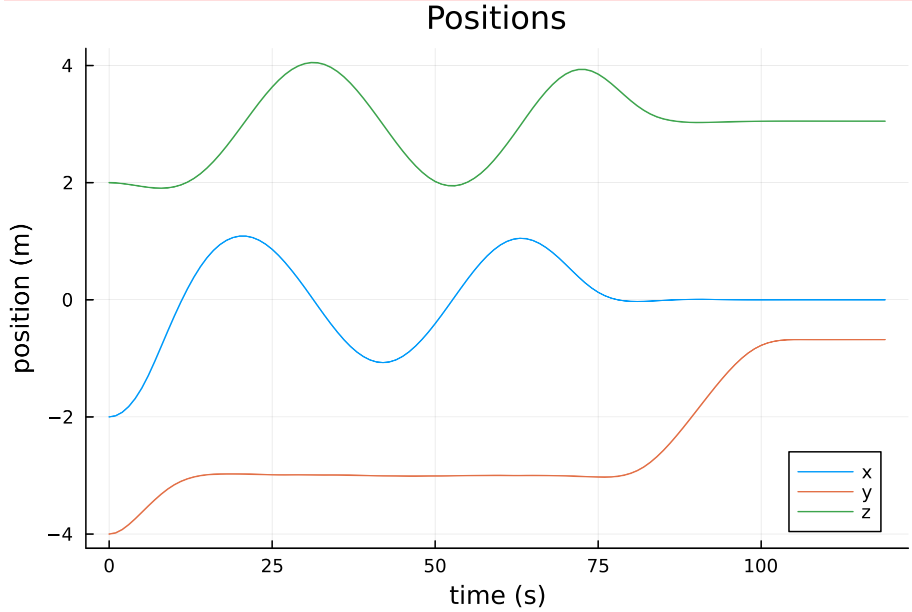
where N in this case is N_{mpc}. This allows for the MPC controller to "think" about the future states in a way that the LQR controller cannot. By updating the reference trajectory window (\tilde{x}_{ref}) at each step and updating the initial condition (x_{IC}), the MPC controller is able to "react" and compensate for the sim to real gap.

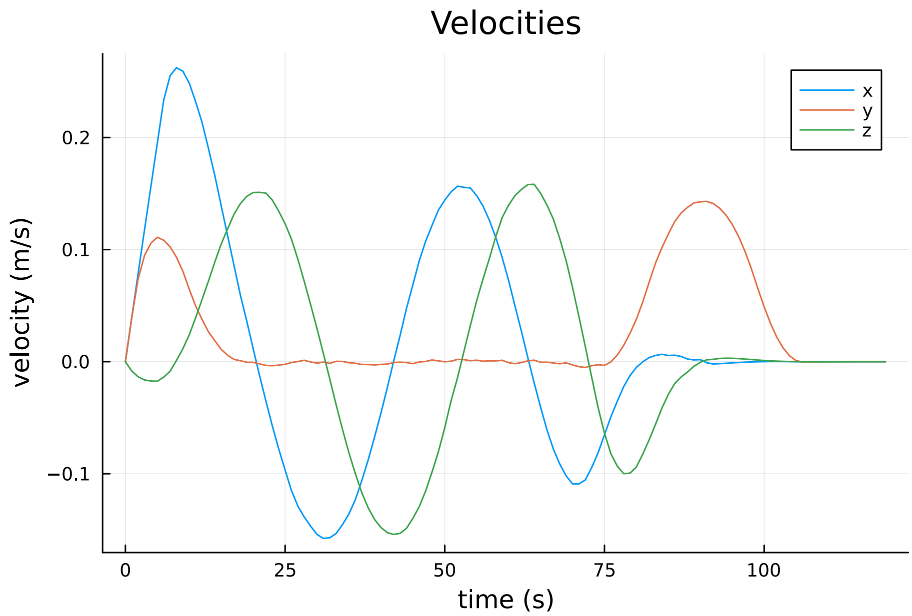
You will now implement a function `convex_mpc` where you setup and solve this optimization problem at every timestep, and simply return u_1 from the solution.

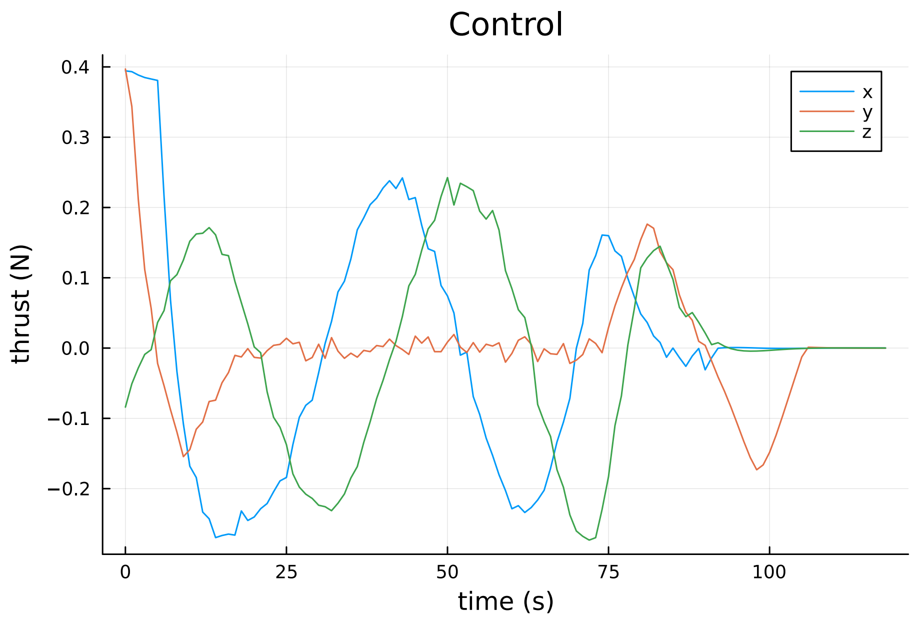
In [40]:
```julia
"""
`u = convex_mpc(A,B,X_ref_window,xic,xg,u_min,u_max,N_mpc)`

setup and solve the above optimization problem, returning the
first control u_1 from the solution (should be a length nu
Vector{Float64}).
"""
function convex_mpc(A::Matrix, # discrete dynamics matrix A
                    B::Matrix, # discrete dynamics matrix B
                    X_ref_window::Vector{Vector{Float64}}, # reference trajectory for this window
                    xic::Vector, # current state x
                    xg::Vector, # goal state
                    u_min::Vector, # lower bound on u
                    u_max::Vector, # upper bound on u
                    N_mpc::Int64,  # length of MPC window (horizon)
                    )::Vector{Float64} # return the first control command of the solved policy

    # get our sizes for state and control
    nx,nu = size(B)

    # check sizes
    @assert size(A) == (nx, nx)
    @assert length(xic) == nx
    @assert length(xg) == nx
    @assert length(X_ref_window) == N_mpc

    # LQR cost
    Q = diagm(ones(nx))
    R = diagm(ones(nu))
    Qf = 10*Q

    # variables we are solving for
    X = cvx.Variable(nx,N_mpc)
    U = cvx.Variable(nu,N_mpc-1)

    # TODO: implement cost function
    obj = sum(0.5 * cvx.quadform(X[:, i] - X_ref_window[i], Q) for i in 1:N_mpc-1) +
        sum(0.5 * cvx.quadform(U[:, i], R) for i in 1:N_mpc-1) + 0.5*(cvx.quadform(X[:, N_mpc] - X_ref_window[N_


    # create problem with objective
    prob = cvx.minimize(obj)

    # TODO: add constraints with prob.constraints = vcat(prob.constraints, ...)
    constraints = []
    push!(constraints, X[:, 1] == xic)
    for i in 1:N_mpc-1
        push!(constraints, X[:, i+1] == A*X[:, i] + B*U[:, i])
        push!(constraints, X[2, i] <= xg[2])
        push!(constraints, U[:, i] >= u_min)
        push!(constraints, U[:, i] <= u_max)
    end

    prob.constraints = constraints

    # solve problem
    cvx.solve!(prob, ECOS.Optimizer; silent = true)

    # get X and U solutions
    X = X.value
    U = U.value

    # return first control U
    return U[:,1]
end

@testset "convex mpc" begin

    # create our discrete time model
    dt = 1.0
    A,B = create_dynamics(dt)

    # get our sizes for state and control
    nx,nu = size(B)
```

```julia
    # initial and goal states
    x0 = [-2;-4;2;0;0;.0]
    xg = [0,-.68,3.05,0,0,0]

    # bounds on U
    u_max = 0.4*ones(3)
    u_min = -u_max

    # problem size and reference trajectory
    N = 100
    t_vec = 0:dt:((N-1)*dt)
    X_ref = [desired_trajectory(x0,xg,N,dt)...,[xg for i = 1:N]...]

    # MPC window size
    N_mpc = 20

    # sim size and setup
    N_sim = N + 20
    t_vec = 0:dt:((N_sim-1)*dt)
    X_sim = [zeros(nx) for i = 1:N_sim]
    X_sim[1] = x0
    U_sim = [zeros(nu) for i = 1:N_sim-1]

    # simulate
    @showprogress "simulating" for i = 1:N_sim-1

        # get state estimate
        xi_estimate = state_estimate(X_sim[i], xg)

        # TODO: given a window of N_mpc timesteps, get current reference trajectory
        if i + N_mpc - 1 > length(X_ref)
            X_ref_tilde = X_ref[i:end]  # Truncate to available states
        else
            X_ref_tilde = X_ref[i:i+N_mpc-1]
        end

        # TODO: call convex mpc controller with state estimate
        u_mpc = convex_mpc(A, B, X_ref_tilde, xi_estimate, xg, u_min, u_max, N_mpc)

        # commanded control goes into thruster model where it gets modified
        U_sim[i] = thruster_model(X_sim[i], xg, u_mpc)

        # simulate one step
        X_sim[i+1] = A*X_sim[i] + B*U_sim[i]
    end



    # -------------plotting/animation---------------------------
    Xm = mat_from_vec(X_sim)
    Um = mat_from_vec(U_sim)
    display(plot(t_vec,Xm[1:3,:]',title = "Positions",
                 xlabel = "time (s)", ylabel = "position (m)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec,Xm[4:6,:]',title = "Velocities",
            xlabel = "time (s)", ylabel = "velocity (m/s)",
                 label = ["x" "y" "z"]))
    display(plot(t_vec[1:end-1],Um',title = "Control",
            xlabel = "time (s)", ylabel = "thrust (N)",
                 label = ["x" "y" "z"]))


    display(animate_rendezvous(X_sim, X_ref, dt;show_reference = false))
    # -------------plotting/animation---------------------------

    # tests
    @test norm(X_sim[end] - xg) < 1e-3 # goal
    xs=[x[1] for x in X_sim]
    ys=[x[2] for x in X_sim]
    zs=[x[3] for x in X_sim]
    @test maximum(ys) <= (xg[2] + 1e-3)
    @test maximum(zs) >= 4 # check to see if you did the circle
    @test minimum(zs) <= 2 # check to see if you did the circle
    @test maximum(xs) >= 1 # check to see if you did the circle
    @test maximum(norm.(U_sim,Inf)) <= 0.4 + 1e-3 # control constraints satisfied

end
```

```
simulating 100%|██████████████████████████████████████████| Time: 0:00:01
┌ Info: Listening on: 127.0.0.1:8716, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8716
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```







```
Test Summary: | Pass  Total  Time
convex mpc    |    6      6  2.5s
Test.DefaultTestSet("convex mpc", Any[], 6, false, false, true, 1.740013861301e9, 1.740013863808e9, false, "c:\\
CMU\\SEM II\\OCRL\\HW2_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X16sZmlsZQ==.jl")
```